Interprocedural Slicing of Multithreaded Programs with Applications to Java

MANGALA GOWRI NANDA
IBM, India Research Laboratory
and
S. RAMESH
Indian Institute of Technology, Bombay

Slicing is a well-known program reduction technique where for a given program P and a variable of interest v at some statement p in the program, a program slice contains those set of statements belonging to P that affect v. This article presents two algorithms for interprocedural slicing of concurrent programs–a *context-insensitive* algorithm and a *context-sensitive* algorithm. The context-insensitive algorithm is efficient and correct (it includes every statement that may affect the slicing criterion) but is imprecise since it may include certain extra statements that are unnecessary. Precise slicing has been shown to be undecidable for concurrent programs. However, the context-sensitive algorithm computes correct and reasonably precise slices, but has a worst-case exponential-time complexity. Our context-sensitive algorithm computes a closure of dependencies while ensuring that statements sliced in each thread belong to a *realizable* path in that thread.

A realizable path in a thread with procedure calls is one that reflects the fact that when a procedure finishes, execution returns to the site of the most recently executed call in that thread. One of the novelties of this article is a practical solution to determine whether a given set of statements in a thread may belong to a realizable path. This solution is precise even in the presence of recursion and long call chains in the flow graph.

The slicing algorithms are applicable to concurrent programs with shared memory, interleaving semantics, explicit `wait/notify` synchronization and monitors. We first give a solution for a simple model of concurrency and later show how to extend the solution to the Java concurrency model. We have implemented the algorithms for Java bytecode and give experimental results.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Multithreading, data dependence, interference dependence, context-sensitivity, strongly connected regions, program slicing

This work was partially supported by the Indo-US project titled "Programming Dynamical Real-Time Systems".

Authors' addresses: M. G. Nanda, IBM, India Research Laboratory, Block 1, IIT, New Delhi 110016, India; email: mgowri@in.ibm.com; S. Ramesh, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, Powai, Mumbai 400076; email: ramesh@cse.iitb.ac.in.

1. INTRODUCTION

Slicing is a program reduction technique that is useful in debugging [Agrawal et al. 1993], program maintenance [Gallagher and Lyle 1991], Y2K compliance transformations [Nanda et al. 1999], reverse engineering [Beck and Eichmann 1993], program testing [Harman and Danicic 1995] and applications that involve understanding program behavior. The *slice* of a program with respect to a program point p and a variable x, consists of all statements and predicates of the program that might affect the value of x at point p [Weiser 1984]. More details may be found in Binkley and Gallagher [1996] and Tip [1995].

In this article we show how to compute the interprocedural slice of concurrent programs. We consider two models of concurrency - a simpler model where concurrency is represented by the classical `cobegin-coend` [Dijkstra 1968] statements and the Java concurrency model.

The original slicing algorithm by Weiser [Weiser 1984] was based on iterative data flow analysis. Subsequently Ottenstein and Ottenstein [Ottenstein and Ottenstein 1984] introduced the notion of slicing using Program Dependence Graphs (PDG). A PDG is a graph in which nodes, representing assignments and conditions in the program, are connected by control and data dependence edges. The slice is defined with respect to a given node in this graph as the set of all the nodes on which the given node is directly or transitively dependent. Thus, given the PDG, the slice can be computed by a simple reachability algorithm [Ottenstein and Ottenstein 1984; Horwitz et al. 1989].

Venkatesh [Venkatesh 1991] classifies static slicing algorithms as "executable" and "closure" slices. Closure slices contain the set of statements that are related to the variable of interest through a closure of dependences and are not necessarily either syntactically correct or executable programs, that is, programs which on execution preserve the behavior of the original program. Weiser's [1984] algorithm produces executable slices. However, his algorithm does not produce precise slices for programs with procedures since it fails to account for the calling context of procedures. Horwitz et al. [1990] were the first to address the issue of calling contexts and gave a closure based context-sensitive slicing algorithm for slicing programs with procedures. Papers on generating semantically correct slices for sequential programs include Harman et al. [2003], Yang et al. [1992], and Binkley et al. [1995]. In this article, we present a closure algorithm for generating context-sensitive slices for concurrent programs. This work appeared in the Ph.D. dissertation of the first author [Nanda 2001].

In slicing programs, there are two main issues:

(1) *Correctness*. A slice is correct if it includes, at least, all the program statements that affect the criterion. In the extreme case, the complete program is always a correct slice. A slice is incorrect if it excludes some program statements that should be in the slice.
(2) *Precision*. Given two correct slices, one slice is more precise than the other if it contains fewer program statements.

Ideally, one would like to compute the correct slice that is the most precise. But this is, in general, not computable [Weiser 1984; Müller-Olm and Seidl

2001; Ramalingam 2000]. Hence, we try to compute a slice that is correct and as precise as possible. There are three main sources of imprecision in closure based interprocedural slicing of concurrent programs—(1) interprocedural paths are not transitive, (2) interference dependence is not transitive and (3) backwards reachability in a dependence graph may give imprecise results.

(1) A *realizable* path is one that corresponds to a legal call/return sequence where each return statement brings control back to the point just after the corresponding procedure call was made [Sharir and Pnueli 1981]. Interprocedural paths may not be transitive. A realizable path from n_1 to n_2 and a realizable path from n_2 to n_3 does not imply that n_1, n_2 and n_3 lie along a realizable path. The problem of intransitivity of paths in interprocedural slicing of sequential programs has been solved by Horwitz et al. [1990]. A context-sensitive slice is one that computes only those nodes that lie on a realizable path to the node that represents the slicing criterion. It has been shown [Harman et al. 2003], that a context-sensitive slice may be more precise than a context-insensitive slice.

(2) Interference dependence arises when a node uses a variable that was defined in a parallel executing thread. Interference dependence is not transitive. Consider two nodes n_i^1 and n_k^1 in a thread θ_1 and a node n_j^2 in a thread θ_2. An interference dependence from n_i^1 to n_j^2 and an interference dependence from n_j^2 to n_k^1 does not imply a dependence from n_i^1 to n_k^1 unless there is a path from n_i^1 to n_k^1 in θ_1. This problem was first identified and solved by Krinke [Krinke 1998].

(3) Müller-Olm and Seidl [2001] show that backward reachability in the dependence graph can give sub-optimal results when slicing concurrent programs. This is because of a further weakness in interference dependence. Their results show that an interference dependence from n_i^1 to n_j^2 and an interference dependence from n_j^2 to n_k^1 does not imply a dependence from n_i^1 to n_k^1 *even* if there is a path from n_i^1 to n_k^1 in θ_1. This is because the definition at n_i^1 may get killed along concurrent threads.

In this article, we present a context-sensitive slicing algorithm for concurrent programs. Due to the limitations of backward reachability in the presence of interference dependences, our algorithm generates a conservative slice. A conservative slice is one that includes every node that is required as well as some unnecessary nodes.

In order to compute a context-sensitive slice it is necessary to ensure that nodes sliced in a particular thread form a realizable path in that thread.[1] We give a practical solution to determine whether a given set of nodes belonging to a thread lie on a realizable path in that thread. We call this the "Realizable Path" problem. A more formal definition is given in Section 5. We give a precise solution to the realizable path problem. Essentially, we collapse every strongly connected region in the call graph into a single node each. On the resultant acyclic interprocedural call graph, we generate a topological ordering of the nodes that performs a virtual inlining of called procedures. The topological ordering renders the interprocedural call graph into an equivalent intraprocedural call graph. Then, when adding a node to a path it is not necessary to

[1]Note that computation of realizable paths pertains to a *single* thread of a concurrent program.

check the entire path for the existence of a realizable path—it is sufficient to check it against the previously added node based on the topological numbering. Thus, path reachability is performed efficiently and precisely.

The context-sensitive algorithm has exponential complexity. We also give a more efficient but less precise, context-insensitive slicing algorithm. The algorithms are described using the `cobegin-coend` model of concurrency. However, we show how to map the Java concurrency model to the `cobegin-coend` model (with certain limitations). We have implemented both the algorithms on Java bytecode and tested them on Java programs (maximum size of 10^5 statements). Although the context-sensitive algorithm has exponential complexity we found that, with certain optimizations, it was practical for the programs that we used for testing.

Recently, Krinke [Krinke 2003] has published a solution to context-sensitive interprocedural slicing of concurrent programs. To ensure that nodes sliced in a thread belong to a realizable path, Krinke maintains "callstrings" with each node in the slice. The callstrings keep track of the calling context. However, when slicing with callstrings it is not possible to make use of (intra-thread) summary edges and this makes the algorithm extremely expensive, not only in terms of maintaining callstrings but also in terms of visiting called procedures repeatedly for each calling context. Further, the callstrings approach suffers from combinatorial explosion of the callstrings and is usable only if the length of the callstrings is limited to 2 or 3 elements—which decreases the precision. To partially improve the efficiency, Krinke computes a "chop" of the graph between the slicing criterion and every node in the thread that has an incoming interference dependence edge. Nodes that are not in the chop may be sliced using summary edges. However, the remaining nodes must be sliced using the expensive and imprecise callstring approach. Further, maintenance of chops may be complex in the case that there are multiple slicing criterion in a single thread. We compare our algorithm with Krinke's in Section 10.

An algorithm for interprocedural slicing of concurrent programs has been presented by Zhao [Zhao 1999]. However, we show that this algorithm is neither context-sensitive nor correct.

Contributions

—A context-insensitive slicing algorithm for concurrent programs which is correct and has polynomial complexity.

—An algorithm to determine whether a given set of nodes may form a realizable path in a thread.

—A context-sensitive slicing algorithm for concurrent programs which is comparatively more precise but has a worst case exponential complexity.

—Optimizations on the basic context-sensitive slicing algorithm and experimental evaluation of the algorithms.

1.1 Organization

In Section 2, we give some background information about slicing of sequential and concurrent programs, and in Section 3, we motivate the issues of correctness

and precision of interprocedural slices for concurrent programs. In Section 4, we give a context-insensitive slicing algorithm for a simple concurrency model. Section 5 gives an algorithm to determine whether a given set of nodes form a realizable path. Using this algorithm we extend the context-insensitive algorithm to a context-sensitive solution by ensuring that the nodes sliced in each thread lie on a realizable path in that thread (Section 6). In Section 7, we extend the algorithm to programs with nested threads and threads nested within loops. In Section 8, we extend the algorithm to concurrent Java. We give experimental results in Section 9. Section 10 gives information about related work and Section 11 concludes this article.

2. BACKGROUND

As much work has been done in the area of slicing, we give a brief overview of the current state of the art in sequential and concurrent programs.

2.1 Interprocedural Slicing of Sequential Programs

Unlike intraprocedural slicing, mere reachability produces imprecise slices in programs with procedure calls. The interprocedural slice of a program is computed on the *system dependence graph (SDG)* using a two-phase algorithm [Horwitz et al. 1990]. Each procedure in the program is represented by a PDG, and the SDG is built by joining the PDGs with special edges. At every procedure there is a *formal-in* node for each global variable and parameter that may be modified or referenced and a *formal-out* node for each global variable and parameter that may be modified by the procedure. At each call site, there is an *actual-in* node for each formal-in node and an *actual-out* node for each formal-out node. There is a *call* edge from the call node to the `ENTRY` node of the called procedure. *Parameter-in* edges connect corresponding actual-in nodes to formal-in nodes, and *parameter-out* edges connect formal-out nodes to actual-out nodes. *Summary* edges connect actual-in nodes to actual-out nodes to represent transitive flow dependencies induced by called procedures. The slicing criterion is a node in the SDG.

Horwitz et al. [1990] circumvent the calling context problem using summary edges in a two-phase slicing algorithm. In the first phase the algorithm identifies the nodes that can be reached from the slicing criterion, s in procedure P, without descending into procedures called by P. The algorithm traverses data, control, summary, call and parameter-in edges and ignores parameter-out edges. In the second phase the algorithm identifies nodes that reach s from procedures (transitively) called by P. The algorithm traverses data, control, summary and parameter-out edges and ignores parameter-in and call edges. The summary edges allow the algorithm to slice *across* an entire procedure without descending into it.

2.2 Intraprocedural Slicing of Concurrent Programs

Consider the simplest model of concurrent programs that consist of processes or threads which interact via shared variables with atomic memory reads and writes. Threads share the same address space and execute concurrently with

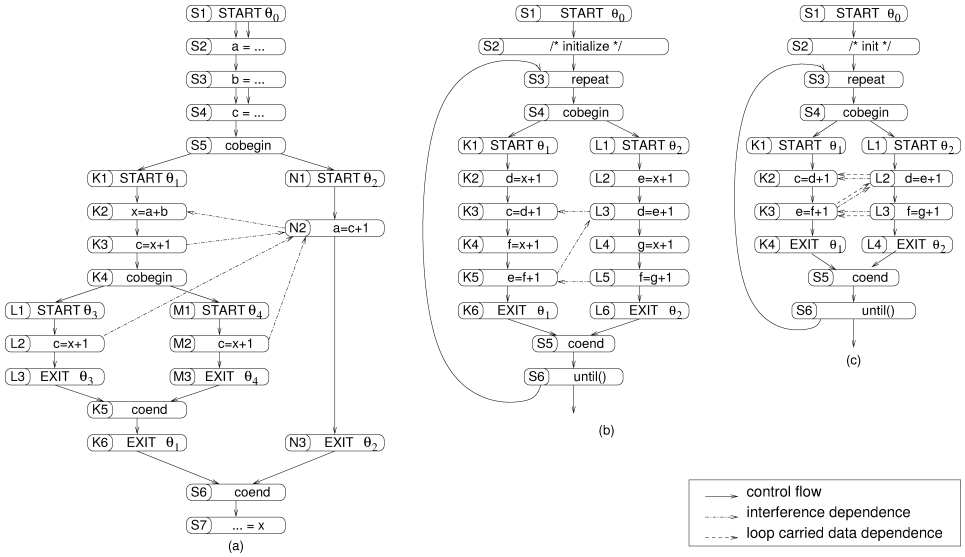


Fig. 1. (a) A threaded CFG with nested threads, (b), (c) threaded CFGs with threads nested within a loop.

each other with complete interleaving semantics. There is no explicit synchronization between the threads. At the language level the classical `cobegin-coend` construct is used to express the parallelism. Threads generated by a `cobegin` statement synchronize at the corresponding `coend` statement. There is no other synchronization between the threads. An abstract representation of the concurrent programs is the *threaded control flow graph* (TCFG) [Krinke 1998] as shown in Figure 1.

Let `START` θ_i represent the `START` node of a thread θ_i and `START` θ_j represent the `START` node of a thread θ_j, then the two threads may execute in parallel if the closest common ancestor of `START` θ_i and `START` θ_j is a `cobegin` node. We define a function $||(n_i, n_j)$ to be true for two nodes n_i and n_j, if they belong to threads that may execute in parallel. Interference dependence is defined as:

Definition 1. A node n_1 is interference dependent on a node n_2 if n_2 defines a variable v, n_1 uses the variable v, and n_1 and n_2 execute in parallel.

The *Threaded PDG* (TPDG) is an extension of the PDG to concurrent programs. TPDGs capture, besides data and control dependencies, additional dependencies arising out of *interference* between concurrent execution of threads. Interference dependence is fundamentally different from data or control dependence as it is intransitive. In Figure 1(a), `K2` is interference dependent on `N2` and `N2` is interference dependent on `K3`. Yet `K2` cannot be dependent on `K3` in any execution of the program. Hence a slicing algorithm that computes a simple transitive closure on the TPDG could generate an imprecise slice.

2.2.1 *The Basic Algorithm.* The reachability algorithm can be extended to compute comparatively precise slices [Krinke 1998]. The slicing algorithm

starts at the slicing criterion s and traverses backwards along data, control and interference dependence edges. At each step the algorithm maintains a tuple of nodes indicating the last node traversed in each thread. When following an interference dependence edge from a node n_j in a thread θ_j to a node n_i in a thread θ_i, it checks if there is a path from n_i to the last node traversed in θ_i. For example, in Figure 1(a) to find the slice of K2 in θ_1, the algorithm creates the tuple [$\perp$, K2, $\perp$, $\perp$, $\perp$] which has five elements—one for each thread in the graph. The tuple indicates that K2 was the last node visited in θ_1 and $\perp$ indicates that no node has been visited in the corresponding thread. K2 is interference dependent on N2 in θ_2. The node corresponding to θ_2 in the tuple is $\perp$ and so N2 is added to the slice with the tuple [$\perp$, K2, N2, $\perp$, $\perp$]. Next N2 is interference dependent on K3 in θ_1. The tuple node corresponding to θ_1 is K2 and there is no path from K3 to K2 and hence K3 is rejected from the slice. However, this algorithm is imprecise in the presence of nested threads. N2 is interference dependent on L2 in θ_3 and M2 in θ_4. The tuple nodes corresponding to θ_3 and θ_4 are $\perp$ and so L2 and M2 would get added to the slice (which is imprecise).

2.2.2 *Nested Threads.* Krinke's algorithm has been extended to handle nested threads and threads nested within loops [Nanda and Ramesh 2000]. We give an informal description of the extensions. To handle nested threads the algorithm is extended as follows. In Figure 1(a) to find the slice of K2 in θ_1, the algorithm creates the tuple [K2, K2, $\perp$, K2, K2]. The node corresponding to θ_1 is marked K2 as this was the last node traversed in θ_1. The nodes corresponding to θ_0, θ_3 and θ_4 are *also* marked K2 since each of these threads may execute sequentially with θ_1 (this is required to handle nested threads). K2 is interference dependent on N2 in θ_2. The tuple node corresponding to θ_2 is $\perp$ and so N2 is added to the slice with the tuple [N2, K2, N2, K2, K2]. Since θ_0 executes sequentially with θ_2, the tuple corresponding to θ_0 is also updated to N2. The other tuple nodes are not changed. N2 is interference dependent on K3 in θ_1, L2 in θ_3 and M2 in θ_4. The tuple node corresponding to these threads is K2. Since there is no path from K3, L2 or M2 to K2, each of these nodes will be rejected.

2.2.3 *Threads Nested in Loops.* In Figure 1(b), K3 is interference dependent on L3 which is interference dependent on K5. There is a path from K5 to K3 induced by the loop and so the slicing algorithm would add K5 to the slice of K3. But it is not possible for K3 to be transitively dependent on K5 as the definition at K5 is killed by the definition at L2 and the definition at L3 is killed by the definition at K2. On the other hand, in Figure 1(c) K2 is interference dependent on L2 which is interference dependent on K3. In this case it is correct to add K3 to the slice of K2. To handle threads nested within loops, the algorithm differentiates between data dependences and loop-carried data dependences that cross thread boundaries. In Figure 1(c), there are two edges between K2 and L2, an interference dependence edge and a loop-carried data dependence. A loop is induced by a backedge in which the destination dominates the source [Aho et al. 1986]. Let n_b be the source of the backedge of the loop that induces a loop-carried data dependence. Then each loop-carried data dependence is treated as a traversal through n_b and tuples are updated accordingly. In addition when

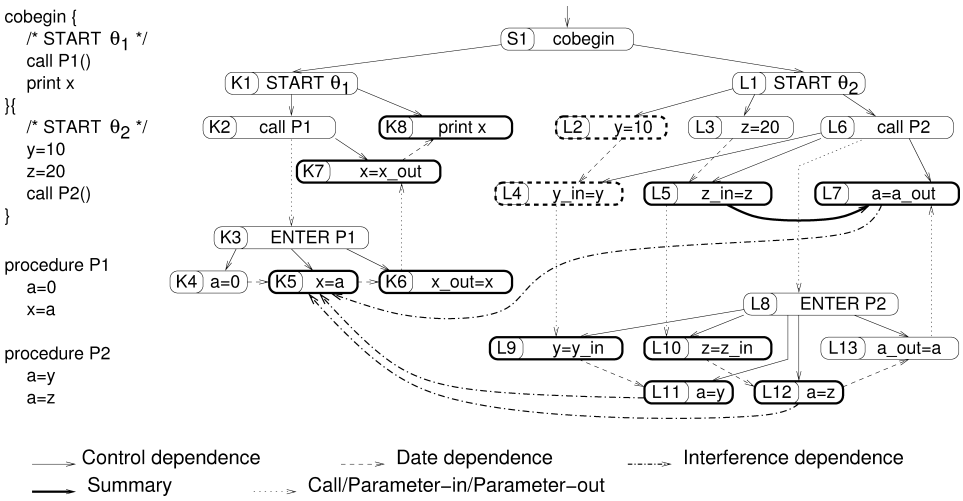


Fig. 2. Example showing incorrectness of the two-phase algorithm. The nodes in the thickened boxes help navigate the slice of `print x`. The nodes in the dashed boxes are the ones that do not get added to the slice, thus generating an incorrect slice.

traversing an interference dependence edge from n_i in θ_i to n_j in θ_j, let n'_j be the last node visited in θ_j. Then the reachability test from n_j to n'_j is restricted to paths enclosed within relevant subregions rather than the entire TCFG. The relevant region is defined as the region enclosed between the `cobegin` node that is the closest common ancestor of n_i and n_j and its corresponding `coend` node. Details and proof of correctness may be found in Nanda and Ramesh [2000]. Note that this approach is limited to structured programs as loop-carried data dependence is not defined in irreducible CFGs.

3. MOTIVATION—INTERPROCEDURAL SLICING OF CONCURRENT PROGRAMS

Consider simple concurrent programs with procedures based on `cobegin-coend` parallelism where threads are not nested and threads may not be nested within loops. The concurrent program is represented by an *interprocedural threaded control flow graph (ITCFG)*. A *threaded system dependence graph* (TSDG) is an SDG with interference dependence edges (as shown in Figure 2). The complete set of edges, E in the TSDG are the data dependence, control dependence, interference dependence, call, summary, parameter-in and parameter-out edges which are denoted by E_{dd}, E_{cd}, E_{id}, E_c, E_s, E_{pi} and E_{po} respectively. The slicing criterion is defined as a node in the TSDG.

Müller-Olm and Seidl [2001] define for a program point p of a program P, the optimal slice $\mathcal{S}_{opt}(p)$ is the set of statements n_i that affect p given that all conditionals in P are interpreted as non-deterministic choices. They have also shown that $\mathcal{S}_{opt}(p)$ is not computable.

On the ITCFG, G^* of a concurrent program, we define:

Definition 2. A realizable path in a thread of a concurrent program is a path in which "returns" are matched with corresponding "calls" [Reps et al. 1994].

Definition 3. An *interprocedural threaded witness* is an ordered sequence of nodes $\langle n_1, n_2, \ldots, n_k \rangle$ belonging to G^* such that any subsequence of nodes $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ belonging to the same thread, θ_i, form a realizable path in θ_i.

Now we define our notion of slice, which is weaker than optimal slices. Informally, the *context-sensitive* slice $\mathcal{S}(p)$ of a TSDG at a node p consists of all nodes n on which p is transitively dependent in such a manner that only interprocedural threaded witnesses are generated in the slice. $\mathcal{S}(p)$ represents the slice that is both correct and context-sensitive. More formally,

Definition 4

$$\mathcal{S}(p)=\{q \mid \mathcal{P} = \langle n_1, \ldots, n_k \rangle, \; q = n_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} n_k = p, e_i \in E' \; E' = E_{dd} \cup E_{cd} \cup E_{id} \cup E_c \cup E_{pi} \cup E_{po} \; \mathcal{P} \textit{ is an interprocedural threaded witness in the corresponding ITCFG} \}$$

$\mathcal{S}(p)$ is less precise than $\mathcal{S}_{opt}(p)$ [Müller-Olm and Seidl 2001]. However, we believe that it is a reasonable notion and can be computed. We will give an algorithm, which we call the "context-sensitive algorithm" that exactly computes $\mathcal{S}(p)$. This algorithm has exponential complexity. We also give an efficient algorithm, which we term the "context-insensitive algorithm", that computes a superset of $\mathcal{S}(p)$. Both the algorithms compute correct slices although the context-sensitive algorithm computes more precise slices than the context-insensitive one. We introduce three examples to motivate the issues of correctness and precision.

3.1 Correctness Issues

Consider a simple adaptation of the two-phase algorithm for sequential programs [Horwitz et al. 1990] with extensions for interference dependence, where interference dependence edges are traced in both Phase 1 and Phase 2 in addition to the other standard edges [Zhao 1999]. Applying this algorithm to the node `K8` in Figure 2 yields the following results. In Phase 1, `K8` is data dependent on `K7` which is parameter-out dependent on `K6` in procedure `P1`. Therefore, `K6` will be sliced in Phase 2.

In Phase 2, `K6` is data dependent on `K5`. `K5` is data dependent on `K4`, interference dependent on `L11` and `L12` in procedure `P2` in thread θ_2 and interference dependent on `L7` in θ_2. `L12` is data dependent on `L10` which is parameter-in dependent on `L5`. Since the algorithm is now in Phase 2, `L5` will not be added to the slice. However, `L7` is summary dependent on `L5` and hence `L5` will eventually get added to the slice along with its transitive dependencies. Now consider the dependencies arising from `L11`. Of these, `L11` is data dependent on `L9` which is parameter-in dependent on `L4`. Since the algorithm is in Phase 2, `L4` will not be added to the slice. As a result `L4` and nodes that are transitively dependent on it (`L2`) will never get added to the slice. Hence the two-phase algorithm gives an incorrect slice.

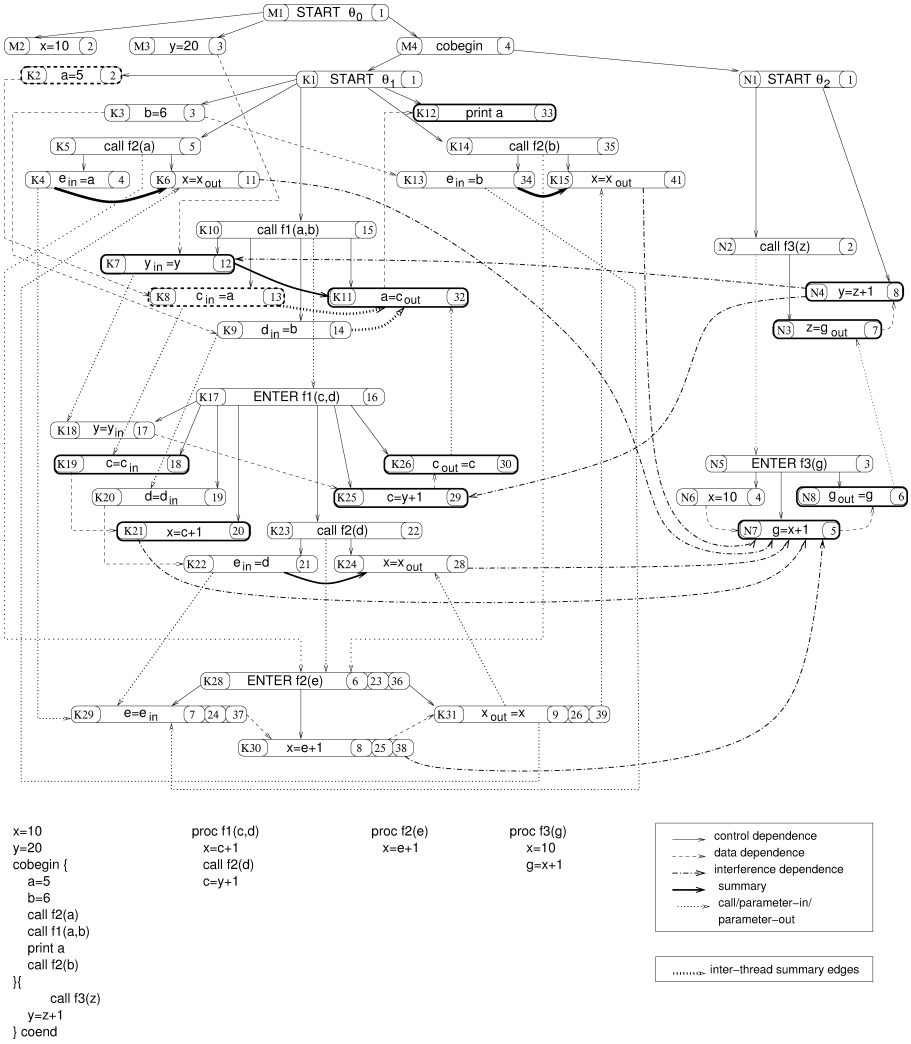


Fig. 3. Illustrating inter-thread summary edges. The nodes are labeled `Mi` in θ_0, `Ki` in θ_1, and `Ni` in θ_2. The thickened boxes do not show a complete slice but help navigate the relevant nodes in the two-phase slice of `K12`. The dashed, thickened boxes show some of the nodes that do not get added to the slice. The inter-thread summary edges are notional in the sense that they show the inter-thread transitive dependency, but they are not explicitly computed by any algorithm. The numbers in circles indicate the topological order of the nodes after inlining and are used in the context-sensitive slicing algorithm.

3.1.1 *Interthread Summary Edges.* Another problem with applying the two-phase algorithm directly is that the summary edges do not adequately represent transitive dependencies in the presence of threads since there may be transitive dependencies that cross thread boundaries. In fact, summary edges are not computable in concurrent programs [Ramalingam 2000].

Consider the slice of `K12` in Figure 3. In Phase 1, we find that `K12` is parameter-out dependent on `K11`. In Phase 2, `K11` is summary dependent on `K7` and

parameter-out dependent on K26. Further, we trace the following dependencies backwards: K26 ← K25 ← N4 ← N3 ← N8 ← N7 ← K21 ← K19. K19 is parameter-in dependent on K8 but since the algorithm is in Phase 2, K8 will not get added to the slice and nor will any of the nodes that it is dependent on (such as K2). Similarly, K9 and its dependencies will also not get added to the slice.

Here the problem is that some inter-thread transitive dependencies are missing. There is a transitive dependence from the formal-in node, K19 to the formal-out node, K26, and from K20 to K26 which should induce inter-thread summary edges from K8 to K11 and K9 to K11, respectively. As these are *inter-thread* transitive dependencies they are not considered by the standard [Horwitz et al. 1990] intra-thread summary edge algorithm. As mentioned earlier, this is not computable.

3.2 Precision Issues

Even if it were possible to compute all summary edges, this algorithm remains imprecise and context-insensitive. In Figure 4(a), we show a program and its ITCFG, and the corresponding TSDG is in Figure 4(b). Consider the slice of M6 in Figure 4. Clearly, M6 is dependent on the call to f1 at M5 but not on the call to f1 at M11. But the simple two-phase algorithm adds M11 to the slice of M6 as follows: M6 is parameter-out dependent on M21 in procedure f1 and M21 ← M19 ← N3 ← M13 ← M11.

It might appear that a naive marriage of Krinke's algorithm and Zhao's algorithm would generate precise context-sensitive slices. However, this is not the case because unlike a single procedure program, paths may not be transitive in programs with multiple procedure calls.

3.2.1 *Intransitivity of Interprocedural Paths.* Determining whether there is a realizable path between any *two* nodes in the interprocedural control flow graph of a *sequential* program is possible using standard techniques of interprocedural control flow analysis of programs [Burke 1990; Callahan 1988]. Let the relation Reach(n_i, n_j) be true if there is a realizable path from n_i to n_j and false otherwise. For example, in Figure 4(a), Reach(M8, M20) and Reach(M20, M19) are true but Reach(M11, M8) is false. In programs with procedure calls, the composition of Reach may not be a realizable path. That is, given that Reach(n_a, n_b) and Reach(n_b, n_c) are true, the composition Reach$(n_a, n_b) \circ$ Reach(n_b, n_c) does not imply that $\langle n_a, n_b, n_c \rangle$ is a realizable path. For example, Reach(M8, M20) ∘ Reach(M20, M19) does not imply that there is a realizable path through ⟨M8, M20, M19⟩.

As a consequence, a simple extension of Krinke's algorithm to concurrent interprocedural slicing generates context-insensitive slices. Consider a simple extension of the two-phase algorithm where we keep track of the last node visited in each thread. The slice of M14 in Figure 4(b) would trace the following dependencies: In Phase 1, M14 ← M12 ← M21. In Phase 2, M21 ← M19 ← N3. Since N3 is in θ_2 the algorithm remembers that the last node visited in θ_1 was M19. Next, N3 ← M20. When the algorithm re-enters θ_1 it finds that there

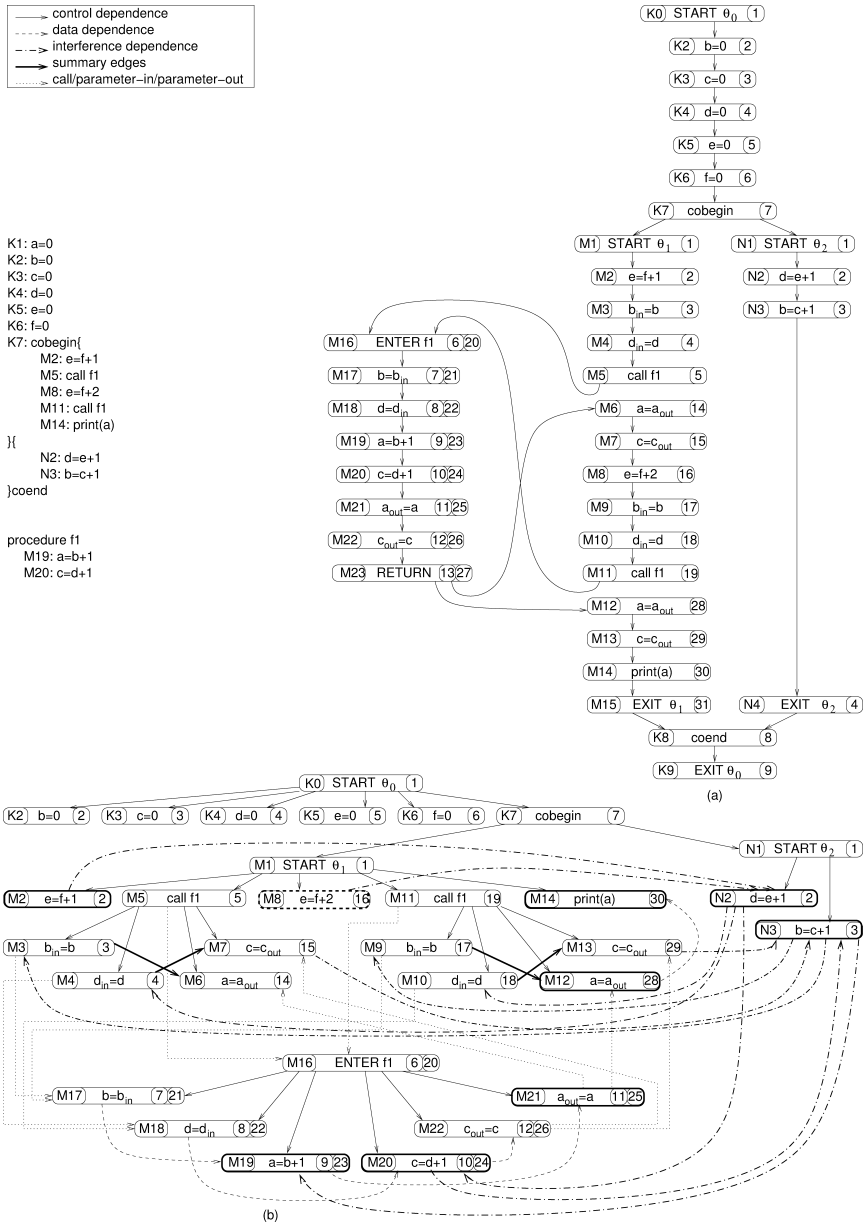


Fig. 4. (a) An ITCFG. The nodes are labeled `Ki` in θ_0, `Mi` in θ_1 and `Ni` in θ_2. The numbers in the circles indicate the topological order of the nodes after inlining the calls. (b) The corresponding TSDG, depicting intransitivity of paths. The darkened nodes trace part of a context-insensitive slice of `M14`. The darkened dashed node would not be added in a context-sensitive slice.

is a realizable path from M20 to M19 (Figure 4(a)) and so M20 is added to the slice.

Note that the path from M20 to M19 is an interprocedural path that corresponds to an execution of M20 from the call at M5 followed by an execution of M19 from the call at M11. However, the path information has no way of keeping track of the calling contexts associated with each node.

Further, M20 $\leftarrow$ N2. On re-entering θ_2, the algorithm checks that there is a path from N2 to N3. Now, N2 is interference dependent on M2 and M8. Both have a path to M20 and so both will be added to the slice. But adding M8 violates the realizable path condition. To understand why, observe that the path traced by the algorithm within θ_1 is as follows : M8 $\rightarrow$ M20 $\rightarrow$ M19 $\rightarrow$ M21 $\rightarrow$ M12 $\rightarrow$ M14. The subsequence of nodes in θ_1 is $\langle$ M8, M20, M19, M21, M12, M14 $\rangle$. Although there is a *realizable* path between every pair of nodes in the sequence, yet the *composition* of the realizable paths is not a realizable path in θ_1.

3.3 Recapitulation

This section has highlighted the following points

—The two-phase algorithm when applied to concurrent programs may give incorrect slices.

—In concurrent programs, computation of summary edges is undecidable as the summary edges in a concurrent program may be induced by cross-thread dependences. Cross-thread dependencies must be computed without using summary edges else some nodes may be missed by a slicing algorithm, resulting in an incorrect slice.

—To compute a context-sensitive slice it is necessary to be able to determine whether a set of nodes belong to a realizable path.

In Section 4, we will explain how to solve the correctness problem. In Section 5, we solve the realizable path problem and in Section 6 we put the two solutions together to compute correct, context-sensitive slices.

4. A CONTEXT-INSENSITIVE SLICING ALGORITHM

In this section we give a fast slicing algorithm which computes correct but possibly imprecise slices.[2] First we make an observation about two-phase slicing of sequential programs (or a single thread of a concurrent program). In a sequential program:

—When computing the slice of a node n_i, for every node, n_j, that is added to the slice in Phase 1, the slice of n_j is available as a subset of the slice of n_i that is computed. That is, every node that n_j is dependent on gets added to the slice.

—If the node n_j is added to the slice in in Phase 2, then only the relevant *subset* of the calling contexts of n_j are added to the slice of n_i.

[2]Note that this algorithm is intra-thread context-sensitive, but inter-thread context-insensitive.

We need to extend the two-phase algorithm to handle inter-thread dependencies. Since inter-thread summary edges are not computable we need to capture these dependencies while traversing interference dependence edges. The basic idea behind the interprocedural slicing algorithm for concurrent programs is as follows:

—A node n_i that is reached via an interference dependence edge needs to be sliced in Phase 1, since we need to find all the nodes that it is dependent on. So whether the interference dependence edge is traversed in Phase 1 or Phase 2, n_i must be sliced in Phase 1. This implies that if an interference dependence edge is discovered during Phase 2, we need to run a subsequent pass in Phase 1 for n_i (and its corresponding Phase 2). This (as we will prove later) also solves the problem of transitive dependencies that cross threads.

—A node n_j that is added to the slice in Phase 2 may be traversed again in a subsequent Phase 1 due to a transitive dependence that includes an interference dependence edge. Then although n_j has been sliced before it will have to be sliced again in Phase 1, since Phase 2 generates only a subset of all the nodes that n_j is dependent on. To handle this we color the nodes with three colors rather than two colors as in the two-phase algorithm.

4.1 The Algorithm

The slicing algorithm essentially puts a loop around the traditional two-phase algorithm [Horwitz et al. 1990]. The purpose of the loop is to ensure that nodes reached via an interference dependence edge are sliced in Phase 1. Our slicing algorithm uses three lists. Each time a node is reached through an interference dependence edge, it is added to the outermost worklist w_0 and subjected to a complete two-phase slice. The algorithm iteratively applies a 2-phase slice to every node in w_0.

Our algorithm uses three colors for marking a node. We call the colors `phase1`, `phase2` and `undefined` and define a transitive order `undefined` < `phase2` < `phase1` on them. Initially all the nodes in the TSDG are colored `undefined`.

In Phase 1, for the nodes reached by tracing backwards the data dependence, control dependence, summary dependence, parameter-in dependence, and call dependence edges we check the color of the node. If it has already been colored `phase1` we do not need to slice it again, else we color it `phase1` and insert it into worklist w_1. For nodes reached via interference dependence edges, if they have been colored `phase1` already, we ignore them, else they are colored `phase1` and added to the outermost worklist w_0. For a node reached via a parameter-out dependence edge, if it is still colored `undefined` it is colored `phase2` and added to the worklist w_2. If it had already been colored `phase1` or `phase2` it need not be added to the worklist again.

In Phase 2, also nodes reached via interference dependence edges are colored `phase1` and added to w_0 if they are not already colored `phase1`. Nodes reached via data dependence, control dependence, summary dependence, and parameter-out edges are colored `phase2` and added to the worklist w_2 if they

```
Input: the slicing criterion s, the TSDG
Output: the slice S = every node in the TSDG that has been colored phase1 or phase2
Declare: undefined < phase2 < phase1
Initialization:
for each node x ∈ TSDG do
    x.color = undefined
w0 = {s}

begin
while w0 ≠ ∅ do
    remove next element x from w0
    w1 = {x}

    /* Phase 1 */
    while w1 ≠ ∅ do
        remove next element x from w1
        for all y | (y, x) ∈ E_id
            if y.color < phase1 then
                y.color = phase1
                w0 = w0 ∪ y
        for all y | (y, x) ∈ E_dd ∪ E_cd ∪ E_s ∪ E_pi ∪ E_c
            if y.color < phase1 then
                y.color = phase1
                w1 = w1 ∪ y
        for all y | (y, x) ∈ E_po
            if y.color < phase2 then
                y.color = phase2
                w2 = w2 ∪ y
    endwhile

    /* Phase 2 */
    while w2 ≠ ∅ do
        remove next element x from w2
        for all y | (y, x) ∈ E_id
            if y.color < phase1 then
                y.color = phase1
                w0 = w0 ∪ y
        for all y | (y, x) ∈ E_dd ∪ E_cd ∪ E_s ∪ E_po
            if y.color < phase2 then
                y.color = phase2
                w2 = w2 ∪ y
    endwhile

endwhile
end
```

Fig. 5. The context-insensitive three-color iterated two phase slicing algorithm.

have not already been colored either `phase1` or `phase2`. Parameter-in and call edges are ignored.

The final slice consists of every node that has been colored either `phase1` or `phase2`. The algorithm is given in Figure 5.

Example 1. Let us slice the program in Figure 2. Initially `K8` is added to the worklist w_0. Then `K8` is extracted and a 2-phase algorithm is applied to it. We

use the shorthand notation "Ki ← Kj" to indicate that Ki is dependent on Kj. The steps are as follows:

—Iteration 1:
  —Phase 1: K8 is data dependent on K7 which is parameter-out dependent on K6 in procedure P1. K6 is inserted into w_2.
  —Phase 2: In Phase 2, K6 is data dependent on K5. K5 is data dependent on K4, and K4 is inserted into w_2. K5 is interference dependent on L11 and L12 in procedure P2 in thread θ_2 and interference dependent on L7 in θ_2. L11, L12 and L7 are inserted into w_0. Next K4 is sliced. No further dependencies are found and the first iteration ends.
—Iteration 2: In the next iteration, let us assume L7 is processed.
  —Phase 1: L7 is parameter-out dependent on L13 and therefore L13 is added to w_2. Then, L7 ← call P2 ← START θ_2. Also L7 ← L5 ← L3.
  —Phase 2: L13 ← L12. But L12 is already colored phase1. No other dependencies are found and this completes the second iteration.
—Iteration 3: Let L11 be extracted from w_0 in the next iteration.
  —Phase 1: L11 ← L9 ← L4 ← L2.
  Nothing gets added to w_2 and so the Phase 2 is empty. This completes the iteration.
—Iteration 4: This iteration starts with L12 and nothing new is found in the iteration.

Thus, the algorithm generates a correct slice.

Example 2. Consider the slice of K12 in Figure 3.

—Iteration 1:
  —Phase 1: K12 ← K1 ← M4 ← M1. K12 ← K11. (K11 is added to w_2.)
  —Phase 2: K11 ← K7 ← K10. K11 ← K26 ← K25 ← K18 ← K17. K25 ← N4. (N4 is added to w_0).
—Iteration 2:
  —Phase 1: N4 ← N1. N4 ← N3 ← N8 (N8 is added to w_2).
  —Phase 2: N8 ← N7 ← N6 ← N5. N7 ← K15. N7 ← K6. N7 ← K24. N7 ← K30. N7 ← K21. All of K15, K6, K24, K30, K21 are added to w_0.
—Iteration 3:
  —Phase 1: K21 ← K17. K21 ← K19 ← K8 ← K2. Since the algorithm is in Phase 1, the parameter-in edge is traversed.
—Iteration 4:
  —Phase 1: K24 ← K22 ← K20 ← K9 ← K3. K24 ← K31 (K31 is added to w_2).
  —Phase 2: K31 ← K30 ← K29. Since the algorithm is now in Phase 2, the parameter-in edge to K4 is not traversed.
—Iteration 5:
  —Phase 1: K30 ← K29. K29 ← K4 ← K5. K29 ← K22. K29 ← K13 ← K3. In Iteration 4, K30 was colored phase2. In this iteration it is colored phase1 and sliced again. This time the parameter-in edges are also traversed.

—Iteration 6:
  —Phase 1: K6 ← K4 ← K5.
—Iteration 7:
  —Phase 1: K15 ← K14. K15 ← K13.

Note that the slice computed is correct but not context-sensitive. The call to $f2(b)$ at K14 is added to the slice although it is not possible for the value of a at K12 to be affected by the call at K14. The other unnecessary nodes added to the slice are K15 and K13.

We do not give a formal proof of correctness of the algorithm, but in Appendix B, we give an informal argument based on the proof for the context-sensitive algorithm.

4.2 Complexity

Each node in the graph may be colored at most twice. Hence, the maximum number of nodes handled by the algorithm is $2 * N$ and the runtime complexity of the algorithm is $O(N + E)$, where E is the number of edges.

The cost of construction is governed by the cost of generating the control dependence, data dependence, interference dependence and summary edges. Control dependence edges are determined on a per-thread basis with additional control dependence edges from a cobegin node to the corresponding START θ_i nodes and from EXIT θ_i nodes to the corresponding coend node. These can be generated in time linear in the size of the graph [Johnson and Pingali 1993]. Data and interference dependence can be generated using algorithms of Rugina and Rinard [1999], Salcianu and Rinard [2001], and Nanda and Ramesh [2003]. Novillo et al. [1998] further explain how to reduce the number of interference dependence edges in the presence of monitors. The cost of generating summary edges is the same as in sequential programs [Reps et al. 1994].

5. REALIZABLE PATHS IN A SEQUENTIAL THREAD

In the previous section, we showed that the algorithm is context insensitive as it simply includes into the slice all the nodes that are reached during the traversal irrespective of whether all the nodes in a thread form a realizable path. In order to generate a context-sensitive slice it is necessary to determine whether a given set of nodes may form a realizable path. Given a set of nodes that form a realizable path in a thread, we show in this section how to determine whether another node may be added to the set without violating the realizable path condition.

In Figure 6, Reach(S3, K1) and Reach(K1, S8) are true but Reach(S3, S8) is false. Further, consider the realizable path (S4, K1, K2, S5, S9, S10, S11, K1, K2, S12). From this path we see that there is a realizable path from K2 to K1. That is Reach(K2, K1) is true. Similarly, Reach(K1, S9) is true and in this case Reach(K2, S9) is also true. It might appear that the concatenation of the path from K2 to K1 and the path from K1 to S9 may be a realizable path. Yet, there is no realizable path $\texttt{K2} \xrightarrow{+} \texttt{K1} \xrightarrow{+} \texttt{S9}$. On the other hand, there is a realizable path $\texttt{K2} \xrightarrow{+} \texttt{K1} \xrightarrow{+} \texttt{S13}$.

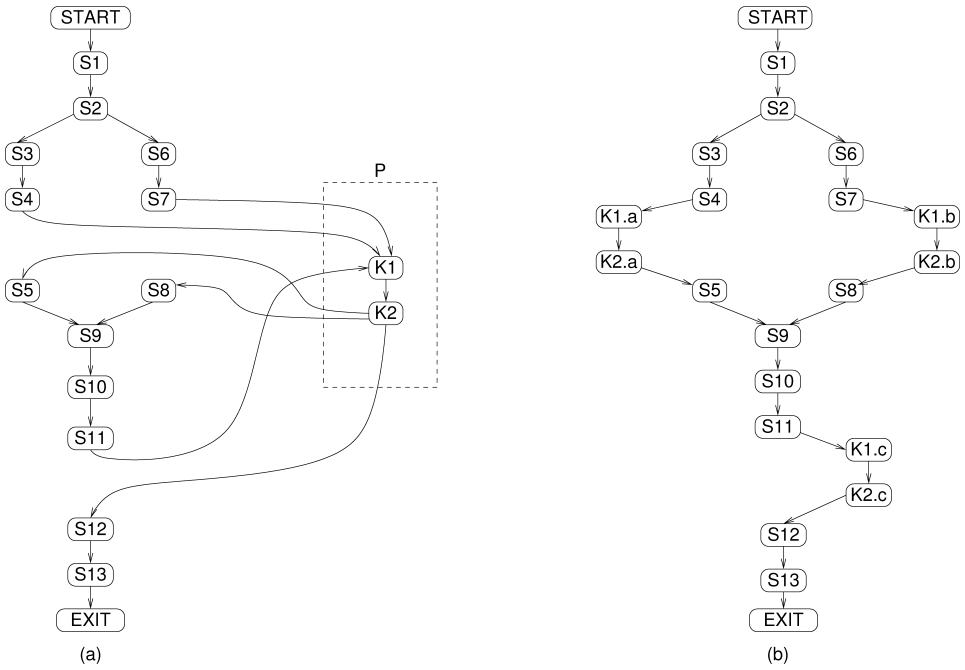


Fig. 6. (a) A program ICFG and (b) The ICFG with procedures inlined.

We are now ready to state the problem to be solved: *Given a sequential interprocedural control flow graph, G, and a set of nodes* n_1, n_2, $\cdots$, n_k *belonging to G we wish to find out if there is a realizable path* $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$. We term this problem the *"Realizable Path"* problem.

Given a set of nodes, we wish to identify whether they form a part of a realizable path. A brute force solution would generate all possible realizable paths in the graph using any standard technique [Sharir and Pnueli 1981; Reps et al. 1994] and then check if the given set of nodes forms a subsequence of any one of the realizable paths.

Another solution to the *Realizable Path* problem could be to inline all the procedures. In Figure 6 we have shown a program with the procedures inlined. Then the problem reduces to the intraprocedural case and can be solved easily. This solution obviously requires unbounded space when there are recursive procedures. Another, not so obvious, problem with this solution is that a given node (for example, `K1` in Figure 6) may occur in multiple locations in the expanded graph. To find a realizable path that includes `K1` we need to trace every path through each location of `K1`. In the presence of loops, the length of a path may be unbounded.

In this section, we propose a solution based on interval analysis.

5.1 An Interval-Based Approach

First we introduce some standard terminology. $G = (V, E, n_0)$ is an interprocedural control flow graph (or *ICFG*) where V represents the set of vertices, E

is the set of edges and n_0 is the distinguished start node. A *region* R of G is a (possibly interprocedural) subgraph of G such that an edge (n_x, n_y) of G is in R if and only if n_x and n_y are both in R. A node n_x is an *entry node* of R if there is an edge (n_w, n_x) of G such that n_w is not in R. A node n_y is an *exit node* of R if there is an edge (n_y, n_z) of G such that n_z is not in R. A *strongly connected region* (SCR) is a region such that every node in it is reachable from every other node.

The *interval order* of the nodes of G is the order in which they are visited by a reverse postorder traversal (i.e., postorder traversal on the reverse graph, rooted at the `EXIT` node of the ICFG). If G is **acyclic**, then interval order is a topological ordering [Burke 1990]. Let `NUMBER`(n_i) be the topologically ordered number for any node n_i. Then topological order has the interesting property that `NUMBER`$(n_i) <$ `NUMBER`(n_j) if n_i is a predecessor of n_j in the graph, G.

A *call graph* of a program is a graph where each procedure is uniquely represented by a single node. There is an edge (P, Q) in the call graph if procedure P calls procedure Q.

Since every node in a SCR is reachable from every other node in it, it is obvious that we do not need to test for reachability for nodes within a SCR. This is the basis of our algorithm. We find the largest possible *interprocedural* SCRs in the ICFG and collapse them into single nodes [Burke 1990; Sarkar 1991]. The resultant graph is an acyclic graph and can be ordered in topological order. We use the program in Figure 7 as an example. Briefly, our analysis takes the following steps:

(1) Construct the call graph of the program and find the SCRs in this graph. Collapse the SCRs into single nodes to generate the *CallSCR* graph. Term the nodes in this graph the *call strongly connected components* or *cSCRs*. The cSCRs are shown in Figure 8 and internally consist of one or more procedures.

(2) Determine the intraprocedural SCRs for each non-recursive procedure as follows: ignore all interprocedural edges, and create a dummy edge connecting a call site to the corresponding return site and then apply any standard algorithm [Cormen et al. 1990] to collapse strongly connected regions into a single node. Call these *intraprocedural strongly connected components* or *intraSCRs* for short. They are labeled as χ_i in all the figures. Figure 9 shows the intraSCRs of the example program in Figure 7.

(3) Mutually recursive procedures form a single node (cSCR) in the CallSCR graph. On the subgraph induced by such a cSCR node (i.e., the set of nodes and edges belonging to the procedures in a cSCR node), apply any standard algorithm to determine the strongly connected components in this subgraph. These SCRs are also termed intraSCRs. (See Example 3 and 4 below.)

(4) Finally, we give an algorithm to integrate intraSCRs across procedures to construct the final *interprocedural strongly connected components* or *ISCRs*. The ISCRs of the example program are shown in Figure 10. The ISCRs are numbered as `Zi`. The component intraSCRs are also shown. The resultant ISCR graph is an ICFG that has no cycles, no recursion and in which each

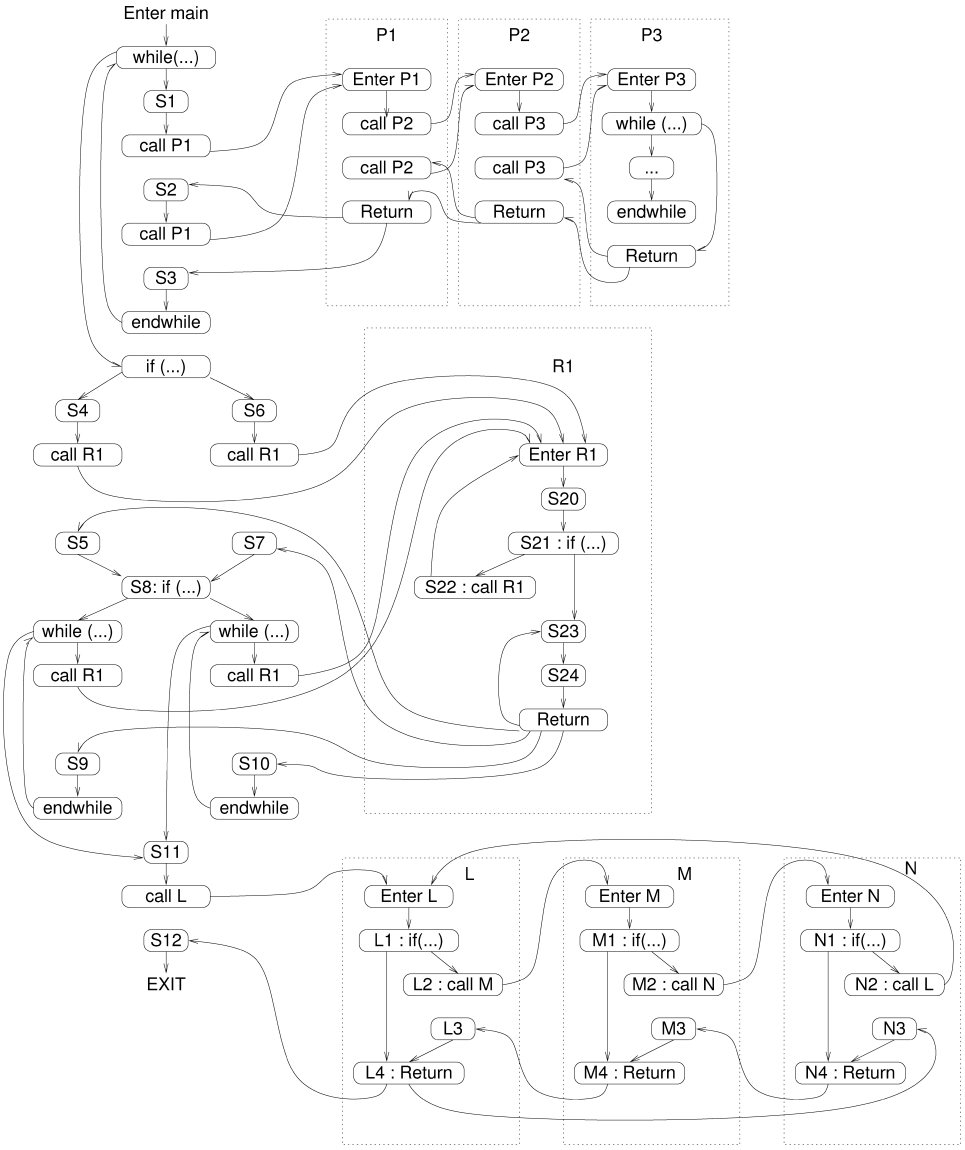


Fig. 7. The ICFG of a program.

node represents an ISCR. Next, we inline any procedure that has not been integrated into an ISCR.

Once this set of information has been calculated we can calculate whether a given set of nodes form a realizable path.

Example 3. Consider the case of the self recursive procedure, `R1` in Figure 7. The edge from `S22(call R1)` to the `Enter R1` vertex generates an intraSCR containing all the statements between `Enter R1` and `S22`. Similarly, the edge from

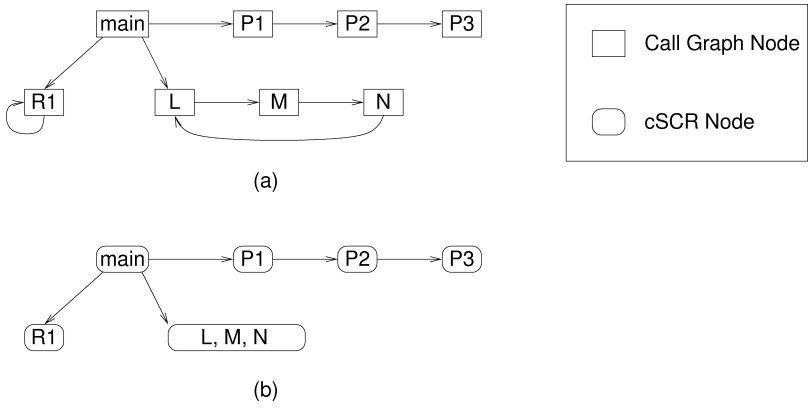


Fig. 8. (a) The call graph of the program in Figure 7 and (b) the CallSCR graph which is formed by collapsing SCRs in the call graph. Each node in the CallSCR graph is called a cSCR node and contains one or more procedure.

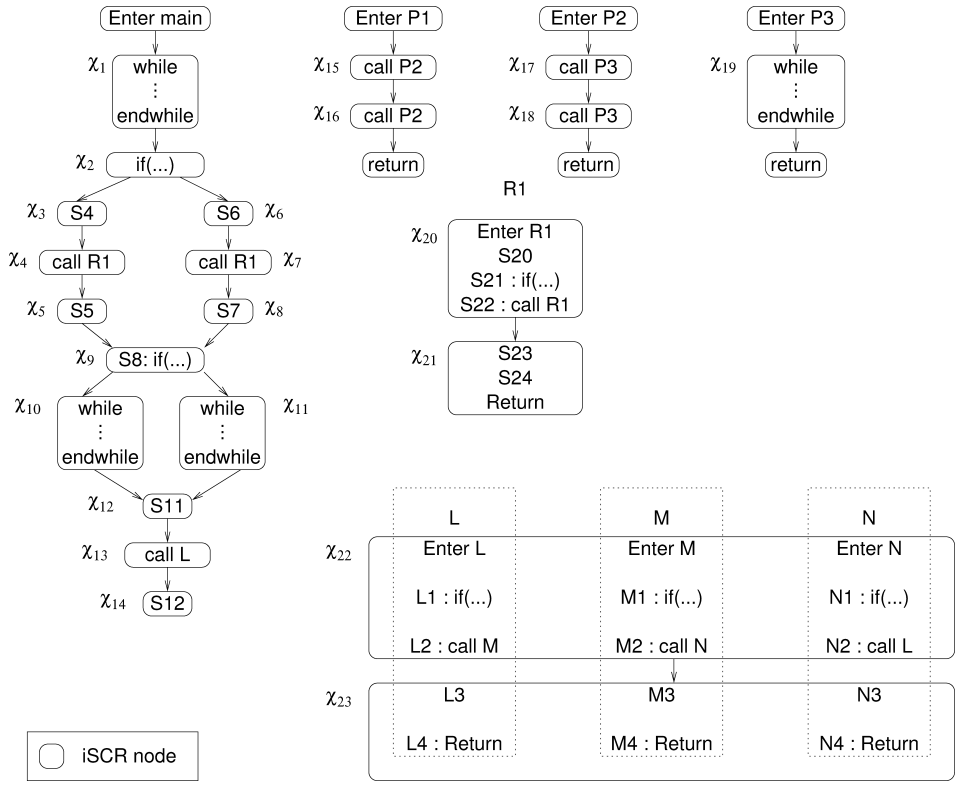


Fig. 9. The *intraprocedural* and *recursion induced* strongly connected components (intraSCRs), labeled χ_i and connected by intraprocedural control flow edges. The interprocedural edges have not been shown.

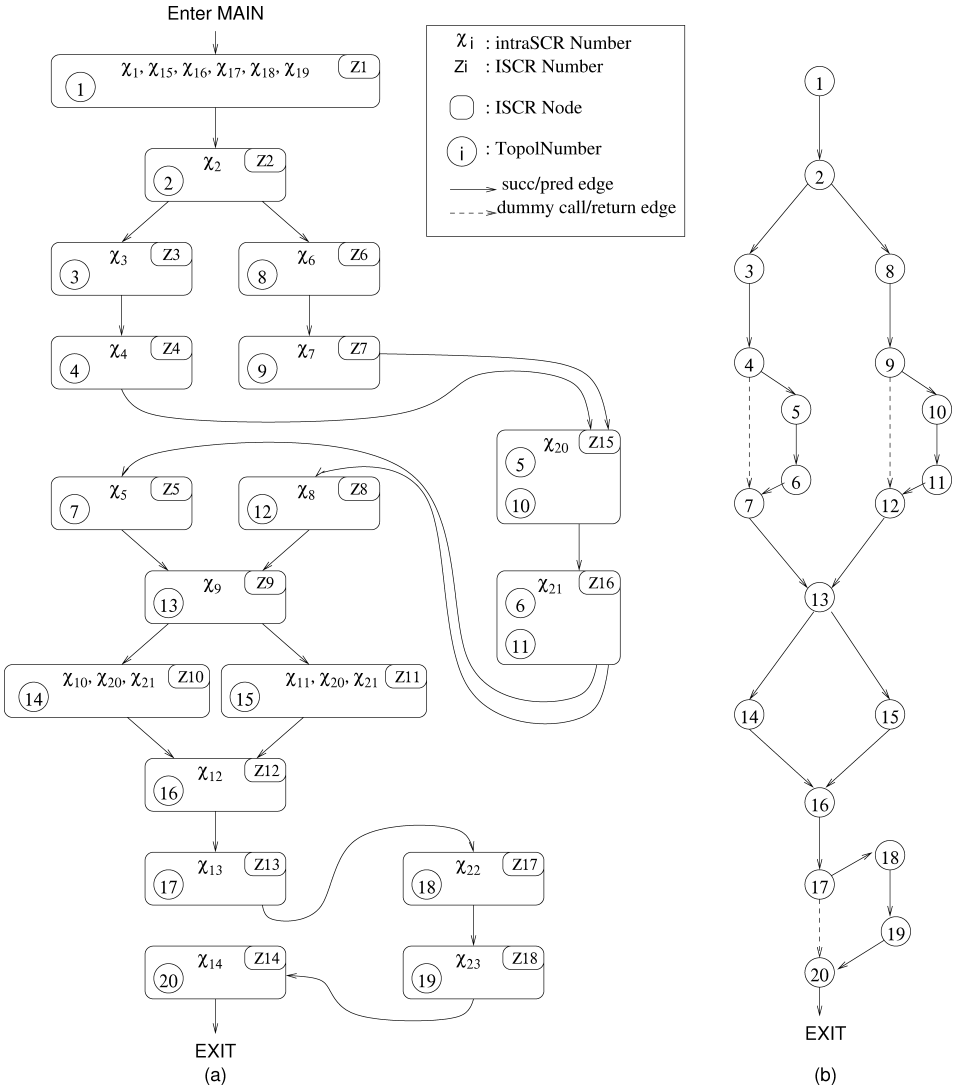


Fig. 10. (a) The ISCR graph for the program in Figure 7 consisting of Interprocedural Strongly Connected Components (ISCRs). `Zi` are the names of the ISCRs, the list of intraSCRs associated with each ISCR are shown as χ_i and the numbers in circles represent the topological order of the ISCRs after inlining. (b) The "TopologicalNumber Graph" showing the edges connecting the TopolNumbers.

the `Return` node to `S23` induces another intraSCR and the edge `S21` $\rightarrow$ `S23` induces the connection between the two intraSCRs. In Figure 9, these two intraSCRs are labeled χ_{20} and χ_{21}. This is logically correct since every node in χ_{20} is reachable from every other node along some recursive path and similarly for χ_{21}. Also, every node in χ_{21} is reachable from every node in χ_{20}.

Example 4. Consider the multiway recursion in the procedures `L`, `M` and `N`. In the region induced by these procedures we calculate the SCRs. For the

purpose of determining the SCRs within this region we ignore the interprocedural call edge from `call L` in `main` to `Enter L` but not the edge from `call L` in procedure N to `Enter L`. That is, we include only those interprocedural call and return edges between the procedures `L`, `M` and `N` that are internal to the cSCR under consideration. From Figure 9, we see that there are two interprocedural SCRs in the region labeled χ_{22} and χ_{23} induced by the two interprocedural loops `Enter L` $\rightarrow$ `L1` $\rightarrow$ `L2` $\rightarrow$ `Enter M` $\rightarrow$ `M1` $\rightarrow$ `M2` $\rightarrow$ `Enter N` $\rightarrow$ `N1` $\rightarrow$ `N2` $\rightarrow$ `Enter L` and `L3` $\rightarrow$ `L4` $\rightarrow$ `N3` $\rightarrow$ `N4` $\rightarrow$ `M3` $\rightarrow$ `M4` $\rightarrow$ `L3`. There is an external edge from `call L` in the main procedure to χ_{22} and an external edge from χ_{23} to `S12` in `main`.

5.2 Building the ISCR Graph

This section gives details of step (4) given above. After performing steps (1), (2) and (3), each procedure in the ICFG (including `main`) consists of intraSCRs connected by control flow edges. The procedures are also connected by call and return edges as in the original ICFG. An intraSCR that has multiple ICFG nodes has clearly been generated by merging ICFG nodes in a loop and is termed a multi-node intraSCR. An intraSCR node that has exactly one ICFG node is termed a singleton intraSCR.

We now try to combine as many intraSCRs (across all procedures) as possible into a single *interprocedural SCR* or ISCR. The algorithm is very simple. The algorithm walks through the collapsed ICFG starting with the `Enter main` node in the `main` procedure. For each multi-node intraSCR that contains a call site ($P \rightarrow Q$), we delete the call edge and insert the intraSCRs in Q into the parent ISCR. Then we traverse the call SCR graph from Q and insert all the intraSCRs in the procedures that are called directly or transitively from Q. Since a procedure may be integrated from multiple locations, this process may have to be repeated from each location. To avoid this, we use the CallSCR graph to cache the list of nodes reachable transitively from a procedure. The algorithm works in two phases.

In the first phase, we walk through the CallSCR graph, starting at the root (the cSCR representing the `main` procedure) and build the cache for each procedure. The algorithm is given in Figure 11. In the second phase we start with the `main` procedure and walk through the intraSCR graph and integrate as many intraSCRs as possible into a single ISCR. Each intraSCR has an attribute *multi* which is true if the intraSCR has multiple nodes (i.e., it has at least one internal edge) and is false otherwise (if it is a singleton node). The corresponding ISCR also inherits this attribute. (This is used by the Reach algorithm in Section 5.3.) A call statement may be part of a multi-node intraSCR or it may belong to a singleton intraSCR. If it is a single statement, then the corresponding ISCR generated from it is marked to be of type `CALL_SITE` and we introduce a dummy ISCR after it and label it to be of type `RETURN_SITE` with attribute *multi* set to false. If the call statement is part of a multi-node intraSCR, then the called procedure is integrated into the current ISCR and the call edge will be deleted.

At the end of this phase, the interprocedural SCRs are in place in the interprocedural SCR graph. The ISCR graph is shown in Figure 10. As we can

```
procedure CacheTransitive ( cSCR cnode )
    if cnode.mark == true then
        return
    cnode.mark = true
    for each procedure P ∈ cnode do
        for each intraSCR q ∈ P do
            cnode.cache = cnode.cache ∪ q
    for each successor x of cnode do
        CacheTransitive(x)
        cnode.cache = cnode.cache ∪ x.cache


procedure BuildISCRGraph ( intraSCR q )
    if q.mark ==true then
        return
    q.mark =true
    ISCR = {q}
    ISCR.multi = q.multi
    if q.multi == true then
        for each call site n_x → P in q do
            ISCR = ISCR ∪ P.cache
            delete the edge n_x → P
    else /* single node */
        if q is a call site n_x → P then
            BuildISCRGraph (P.entryIntraSCR)
            ISCR.type = CALL_SITE
            Introduce a dummy ISCR of type RETURN_SITE with attribute
                                                        multi = false
    for each q_s ∈ successor(q) do
        BuildISCRGraph (q_s)
```

Fig. 11. Algorithm to build the ISCR graph. In `CacheTransitive`, *cnode* is a node in the CallSCR graph. For each procedure P, in the *cnode*, we find the set of intraSCRs belonging to P and every intraSCR belonging to a procedure (transitively) called by P and cache the set at *cnode*. In `BuildISCRGraph`, q is an intraSCR node. If q is a multi-node SCR which has procedure calls, then every intraSCR in the cache of the called procedure is added to the current q to generate the final ISCR node.

see in the ISCR graph, several intraSCRs may be combined into a single ISCR, as for example, χ_1, χ_{15}, χ_{16}, χ_{17}, χ_{18} and χ_{19} are all in the ISCR numbered `Z1`. Also, it is possible that a single intraSCR may be tagged to several ISCRs, as for example, χ_{20} is tagged to ISCRs labeled `Z10`, `Z11` and `Z15`. In the example in Figure 10, the procedures `P1`, `P2` and `P3` have got integrated into the ISCR labeled `Z1`; procedures `L`, `M` and `N` have generated the two ISCRs `Z17` and `Z18`, but there is a call edge from `Z13` to `Z17` and a return edge from `Z18` to `Z14`. The case of `R1` is interesting. `R1` has been integrated into `Z10` and `Z11` so there is no call edge from `Z10` or from `Z11`. But there is a call edge from `Z4` and `Z7` to `Z15` and a corresponding return edge from `Z16` to `Z5` and `Z8`, respectively.

At this stage, we are assured that the resultant graph is acyclic and the ISCR nodes can be ordered in topological order after inlining all the remaining procedure calls that did not get integrated into an ISCR. We give an efficient method for inlining that does not require making actual copies of the procedures. Briefly,

```
/* initialization */ count = 0
for each ISCR z of the ISCR Graph do
    z.top_set = φ; z.mark = false

procedure GenerateTopologicalNumbers ( ISCR z )
    if z.mark == true then return
    for each predecessor z_p of z do
        switch z_p.type
            case RETURN_SITE :
                PushStack (z_p.corres_call_site)
                GenerateTopologicalNumbers (z_p)
                reset marks in called procedure
            case CALL_SITE :
                if z_p == TopStack() then
                    PopStack() ; GenerateTopologicalNumbers (z_p)
            default :
                GenerateTopologicalNumbers (z_p)
        endswitch
    endfor
    z.number = count ++; z.mark = true
    z.top_set = z.top_set ∪ ⟨z.number⟩
    return
```

Fig. 12. The algorithm to generate topological ordering.

the algorithm walks through the graph in reverse postorder maintaining a stack of the call sites to ensure that only realizable paths are traversed. The nodes are numbered in reverse postorder. A procedure node that is visited more than once gets multiple numbers. So inlining is simulated by the simple expedient of using a single integer to number the position of the inlined procedure in the graph. The topological ordering is shown enclosed in circles in Figure 10 and for ease of exposition we refer to them as "`TopolNumbers`". Each ISCR has one or more unique `TopolNumbers` associated with it. For example, the ISCR `Z15` has two `TopolNumbers` 5 and 10, corresponding to each call site while the ISCR `Z17` has only one `TopolNumber` 18.

The algorithm for generating the topological order uses the following notation: Each node is of type `CALL_SITE`, `RETURN_SITE` or `DEFAULT`. Each ISCR has the following attributes:

—*z.number* holds the value of the latest `TopolNumber` assigned to the z and is initially set to zero;

—The attribute *multi* indicates whether the ISCR has at least one internal edge. If it is true, then it implies that all nodes in the ISCR are reachable from each other (else it is a singleton node).

The algorithm starts with the `EXIT` node of the ICFG and visits the nodes in reverse postorder. The global variable *count* is initialized to 0. The numbering scheme allows a procedure node to be re-entered for each corresponding call site without looping endlessly or missing any call sites. The call stack ensures that procedures are entered and exited along realizable paths. The algorithm is given in Figure 12.

```
procedure Reach ( TopolNumber from, TopolNumber to )
    if from == to then
        return true
    if from > to then
        return false
    if to is a return site then
        callnumber = value of the corresponding call site (using dummy edge)
        if callnumber ≤ from then
            /* Skip the called procedure */
            return Reach ( from, callnumber )
        else
            /* Descend into the called procedure */
            prednumber = predecessor of to
            return Reach ( from, prednumber )
    else
        for each predecessor, prednumber of to do
            if Reach ( from, prednumber ) == true then
                return true
    return false
```

Fig. 13. The Reach algorithm.

5.3 The Reach Algorithm

To summarize the construction so far: Each node in the ICFG of a thread belongs to an intraSCR. The intraSCR may be a multi-node intraSCR or a single-node intraSCR. Each intraSCR belongs to one or more ISCRs. Each ISCR is numbered in topological order and may have one or more `TopolNumbers`. This is equivalent to inlining the ISCRs. Since the `TopolNumbers` represent a single-procedure program, it is easy to determine reachability on them. We build a *TopologicalNumber* graph which has as its nodes `TopolNumbers` and whose edges represent successor/predecessor relationships on the `TopolNumbers`. The TopologicalNumber graph is shown in Figure 10(b). Once this is generated, we no longer explicitly need the ISCR graph and so its edges can be deleted.

In this section, we explain how to efficiently determine reachability on the TopologicalNumber graph. Given two `TopolNumbers`, t_x and t_y, we give an algorithm, Reach to determine whether there is a path from t_x to t_y. Clearly, if the value of t_x is larger than that of t_y, then there is no path from t_x to t_y. A simple solution would be to walk back along the TopologicalNumber graph from t_y until we reach t_x or a number smaller then t_x. If we reach t_x, then there is a path, else there is no path. However, the number of `TopolNumbers` can be exponential in the depth of the call graph and hence this is not an efficient solution. The Reach algorithm shown in Figure 13 performs an optimization that skips across called procedures where possible. As a result, the algorithm's complexity is linear in the size of the ISCR graph.

The `TopolNumbers` that represent call and return sites are marked accordingly. In addition, we keep a dummy edge from the `TopolNumber` associated with a call site to the `TopolNumber` associated with the corresponding return

site. For example, we create a dummy edge from 9 to 12 and from 4 to 7 in Figure 10(b).

The algorithm starts from the TopolNumber with the larger value and walks back along the TopologicalNumber graph. Let the source TopolNumber value be t_x and the destination TopolNumber value be t_y. When the algorithm reaches a return site, it uses the dummy edge to check if the call site value is smaller than t_x. If the call site value is smaller than t_x then it descends into the called procedure, else it can use the dummy edge to skip the called procedure and continue traversing backwards from the call site. Thus, the traversal enters a procedure only if t_x is in the called procedure or in a procedure called (transitively) from the called procedure; otherwise, it keeps going up the TopologicalNumber graph. Thus, the maximum number of TopolNumbers visited by the algorithm is at most the number of ISCRs. The algorithm, Reach, to determine whether there is a path from a TopolNumber t_x to another TopolNumber t_y is given in Figure 13. An initial invocation of Reach(t_x, t_x) trivially returns false if t_x belongs to an ISCR with attribute *multi* set to false.

5.4 The Realizable Path Algorithm

The Reach algorithm described in the previous section determines reachability from one TopolNumber to another. However, we need reachability from one node in the ICFG to another. So we determine the set of TopolNumbers that a node can map to and use this as the basis for determining reachability on the nodes of the ICFG. We now present the *ValidPath* algorithm to determine whether a *set* of nodes belong to a realizable path. The algorithm for determining the existence of a path is given in Figure 14. As a node belongs to a unique intraSCR, we have a function getIntraSCR(node n_i) that returns the intraSCR that n_i belongs to. The procedure GetTopologicalNumberSet(*intraSCR*) does the following: It finds all the ISCRs that the intraSCR belongs to and returns the set of TopolNumbers associated with these ISCRs. GetTopologicalNumberSet(χ_{20}), for example, would find the ISCRs Z10, Z11 and Z15 and the corresponding TopolNumbers set is (5, 10, 14, 15). The overloaded procedure GetTopologicalNumberSet(node n_i) returns GetTopologicalNumberSet(getIntraSCR(n_i)).

RealizablePath(n_i, n_j, μ) returns all possible TopolNumbers associated with n_i if n_j is $\perp$. That is to say that n_i is the first node visited in a given thread and hence all paths to n_i are realizable. If n_j is not $\perp$, then in RealizablePath(n_i, n_j, μ), μ is a proper subset of all the TopolNumbers associated with n_j and represents the valid numbers associated with n_j at a particular position in the path. The procedure IntraSCRPath(q_i, μ) called by RealizablePath locates every TopolNumber associated with q_i that reaches at least one of the TopolNumbers in μ. If it returns $\emptyset$ it implies that there is no realizable path from n_i to n_j.

Example 5. As an example, we will walk through the algorithm to check if ⟨ S24, S20, S8 ⟩ in Figure 7 is a realizable path. The nodes are processed in reverse order. S8 $\in \chi_9$ and therefore μ is initialized to the set of TopolNumbers associated with χ_9, which is (13). S20 $\in \chi_{20}$ and the TopolNumbers associated with χ_{20} are (5, 10, 14, 15). Of these, we need to check only 5 and 10 since 14 and

```
procedure ValidPath(NodeSet ⟨n_1, n_2, ··· , n_k⟩)
    μ = GetTopologicalNumberSet(n_k)
    for i = k − 1 downto 1 do
        μ' = RealizablePath(n_i, n_{i+1}, μ)
        if μ' == ∅ then
            return false
        μ = μ'
    endfor
    return true

procedure IntraSCRPath(intraSCR q_i, TopolNumberSet μ)
    newset = ∅
    ν = GetTopologicalNumberSet(q_i)
    for each TopolNumber t_x ∈ ν do
        for each TopolNumber t_y ∈ μ do
            if t_x ≤ t_y and Reach(t_x, t_y) == true then
                newset = newset ∪ t_x
                break
        endfor
    endfor
    return newset

procedure RealizablePath(Node n_i, Node n_j, TopolNumberSet μ)
    if n_j == ⊥ then
        return GetTopologicalNumberSet(n_i)
    return IntraSCRPath(getIntraSCR(n_i), μ)
```

Fig. 14. The Realizable Path algorithm.

15 are larger than 13 and hence cannot precede 13. We have both Reach(5, 13) is true and Reach(10, 13) is true. So now *newset* has (5, 10), which is returned by RealizablePath. Next we check S24 $\in \chi_{21}$ whose TopolNumbers are (6, 11, 14, 15). Of these none are smaller than 5 and only 6 is smaller than 10 but Reach(6, 10) is false and hence this is not a realizable path.

The correctness of the algorithm follows from the correctness of the RealizablePath algorithm. The correctness of the RealizablePath algorithm can be stated as

THEOREM 1. *For any two nodes n_i and n_j in the ICFG, RealizablePath(i, j, μ) returns all and only the* TopolNumbers *associated with n_i that have a path to some* TopolNumber *associated with n_j in μ.*

Since every node belongs to a unique intraSCR node, the correctness of the ValidPath algorithm may be stated as

THEOREM 2. *Given a set of intraSCR nodes and the corresponding set of* Topol-Numbers *$\langle\langle q_1, \mu_1\rangle, \langle q_2, \mu_2\rangle, \ldots, \langle q_n, \mu_n\rangle\rangle$, such that μ_n = GetTopologicalNumberSet(q_n) and μ_i = IntraSCRPath(q_i, μ_{i+1}), for $1 \leq i < n$, then $\langle q_1, q_2, \ldots, q_n\rangle$ is a realizable path if and only if $\mu_i \neq \emptyset$ for $1 \leq i \leq n$.*

COROLLARY 1. *Given a set of intraSCR nodes and the corresponding set of* TopolNum-bers *$\langle\langle q_1, \mu_1\rangle, \langle q_2, \mu_2\rangle, \ldots, \langle q_n, \mu_n\rangle\rangle$, such that μ_n =*

GetTopologicalNumberSet(q_n) *and* μ_i = IntraSCRPath(q_i, μ_{i+1}), *for* $1 \leq i < n$, *then* $\langle q_0, q_1, q_2, \ldots, q_n \rangle$ *is a realizable path if and only if* $IntraSCRPath(q_0, q_1, \mu_1) \neq \emptyset$.

Thus, by Corollary 1, it is clear that to add a node to an existing realizable path, it is sufficient to check for a path from the new node to some *valid* `TopolNumber` of the last node added to the path. It is not necessary to run ValidPath on the entire set of nodes in the path or even to keep track of all the previous nodes in the path. Thus, the RealizablePath algorithm can be used to *incrementally* determine whether a set of nodes form a realizable path.

We give a proof of correctness of the RealizablePath algorithm in Appendix A.

5.5 Complexity

Since we are using a form of procedure inlining, the space and time complexity can in the worst case grow exponentially. If there is a long call chain, of length y, where each procedure in the chain is called from x sites then a procedure at the bottom of the call chain may get inlined x^y times. However, we use only one integer to represent the inlined procedure and hence we found that the space requirement was practical for our sample programs.

The steps in building the interprocedural SCR graph involve finding SCRs in each procedure and finding SCRs in the call graph. There are standard near-linear algorithms for doing this. Integrating the intraprocedural SCR graphs with the call SCR graph to generate the interprocedural SCR graph requires a single walk through the program and also takes linear time. The algorithm for inlining the procedures and calculating the `TopolNumbers` has potentially exponential performance if there are very long chains of procedures that need to be integrated.

The time complexity of the RealizablePath algorithm is linear on the number of `TopolNumbers`. However, the number of `TopolNumbers` can be exponential in the depth of the call graph, and hence the algorithm has exponential complexity. However, despite this, in practice the algorithm is very fast. We have tested it on several Java benchmark programs and the results are given in Section 9.

6. A CONTEXT-SENSITIVE SLICING ALGORITHM

6.1 The Algorithm

To simplify the presentation, we assume that threads are not nested and threads are not nested within loops. We discuss these extensions in Section 7.1 and 7.2. The context-sensitive algorithm (Figure 15) is an extension of the context-insensitive algorithm with modifications to ensure that only realizable paths are traversed.

The algorithm takes as input the slicing criterion, the ISCR graph of each thread and the TSDG. It starts with the slicing criterion, s, and inserts it into the outermost list w_0. It keeps extracting a node from w_0 and applies a 2-phase algorithm to it. The same coloring scheme as in the context-insensitive

Input: the slicing criterion s, the TSDG, the TopologicalNumber graph
Output: the slice $\mathcal{S}$
$\mathcal{S} = \emptyset$
mu = GetTopologicalNumberSet(s)
$C = (s, [t_0, \ldots, t_n], \text{phase1}),$ $\begin{cases} \textbf{if } \theta(s) == \theta_i \textbf{ then } t_i.node = s,\ t_i.\mu = mu \\ \textbf{else } t_i.node = \perp,\ t_i.\mu = \emptyset \end{cases}$
w0 = $\{C\}$

while w0 $\neq \emptyset$ **do**
  remove next element from **w0** and insert into **w1**
  while w1 $\neq \emptyset$ **do** /* Phase 1 */
    remove next element $c = (x, T, color)$ from **w1**
    for all $y \mid (y, x) \in E_{id}$
      $t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
      Insert (y, RealizablePath (y, t, mu), T, `phase1`, **w0**)
    for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{pi} \cup E_c$
      $t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
      Insert (y, RealizablePath (y, t, mu), T, `phase1`, **w1**)
    for all $y \mid (y, x) \in E_{po}$
      $mu = T[\theta(x)].\mu$
      Insert (y, RealizablePath (y, x, mu), T, `phase2`, **w2**)
  endwhile

  while w2 $\neq \emptyset$ **do** /* Phase 2 */
    remove next element $c = (x, T, color)$ from **w2**
    for all $y \mid (y, x) \in E_{id}$
      $t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
      Insert (y, RealizablePath (y, t, mu), T, `phase1`, **w0**)
    for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{po}$
      $mu = T[\theta(x)].\mu$
      Insert (y, RealizablePath (y, x, mu), T, `phase2`, **w2**)
  endwhile
endwhile

procedure Insert (Node: y, TopolNumberSet: mu, Tuple: T, int color, List: $\mathcal{L}$)
  if $mu == \emptyset$ **then return**
  $T[\theta(y)].node = y$
  $T[\theta(y)].\mu = mu$
  $c' = (y, T, color)$
  if c' has not already been calculated **then**
    $\mathcal{S} = \mathcal{S} \cup \{y\}, \mathcal{L} = \mathcal{L} \cup \{c'\}$

Fig. 15. The context-sensitive interprocedural slicing algorithm.

algorithm is applied to the elements of the worklists in the context-sensitive algorithm also.

When adding a node, n_i in some thread θ_i, to the slice, we need to check if there is a realizable path in θ_i that includes n_i and the set of nodes already added to the slice in θ_i. Therefore, when computing the slice we need to carry sufficient information to capture the entire path currently sliced in each thread. A tuple of

nodes containing one node per thread is maintained (similar to the tuples used in the intraprocedural algorithm [Krinke 1998; Nanda and Ramesh 2000]). In the tuple, the node corresponding to a thread is the last node reached (so far) in that thread. Along with the node we also keep the valid set of `TopolNumbers` associated with the node and the color of the node. So each element in the state tuple consists of three items, (i) the last node visited by the algorithm in that thread, (ii) the valid set of `TopolNumbers` associated with that node and (iii) the color of the node. When a node n_i in thread θ_i is visited, we update the tuple entry for that thread. Later if the algorithm visits a node, n_j, in the same thread θ_i we check if there is a *realizable* path from n_j to n_i using the RealizablePath test. If so, n_j is added to the slice (along with the valid set of `TopolNumbers`) else it is rejected.

We use the notation $\{n_i, \langle a, b, \ldots, k\rangle\}$ to represent a node numbered n_i with the set of `TopolNumbers` $\langle a, b, \ldots, k\rangle$. We use the symbol $\perp$ in the state tuple to indicate that the corresponding thread has not yet been visited. The function $\theta(n_i)$ returns the index of the thread to which n_i belongs.

Example 6. Let us apply this algorithm to `K12` in Figure 3. In the figure, the nodes are numbered on the left for identification and the `TopolNumbers` are in circles on the right. For simplicity, we use the short notation n_i to represent a node and the expanded form $\{n_i, \langle a, b, \ldots, k\rangle\}$ only at important points.

—Iteration 1:
  —Phase 1: {`K12`, [($\perp$, $\emptyset$), (`K12`, $\langle 33\rangle$), ($\perp$, $\emptyset$)]} $\leftarrow$ `K1` $\leftarrow$ `M4` $\leftarrow$ `M1`.
  The notation {`K12`, [($\perp$, $\emptyset$), (`K12`, $\langle 33\rangle$), ($\perp$, $\emptyset$)]} indicates that the node sliced is `K12`. The state of θ_0 is ($\perp$, $\emptyset$) indicating that it has not yet been visited. The state of θ_1 is (`K12`, $\langle 33\rangle$) indicating that the last node visited in θ_1 was `K12` and the corresponding valid `TopolNumber` set is $\langle 33\rangle$. The state of θ_2 is also ($\perp$, $\emptyset$) indicating that it has not yet been visited.
  `K12` $\leftarrow$ `K11`. (`K11` is added to w_2.)
  —Phase 2: `K11` $\leftarrow$ `K7` $\leftarrow$ `K10`. `K11` $\leftarrow$ `K26`. `K26` $\leftarrow$ `K25` $\leftarrow$ `K18` $\leftarrow$ `K17`.
  {`K25`, [($\perp$, $\emptyset$), (`K25`, $\langle 29\rangle$), ($\perp$, $\emptyset$)]} $\leftarrow$ {`N4`, [($\perp$, $\emptyset$), (`K25`, $\langle 29\rangle$) (`N4`, $\langle 4\rangle$)]}. (`N4` is added to w_0). The algorithm records the fact that the last node visited in θ_1 was `K25` with `TopolNumber` set $\langle 29\rangle$.
—Iteration 2:
  —Phase 1: `N4` $\leftarrow$ `N1`. `N4` $\leftarrow$ `N3` $\leftarrow$ `N8` (`N8` is added to w_2).
  —Phase 2: `N8` $\leftarrow$ `N7` $\leftarrow$ `N6` $\leftarrow$ `N5`.
  {`N7`, [($\perp$, $\emptyset$), (`K25`, $\langle 29\rangle$) (`N7`, $\langle 5\rangle$)]} $\leftarrow$ {`K15`, [($\perp$, $\emptyset$), (`K15`, $\langle 41\rangle$), (`N7`, $\langle 5\rangle$)]}. In this case, since there is no path from the `TopolNumber` $\langle 41\rangle$ in θ_1 to $\langle 29\rangle$, this node is rejected by the slice giving a context-sensitive solution.
  `N7` $\leftarrow$ `K6`. `N7` $\leftarrow$ `K24`.
  {`N7`, [($\perp$, $\emptyset$), (`K25`, $\langle 29\rangle$) (`N7`, $\langle 5\rangle$)]} $\leftarrow$ {`K30`, [($\perp$, $\emptyset$), (`K30`, $\langle 8, 25\rangle$), (`N7`, $\langle 5\rangle$)]}. The `TopolNumber` set of `K30` is $\langle 8, 25, 38\rangle$ of which there is no path from $\langle 38\rangle$ to $\langle 29\rangle$. Hence RealizablePath returns a subset of the possible `TopolNumbers` $\langle 8, 25\rangle$ which are the valid numbers for the node at this point in the slice.
  `N7` $\leftarrow$ `K21`.

All of K6, K24, K30, K21 are added to w_0.

—Iteration 3:
  —Phase 1: K21 ← K17 ← K10. K21 ← K19 ← K8 ← K2. Since the algorithm is in Phase 1, the parameter-in and call edge is traversed.
  —Phase 2: No nodes were added to w_2 and so there is no Phase 2.

—Iteration 4:
  —Phase 1: K24 ← K22 ← K20 ← K9 ← K3. K24 ← K31 (K31 is added to w_2).
  —Phase 2: K31 ← K30 ← K29. Since the algorithm is now in Phase 2, the parameter-in edge to K4 is not traversed.

—Iteration 5:
  —Phase 1: {K30, [(⊥, ∅), (K30, ⟨8, 25⟩), (N7, ⟨5⟩)]} ← {K29, [(⊥, ∅), (K29, ⟨7, 24⟩), (N7, ⟨5⟩)]}. Again, a subset of the TopolNumbers is allowed.
  K29 ← K4 ← K5. K29 ← K22. {K29, [(⊥, ∅), (K29, ⟨7, 24⟩), (N7, ⟨5⟩)]} ← {K13, [(⊥, ∅), (K13, ⟨34⟩), (N7, ⟨5⟩)]}. In this case, K13 is rejected as there is no path from ⟨34⟩ to ⟨7, 24⟩. In Iteration 4, K30 was colored phase2. In this iteration it is colored phase1 and sliced again.

—Iteration 6:
  —Phase 1: K6 ← K4 ← K5.

Example 7. Consider the slice of M14 in Figure 4(b) again. In Figure 4(a), we have shown the topological ordering for each thread in circles on the right end of each node. The numbers are *local* to a thread. The ISCRs in $f1$ have two numbers each corresponding to the two call sites. All others have only one TopolNumber each. The TopolNumbers are also depicted in the corresponding TSDG shown in Figure 4(b).

The algorithm starts by inserting {M14, [(⊥, ∅), (M14, ⟨30⟩), (⊥, ∅)} into w_0. Here the state tuple associated with θ_0 and θ_2 is (⊥, ∅) as these threads have not yet been visited and the state tuple for θ_1 is (M14, ⟨30⟩) indicating that the last node visited in θ_1 was M14 and its corresponding valid TopolNumbers set is ⟨30⟩.

—Iteration 1:
  —Phase 1: M14 ← M12 ← M21.
  —Phase 2: M21 ← M19 ← N3 in θ_2. The element inserted into w_0 will be {N3, [(⊥, ∅), (M19, ⟨9, 23⟩), (N3, ⟨3⟩)]}. The state tuple for θ_0 is still (⊥, ∅), for θ_1 it is (M19, ⟨9, 23⟩) indicating that M19 was the last node visited in θ_1 and its corresponding valid TopolNumbers are 9 and 23.

—Iteration 2:
  —Phase 1: N3 ← M20. The possible TopolNumbers for M20 are 10 and 24 but RealizablePath(M20, M19, ⟨9, 23⟩) returns ⟨10⟩ since Reach(24, 9) and Reach(24, 23) are false but Reach(10, 24) is true. The element inserted into w_0 now is {M20, [(⊥, ∅), (M20, ⟨10⟩), (N3, ⟨3⟩)]} indicating that not all TopolNumbers associated with M20 are now valid.

—Iteration 3:
  —Phase 1: M20 ← N2. Insert {N2, [(⊥, ∅), (M20, ⟨10⟩), (N2, ⟨2⟩)]} into w_0.

—Iteration 4:
  —Phase 1: N2 ← M8 and N2 ← M2. However, RealizablePath(M8, M20, ⟨10⟩) returns ∅ since Reach(16, 10) is false and hence the algorithm rejects M8 from the slice but correctly accepts M2 into the slice as Reach(2, 10) is true.

PROOF OF CORRECTNESS. Let p be the slicing criterion in a given TSDG; let S_p be the slice computed by the algorithm. Then the correctness of the algorithm can be stated as

THEOREM 3. $\mathcal{S}(p) = S_p$

The proof is given in Appendix B.

6.2 Complexity

In intraprocedural slicing, each node in the slice may be inserted into the worklist $O(n^{t-1})$ times [Nanda and Ramesh 2000], where n is the number of nodes in a thread and t is the number of threads. This gives a slicing complexity of $O(N^t)$, where N is the number of nodes in the graph. A node may be represented by an exponential number of TopolNumbers and for each TopolNumber it may be inserted into the slice an exponential number of times. This makes the complexity of the algorithm doubly exponential $O(N^{p^t})$, where p is the calling depth of the call graph. However, in Section 9, we show that with certain optimizations, the algorithm is practical.

7. EXTENDING THE PROGRAM MODEL

The slicing algorithm essentially requires information regarding (1) whether two threads may execute in parallel, (2) the reachability of one node from another, that is, RealizablePath computation, (3) control dependence, (4) data dependence, and (5) interference dependence. In this section, we show how to compute reachability between nodes in the presence of nested threads and threads nested within loops. In the next section we show how to compute these factors for Java programs.

7.1 Nested Threads

Consider a slice beginning with K4 in θ_1 in Figure 16(a). K4 is interference dependent on N2 in θ_2. N2 is interference dependent on L2 in θ_3. Since, the thread θ_3 has never been visited before, the algorithm would add L2 to the slice although it is clear that K4 is not dependent on L2. In the intraprocedural slicing algorithm [Nanda and Ramesh 2000], when a node n_i in θ_i is visited, we update every element of the tuple that corresponds to a thread that does not execute in parallel with θ_i. A similar technique is used for interprocedural slicing: if n_i is dependent on some node $[n_j, T_j, color_j]$, the valid TopolNumber set associated with n_i is μ_i and $color_i$ is determined as explained before, then we create a new tuple $[n_i, T_i, color_i]$ such that

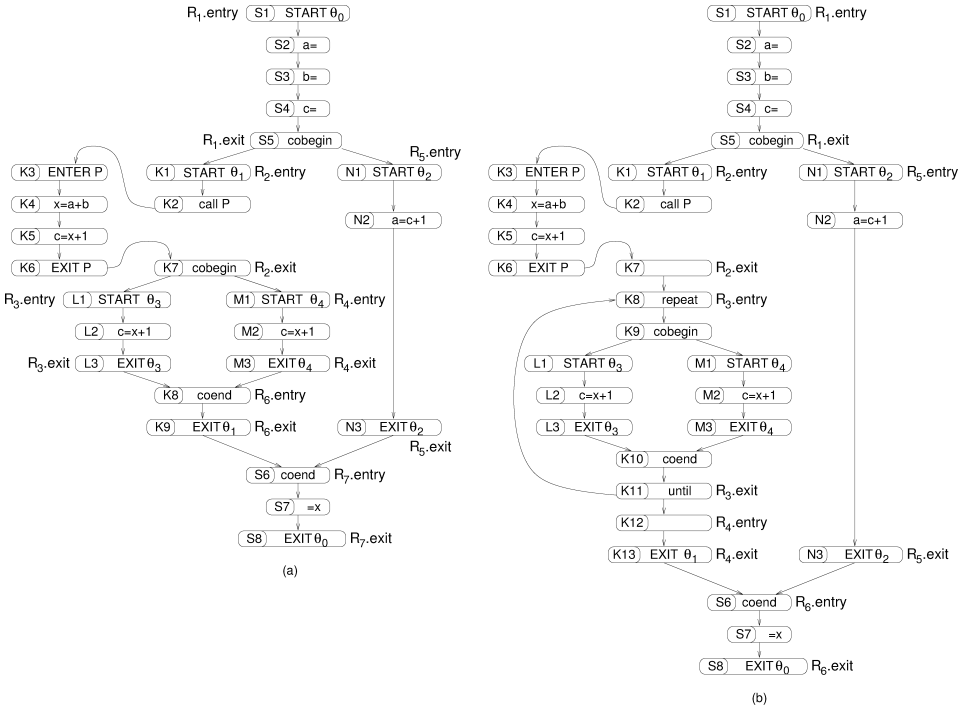


Fig. 16. (a) Nested threads and (b) Threads nested within loops.

for each thread θ_k
    if $||(\theta(n_i), \theta_k)$ is false then
        $T_i[\theta_k].node = n_i$
        $T_i[\theta_k].\mu = \mu_i$
        $T_i[\theta_k].color = color_i$
    else
        $T_i[\theta_k].node = T_j[\theta_k].node$
        $T_i[\theta_k].\mu = T_j[\theta_k].\mu$
        $T_i[\theta_k].color = T_j[\theta_k].color$

where $||(n_i, n_j)$ has been defined earlier as a function that returns true for two nodes n_i and n_j, if they belong to threads that may execute in parallel. Here we overload the function to $||(\theta_i, \theta_j)$ which returns true if two threads θ_i and θ_j may execute in parallel or false otherwise. Consider the slice of K4 in Figure 16(a).

—K4 is inserted into w_1 as {K4, [(K4, $\mu_{(K4)}$, phase1), (K4, $\mu_{(K4)}$, phase1), ($\perp$, $\emptyset$, undefined), (K4, $\mu_{(K4)}$, phase1), (K4, $\mu_{(K4)}$, phase1)]}.
Since, θ_1 executes sequentially with θ_0, θ_3 and θ_4, they are marked in the same way as θ_1.

—K4 is interference dependent on N2 which is added to the slice as {N2, [(N2, $\mu_{(N2)}$, phase1), (K4, $\mu_{(K4)}$, phase1), (N2, $\mu_{(N2)}$, phase1), (K4, $\mu_{(K4)}$, phase1), (K4, $\mu_{(K4)}$, phase1)]}.

—L2 is interference dependent on N2, L2 belongs to θ_3 and the tuple entry for θ_3 has (K4, $\mu_{(K4)}$, phase1). Since the RealizablePath algorithm is applied only to a single thread we need some way to determine whether there is a path from L2 to K4.

We need some algorithm to determine whether there is a path from a node in some thread θ_i to another node in a sequentially executing thread θ_j. The threaded control flow graph has a properly nested structure and hence it is easy to divide it into a set of regions that have a proper global ordering.

Each thread is broken into a set of single-entry-single-exit regions, where the entry node is either the START θ_i node or a coend node and the exit node of the region is either a cobegin node or the EXIT θ_i node of the thread. In Figure 16(a), θ_0 is broken into two regions, $\mathcal{R}_1$ and $\mathcal{R}_7$. The region $\mathcal{R}_1$ has its entry node as S1 (START θ_0) and the exit node is S5 (the cobegin node). The region $\mathcal{R}_7$ has the entry node S6 (coend) and the exit node is S8 (EXIT θ_0). Similarly, θ_1 is broken into two regions, $\mathcal{R}_2$ and $\mathcal{R}_6$ and the other threads have only one region each.

We calculate reachability for these regions and store the information in bitvectors. A node in a thread may belong to one or more regions. Each TopolNumber of a node maps to a specific region. When we apply RealizablePath(n_i, n_j, μ), if n_i and n_j belong to different threads then RealizablePath() has the following functionality

—Let $\mathcal{R}_j$ be the regions associated with μ.
—Let $\mathcal{R}_i$ be a region associated with n_i that has a path to $\mathcal{R}_j$
—For every such $\mathcal{R}_i$ return the TopolNumbers of n_i that map to some region in $\mathcal{R}_i$ or return $\emptyset$ if $\mathcal{R}_i$ is empty.

In the example, there is no path from $\mathcal{R}_3$ (the region to which L2 belongs) to $\mathcal{R}_2$ (the region to which K4 belongs) and hence, L2 is correctly rejected from the slice.

7.2 Threads Nested within Loops

When threads are nested within loops, loop-carried dependences that cross thread boundaries may give rise to a conservative slice. This has been explained in detail in the context of intraprocedural slicing [Nanda and Ramesh 2000] and a brief summary is in Section 2.2. In intraprocedural slicing, a loop carried data dependence from a thread θ_i to θ_j is treated as a sequential data dependence even if θ_i and θ_j may execute in parallel. In addition, reachability between two nodes, n_i and n_j, is calculated on the region defined by the closest enclosing cobegin-coend region. However, in interprocedural slicing, we could not find a way to determine reachability within a region and maintain the calling context information that is stored in the μ component of each tuple.

Instead, we use the conservative estimate, that any node within a loop is reachable from any other node in the loop. If a loop encloses a cobegin-coend construct, then we create a separate region for the entire loop and every node in the loop belongs to this region. In Figure 16(b) we show the regions created when a cobegin-coend construct is nested within a loop. In the example, θ_1 is divided into three regions as shown. We need not determine topological ordering for θ_3

and θ_4, as every node in θ_3 is reachable from every other node and similarly for θ_4. We assign all the nodes within the loop to the same ISCR. The rest of the algorithm remains unchanged.

8. SLICING CONCURRENT JAVA PROGRAMS

Unlike the structured `cobegin-coend` parallelism construct, Java supports `fork-join` parallelism, monitors and explicit `wait/notify` synchronization. The `fork-join` parallelism affects the $||(i, j)$ function, the synchronization gives rise to additional dependencies and monitors affect the interference dependence. In this section we explain how to handle these variations.

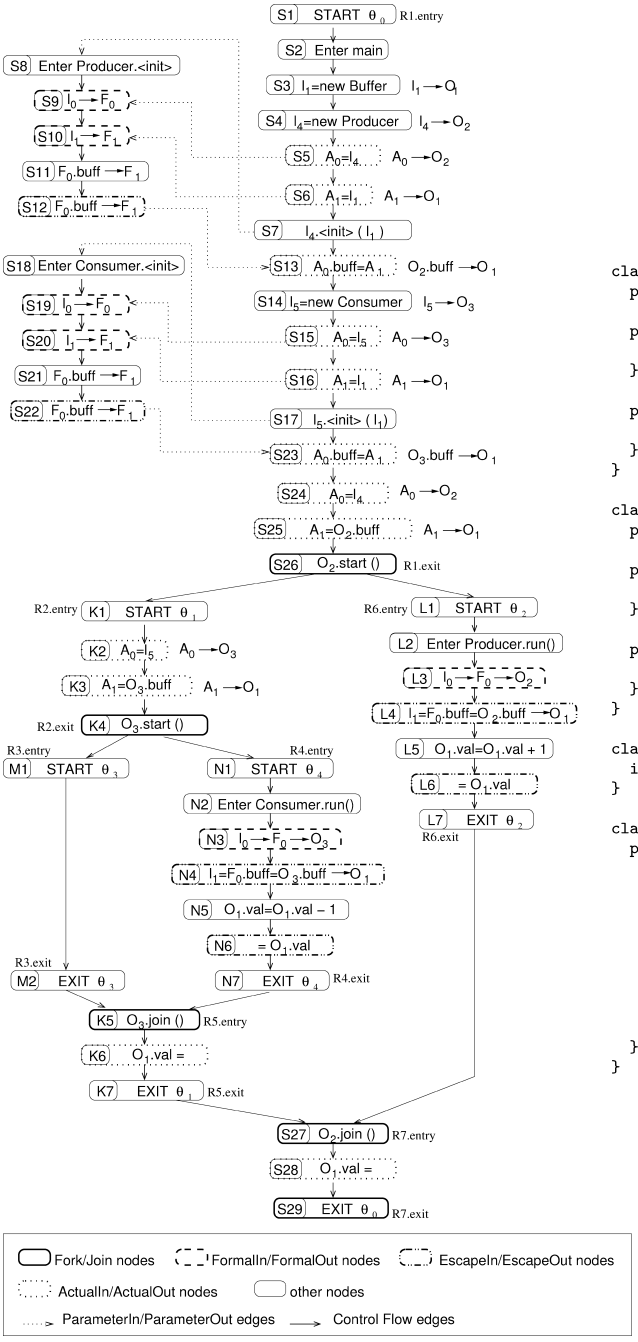
8.1 The Control Flow Graph

8.1.1 *The Basic Thread Model.* Figure 17 shows the ITCFG for a minimalistic producer consumer program written in Java. A method that is used in more than one thread is replicated in each thread. At the call site `p.start()` of an object `p` that extends the `Thread` class or implements the `Runnable` interface, we create a "`fork`" node. At a `fork` node, the parent thread generates a child thread which continues to execute in parallel with the remainder of the parent thread. The child thread continues execution at the `run` method that `p` must implement. In order to map the Java model to the `cobegin-coend` model, at a `fork` node, we create two new threads–one for the child thread and one for the parent thread. In Figure 17, `S26` is a `fork` node where two threads θ_1 and θ_2 are created. At `K1` in θ_1, the `main` method continues execution and at `L1` in θ_2, the `Producer.run()` method gets executed.

8.1.2 *Threads Created in Loops and Procedures.* A thread may be created multiple times within a program if it the thread creation statement occurs within a loop; a recursive call; or if the thread is created within a procedure which is invoked from more than one call site and there is a path from one call site to another. In such cases, we create two representative threads at the `fork` node, as two threads are sufficient to capture inter-thread communication. For a thread created within a procedure, for each call site, we compute the set of other threads that it may interact with.

8.1.3 *Open-Ended Threads.* The child thread may be open-ended or it may `join` the parent thread at some later point. The semantics of an open-ended thread imply that the thread may interfere with any thread that is created subsequently. This gives a conservative estimate of the interference dependence computation and the $||(i, j)$ function.

8.1.4 *The Semantics of* `Join`. The semantics of a `join` node are mapped to the semantics of a `coend` node as follows—in the method that issues the `join` command, we insert a thread exit node just before the call to `join`, then create a `join` node from where the method continues execution. In Figure 17, `M2` is a thread exit node created just before the call to O_3`.join()`. `K5` is the `join` node and it has an incoming edge from `M2` and from `N7` which is the thread exit node for `Consumer` thread. Mapping a `fork` node to a `join` node is achieved




```
class Producer extends Thread {
  private Buffer buff;

  public Producer (Buffer b) {
    this.buff = b;
  }

  public void run () {
    buff.val ++;
  }
}

class Consumer extends Thread {
  private Buffer buff;

  public Consumer (Buffer b) {
    this.buff = b;
  }

  public void run () {
    buff.val --;
  }
}

class Buffer {
  int val = 0;
}

class ProducerConsumerTest {
  public static void main
    ( String[] args ) {

    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);
    p.start();
    c.start();
    c.join();
    p.join();
  }
}
```

Fig. 17. Java threads.

by the pointer and escape analysis [Nanda and Ramesh 2003]. However, the ITCFG construction has the following limitations: If the `fork` and `join` nodes are in different methods we are unable to construct a suitable `join` node and so we conservatively ignore the `join` command and treat the threads as open-ended threads. In addition, if the `fork-join` construct does not generate properly nested threads, then also we conservatively ignore the `join` command. For example, in Figure 17, if the code in `ProducerConsumerTest` had been `p.join(); c.join()` instead of `c.join(); p.join()`, then the threads would not have a properly nested structure and we would have to treat the threads as open-ended threads.

8.2 The $||(i, j)$ Function

In `fork-join` parallelism, $||(n_i, n_j)$ is true if the nearest common ancestor of n_i and n_j, along some path, is a `fork` node.

Note. The $||(n_i, n_j)$ function is conservative and does not take into account the ordering induced by `wait-notify` synchronization.

8.3 RealizablePath Computation

RealizablePath computation is explained in Section 5. As in the `cobegin-coend` parallelism, we define a set of regions on the control flow graph and determine a global ordering across the regions. We define a function ThreadRegion(n_i) which returns the ThreadRegion to which n_i belongs. We also compute REACH($\mathcal{R}_i, \mathcal{R}_j$) over the entry nodes of all the ThreadRegion and store it in small bitvectors. For example, $\mathcal{R}_j$.REACH will have one bit for each ThreadRegion and the bit for $\mathcal{R}_i$ will be set if there is a path from $\mathcal{R}_i$ to $\mathcal{R}_j$. Then, to find out if there is a path n_i to n_j in the ITCFG, we first check if they belong to the same region. If yes, we apply the RealizablePath algorithm. If not, we need to check if there is a path from the ThreadRegion to which n_i belongs to the ThreadRegion to which n_j belongs (i.e., ThreadRegion(n_i) $\in$ ThreadRegion(n_j).REACH).

8.4 Control Dependence

Control dependence is calculated using standard techniques [Horwitz et al. 1990] for intraprocedural and interprocedural analysis. At the inter-thread level, we generated control dependence edges from the `START` node of each thread to the corresponding `fork` node from which it was generated. In addition, in Java, `exit()` statements and statements that throw exceptions can affect the control dependence of programs with procedure calls [Sinha et al. 1999; Harrold et al. 1998].

8.5 Data Dependence

For every variable that is referenced before it is used in a procedure we need to create a formal-in node and for every variable that is modified in a procedure we need to create a formal-out node. In Java, this form of MOD/REF analysis is achieved using a technique known as escape analysis [Nanda and Ramesh 2003; Whaley and Rinard 1999; Choi et al. 1999]. Since Java supports only call-by-value parameter passing, only a static variable or a field of a formal reference

parameter may belong to the REF set. These are called EscapeIn variables. Similarly the MOD set is referred to as EscapeOut. The escape analysis generates EscapeIns and EscapeOuts for upwards exposed reads and downwards exposed writes of "escape" variables. The corresponding actual-in and actual-out nodes are also created.

Java has a simple object model in which all reference-type variables point to a heap-allocated object. Thus, the inter-thread pointer analysis converts every field $v.f$ to the form $O_i.f$ where O_i is a symbolic heap location.

For interference dependence we determine which objects escape a thread. Such an object that is assigned in one node and referenced in a parallel executing node, generates an interference dependence edge. For example in Figure 17, three symbolic locations O_1, O_2 and O_3 are generated at S3, S4 and S14. The pointer analysis propagates these locations and determines that at N5 and L5, the variable $O_1.val$ is being used and defined. Since N5 and L5 may execute in parallel, an interference dependence edge is generated from N5 to L5 and from L5 to N5.

In Java, monitors are defined by the synchronized keyword. For synchronized blocks of code, only downward exposed definitions and upward exposed references are used to determine the interference between threads [Nanda and Ramesh 2003; Novillo et al. 1998].

8.6 Synchronization Dependence

Synchronization primitives wait and notify are handled as follows: Synchronization edges are drawn from a notify node to the corresponding wait nodes. Each object has an implicit lock associated with it. wait is treated as having an implicit read on the lock variable and notify is treated as having an implicit write on the lock variable. Then, the data dependence calculation automatically detects the dependence between a write on a $O_i.\langle$lock$\rangle$ at the O_i.notify statement and a read on $O_i.\langle$lock$\rangle$ at the O_i.wait statement. In addition, a wait statement implies the release of all locks held by the object and hence this is treated as an edge of a monitor section and all definitions reaching the wait statement are downward exposed and visible to other threads.

Analysis based on synchronization dependence [Krishnamurthy and Yelick 1995] may generate a partial order between the synchronized blocks that may further reduce the interference dependence edges.

9. IMPLEMENTATION

Although the complexity of the context-sensitive algorithm is doubly exponential, its performance can be improved by applying optimizations similar to the optimizations for intraprocedural slicing [Nanda and Ramesh 2000].

We start with a brief description of these optimizations for intraprocedural slicing [Nanda and Ramesh 2000]. To slice K3 in Figure 18, the algorithm would find that K3 is data dependent on K2 and insert K2 with the tuple [K2, K2, $\perp$, $\perp$]. Next the algorithm may determine that K3 is interference dependent on L2 which is interference dependent on K2. K2 would be inserted into the worklist again with the new tuple [K2, K2, L2, $\perp$]. In general, a node may be inserted into

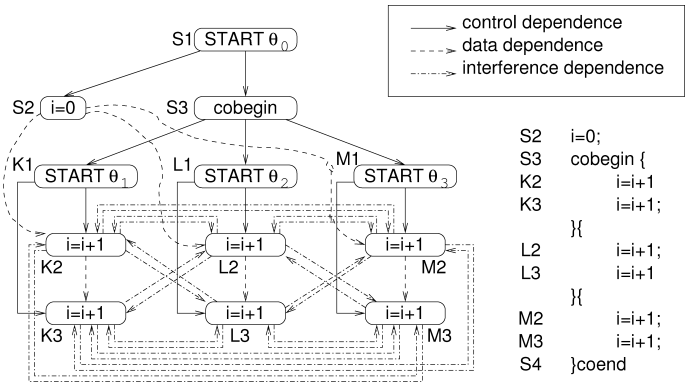


Fig. 18. Threaded PDG for a concurrent single-procedure program.

a worklist $O(n^{t-1})$ times, where n is the number of nodes in a thread and t is the number of threads.

There is a concept of a less restrictive tuple versus a more restrictive tuple. For example, the tuple [K2, K2, $\perp$, $\perp$] is less restrictive than the tuple [K2, K2, L2, $\perp$]. The first tuple says that any node in θ_2 and θ_3 may be visited, whereas the second tuple says that any node in θ_3 may be visited but in θ_2 *only* those nodes can be visited *that have a path to* L2. For example, although K2 is interference dependent on L3, the tuple [K2, K2, L2, $\perp$] would not allow the insertion of L3 since there is no path from L3 to L2 in θ_2. Clearly, the paths that can be traversed from the second tuple is a subset of the paths admissible from the first tuple.

The essence of the intraprocedural optimizations is to ensure that less restrictive tuples are sliced first and more restrictive tuples are discarded. In order to ensure that the slicing algorithm "finds" less restrictive tuples first, the algorithm gives nodes reached via interference dependence edges a low priority.

We extend the same notion to interprocedural slicing as follows: Inside the procedure Insert (Figure 19, 20 in the appendix), let *mu* have k TopolNumbers $t_1, t_2, \ldots, t_k$. Instead of creating one worklist element, we create k worklist elements. Each worklist element has one TopolNumber from the set *mu*. The data structures do not change, except that now in a tuple T, $T[i].\mu$ is either the empty set or it is a set with exactly one element. This is equivalent to inlining the procedures in the ISCR graph.[3] Then, we apply the optimization used in intraprocedural slicing as follows:

(1) Given that a node n_i has been inserted into the worklist with a tuple $[(n_1, \mu_1), (n_2, \mu_2), \ldots, (n_k, \mu_k)]$, if it gets a new tuple $[(n'_1, \mu'_1), (n'_2, \mu'_2), \ldots, (n'_k, \mu'_k)]$, where for all i, either n_i is $\perp$ or there is a realizable path from (n'_i, μ'_i) to (n_i, μ_i), then the new tuple is redundant and need not be added to the slice.

[3]Recall that the ISCR graph has no loops and no recursion.

(2) Nodes reached through interference dependence need to be given a low priority. This is already ensured by the algorithm since nodes reached via interference dependence edges are placed in the outermost queue which is processed only after nodes reached through other edges.

The complexity of the algorithm does not change but the running time of the algorithm was observed to improve on our sample programs.

By breaking each `TopolNumber` set into its individual components, it might appear that the number of elements inserted into the worklist is going to increase exponentially. However, note that, by the optimization, if there is a path from one `TopolNumber` t_i to another, t_j in the set mu, then a worklist element will be created only for t_j. Only in the case when a procedure is called from multiple branches of a switch which is not embedded within a loop do we actually end up inserting a large number of elements into the worklist. In practice, the improvement in slicing time to be gained by avoiding unnecessary paths, far outweighs the overhead of breaking the `TopolNumber` set into individual worklist elements.

Another practical optimization that we built into our implementation is as follows: if a thread has no incoming interference dependence edges (i.e., there is no node in the thread which is the source of an interference dependence edge) then we need not apply RealizablePath or maintain tuples for any node in that thread. This thread can never be reached via an interference dependence edge and hence may be sliced as a sequential thread. With this optimization, the algorithm reduces to the sequential two-phase algorithm in the absence of threads or interference dependence edges. The complete algorithm is given in Appendix C.

9.1 Experimental Results

The algorithms have been tested on a uniprocessor 2.66GHz Intel Pentium 4 workstation with 2GB of memory running under Linux. The algorithms analyze Java bytecode.

The characteristics of the programs have been described in Table I: `pc` is a simple Producer-Consumer program, `multi` is a multithreaded program that downloads a web site and all its links so that it can be browsed offline, `raytracer` is a multithreaded ray tracer, `hanoi` is an IBM benchmark program, `Slice` is a public domain applet obtained from Eric Ruf's web page [Ruf 2000], `mandelbrot` is Simon Arthur's applet to explore the Mandelbrot set, and `instantdb` is a database browser that comes with the `InstantDB` database. In addition, we ran our slicer on all the programs in the Java Grande [JavaGrand] suite, but in the interests of space we include results for one program from each of its three classes of applications (`sync`, `series` and `montecarlo`). We give a break-up of data based on user code and library code. Note that the degree of concurrency in our sample programs is quite small and so the experimental results do not test the limits of scalability that may arise due to higher concurrency.

For the slicing criterion, we chose every node that was the destination of an interference dependence edge. This resulted in approximately 30 to 2000 slicing criteria depending on the program. The results are given for averages

Table I. Benchmark Programs

Program	Number of Classes		Methods		Statements		Threads	Degree of Concurrency
	User	Lib	User	Lib	User	Lib		
pc	4	148	9	224	2932	32536	3	3
multi	10	148	95	397	16659	57917	3	3
raytracer	8	242	19	577	18722	210676	5	2
slice	13	244	67	678	34055	206225	5	2
hanoi	29	241	104	594	87464	234472	4	2
mandelbrot	18	276	51	727	87872	407081	4	2
instantdb	3	319	34	814	80008	466105	4	2
sync	6	152	52	578	53641	116910	5	3
series	6	152	44	270	39145	79969	3	3
montecarlo	18	160	259	514	197240	120250	3	3

Table II. Build Mitime (seconds)

Program	CFG + CD + DD + ID	Summary Edges	IntraSCR + ISCR Graph Topological Ordering
pc	0.48	0.18	0.01
multi	1.32	0.57	0.04
raytracer	17.79	3.83	1.01
slice	17.06	6.74	0.81
hanoi	45.23	35.16	1.85
mandelbrot	140.36	94.39	12.24
instantdb	166.47	64.12	13.17
sync	19.49	1.25	0.96
series	4.27	1.45	1.24
montecarlo	15.72	10.82	5.13

over all the slicing criteria as well as for the criterion node that generated some maximum or minimum size.

In Table II, we give the time to build the various components of the threaded system dependence graph. The first column gives the time to parse the program and calculate the control, data and interference dependence edges. The second column gives the time to build the summary edges. The third column gives the time to build the ISCR graph and to generate the topological ordering for all the threads. Clearly, the time to build the ISCR graph and the `TopolNumbers` is a small fraction of the total time to build the threaded system dependence graph.

9.1.1 *Performance Analysis.* In Table III, we give the average and maximum time taken to compute a slice for the context-insensitive algorithm (column "I"), the context-sensitive algorithm without optimizations (column "V") and the context-sensitive algorithm with optimizations (column "O"). The minimum time, in each case was too small to measure. The computation pace (time per node) was too small to measure. The context-sensitive algorithm without optimizations takes a long time to compute and for many programs it was found to be impractical (i.e., it took more than 20 minutes of CPU time to generate the slice for most nodes).

Table III. Total Time (in seconds) to Slice for the Context-Insensitive Algorithm (I), the Context-Sensitive Algorithm without Optimization (V) and the Context-Sensitive Algorithm with Optimization (O); the Number of Nodes in the Slice for (V) and (O); and Computation Pace (time in seconds per node) for the Context-Sensitive Algorithm with Optimization (O)

	Total Slice Time					
	I		V		O	
Program	Ave	Max	Ave	Max	Ave	Max
pc	0.0	0.01	0.0	0.0	0.0	0.0
multi	0.001	0.02	0.02	0.07	0.006	0.03
raytracer	0.014	0.03	0.12	0.25	0.035	0.09
slice	0.014	0.03	—	—	0.02	0.07
hanoi	0.04	0.08	—	—	0.05	0.18
mandelbrot	0.08	0.11	—	—	0.13	0.4
instantdb	0.11	0.14	—	—	0.20	0.41
sync	0.02	0.03	—	—	0.10	0.19
series	0.002	0.01	0.03	0.05	0.001	0.01
montecarlo	0.03	0.09	—	—	0.08	0.16

Table IV. Worklist Analysis for the Context-Sensitive Algorithm without Optimization (V) and The Context-Sensitive Algorithm with Optimization (O). Values are for the Slicing Criterion that had the Largest Ratio of Nodes in the Worklist to Nodes in the Slice

	Max Inserts Per Node			Total Elements in Worklists			Actual Nodes in Slice		Ratio		
Program	I	V	O	I	V	O	I	V/O	I	V	O
pc	2	5	2	82	218	83	77	61	1.1	3.6	1.4
multi	2	97	23	6083	64330	5204	5644	1935	1.1	33.3	2.7
raytracer	2	189	17	16547	61696	34843	13941	9542	1.2	6.5	3.65
slice	2	—	18	19235	—	32338	14933	9952	1.3	—	3.25
hanoi	2	—	58	36144	—	69290	28762	13366	1.3	—	5.2
mandelbrot	2	—	55	57018	—	156558	43221	23684	1.3	—	6.6
instantdb	2	—	8	71374	—	50134	59624	21632	1.2	—	2.3
sync	2	—	30	17458	—	24832	16385	10622	1.07	—	2.3
series	2	85	4	613	16649	628	613	329	1.0	50.6	1.9
montecarlo	2	—	26	34650	—	33705	32480	17317	1.07	—	1.9

Another measurement of the performance of the algorithm is the number of nodes inserted into the worklist. In the context-insensitive algorithm, some nodes get sliced twice as they get colored `phase2` and later `phase1`. Hence, the number of nodes handled by the worklist is greater than the total number of nodes in the slice. In the context-sensitive algorithm, a node may be inserted into the worklist with potentially exponential number of tuples. In Table IV, we give the maximum number of times a single node is inserted into the worklist with a different tuple and the total number of nodes handled by the worklists for the context-insensitive algorithm ("I"), for the context-sensitive algorithm without optimization ("V") and for the context-sensitive algorithm with optimization ("O"). As a base for comparison, we also show the actual number of nodes in the slice (which is the same for context sensitive slice with and without optimizations). In the last column, we show the ratio of nodes handled by the

Table V. Precision Analysis. Average Number of Nodes Sliced in the Context-Insensitive (1) and the Context-Sensitive Algorithms (2) and the Percentage Range in Reduction of Nodes (3)

Program	Context-Insensitive (1)	Context-Sensitive (2)	(3)
pc	68	37	40% to 48%
multi	2010	711	74% to 88%
raytracer	8079	5574	1.5% to 83.3%
slice	6695	2911	4.8% to 84.6%
hanoi	22566	8194	5.8% to 90%
mandelbrot	34318	17903	2.8% to 78.4%
instantdb	40420	13079	61.3% to 73.9%
sync	14604	8101	34.1% to 80%
series	430	295	23.5% to 46.7%
montecarlo	22194	11820	32.9% to 46%

worklists to the actual number of nodes in the slice. The values are shown for the slicing criterion that generated the largest ratio (worst case).

Although the total number of nodes handled by the worklists is, in general, a small (< 10) multiple of the number of nodes in the slice, we observe that the number of times a single node is inserted into the slice can be much higher (e.g., a maximum of 189 in raytracer). The improvement due to the optimizations is also clearly visible as a drop in the maximum insertions per node as well as a drop in the total worklist count.

9.1.2 *Precision Analysis.* In Table V, we show the average number of nodes sliced by the context-insensitive algorithm and the context-sensitive algorithm. (Obviously, there is no difference in precision between the context-sensitive algorithm with and without optimization.) In the last column we give the improvements to be gained by using a context-sensitive algorithm rather than a context-insensitive algorithm. For example, in `hanoi` the context-sensitive algorithm generated between 5.8% to 90% fewer nodes than the context-insensitive algorithm. The gain in precision from context-insensitive to context-sensitive analysis may be worth the extra computing power.

We also implemented Zhao's algorithm and found that his algorithm misses nodes in some programs. In the case of the programs `raytracer`, `slice`, `hanoi` and `mandelbrot` Zhao's algorithm generated between 0% to 8% fewer nodes than the context-insensitive algorithm, whereas in `pc` and `multi` there was no difference between the two algorithms.

9.1.3 *The ISCR Graph.* In Table VI, we give statistics related to the ISCR graph. It is interesting to note the reduction in the size of the graph when using an interval-based approach. For each program, we give statistics for the thread with the maximum number of ISCRs generated. The table gives the original number of statements in the thread; the number of intraSCRs and ISCRs; and the largest value of a `TopolNumber`. The maximum number of ISCR nodes visited by a single call to Reach is equal to the maximum number of ISCRs in the graph. As can be seen from the table, the maximum number of ISCRs is much smaller than the largest value of a `TopolNumber` and hence Reach is very

Table VI. The ISCR Graph

Program	LOC	No. of IntraSCR	No. of ISCR	Max Value of a `TopolNumber`	Max Numbers at an IntraSCR
pc	54	9	6	6	1
multi	55049	2461	1336	13174	348
raytracer	207249	4212	1636	20425	672
slice	212155	4965	2474	135166	4031
hanoi	290059	5170	2629	289155	8256
mandelbrot	449266	6316	3479	703312	19104
instantdb	527054	7123	4185	488878	10740
sync	129091	2847	1167	257492	7786
series	118276	2995	1610	391444	11861
montecarlo	239018	4208	2150	1269957	38366

fast. The last column gives the maximum number of `TopolNumbers` associated with a single intraSCR. This governs the number of iterations in the two loops in IntraSCRPath called by RealizablePath.

The maximum number of `TopolNumbers` is also an indication of the problems of using algorithms based on procedure inlining. Initially we tried using a bitvector solution to compute Reach with one vector for each `TopolNumber` and in each vector, one bit for each `TopolNumber` indicating reachability. This would have required $703312 * 703312$ bits (roughly 60GigaBytes of RAM) for `mandelbrot`. In contrast, we now use one integer for each `TopolNumber` with a requirement of roughly 2.5 MB (assuming 4 bytes per integer) which is more manageable.

9.1.4 *Remarks.* The context-insensitive algorithm is very fast and requires little overhead in terms of memory (it does not require calculation of the ISCR graph or topological ordering) but is definitely less precise than the context-sensitive algorithm. The context-sensitive algorithm has exponential complexity, but, in general, few nodes display this exponential behavior and most nodes get inserted into the worklist a small constant number of times. With the optimizations, the algorithm was found to be practical for our sample benchmark programs. Note, however that our sample programs display a low degree of concurrency. We have not tested the limits of scalability that may arise due to higher concurrency.

In our tests, we have analyzed complete programs including all library methods (except native methods). Often we are not interested in slicing library methods. There are standard techniques [Horwitz et al. 1990] for analyzing sequential programs in the absence of library code and it may be possible to extend these techniques to concurrent programs (we leave this for future work) and would considerably increase the speed of slicing. However, our experience has been that while there are rarely errors in library code, errors in user programs often occur due to incorrect usage of library code. Hence, it is important not to neglect library code in the analysis.

In applications such as model checking [Millet and Teitelbaum 1998] and formal verification where it is important to get as small a slice as possible, it may be well worth the additional computation to generate a context-sensitive slice.

9.1.5 *Limitations of the Implementation.* Our data dependence analysis generates synchronization dependence edges but we do not apply the synchronization analysis to determine a partial order based on `wait-notify` synchronization. Hence we do not eliminate interference dependence edges based on synchronization analysis. Java supports virtual method calls, so we use type information to refine the call graph. However, we do not support dynamic class loading. This assumption enables us to perform whole program analysis including construction of a static call graph.

10. RELATED WORK

The entire work in this article consisting of context-insensitive and context-sensitive algorithms, along with the correctness proofs and experimental results, appeared as part of the first author's thesis work, reviewed by an international panel of examiners and approved in November 2001 [Nanda 2001]. Recently, a very similar algorithm has been published [Krinke 2003]. While the basic ideas of path folding and slice computation are the same in Krinke's article, the underlying methodology in our algorithm is very different. Krinke's algorithm has been described in the introduction. Essentially he uses callstrings to capture the calling context. The callstrings need to be truncated to 2 or 3 elements to avoid a combinatorial explosion of callstrings. Chops need to be computed between the slicing criterion and every node in the thread that has an incoming interference dependence edge. Nodes that are not in the chop may be sliced using summary edges. However, the remaining nodes must be sliced using the expensive and imprecise callstring approach. Here we would like to add the following remarks: We believe that (1) Our algorithm is more precise since our realizable path algorithm is more precise—it does not suffer from limitations of truncated callstrings; (2) Our algorithm is more efficient – it uses summary edges effectively and has additional optimizations. It may be noted that both approaches handle fork-join conservatively.

For concurrent programs, most of the work on slicing has been done on intraprocedural slicing. Cheng [1993] presents an approach for slicing programs where interprocess communication is channel-based. As Tip [1995] notes, Cheng does not state or prove any property of the slices computed. Krinke [1998] gives a precise slicing algorithm for programs with shared memory. This however becomes imprecise in the presence of nested threads and threads nested within loops. Nanda et al. [2000] give a more efficient slicing algorithm which remains precise in the presence of all program constructs. None of these handle issues such as synchronization and monitors.

Hatcliff et al. [1999] analyze Java programs with monitors and synchronization. They introduce several new dependencies including a *ready dependence*, where a node, n_i, is ready dependent on another node, n_j, if n_j's failure to complete could imply that n_i never executes. For example, a `wait` statement is ready dependent on the corresponding `notify` statement and all statements that may be reached from a `wait` statement are ready dependent on the `wait` statement. However, they ignore the imprecision introduced by the intransitivity of interference dependence. More recently, Ranganath et al. [2004] show how to reduce

the number of interference dependence edges for concurrent Java programs. Though they do not give experimental results on slicing, this should improve the precision and speed of slicing. Chen and Xu [2001] give an algorithm that handles the imprecision due to interference dependence to some extent. However, they require to inline all procedures that contain the source or destination of an interference or synchronization dependence edge.

Intraprocedural slicing of programs has a limitation in that it is often impractical to slice realistic programs. Procedures need to be inlined causing the control flow graph to grow potentially without bounds. Programs with recursion either cannot be analyzed or are analyzed conservatively.

Millet and Teitelbaum [1999] give an algorithm for slicing Promela. They also use an extension of the SDG but it is not clear whether their algorithm is context-sensitive.

11. CONCLUSION

In this article, we have given a *context-insensitive* interprocedural solution to slicing concurrent programs that is efficient and *correct*. Then we have given a *context-sensitive* interprocedural slicing algorithm for concurrent programs which is both correct and comparatively more precise. We have shown how to extend the analysis to handle nested threads. We have implemented the algorithm for Java programs and give statistics on a set of benchmark programs. Although the context-sensitive solution is exponential in complexity, we show that it may be practical for some programs.

The context-sensitive algorithm may generate conservative results when threads are nested within loops. The limitations of the context-sensitive algorithm are due to the limitations of determining realizable paths in concurrent programs with procedure calls. However, despite the limitations, our experiments show that the context-sensitive algorithm is more precise than a context-insensitive algorithm. We plan to use the output of the slicing algorithm for efficient verification of Java programs, after which we will have a better idea about the usefulness of context-sensitivity versus context-insensitivity.

APPENDIX

A. CORRECTNESS OF REALIZABLE PATH ALGORITHM

In this appendix, we give the proof of Theorem 1, Theorem 2 and Corollary 2 defined in Section 5.4 and restated below for convenience.

THEOREM 1. *For any two ICFG nodes n_i and n_j in the ISCR graph, RealizablePath (n_i, n_j, μ) returns all and only the* `TopolNumbers` *associated with n_i that have a path to some* `TopolNumber` *associated with n_j in μ.*

THEOREM 2. *Given a set of intraSCR nodes and the corresponding set of* `Topol-Numbers` $\langle\langle q_1, \mu_1\rangle, \langle q_2, \mu_2\rangle, \ldots, \langle q_n, \mu_n\rangle\rangle$, *such that* $\mu_n = \text{GetTopologicalNumberSet}(q_n)$ *and* $\mu_i = \text{IntraSCRPath}(q_i, \mu_{i+1})$, *for* $1 \leq i < n$, *then* $\langle q_1, q_2, \ldots, q_n\rangle$ *is a realizable path if and only if* $\mu_i \neq \emptyset$ *for* $1 \leq i \leq n$.

COROLLARY 2. *Given a set of intraSCR nodes and the corresponding set of* `Topol-Numbers` $\langle\langle q_1, \mu_1\rangle, \langle q_2, \mu_2\rangle, \ldots, \langle q_n, \mu_n\rangle\rangle$, *such that* $\mu_n = \text{GetTopologicalNumberSet}(q_n)$ *and* $\mu_i = \text{IntraSCRPath}(q_i, \mu_{i+1})$, *for* $1 \leq i < n$, *then* $\langle q_0, q_1, q_2, \ldots, q_n\rangle$ *is a realizable path if and only if* $\mathit{IntraSCRPath}(q_0, \mu_1) \neq \emptyset$.

By construction, every ITCFG node n_i belongs to exactly one intraSCR node q_i; every intraSCR node q_i is part of at least one ISCR node in the ISCR graph. Each ISCR node has at least one `TopolNumber` associated with it. This follows from the assumption that there is a path from `ENTRY` to every node in the ICFG and a path from every node to `EXIT` in the ICFG. The proof uses the following fact: for an intraSCR node q_i, $\text{GetTopologicalNumberSet}(q_i)$ returns *all* the `TopolNumbers` of all the ISCR nodes in the ISCR graph to which q_i belongs.

By construction, every intraprocedural and interprocedural loop is collapsed into a single node. Therefore, there are no loops in the ISCR graph. Also, by construction, the topological numbering inlines all the procedures in the ISCR graph. Hence, we have the following lemmas.

LEMMA 1. *The ISCR graph has no loops.*

LEMMA 2. *Topological numbering reduces the program to a single procedure with no loops.*

LEMMA 3. *For any two* `TopolNumbers` t_i *and* t_j *in the ISCR graph* $\mathit{Reach}(t_i, t_j)$ *is true if and only if there is a path from* t_i *to* t_j. *Also, if there is a path from* t_i *to* t_j, *then* $\mathit{Reach}(t_i, t_j)$ *is true.*

PROOF. This follows from Lemma 1 and Lemma 2.

PROOF OF THEOREM 1. The proof follows from Lemma 3 and the $\text{IntraSCRPath}(q_i, \mu)$ algorithm, which returns every `TopolNumber` of $t_i \in q_i$ such that $\text{Reach}(t_i, t_j)$ is true for some $t_j \in \mu$ and rejects any `TopolNumber` of t_i if $\text{Reach}(t_i, t_j)$ is false for all $t_j \in \mu$.

LEMMA 4. *Given any three* `TopolNumbers` t_i, t_j *and* t_k, *a path from* t_i *to* t_j *and a path from* t_j *to* t_k *implies there is a realizable path through* $\langle t_i, t_j, t_k\rangle$.

PROOF. By Lemma 2 and by property of intraprocedural paths.

PROOF OF THEOREM 2. By Theorem 1, $\text{IntraSCRPath}(q_i, \mu)$ returns a nonempty set if and only if there is a path from some `TopolNumber` of q_i to some `TopolNumber` in μ. Hence, there is a path from q_i to q_{i+1} if and only if $\mu_i = \text{IntraSCRPath}(q_i, \mu_{i+1})$ and $\mu_i \neq \emptyset$. If any μ_i is the empty set, then there is no path from that q_i to q_{i+1} and there cannot be a realizable path. Conversely, if $\mu_i \neq \emptyset$ for $i \leq 1 \leq n$, then there is a path from every q_i to q_{i+1} and by Lemma 4 there is a realizable path through $\langle q_1, q_2, \ldots, q_n\rangle$. Hence, the theorem.

PROOF OF COROLLARY 2. By Theorem 2, there is a realizable path through $\langle q_1, q_2, \ldots, q_n\rangle$. By Theorem 1, there is a path from q_0 to q_1, and by Lemma 4, there is a realizable path through $\langle q_0, q_1, q_2, \ldots, q_n\rangle$.

Thus, to add a new node to the realizable path, it is sufficient to apply IntraSCRPath to the last node added to the path and its corresponding set of `TopolNumbers`.

B. CORRECTNESS OF THE CONTEXT-SENSITIVE SLICING ALGORITHM

In this appendix, we give a proof of Theorem 3 defined in Section 6 and restated below

THEOREM 3. $\mathcal{S}(p) = S_p$

Definition 5. There is an *intra*-thread transitive dependence from a node n_i to a node n_j, if there is a sequence of dependencies $n_i \xrightarrow{e_i} \cdots \xrightarrow{e_k} n_j$ such that none of e_i is an interference dependence edge.

THEOREM 4. *If there is an* intra-*thread transitive dependence from a FormalIn node to a FormalOut node at a call site of a procedure, then there is a summary edge from the corresponding ActualIn node to the corresponding ActualOut node. Conversely, if there is a summary edge from an ActualIn node to an ActualOut node, then there is an* intra-*thread transitive dependence from the corresponding FormalIn node to the corresponding FormalOut node* [Horwitz et al. 1990].

LEMMA 5. *A tuple* $(n_1, T_1, color_1)$ *is added to the one of the lists* w_0, w_1, *or* w_2, *at some stage if and only if there exists a sequence of tuples* $(n_i, T_i, color_i)$, $i = 1, \ldots, n$ *that is added to one of these lists where the tuples satisfy the following properties:*

(1) *For* $i = n$
  —$n_n = p$
  —$T_n(\theta_j).node = \bot$ `if` $\theta_j \neq \theta(n_n)$
    $T_n(\theta_j).node = n_n$ `if` $\theta_j == \theta(n_n)$
  —$color_n =$ `phase1`.

(2) *For* $i = 1, \ldots, n-1, n > 1$
  —$n_i \xrightarrow{e_i} n_{i+1}$, $e_i \in \{cd, dd, c, s, pi, po, id\}$
  —$color_i \in \{phase1, phase2\}$
  —*For* $j = 1, \ldots, |\theta|$
    —$T_i[\theta_j].node = T_{i+1}[\theta_j].node$, `if` $\theta_j \neq \theta(n_i)$
    $= n_i$, `if` $\theta_j == \theta(n_i)$
    —$T_i[\theta_j].\mu = RealizablePath(n_i, T_{i+1}[\theta_j].node, T_{i+1}[\theta_j].\mu)$ `if` $\theta_j \neq \theta(n_i)$
    $= T_{i+1}[\theta_j].\mu$ `if` $\theta_j == \theta(n_i)$
    —$T_i[\theta_j].\mu \neq \emptyset$

(3) *For* $i = 1, \ldots, (n-1)$, one *of the following holds*
  —$e_i \in \{cd, dd, s\}$, $color_i = color_{i+1}$
  —$e_i = po$, $color_i \leq color_{i+1}$
  —$e_i \in \{pi, c\}$, $color_i = color_{i+1} =$ `phase1`
  —$e_i = id$, $color_i =$ `phase1`

PROOF. There are two parts to this proof:
"If" part: This is obvious.
"Only If" part: Assume that $(n_1, T_1, color_1)$ is added to one of the lists. We need to show that there exists a sequence of tuples $(n_i, T_i, color_i)$, $i = 1, \ldots, n$ having the desired properties that get added to one of the lists. This follows by computational induction from the following two facts:

(1) The tuple $(p, T, phase1)$ has such a sequence, where $T[\theta].node = \perp$, for all $\theta \neq \theta(p)$ and $T[\theta(p)].node = p$.
(2) If every tuple that is already added to one of the lists has such a sequence, then every new tuple that gets added because of these tuples, has a sequence.

(1) is obvious. (2) follows from the fact that the sequence for the new tuple is got from the sequence corresponding to one of the tuples already in the list.

LEMMA 6. *A node n_i is colored* `phase2` *if and only if there exists a sequence of tuples $n_i \rightarrow \cdots \rightarrow n_j \rightarrow n_k$, such that n_j is a FormalOut node in the same procedure as n_i and n_j is also colored* `phase2`*; and n_k is an ActualOut node which may be colored* `phase1` *or* `phase2` *(n_j is parameter-out dependent on n_k).*

PROOF. This follows from Definition 5 and the fact that all nodes reached by interference dependence edges are colored `phase1`. Hence it is not possible to enter `phase2` except via a parameter-out dependence edge originating at an *ActualOut* node.

LEMMA 7. *Given a node n_j and a* `TopolNumber` *set μ such that $\mu \neq \emptyset$ and another node n_i such that $n_i \xrightarrow{*} n_j$ and for each edge e_i in the path from n_i to n_j we have $e_i \in \{dd, cd, s, po, c, pi\}$, then RealizablePath$(n_i, n_j, \mu) \neq \emptyset$.*

PROOF. By property of data dependence, control dependence, summary dependence, parameter-out dependence, call dependence and parameter-in dependence, there is a realizable path from n_i to n_j.

THEOREM 5. $\mathcal{S}(p) \subseteq S_p$.

Proof. Assume $n \in \mathcal{S}(p)$, then by definition of $\mathcal{S}(p)$, there is a threaded witness from n to p. Let the threaded witness be $\langle n = n_1, n_2, \ldots, n_n = p \rangle$. We shall construct a sequence of tuples $(n_i, T_i, color_i), i = 1, \cdots, n$, that satisfy the property mentioned in Lemma 7, thereby proving the theorem.

Let T_n and $color_n$ be as given in (1) of the property of Lemma 5.

Now, we define inductively T_i given T_{i+1} for $i = 1, \ldots (n-1)$. For each pair of nodes in the witness, n_i and n_{i+1}, let $n_i \xrightarrow{e} n_{i+1}$, where $e \in E'$, that is, e is of type 'cd', 'dd', 'c', 'pi', 'po', or 'id' (but not 's'). The definition of T_i depends upon the type of e. We shall define T_i for each of these types:

(1) e is 'cd' or 'dd' and n_i and n_{i+1} belong to the same thread $\theta(n_i)$:
In the tuple T_{i+1} associated with n_{i+1}, let $T_{i+1}[\theta(n_i)].\mu$ be μ_{i+1}. Then at n_i we have the tuple T_i such that $T_i[\theta(n_i)].\mu = \text{RealizablePath}(n_i, n_{i+1}, \mu_{i+1})$. Since n_i and n_{i+1} belong to a threaded witness and since n_i and n_{i+1} belong

to the same thread, there must be a (realizable) path from n_i to n_{i+1} and by Lemma 7, we get $T_i[\theta(n_i)].\mu \neq \emptyset$. Also from the algorithm, we get $color_i = color_{i+1}$.

(2) e is 'id' (Obviously, n_i and n_{i+1} belong to different threads) or case 'cd', 'dd' such that n_i and n_{i+1} belong to different threads:
In the tuple T_{i+1} associated with n_{i+1}, let n' be the previous node belonging to $\theta(n_i)$ in the threaded witness. Then $T_{i+1}[\theta(n_i)].node = n'$ and let $T_{i+1}[\theta(n_i)].\mu$ be μ'. Then, at n_i we have the tuple T_i such that $T_i[\theta(n_i)].\mu =$ RealizablePath(n_i, n', μ'). If n' is $\perp$ then RealizablePath(n_i, n', μ') will return GetTopologicalNumberSet(n_i) and hence is not an empty set. Else, since n_i and n' belong to a threaded witness and since n_i and n' belong to the same thread, there must be a path from n_i to n' and by Lemma 7, we get $\mu' \neq \emptyset$.

Also from the algorithm, we get $color_i = color_{i+1}$ for 'cd' and 'dd' and for 'id' edges $color_i = phase1$.

(3) e is 'pi' : if $color_{i+1}$ is `phase1` then all the properties of Lemma 5 are met in the same way as item 1. If $color_{i+1}$ is `phase2`, then by Lemma 6 there is a sequence of nodes $\langle n_i, n_{i+1}, \ldots, n_f, n_a \rangle$, such that all the nodes belong to $\theta(n_i)$. Further n_f is a *FormalOut* node with color `phase2`, n_a is an *ActualOut* node with color `phase1` or `phase2`. Also by Lemma 6, n_a is in the sequence. Further, by Theorem 4, n_a is summary dependent on n_i.

Let the state tuple associated with n_a be T_a such that $T_a[\theta(n_i)].\mu = \mu_a$. Then at n_i we have the tuple T_i such that $T_i[\theta(n_i)].\mu = \mu_i =$ RealizablePath(n_i, n_a, μ_a). By Theorem 4, there is a path from n_i to n_a and by Lemma 7, we get $\mu_i \neq \emptyset$.

Also from the algorithm, we get $color_i = color_{i+1}$ since there is summary edge from n_i to n_a.

(4) case 'c' : if $color_{i+1}$ is `phase1` then all the properties of Lemma 5 are met in the same way as for item 1. If $color_{i+1}$ is `phase2`, then by Lemma 6 there is a sequence of nodes $n_i, \ldots, n_f, n_a$, such that n_f is a *FormalOut* node with color `phase2`, n_a is an *ActualOut* node with color `phase1` or `phase2`, and n_a is call dependent on n_i (by construction).

Let the state tuple associated with n_a be T_a such that $T_a[\theta(n_i)].\mu = \mu_a$. Then at n_i we have the tuple T_i such that $T_i[\theta(n_i)].\mu = \mu_i =$ RealizablePath(n_i, n_a, μ_a). By construction, there is a path from n_i to n_a and by Lemma 7, we get $\mu_i \neq \emptyset$.

(5) e is 'po' : $color_i \leq color_{i+1}$ and all the properties of Lemma 5 are met in the same way as for item1.

Now construct the tuples $(n_i, T_i, color_i), i = 1, \cdots, n$. It is easy to see that this sequence satisfies the required properties of Lemma 5. Hence the theorem.

THEOREM 6. *$S_p \subseteq \mathcal{S}(p)$. If a node q is added to the slice S_p, then it is in $\mathcal{S}(p)$*

PROOF. If q is added to the slice then by Lemma 5 there is a sequence of tuples $(n_i, T_i, color_i), i = 1, \ldots, n$ with $n_1 = q$ that obey the properties of tuples

given in Lemma 5. We show that these properties imply that there is a threaded witness for q. We have $n_i \xrightarrow{e_i} n_{i+1}$, where $e_i \in \{cd, dd, c, s, pi, po, id\}$. In order to show that this sequence forms a threaded witness, we need to show that for every thread θ_i, the subsequence of nodes $n_1^i, \ldots, n_k^i$ belonging to thread θ_i form a realizable path.

Let n_i and n_{i+1} be any two consecutive nodes added to the slice. We show inductively that if n_{i+1} forms a part of a threaded witness, then n_i also forms a threaded witness. Consider the different dependence edges from n_i to n_{i+1}.

(1) case 'pi', 'po' or 'cd', 'dd' such that n_i and n_{i+1} belong to the same thread (i.e., $\theta(n_i) == \theta(n_{i+1})$). (In the case of 'pi' and 'po', $\theta(n_i) == \theta(n_{i+1})$ is always true):
Let μ_{i+1} be the set of `TopolNumbers` associated with n_{i+1}. Then in the state tuple for n_{i+1}, we have $T_{i+1}[\theta(n_i)].\mu = \mu_{i+1}$. In the state tuple for n_i, we have $T_i[\theta(n_i)].\mu = \mu_i$ such that $\mu_i = \text{RealizablePath}(n_i, n_{i+1}, \mu_{i+1})$. By Lemma 5, $\mu_i \neq \emptyset$. Then by Corollary 2 (Section A) there is a realizable path from n_i to n_{i+1} and hence there is a threaded witness for n_i.

(2) case 'id' or 'cd', 'dd' such that n_i and n_{i+1} belong to different threads (i.e., $\theta(n_i) \neq \theta(n_{i+1})$):
Let the last node visited by the algorithm in $\theta(n_i)$ be some n'. If n' is $\perp$ then n_i is the first node in its thread and it obviously forms a realizable path in its thread. If n' is not $\perp$, then in the state tuple associated with n_{i+1}, we have $T_{i+1}[\theta(n_i)].node = n'$ and $T_{i+1}[\theta(n_i)].\mu = \mu'$. In the state tuple associated with n_i, we have $T_i[\theta(n_i)].\mu = \mu_i$ such that $\mu_i = \text{RealizablePath}(n_i, n', \mu')$. By Lemma 5 $\mu_i \neq \emptyset$. Then by Corollary 2 there is a realizable path from n_i to n' and hence there is a threaded witness for n_i.

(3) case 's' : In a threaded witness, no pair of adjacent nodes are related by the summary edges. So, in order to show the existence of threaded witness, we first replace every pair n_i, n_{i+1}, that are related by a summary edge by a sequence of nodes that are related by 'cd', 'dd', 'pi' and 'po' edges alone. This is always possible thanks to the property of summary edges (Theorem 4). Also, by Definition 5 and Theorem 4 there is a transitive dependence from n_i to n_{i+1} composed of only 'cd', 'dd', 'pi', 'po' edges. Then we can apply case 1 of this proof to the sequence of nodes from n_i to n_{i+1} that have only 'cd', 'dd', 'pi', 'po' edges. Thus, there is a realizable path from n_i to n_{i+1} and the nodes belong to a threaded witness.

Thus, for any n_i added to the slice, there is a threaded witness in $\theta(n_i)$. Hence, the theorem.

B.1 Correctness of the Context-Insensitive Algorithm

Let S_p^i be the slice computed by the context-insensitive algorithm, then the correctness can be stated as

THEOREM 7. $\mathcal{S}(p) \subseteq S_p^i$.

For the context-insensitive algorithm we have the following lemma:

Input: the slicing criterion s, TSDG
Output: the slice $\mathcal{S}$
mu = GetTopologicalNumberSet(s)
for each $m \in mu$ **do**
$C = (s, [t_0, \ldots, t_n], \text{phase1}), \begin{cases} \textbf{if } ||(\theta(s), \theta_i) \text{is } \textbf{false then } \ t_i.node = s, \ t_i.\mu = \{m\} \\ \textbf{else } t_i.node = \perp, \ t_i.\mu = \emptyset \end{cases}$
Insert $(s, \{m\}, [t_0, \ldots, t_n]$, `phase1`, **w0**)
endfor

while w0 $\neq \emptyset$ **do**
remove next element from **w0** and insert into **w1**
while w1 $\neq \emptyset$ **do** /* Phase 1 */
remove next element $c = (x, T, color)$ from **w1**
for all $y \mid (y, x) \in E_{id}$
$t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
Insert (y, RealizablePath' (y, t, mu), T, `phase1`, **w0**)
for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{pi} \cup E_c$
$t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
Insert (y, RealizablePath' (y, t, mu), T, `phase1`, **w1**)
for all $y \mid (y, x) \in E_{po}$
$mu = T[\theta(x)].\mu$
Insert (y, RealizablePath (y, x, mu), T, `phase2`, **w2**)
endwhile

while w2 $\neq \emptyset$ **do** /* Phase 2 */
remove next element $c = (x, T, color)$ from **w2**
for all $y \mid (y, x) \in E_{id}$
$t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
Insert (y, RealizablePath' (y, t, mu), T, `phase1`, **w0**)
for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{po}$
$mu = T[\theta(x)].\mu$
Insert (y, RealizablePath (y, x, mu), T, `phase2`, **w2**)
endwhile
endwhile

Fig. 19. The context-sensitive interprocedural slicing algorithm with optimizations.

LEMMA 8. *A tuple* $(n_1, color_1)$ *is added to the one of the lists* w_0, w_1, *or* w_2, *at some stage if and only if there exists a sequence of tuples* $(n_i, color_i)$, $i = 1, \ldots, n$ *that is added to one of these lists where the tuples satisfy the following properties:*

(1) *For* $i = n$
—$n_n = p$
—$color_n =$ `phase1`.
(2) *For* $i = 1, \ldots, n-1, n > 1$
—$n_i \xrightarrow{e_i} n_{i+1}$, $e_i \in \{cd, dd, c, s, pi, po, id\}$
—$color_i \in \{phase1, phase2\}$
(3) *For* $i = 1, \ldots, (n-1)$, one *of the following holds*
—$e_i \in \{cd, dd, s\}$, $color_i = color_{i+1}$
—$e_i = po$, $color_i \leq color_{i+1}$
—$e_i \in \{pi, c\}$, $color_i = color_{i+1} =$ `phase1`
—$e_i = id$, $color_i =$ `phase1`

```
procedure Insert ( Node: y, TopolNumberSet: mu, Tuple: T, int color, List: L )
    if mu == ∅ then return
    for each m ∈ mu do
        for all i s.t. ||(θ_i, θ(y)) is false do
            T[i].node = y
            T[i].μ = {m}
        endfor
        for each previous tuple, T', of y do
            redundant = true
            for all i do
                if RealizablePath''(T[i], T'[i]) == false then
                    redundant = false
                    break
            endfor
            if redundant == true then return
        endfor
        c' = (y, T, color)
        S = S ∪ {y}
        L = L ∪ c'
    endfor

procedure RealizablePath' ( Node: y, Node: t, TopolNumberSet: mu )
    if θ(y) == θ(t) then
        return RealizablePath(y, t, mu)
    if ThreadRegion(y) ∈ThreadRegion(t).REACH then
        return GetTopologicalNumberSet(y)
    else
        return ∅

procedure RealizablePath'' (
        (Node: x, TopolNumberSet: {Mx}),
        (Node: y, TopolNumberSet: {My}) )
    if y == ⊥ return true
    if Reach(Mx, My) return true
    return false
```

Fig. 20. The context-sensitive interprocedural slicing algorithm with optimizations—continued.

This is identical to Lemma 5 without the RealizablePath restriction. The proof of Lemma 8 is similar to the proof of Lemma 5 and the proof of Theorem 7 proceeds along the same lines as Theorem 5, using Lemma 8 instead of Lemma 5.

C. ALGORITHM FOR OPTIMIZED INTERPROCEDURAL SLICING OF CONCURRENT PROGRAMS

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their detailed comments on an earlier draft.

REFERENCES

AGRAWAL, H., DEMILLO, R., AND SPAFFORD, E. 1993. Debugging with dynamic slicing and backtracking. *Softw.—Pract. Exper. 23*, 6, 589–616.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

BECK, J. AND EICHMANN, D. 1993. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 509–518.

BINKLEY, D. AND GALLAGHER, K. 1996. Program slicing. *Advances in Computers 43*.

BINKLEY, D., HORWITZ, S., AND REPS, T. 1995. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Meth. 4*, 1 (Jan.), 3–35.

BURKE, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Prog. Lang. Syst. 12*, 3 (July).

CALLAHAN, D. 1988. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*.

CHEN, Z. AND XU, B. 2001. Slicing concurrent Java programs. *ACM SIGPLAN Notices 36*, 4 (Apr.), 41–47.

CHENG, J. 1993. Slicing concurrent programs. In *Automated and Algorithmic Debugging 1st International Workshop*. Lecture Notes in Computer Science, vol 749. Springer-Verlag, New York.

CHOI, J.-D., GUPTA, M., SREEDHAR, V. C., AND MIDKIFF, S. 1999. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to algorithms*. MIT press, Cambridge MA.

DIJKSTRA, E. W. 1968. *Programming Languages*. Academic Press, London, England.

GALLAGHER, K. B. AND LYLE, J. R. 1991. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng. 17*, 8, 751–761.

HARMAN, M., BINKLEY, D., AND DANICIC, S. 2003. Amorphous program slicing. *J. Syst. Softw. 68*, 1, 45–64.

HARMAN, M. AND DANICIC, S. 1995. Using program slicing to simplify testing. *Softw. Test. Verifi. Reliab.*, 143–162.

HARROLD, M. J., ROTHERMEL, G., AND SINHA, S. 1998. Computation of interprocedural control dependence. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York. 11–20.

HATCLIFF, J., CORBETT, J., DWYER, M., SOKOLOWSKI, S., AND ZHENG, H. 1999. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the Conference on International Static Analysis Symposium*.

HORWITZ, S., PRINS, J., AND REPS, T. 1989. Integrating noninterfering versions of programs. *ACM Trans. Prog. Lang. Syst. 11*, 3, 345–387.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst. 12*, 26–60.

JAVAGRAND. Benchmark suite—thread version 1.0. URL http://www.epcc.ed.ac.uk/computing/ research_activities/java_grande/.

JOHNSON, R. AND PINGALI, K. 1993. Dependence based program analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York. 78–89.

KRINKE, J. 1998. Static slicing of threaded programs. In *Proceedings of the Conference on Program Analysis for Software Tools and Engineering (PASTE 98)*. 35–42.

KRINKE, J. 2003. Context-sensitive slicing of concurrent programs. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*. ACM, New York.

KRISHNAMURTHY, A. AND YELICK, K. 1995. Optimizing parallel programs with explicit synchronization. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York. 196–204.

MÜLLER-OLM, M. AND SEIDL, H. 2001. On optimal slicing of concurrent programs. In *Proceedings of the ACM Symposium on Theory of Computing (STOC 2001)*. ACM, New York. 647–656.

MILLET, L. AND TEITELBAUM, T. 1998. Slicing Promela and its application to model checking, simulation, and protocol understanding. In *Proceedings of the 4th Workshop on Automata Theoretic Verification with the SPIN Model Checker*.

MILLET, L. AND TEITELBAUM, T. 1999. A case for channel dependence analysis: Slicing Promela. In *Proceedings of the 1999 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*.

NANDA, M. G. 2001. Slicing concurrent Java programs: Issues and solutions. Ph.D. thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. http://www.cse.iitb.ernet.in/~ramesh/gow-thesis.ps.gz; Examiners: Dr. Frank Tip, IBM T. J. Watson Research Center and Prof. R. Mall, Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur.

NANDA, M. G., BHADURI, P., OBEROI, S., AND SANYAL, A. 1999. An application of compiler technology to the year 2000 problem. *Softw.—Pract. Exper.*

NANDA, M. G. AND RAMESH, S. 2000. Slicing concurrent programs. In *Proceedings of the International Symposium on Software Testing and Analysis*.

NANDA, M. G. AND RAMESH, S. 2003. Pointer analysis of multithreaded Java programs. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, New York.

NOVILLO, D., UNRAU, R., AND SCHAEFFER, J. 1998. Concurrent SSA form in the presence of mutual exclusion. In *Proceedings of the International Conference on Parallel Processing*.

OTTENSTEIN, K. AND OTTENSTEIN, L. 1984. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM, New York. 177–184.

RAMALINGAM, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22*, 2, 416–430.

RANGANATH, V. P. AND HATCLIFF, J. 2004. Pruning interference and ready dependence for slicing concurrent java. In *Proceedings of the 13th International Conference on Compiler Construction (CC)*.

REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. 1994. Speeding up slicing. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering* (New Orleans, LA). ACM, New York. 11–20.

RUF, E. 2000. Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York. 208–218.

RUGINA, R. AND RINARD, M. 1999. Pointer analysis for multithreaded programs. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York.

SALCIANU, A. AND RINARD, M. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York.

SARKAR, V. 1991. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Devel. 35*, 779–804.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, 189–233.

SINHA, S., HARROLD, M. J., AND ROTHERMEL, G. 1999. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the International Conference on Software Engineering*. 432–441.

TIP, F. 1995. A survey of program slicing techniques. *J. Prog. Lang. 3*, 121–181.

VENKATESH, R. 1991. The semantic approach to program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10*, 352–357.

WHALEY, J. AND RINARD, M. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York.

YANG, W., HORWITZ, S., AND REPS, T. 1992. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Method. 1*, 3 (July), 310–354.

ZHAO, J. 1999. Slicing concurrent Java programs. In *Proceedings of the IEEE International Workshop on Program Comprehension*. IEEE Computer Society Press, Los Alamitos, CA.

Received January 2004; revised April 2005 and November 2005; accepted February 2006