Noname manuscript No.
(will be inserted by the editor)

Advanced Chopping of Sequential and Concurrent Programs*

Dennis Giffhorn

Received: date / Accepted: date

Abstract A chop for a source statement s and a target statement t reveals the program parts involved in conveying effects from s to t. While precise chopping algorithms for sequential programs are known, no chopping algorithm for concurrent programs has been reported at all. This work introduces six chopping algorithms for concurrent programs, which offer different degrees of precision, ranging from imprecise over context-sensitive to time-sensitive. Our evaluation on concurrent Java programs shows that context-sensitive and time-sensitive chopping reduces chop sizes significantly. We further present an extensive evaluation of chopping algorithms for sequential programs and describe a new, easy to implement chopping technique for sequential programs that computes fast and almost context-sensitive chops.

Keywords Chopping · Slicing · Program analysis · Concurrency · Threads

1 Introduction

A chop $chop(s, t)$ for a source statement s and a target statement t in a program p contains all statements conveying effects from s to t. Chopping is used in a wide range of applications as a preprocessing step identifying the relevant program parts for the main analysis, e.g. for vulnerability signatures (Brumley et al. 2006), path conditions (Snelting et al. 2006), input validation (Liu and Kuan Tan 2008), reducing programs for model checking (Shacham et al. 2007) and for witnesses for illicit information flow (Hammer and Snelting 2009). Such applications can benefit from chopping algorithms that are as precise as possible (i.e. the chops are as small as possible): Foremost, a more precise chop can lead to a more precise analysis result. The more precise chop may also reduce the costs of the main analysis, which may outweigh the increased

* This is an extended version of previous work (Giffhorn 2009)

D. Giffhorn
Adenauerring 20a
Karlsruhe Institute of Technology
Karlsruhe, Germany
Tel.: +49-721-608-7401
E-mail: dennis.giffhorn@kit.edu

```
1 void main()
2   int m = foo();
3   int n = foo();

4 int foo()
5   return 1;
```

```
1 int x,y;

2 thread_1()
3   int p = x;
4   y = p;

5 thread_2()
6   int a = y;
7   x = a;
```

Fig. 1 Examples for imprecise chopping

costs of the more precise chopping algorithm. For example, a path condition (Snelting et al. 2006) between two statements, s and t, is a necessary condition on the program state that a program run has to satisfy in order to reach t, when coming from s. The path condition is composed of all predicates influenced by s and influencing t, which in turn are determined by the chop from s to t. Thus, the more precise the chop, the smaller and more precise is the resulting path condition, and may also be evaluated faster.

A simple way to compute $chop(s, t)$ for s and t is collecting all statements influenced by s and all statements influencing t, and then intersecting those sets. However, such a computed chop may be *context-insensitive*, because different invocations of the same procedure are not distinguished. Consider the program on the left side in Fig. 1: Statement 3 is not influenced by statement 2, hence $chop(2, 3)$ should be empty. But statement 2 influences the statements $\{2, 4, 5\}$, because it calls `foo`, and statement 3 is influenced by the statements $\{3, 4, 5\}$, because it assigns the result of the procedure call to `n`, thus the intersection results in chop $\{4, 5\}$. Reps and Rosay (Reps and Rosay 1995) developed the first *context-sensitive* chopping algorithm for sequential interprocedural programs, which distinguishes different invocations of the same procedure. Their algorithm is the state of the art for chopping sequential programs. We abbreviate it with *RRC* throughout the paper.

The RRC is, however, rather complicated to implement, and its asymptotic running time is not linear to the size of the target program. We present a new chopping technique for sequential programs that is not entirely context-sensitive, but is easy to implement and very fast in practice, offering a genuine alternative for quick deployment. We evaluated the precision and runtime costs of this new technique together with several variants of the RRC on a set of 20 Java programs, providing one of the few published evaluations of sequential chopping algorithms.

Many complementary languages, like Java or C♯, have built-in support for concurrent execution. Applications that leverage chopping to analyze such languages need chopping algorithms suitable for concurrent programs. Unfortunately, the RRC cannot be applied here: Concurrent programs give rise to new kinds of dependences between program statements, which are not covered by that algorithm. We show how to extend Reps and Rosay's algorithm to compute context-sensitive chops in concurrent programs.

Concurrent programs bear a new kind of imprecision, so-called *time travels* (Krinke 2003 (ESEC/FSE)). Consider the program on the right side in Fig. 1, consisting of two concurrent threads that communicate via two shared variables, `x` and `y`. Clearly, $chop(7, 6)$ should be empty, because statement 7 is executed after statement 6 and therefore cannot influence it. But if the chop is computed using intersection, the result is $chop(7, 6) = \{3, 4\}$, because statement 7 influences the statements {3,4,7} and state-

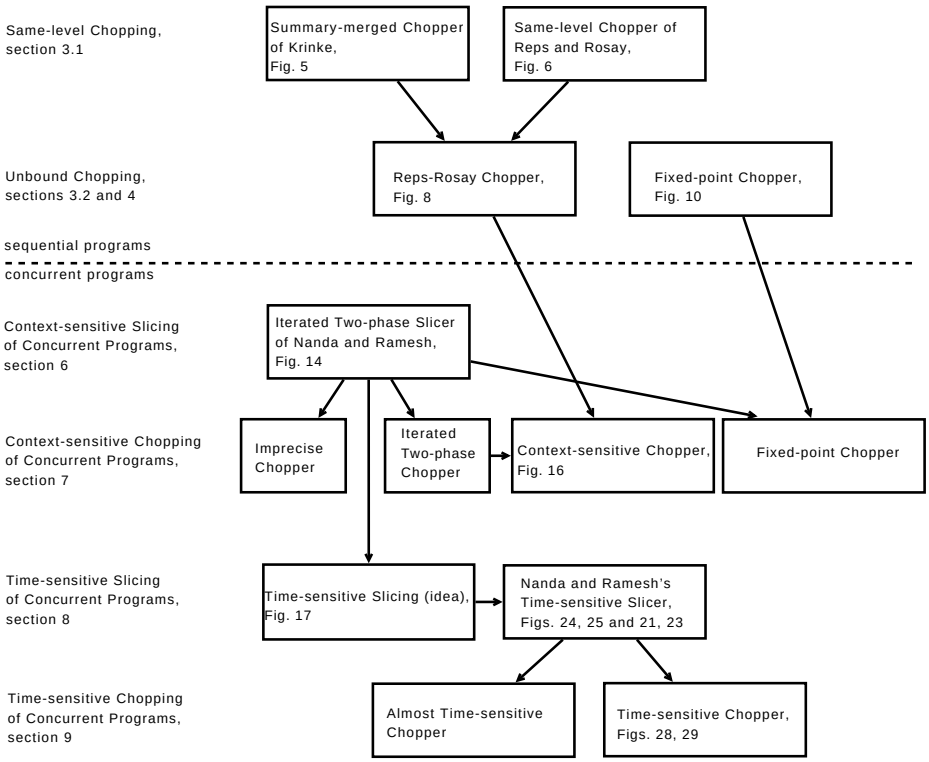

Fig. 2 Relationship between the algorithms presented in this article. The algorithm at the start of an arrow is a prerequisite for the algorithm at the end

ment 6 is influenced by statements {3,4,6}. We show how to avoid such time travels in a chop, resulting in *time-sensitive* chops.

Since time travel detection is expensive and difficult to implement, we present six chopping algorithms for concurrent programs. These algorithms offer different degrees of precision, from imprecise (but fast) over context-sensitive to context- and time-sensitive. We implemented these algorithms and evaluated their precision and runtime costs on a set of concurrent Java programs. Context-sensitive chopping reduced the chop sizes up to 25%, while moderately increasing execution times. Time-sensitive chopping strongly reduced the chop sizes – up to 73%, but at the expense of considerably increased execution times.

1.1 The roadmap

This article consists roughly of two parts: Sections 2 – 5 are concerned with chopping of sequential programs, sections 6 – 14 investigate chopping of concurrent programs. Since this article presents many existing and new algorithms, we want to depict their relationship in Fig. 2. It shows the most important presented algorithms for sequential programs and all presented algorithms for concurrent programs. In case the article contains pseudo code for an algorithm, the corresponding Figure is mentioned. Section

2 introduces *system dependence graphs* and *slicing*, another program analysis technique on which chopping is based. Section 3 introduces chopping of sequential programs and explains the Reps-Rosay chopper. It starts with introducing *same-level chopping*, where the statements of the chopping criterion have to be located in the same procedure. The Reps-Rosay chopper extends same-level chopping to *unbound chopping*, which permits arbitrary chopping criteria. In total, section 3 presents Krinke's *summary-merged chopper* (Krinke 2002), the same-level chopper of Reps and Rosay (Reps and Rosay 1995), and the unbound Reps-Rosay chopper. Section 4 presents our new, almost context-sensitive chopping algorithm for sequential programs, the *fixed-point chopper*. Section 5 concludes the part about chopping sequential programs by presenting one of the few published evaluations of sequential chopping algorithms.

Section 6 extends system dependence graphs and slicing to concurrent programs. For that purpose, it introduces the *iterated two-phase slicer* (I2P) of Nanda and Ramesh (Nanda and Ramesh 2006), a context-sensitive slicer for concurrent programs. Section 7 presents two simple chopping algorithms for concurrent programs, the *imprecise chopper* (IC) and the *iterated two-phase chopper* (I2PC), which are both extensions of the I2P slicer, and extends the fixed-point chopper of section 4 to concurrent programs. These three algorithms are not context-sensitive, but fast and easy to implement. The section proceeds with introducing our *context-sensitive chopper*, CSC, which employs the Reps-Rosay chopper and the I2PC to compute context-sensitive chops in concurrent programs. Section 8 introduces time travels as a source of imprecision in concurrent programs and explains the idea of time-sensitive slicing. Since time-sensitive chopping is based thereon, section 9 gives an in-depth description of Nanda and Ramesh's time-sensitive slicer (Nanda and Ramesh 2006), which currently seems to be the most practical. Eventually, section 10 explains our time-sensitive chopping algorithm, which is evaluated in section 11. Section 12 discusses several issues and future work, section 13 presents related work, and section 14 concludes.

2 Slicing Sequential Programs

Slicing is a program analysis technique that reveals all program parts that influence a given statement c, the *slicing criterion*. The result is the so-called *backward slice*. The dual, the *forward slice*, contains all program parts that are influenced by c. A simple but imprecise chop from s to t can be computed by intersecting the backward slice for t with the forward slice for s (Jackson and Rollins 1994).

Slices are often computed based on *system dependence graphs* (SDG) (Horwitz et al. 1990). A SDG $G = (N, E)$ for program p is a directed graph, where the nodes in N represent p's statements and predicates, and the edges in E represent dependences between them. The SDG is partitioned into *procedure dependence graphs* (PDG) that model the single procedures. In a PDG, a node n is *control dependent* on node m, if m's evaluation controls the execution of n (e.g. m guards a conditional structure). n is *data dependent* on m, if n may use a value computed at m. The PDGs are connected at *call sites*, consisting of a call node c that is connected with the entry node e of the called procedure through a *call edge* $c \rightarrow_{call} e$. Parameter passing and result returning, as well as side effects of the called procedure, are modeled via synthetic *parameter nodes* and *edges*. For every passed parameter there exists an *actual-in node* a_i and a *formal-in node* f_i that are connected via a *parameter-in edge* $a_i \rightarrow_{pi} f_i$. For every modified parameter and returned value there exists an *actual-out node* a_o and a *formal-out node*

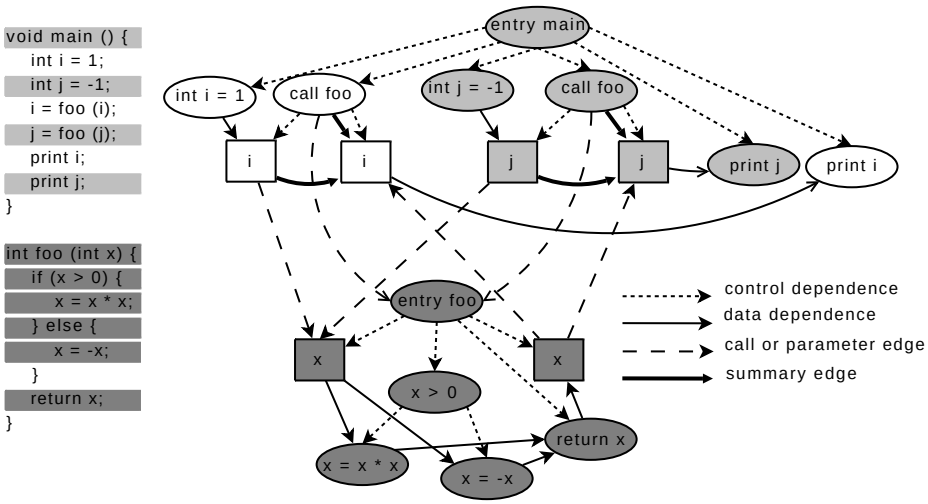

Fig. 3 A system dependence graph. The highlighted nodes are the context-sensitive slice for `print j`. The light gray nodes are visited in phase 1, the dark gray nodes in phase 2

f_o that are connected via a *parameter-out edge* $f_o \rightarrow_{po} a_o$. Formal-in and formal-out nodes are control dependent on entry node e, actual-in and actual-out nodes are control dependent on call node c. So-called *summary edges* between actual-in and actual-out nodes of one call site represent transitive flow from a parameter to a return value in the called procedure. For the purpose of chopping, this standard construction of summary edges is slightly extended: There also exists a summary edge from a call node to each actual-out node of that call site (Reps and Rosay 1995). Figure 3 shows an example SDG, parameter nodes are depicted by rectangles.

2.1 Context-sensitive slicing

A central issue of slicing is *context-sensitivity*. A path in a SDG is context-sensitive if it preserves the calling contexts of procedure calls, i.e. it returns from a procedure to that call site which called the procedure. As context-insensitive paths are infeasible, slicing algorithms – and traversal of SDGs in general – should only consider context-sensitive paths. Figure 3 shows that SDGs contain context-insensitive paths, in that example all paths entering procedure `foo` at call site `i = foo(i)` and leaving it later on towards call site `j = foo(j)`. Computing slices via simple graph reachability in SDGs thus results in context-insensitive slices. For example, the suchlike computed slice for statement `print j` would contain all statements besides `print i`, even though the first call of `foo` does not influence `print j`. Summary edges enable an efficient computation of context-sensitive backward slices in two phases (Horwitz et al. 1990): Phase 1 slices from the slicing criterion only ascending to calling procedures, where summary edges are used to bypass call sites. Phase 2 slices from all visited nodes only descending into called procedures. A two-phase slicer for forward slices works accordingly. This *two-phase* approach is the most established slicing technique for sequential programs. In Fig. 3, the context-sensitive backward slice for statement `print j` is highlighted gray.

The light gray shaded nodes in Fig. 3 are visited in phase 1, the darker gray shaded nodes are visited in phase 2.

In order to reason about context-sensitive chopping, we need a formal definition of context-sensitive paths in SDGs. Reps and Rosay introduced a definition based on a language of matching parentheses:

Definition 1 (Context-sensitive paths in SDGs (Reps and Rosay 1995)) For each call site c, label the outgoing call and parameter-in edges with a symbol $(^e_c$, where e is the entry of the called procedure, and the incoming parameter-out edges with a symbol $)^e_c$. Label all other edges with l.

A path from node m to node n in the SDG of a sequential program is context-sensitive, abbreviated with $m \rightarrow^*_{cs} n$, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal *realizable* by the following context-free grammar H:

$matched \rightarrow matched\ matched \mid (^e_c\ matched\ )^e_c \mid l \mid \epsilon$
$unbalanced\_right \rightarrow unbalanced\_right\ )^e_c\ matched \mid matched$
$unbalanced\_left \rightarrow unbalanced\_left\ (^e_c\ matched \mid matched$
$realizable \rightarrow unbalanced\_right \quad unbalanced\_left$

Nonterminal *matched* describes 'matched' paths: paths that start and end in the same procedure and contain only accomplished procedure calls. 'Unbalanced-right' paths are sequences of matched paths interrupted by unmatched procedure returns, i.e. they start in a procedure p and end in a procedure in which p was called[1]. 'Unbalanced-left' paths are sequences of matched paths interrupted by unmatched procedure calls. They start in a procedure p and end in a procedure that is called by p. A 'realizable' path is a concatenation of an unbalanced-right and an unbalanced-left path. It starts in a procedure p, leaves it towards a procedure q in which p was called, and ends in a procedure r called by q.

A slice for node s is context-sensitive if it contains only nodes lying on context-sensitive paths from or to s:

Definition 2 (Context-sensitive slice)
A context-sensitive backward slice for slicing criterion s in a SDG G consists of the set of nodes

$\{n \mid \exists\ n \rightarrow^*_{cs} s \text{ in } G\}$

A context-sensitive forward slice for s in G consists of the set of nodes

$\{n \mid \exists\ s \rightarrow^*_{cs} n \text{ in } G\}$

3 Chopping Sequential Programs

This section introduces existing chopping techniques for sequential programs. Concerning sequential programs, there exist two different kinds of chopping techniques, *same-level* and *unbound* chopping. Same-level chopping requires from the chopping criterion (s, t) that s and t stem from the same procedure and considers only matched paths from s to t, i.e. it never leaves the procedure towards one of its callers. Unbound

[1] Which in case of recursion may again be p.

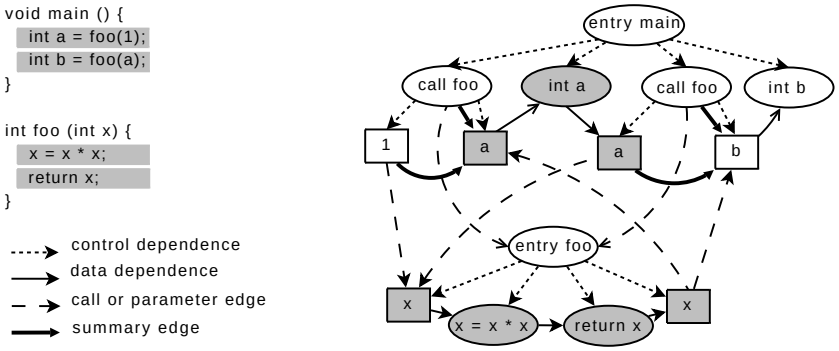

Fig. 4 Same-level vs unbound chopping: The unbound chop for (`return x`, `x=x*x`) consists of the gray shaded nodes, the same-level chop is empty

chopping permits arbitrary chopping criteria and takes all realizable paths from s to t into account. Figure 4 illustrates that difference: The unbound chop from `return x` to `x=x*x` consists of the gray shaded nodes, because the return value of the first call of `foo` is fed to the second. The same-level chop from `return x` to `x=x*x` is empty, because inside one invocation of `foo`, `return x` does not influence `x=x*x`.

In this work, we focus on unbound chopping. However, same-level chopping is employed to compute unbound chops, hence we will briefly explain its state of the art.

3.1 Same-level chopping

Jackson and Rollins suggested computing same-level chops through an iterated approach (Jackson and Rollins 1994). Exploiting summary edges, the approach first computes an intra-procedural chop[2] from s to t. Then, for every traversed summary edge $ai \rightarrow_{su} ao$, it computes another intra-procedural chop for criterion (fi, fo), where fi is the formal-in or entry node connected with the actual-in or call node ai, and fo is the formal-out node connected with the actual-out node ao. This step is repeated until no new summary edge is visited. The same-level chop consists of all nodes visited in the process.

Jackson and Rollins' approach computes a new chop for every pair of formal-in and -out nodes that have a summary edge between the corresponding actual-in and -out nodes included in the chop. Therefore, it may traverse the same edges multiple times – since two of such chops may overlap – and has an asymptotic running time bounded by $O(|E| * MaxFormalIns^2)$, where $MaxFormalIns$ is the maximum number of formal-in nodes in any procedure's PDG. Krinke developed an improved algorithm which relieves that redundancy and is significantly faster in practice (Krinke 2002): If two summary edges of one call site are included in the chop, one does not need to compute the two chops for the corresponding pairs of {formal-in, entry}/formal-out nodes separately. Instead, a single chop between the *set* of corresponding {formal-in, entry} nodes and the *set* of corresponding formal-out nodes exhibits the same precision and traverses a

[2] Intra-procedural chops are commonly computed through intersection of intra-procedural forward and backward slices.

Input: A chopping criterion (s, t).
Output: The same-level chop from s to t.

$W = \emptyset$ // *a worklist*
$M = \emptyset$ // *marks processed summary edges*

`let` C be the intra-procedural chop for (s, t)
`foreach` call site c in C
  // *collect all summary edges at c that lie in the chop*
  // *put them as one element into W*
  $W = W \cup \{\{(ai, ao) \mid ai, ao \in C, \exists$ summary edge $ai \rightarrow_{su} ao$ at $c\}\}$

`repeat`
  $W = W \setminus L$ // *remove one element from the worklist*

  // *build the chopping criterion (S, T) for L*
  $S = \emptyset$
  $T = \emptyset$
  `foreach` $(ai, ao) \in L$
    `let` fi be the formal-in or call node corresponding to ai
    `let` fo be the formal-out node corresponding to ao
    `if` $(fi, fo) \notin M$ // *tuple has not been marked yet*
      $M = M \cup \{(fi, fo)\}$ // *mark tuple as visited*
      $S = S \cup \{fi\}$
      $T = T \cup \{fo\}$

  // *compute the chop for (S, T) and update the worklist*
  `let` C' be the intra-procedural chop for (S, T)
  $C = C \cup C'$
  `foreach` call site c in C'
    $W = W \cup \{\{(ai, ao) \mid ai, ao \in C', \exists$ summary edge $ai \rightarrow_{su} ao$ at $c\}\}$

`until` $W = \emptyset$

`return` C

Fig. 5 SMC: Krinke's summary-merged chopper

smaller number of edges. Krinke's improved algorithm, called *summary-merged chopper* and depicted in Fig. 5, exploits that observation as follows: After computing the initial intra-procedural chop for s and t, all traversed summary edges of visited call sites are collected. Then, for every visited call site, a new chop is computed between the set of corresponding {formal-in, entry} nodes and the set of corresponding formal-out nodes. This procedure is repeated with the new resulting summary edges until there are no more new summary edges left. The resulting chop consists of all nodes visited in the process.

Though significantly faster in practice, the summary-merged chopper has still the same runtime complexity. A technique which is asymptotically faster ($O(|E| * MaxFormalIns)$) has been proposed by Reps and Rosay (Reps and Rosay 1995). We explain that technique on its pseudo code in Figure 6. Starting from the initial intra-procedural chop for (s, t), it computes for every summary edge $ai \rightarrow_{su} ao$ being part of the chop, the corresponding {formal-in, entry}/formal-out pair (fi, fo). Then it stores the forward slice for fi in map F (= forward) and the backward slice for fo in map B (= backward). It further stores for fo the set of actual-out nodes lying in the intra-

Input: A chopping criterion (s, t).
Output: The same-level chop from s to t.

$M = \emptyset$
$B = \emptyset$
$F = \emptyset$
$ActOut = \emptyset$
$W = \emptyset$ // *a worklist*

`let` C be the intra-procedural chop for (s, t)
// *collect all summary edges that lie in the chop*
$W = W \cup \{(m, n) \mid m, n \in C, \exists$ a summary edge $m \rightarrow_{su} n\}$

`repeat`
  $W = W \setminus (m, n)$ // *take next element*
  `let` fi be the formal-in or entry node that corresponds to m
  `let` fo be the formal-out node that corresponds to n

  `if` $(fi, fo) \notin M$
    $M = M \cup \{(fi, fo)\}$

    `if` $B(fo) = \emptyset$
      $B(fo) = backward\_slice(fo)$
      $ActOut(fo) = \{n \in backward\_slice(fo) \mid n$ *is an actual-out node*$\}$

    `if` $F(fi) = \emptyset$
      $F(fi) = forward\_slice(fi)$

    `foreach` $x \in B(fo)$
      `if` $x \in F(fi)$
        $C = C \cup x$
        $B(fo) = B(fo) \setminus \{x\}$
        `if` x *is an actual-out or call node*
          `foreach` summary edge $x \rightarrow_{su} y$ with $y \in ActOut(fo)$
            $W = W \cup \{(x, y)\}$

`until` $W = \emptyset$

`return` C

Fig. 6 Reps and Rosay's same-level chopper

procedural backward slice for *fo*. Then, every node x which lies in both the intra-procedural backward slice for *fo* and the intra-procedural forward slice for *fi* stored in the maps B and F is added to the chop, and is removed from the stored backward slice for *fo*. The removal guarantees that each node is touched at most once. If x is a formal-in or entry node and there is a summary edge $x \rightarrow_{su} y$ to an actual-out node y lying in the intra-procedural backward slice for *fo*, then this edge is added to the worklist. This way, the algorithm iteratively processes the procedures called within the chop.

3.2 Unbound chopping – the algorithm of Reps and Rosay

An unbound chop $chop(s, t)$ for criterion (s, t) takes all realizable paths from s to t into account. Intuitively, $chop(s, t)$ can be computed by intersecting the backward slice of

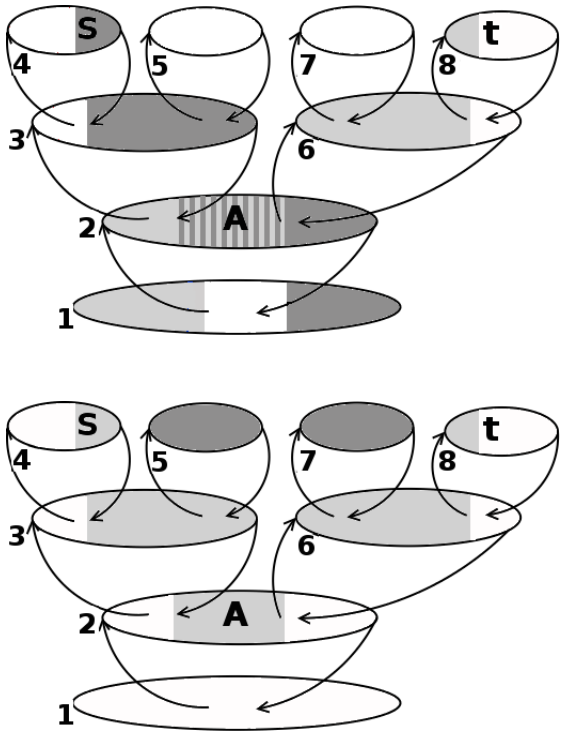

Fig. 7 Schematic overview of the Reps-Rosay chopper for chopping criterion (s, t). The upper part shows step 1, the lower part shows steps 2 and 3

t with the forward slice of s (Jackson and Rollins 1994). However, such a chop may be context-insensitive, even if the underlying slicers are context-sensitive. Consider the program on the left side in Fig. 1 as an example. Statement 3 is not influenced by statement 2, hence the chop for $(2, 3)$ should be empty. But the context-sensitive forward slice for statement 2 and the context-sensitive backward slice for statement 3 both contain statements 4 and 5, and their intersection results in chop $\{4, 5\}$.

Reps and Rosay (Reps and Rosay 1995) developed a sophisticated algorithm that chops programs context-sensitively. It exploits a well-formedness property of SDGs: all interprocedural effects are propagated via call sites. Figure 7 gives a schematic overview: The ovals symbolize procedures, ascending edges are procedure calls and descending edges are returns. The two graphs show how the Reps-Rosay algorithm proceeds in computing $chop(s, t)$. First, it determines the common callers of s and t, i.e. the procedures which (indirectly) call both the procedures of s and t. This is achieved by computing a forward slice for s and a backward slice for t which only ascend to calling procedures. Intersecting them reveals the common callers and the set of nodes A in these procedures that belong to the chop. This is shown in the upper graph. In the next step, the RRC collects the nodes in the procedures leading from A to s or t that belong to the chop. For the procedures leading to s, this is done by intersecting the forward slice of s and the backward slice of A, where the forward slice only ascends to calling procedures and the backward slice only descends into called

Input: A chopping criterion (s,t).
Output: The unbound chop from s to t.

`let` *slc* be a function that computes a same-level chop for a set of chopping criteria.
`let` f_1 be a forward two-phase slicer that only commits phase 1
`let` b_1 be a backward two-phase slicer that only commits phase 1
`let` f_2 be a forward two-phase slicer that only commits phase 2
`let` b_2 be a backward two-phase slicer that only commits phase 2

/ Step 1: collect the nodes in the common callers of s and t */*
$A = f_1(\{s\}) \cap b_1(\{t\})$

/ Step 2: collect the nodes in the procedures leading from A to s and t */*
$C_1 = f_1(\{s\}) \cap b_2(A)$
$C_2 = f_2(A) \cap b_1(\{t\}$
$Chop = C_1 \cup C_2$

/ Step 3: collect the nodes in procedures called underway */*
// Collect all traversed summary edges: distinguishing the branches C_1 *and* C_2
// ensures that the edge has actually been traversed, otherwise ai could have
// been visited in C_1 *and ao in* C_2*, such that the edge has not been traversed.*
$S = \{(ai, ao) \mid \exists\ ai \rightarrow_{su} ao \wedge (ai, ao \in C_1 \vee ai, ao \in C_2)\}$

// build the chopping criteria for the same-level chopper
$W = \{(fi, fo) \mid \exists (ai, ao) \in S : (fi, fo)$ *is* $\{$*formal-in, entry*$\}$*/formal-out pair of* $(ai, ao)\}$
$Chop = Chop\ \cup\ slc(W)$

`return` *Chop*

Fig. 8 RRC: The Reps-Rosay chopper

procedures. For the procedures leading to t this works analogously. The result is shown in the lower graph as light gray highlighted areas. This step ignores the procedures called underway by the visited nodes – in our example procedures 5 and 7. In a third step, these omitted procedures are analyzed via same-level chopping, starting from the summary edges traversed in step 2. The resulting chop consists of the nodes visited in steps 2 and 3. By using A, s and t as a barrier in the second step and employing same-level chopping in the third step, the algorithm maintains context-sensitivity. According to Reps and Rosay, RRC's asymptotic running time is in $\mathcal{O}(|E| * MaxFormalIns)$, if their same-level chopper of Fig. 6 is used for step 3. The authors show that without that optimization its runtime complexity is in $\mathcal{O}(|E| * MaxFormalIns^2)$, because then it basically employs Jackson and Rollins' iterative same-level chopper for step 3. The same holds if Krinke's SMC is used instead, as SMC has the same runtime complexity. Figure 8 shows RRC's pseudo code; function *slc*, which computes the same-level chops, is realized by extending the algorithm in Fig. 6 to handle a set of chopping criteria.

3.2.1 The Reps-Rosay Chopper for Sets of Nodes

Though not explicitly stated by Reps and Rosay (Reps and Rosay 1995), the RRC is also able to compute context-sensitive chops for chopping criteria consisting of sets of nodes S and T, the result being the union of the chops for every pair $(s, t) \in S \times T$. For that purpose, the underlying slicers in the RRC are extended to compute slices for sets

of nodes. We need that extension for computing context-sensitive chops in concurrent programs, and thus prove its correctness in this subsection.

Following grammar H in definition 1, let $m \rightarrow^*_{unbr} n$ denote a SDG path which is *unbalanced-right*. Similarly, let $m \rightarrow^*_{unbl} n$ denote an *unbalanced-left* path. Reps and Rosay define the following operations to compute context-sensitive chops in SDGs (Reps and Rosay 1995):

- $f_{unbr}(S) = \{n \mid \exists s \in S : s \rightarrow^*_{unbr} n\}$ (conforms to f_1 in Fig. 8)
- $f_{unbl}(S) = \{n \mid \exists s \in S : s \rightarrow^*_{unbl} n\}$ (conforms to f_2 in Fig. 8)
- $b_{unbl}(T) = \{n \mid \exists t \in T : n \rightarrow^*_{unbl} t\}$ (conforms to b_1 in Fig. 8)
- $b_{unbr}(T) = \{n \mid \exists t \in T : n \rightarrow^*_{unbr} t\}$ (conforms to b_2 in Fig. 8)

In other words, f_{unbr} is the set of nodes lying on unbalanced-right paths starting at a node $s \in S$, f_{unbl} is the set of nodes lying on unbalanced-left paths starting at a node $s \in S$, b_{unbr} is the set of nodes lying on unbalanced-right paths leading to a node $t \in T$ and b_{unbl} is the set of nodes lying on unbalanced-left paths leading to a node $t \in T$. The operations f_{unbr} and b_{unbl} can be implemented by forward and backward two-phase slicers committing only phase 1, i.e. only ascending to calling procedures, f_{unbl} and b_{unbr} can be implemented by forward and backward two-phase slicers committing only phase 2, i.e. only descending into called procedures (Reps and Rosay 1995).

The RRC further needs a function $SLC(e)$, which takes a summary edge $e = ai \rightarrow_{su} ao$ and computes a same-level chop for the corresponding {formal-in, entry}/formal-out pair. However, its concrete functionality is irrelevant for this proof. As explained in greater detail further above, the RRC performs the following 3 steps to compute the chop $RRC(s,t)$ (Reps and Rosay 1995):

1. $A = f_{unbr}(\{s\}) \cap b_{unbl}(\{t\})$,
2. $Chop = (f_{unbr}(\{s\}) \cap b_{unbr}(A)) \cup (f_{unbl}(A) \cap b_{unbl}(\{t\}))$,
3. For every summary edge e on unbalanced-right paths from s to A or unbalanced-left paths from A to t: $Chop = Chop \cup SLC(e)$.

We claim that the algorithm $RRC(S,T)$ for sets of nodes S and T, consisting of the steps

1. $A = f_{unbr}(S) \cap b_{unbl}(T)$,
2. $Chop = (f_{unbr}(S) \cap b_{unbr}(A)) \cup (f_{unbl}(A) \cap b_{unbl}(T))$,
3. For every summary edge e on unbalanced-right paths from S to A or unbalanced-left paths from A to T: $Chop = Chop \cup SLC(e)$,

computes the same result as the union of the chops $RRC(s,t)$ for all possible pairs of $s \in S, t \in T$.

Lemma 1 $RRC(S,T) = \bigcup_{\substack{s \in S \\ t \in T}} RRC(s,t)$

Proof

'$\supseteq$' Every node $n \in RRC(s,t)$ for $s \in S, t \in T$ is also in $RRC(S,T)$. This follows directly from the definitions of f_{unbr}, f_{unbl}, b_{unbr} and b_{unbl}.

'$\subseteq$' We have to show that for every node $n \in RRC(S,T)$ there exist $s \in S, t \in T$ such that $n \in RRC(s,t)$. We distinguish two cases: n is added to the chop either in step 2 or in step 3.

- n is added in step 2:
  There must exist $s \in S, t \in T, w \in A$ such that either $s \rightarrow^*_{unbr} n \rightarrow^*_{unbr} w \rightarrow^*_{unbl} t$ or $s \rightarrow^*_{unbr} w \rightarrow^*_{unbl} n \rightarrow^*_{unbl} t$ holds. Thus $n \in RRC(s,t)$.
- n is added in step 3:
  This means that n is added to the chop due to the same level chop $SLC(e)$ for a summary edge $e = e_s \rightarrow_{su} e_t$. Thus, there must exist $s \in S, t \in T, w \in A$, such that either $s \rightarrow^*_{unbr} e_s \rightarrow_{su} e_t \rightarrow^*_{unbr} w \rightarrow^*_{unbl} t$ or $s \rightarrow^*_{unbr} w \rightarrow^*_{unbl} e_s \rightarrow_{su} e_t \rightarrow^*_{unbl} t$ holds. Therefore, e is also visited by the chop $RRC(s,t)$ in step 2, which means that $SLC(e)$ is added to that chop. Hence $n \in RRC(s,t)$.

□

Note that this extended algorithm retains the same asymptotic running time, because all employed operations remain the same.

4 An Alternative Chopping Technique for Sequential Programs

In this section, we present a new chopping technique for sequential programs, which is not entirely context-sensitive, but almost as precise in practice, very fast and easy to implement.

Although the RRC is known for 15 years, intersection-based chopping is often considered a convenient alternative for a quick deployment. A well-known optimization computes a forward slice for s and then a backward slice for t restricted to the subgraph traversed by the forward slicer. The resulting backward slice is already the chop, eliminating the intersection. Its runtime complexity is in $\mathcal{O}(|E|)$, like that of the underlying two-phase slicer. Moreover, it already removes some spare nodes from the chop. For example, it detects that $chop(2,3)$ in the program on the left side in Fig. 1 is empty, as statement 3 is not in the forward slice for statement 2.

During our work we made the following observation: Computing another forward slice on the result of the above algorithm may result in an even more precise chop, for which Fig. 9 provides an example. The context-sensitive chop for node pair (2, 8) (statements `int s = 0` and `int t = x` in procedure `main`) consists of the dark gray nodes: Variable `s` is passed as a parameter to `foo` and flows via the return statement into variable `t`. However, if we employ the algorithm above, the resulting chop also contains nodes 16 and 17: The forward slice for node 2 consists of the nodes {2,4,15,18,19,6,7,8,11,16,17,12,13}, the backward slice for node 8, restricted to these nodes, is the set {8,7,6,19,18,17,16,15,4,2}. Nodes 17 and 16 were visited by the forward slicer in the context of the second invocation of `foo`, which does not influence node 8. But that information is not available anymore in the set representing the forward slice, thus the backward slicer traverses from node 18 to the nodes 17 and 16. If we compute a second forward slice for 2 restricted to that backward slice, we are able to remove these spurious nodes. The forward slice visits the nodes {2,4,15,18,6,7,8},

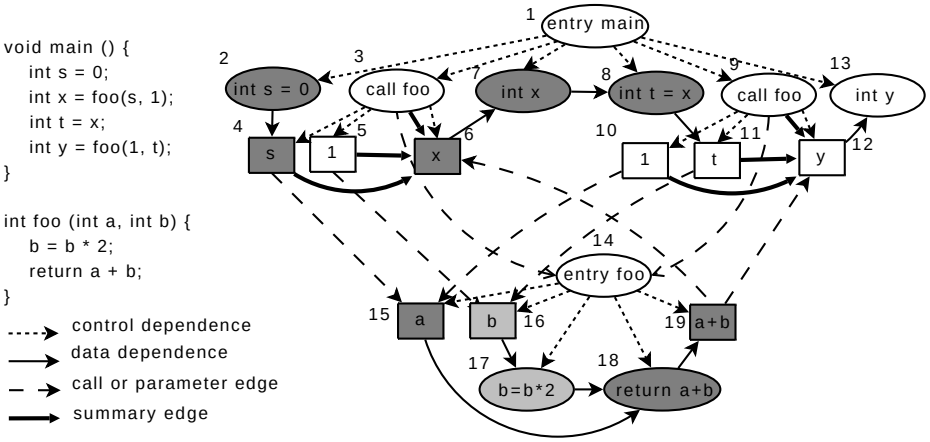

Fig. 9 Chops for chopping criterion $(2, 8)$. The highlighted nodes denote the chop determined by computing the backward slice for 8 on the forward slice for 2. The dark gray nodes denote the context-sensitive chop

---

Input: A chopping criterion (s, t).
Output: The chop from s to t.

`let` $f(c, M)$ be a two-phase forward slicer which only visits the nodes in set M
`let` $b(c, M)$ be a two-phase backward slicer which only visits the nodes in set M

/ Compute the initial chop. */*
$F = f(s, N)$, *// N be the set of all nodes in the SDG*
$Chop = b(t, F)$
`changed = true`

/ Iterate until reaching a fixed-point. */*
`repeat`
  $Tmp = f(s, Chop)$
  $Tmp = b(t, Tmp)$

  // if we have reached a fixed-point, set changed to false
  `if` $(Tmp == Chop)$
    `changed = false`

  $Chop = Tmp$

`until !changed`

`return` *Chop*

Fig. 10 Fixed-point chopping: Computing almost context-sensitive chops

which is in this example the context-sensitive result. This will not always be the case, but repeating that optimization may gradually remove more spurious nodes, resulting in a fixed-point style algorithm shown in Fig. 10. However, fixed-point chopping is not generally context-sensitive. Consider the example in Fig. 11: The context-sensitive chop for (`int s = b`, `int t = a`) consists of the dark gray nodes, but fixed-point chopping additionally includes statement `int b = t`. Although this new algorithm has an

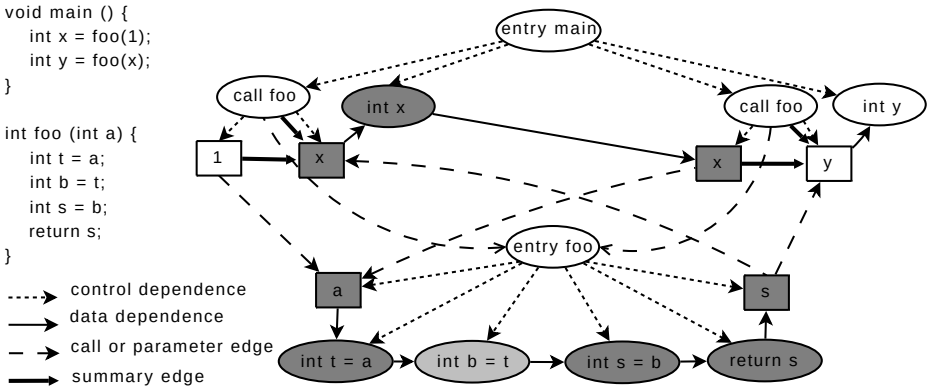

Fig. 11 Fixed-point chopping is not context-sensitive in general

asymptotic runtime complexity of $\mathcal{O}(|E| * |N|)$, our evaluation presented in the next section indicates that the fixed-point is reached very fast, usually after the second iteration of the loop.

5 Evaluation of Sequential Chopping Algorithms

We have implemented the presented algorithms for unbound chopping in Java and evaluated them on a set of 20 Java programs shown in Table 1. The programs in the upper part are small to medium-sized programs taken from the Bandera[3] benchmark, and solve a certain task in a concurrent manner (e.g. LaplaceGrid solves Laplace's equation over a rectangular grid). The other programs are real JavaME[4] applications taken from the SourceForge repository[5]. All these programs contain threads and were also used to evaluate our chopping algorithms for concurrent programs. The sequential chopping algorithms simply treat thread invocations as procedure calls and ignore the other concurrency-related dependences. The implemented algorithms work on SDGs computed by Graf's and Hammer's data flow analysis for Java programs (Graf 2009; Hammer and Snelting 2004). For the evaluation we used a 2.2Ghz Dual-Core AMD workstation with 32GB of memory running Ubuntu 8.04 (Linux version 2.6.24) and Java 1.6.0.

In our evaluation, we measured the average chop sizes and runtime performance of our new proposed algorithm *FC*, the simple intersection-based chopper *IC*, and the RRC. For a more in-depth investigation, we further employed two variants of FC, the algorithms *Opt-0*, which omits the while-loop of FC, and *Opt-1*, which iterates the while-loop only once. Additionally, we combined the RRC with the three same-level chopping algorithms introduced in section 3.1 and examined their runtime differences. In summary, we evaluated the following algorithms:

[3] http://www.cis.ksu.edu/santos
[4] The Java Mobile Edition for mobile devices.
[5] http://sourceforge.net/

Table 1 Statistics of our benchmark programs

Name	Nodes	Edges	Procedures
Example	1687	6148	41
ProdCons	2217	8775	39
DiningPhils	2973	11331	43
AlarmClock	4085	13842	74
LaplaceGrid	10022	100730	95
SharedQueue	17998	139480	122
Daisy	45603	458502	555
DayTime	62594	644400	734
KnockKnock	34667	288736	493
DiskSched	4378	44546	131
EnvDriver	19129	184149	169
Logger	9576	50800	225
Maza	10590	60221	261
Barcode	11025	67849	229
Guitar	13459	87724	307
J2MESafe	17851	125221	309
Podcast	25366	162102	504
CellSafe	40709	845931	524
GoldenSMS	26445	212832	414
HyperM	17847	93068	277

- IC, intersects the forward slice for s with the backward slice for t.
- FC, the fixed-point chopper of Fig. 10.
- Opt-0, computes a backward slice for t on the forward slice for s.
- Opt-1, executes the while-loop of FC only once.
- RRC, the Reps-Rosay Chopper.
- RRC-Unopt, uses Jackson and Rollins' iterative approach to compute the same-level chops in the last step of RRC.
- RRC-SMC, uses Krinke's Summary-Merged Chopper to compute the same-level chops.

For each benchmark program, we randomly determined 10,000 chopping criteria consisting of one source and one target node. The evaluation results are presented in the following subsections.

5.1 Precision

Table 2 shows the average chop size for each chopping algorithm and benchmark program. Since all evaluated RRC variants compute the same chops, they are subsumed by column 'RRC'. The measured values demonstrate that context-sensitive chopping is able to reduce chop sizes significantly: The intersection-based chops (IC) are on average 14.6% bigger than the RRC chops, in the worst case even about 48% (for KnockKnock).

The simple, well-known optimization applied in Opt-0 turns out to be very effective: The Opt-0 chops are on average only 3.2% bigger than the RRC chops. For most programs, the difference lies between 0% and 5%, the only outlier being Maza for which the Opt-0 chop is 32% bigger.

Fixed-point chopping reduces imprecision even more: The FC chops are on average only 0.7% bigger than the RRC chops. For 9 out of 20 programs, the difference is even

Table 2 Average size per chop (number of nodes). Column 'RRC' subsumes our three RRC variants, which always computed the same chops

Name	IC	Opt-0	Opt-1	FC	RRC
Example	54.88	45.36	44.86	44.86	44.82
ProdCons	7.70	7.47	7.40	7.40	7.40
DiningPhils	13.13	12.59	12.49	12.49	12.48
AlarmClock	39.19	38.44	38.10	38.10	38.09
LaplaceGrid	34.95	31.56	31.40	31.40	31.31
SharedQueue	1788.93	1729.17	1721.70	1721.70	1715.52
Daisy	12775.11	11473.73	11445.54	11445.54	11445.53
DayTime	17851.99	16194.70	16041.97	16041.95	16035.79
KnockKnock	1657.52	1184.87	1150.33	1150.33	1129.01
DiskSched	295.77	256.82	254.80	254.80	254.23
EnvDriver	2801.88	2725.90	2725.29	2725.29	2725.26
Logger	150.74	147.40	147.12	147.12	147.12
Maza	371.08	360.98	290.51	290.51	272.90
Barcode	279.78	219.04	213.25	213.25	210.33
Guitar	797.59	749.62	742.93	742.93	741.72
J2MESafe	2362.61	2190.82	2154.21	2154.21	2145.40
Podcast	2518.34	2073.05	2014.39	2014.39	2009.20
CellSafe	13889.89	13442.40	13259.84	13259.84	13150.46
GoldenSMS	1593.33	1381.36	1343.87	1343.87	1333.18
HyperM	495.44	448.93	442.21	442.21	441.96

below 0.1%. Notably, several of our larger programs are amongst these 9 programs (e.g. Daisy and DayTime), so FC's effectiveness is not restricted to small and simple programs. Again, the outlier is Maza, for which the FC chop is 6.5% bigger than the RRC chop.

The differences between Opt-1 and FC are marginally small – in Table 2 they are only visible for DayTime. In fact, only for 17 out of our 200,000 chopping criteria (16 in DayTime and 1 in J2MESafe) Opt-1 and FC computed different results. For these 17 chops, FC needed three loop iterations, two iterations removing spurious nodes and a last one to detect the fixed-point. For the other chops it iterated the loop twice, thus basically performing Opt-1 plus an additional loop iteration detecting the fixed-point.

5.2 Runtime

Table 3 shows the average time in milliseconds needed for one chop. Looking at the measured values for our three RRC variants reveals that the 3rd step of RRC – the computation of the same-level chops – is the critical part concerning runtime performance. The naïve iterative approach taken in RRC-Unopt did not scale well for our larger programs – for Daytime and EnvDriver it was more than 100 times slower than RRC-SMC. Surprisingly, RRC-SMC was the most performant variant, even though the original RRC is asymptotically faster. For 13 out of 20 programs, particularly for the larger programs, it was significantly faster - in the best case (for EnvDriver) even about 8 times.

Given its imprecision, algorithm IC performed rather poorly. Especially for the JavaME programs in our benchmark its runtime performance was often slower than that of RRC-SMC. Opt-0 was by far the fastest algorithm, Opt-1 was always faster than the fastest RRC version at any one time and often even faster than IC. FC was

Table 3 Average time per chop (in milliseconds)

Name	IC	Opt-0	Opt-1	FC	RRC		
					Unopt.	SMC	Orig.
Example	.7	.3	.3	.4	1.1	.9	1.0
ProdCons	.4	.1	.1	.1	.4	.4	.3
DiningPhils	.6	.2	.2	.2	.5	.4	.4
AlarmClock	1.2	.4	.4	.4	.8	.7	.6
LaplaceGrid	2.7	1.0	1.0	1.0	2.6	2.4	2.4
SharedQueue	41.0	28.7	44.1	49.2	26683.4	283.6	1059.0
Daisy	222.2	165.7	282.2	349.2	66013.4	1680.5	2230.6
DayTime	267.1	206.0	359.5	515.4	92964.8	926.3	3913.1
KnockKnock	50.8	29.6	38.9	45.7	251.8	77.3	58.9
DiskSched	8.0	5.8	7.1	9.1	24.2	18.4	10.6
EnvDriver	56.3	42.8	69.7	80.2	12049.4	108.4	846.1
Logger	6.0	3.1	3.8	3.8	10.0	4.4	5.3
Maza	10.5	6.2	7.9	9.6	32.7	9.0	11.4
Barcode	9.8	5.2	6.3	6.8	17.5	5.7	7.6
Guitar	21.7	12.8	17.4	18.8	114.3	18.4	26.4
J2MESafe	41.0	27.1	42.4	56.8	1557.1	89.1	156.1
Podcast	49.2	30.3	44.7	54.9	912.8	46.8	103.3
CellSafe	325.4	270.5	498.7	727.7	50741.9	1093.1	2625.3
GoldenSMS	47.9	27.5	38.9	49.7	1195.6	68.7	136.9
HyperM	17.9	9.6	12.1	14.1	96.7	11.8	18.4

in most cases the slowest amongst the imprecise algorithms, but still competitive to the context-sensitive algorithms. For the JavaME programs, its runtime behavior was similar to that of RRC-SMC, for the other programs it was always considerably faster.

5.3 Study summary

In our opinion, naïve intersection-based chopping as done in algorithm IC turns out to be impractical: In view of its imprecision it exhibits a poor runtime performance. It also has no advantage in being easy to implement, because Opt-0, Opt-1 and FC have a similar implementation effort – one basically needs context-sensitive backward and forward slicers. Algorithms FC and Opt-1 emerge as a genuine alternative to RRC: FC's is in many cases almost as precise as RRC, its runtime is often faster and it is much easier to implement. A very interesting algorithm is Opt-1: Its computed chops were in almost all cases identical to the FC chops and it is noticeably faster than FC.

Concerning context-sensitive chopping, our evaluation shows that the unoptimized version of RRC is not practical; an application should always employ one of the optimized versions. Surprisingly, RRC-SMC was often significantly faster than the original RRC and thus seems to be the most practical variant for middle-sized programs. However, the original RRC is asymptotically faster than RRC-SMC (cf. section 3.2), thus it might be that it is faster for larger programs.

All these results, in particular the comparison between RRC and RRC-SMC, should be verified on a benchmark of larger programs. As our employed SDG generator currently only scales for programs with about 50,000 LOC, this remains future work.

6 Slicing Concurrent Programs

The remaining paper concerns with chopping techniques for concurrent programs. We focus on Java's concurrency model based on threads and shared-memory communication. For differing concurrency mechanisms, the algorithms may have to be adjusted accordingly. This section introduces the kinds of dependences arising from concurrently executing threads, and explains how SDGs can be extended to include these dependences. It also explains context-sensitive slicing of concurrent programs, which is the foundation of our chopping algorithms.

In Java threads are special objects of the `Thread` class, whose behavior is described by the statements in their `run()` procedure. In order to create a thread, a user has to instantiate such a `Thread` object and to call a special procedure `start()`. `start()` spawns a new thread, which executes the `run()` procedure of this thread object. A call of its `join()` procedure terminates the thread. However, joining threads is not mandatory, hence threads may in principle run infinitely. All threads share a single heap for storage of objects and their only interactions consist of (monitor-style) synchronization and communication via shared variables. Threads in Java are totally dynamic. A thread object can be created like any other object, and a subsequent invocation of its `start()` method will create a new thread of execution in the operating system executing its `run()` method. All this may happen in loops or recursion, so there is no static bound on the number of threads. Static detection of the number of threads running in a program execution is generally undecidable.

Due to shared-memory communication, concurrent programs exhibit a special kind of data dependence called *interference dependence* (Krinke 1998): A statement n is interference dependent on statement m, represented in the SDG by an *interference edge* $m \rightarrow_{id} n$, if n may use a value computed at m, and m and n may execute concurrently. Other than standard data dependence, interference dependence ignores the issue of *killing definitions*, i.e. it does not require that there must exist a thread interleaving such that the value computed at m in fact reaches n. Müller-Olm and Seidl have shown that the computation of these killing definitions is undecidable for concurrent programs with procedures (Müller-Olm and Seidl 2001). Thread invocation is modeled similar to procedure calls via *fork sites*, where shared variables are passed as parameters. *Fork* and *fork-in edges* are defined in analogy to call and parameter-in edges. An equivalence to parameter-out edges is not needed, because changes in parameters (the shared variables) are propagated immediately via interference edges. We currently do not model join points of threads, because in many languages like Java or C♯ this would require must-aliasing between the target objects of fork and join: Threads are conservatively assumed to run until the last thread terminates. We call such extended SDGs *concurrent system dependence graphs* (cSDG) (Giffhorn and Hammer 2009). Figure 12 shows an example cSDG. Concurrency also causes several kinds of synchronization-related dependences (Chen et al. 2000; Hatcliff et al. 1999), which are currently excluded by our cSDG generator. Once inserted, synchronization-related dependences can be treated by slicing and chopping algorithms like interference dependences. We will often subsume all concurrency-related edges in cSDGs under the term *concurrency edge*.

The cSDG structure has to consider that threads in Java and likewise languages are created dynamically, such that their concrete number is generally not decidable by a static analysis. Even an upper bound for the number of threads is often not estimable, because threads may be created inside loops or recursion. Thus it is not possible to

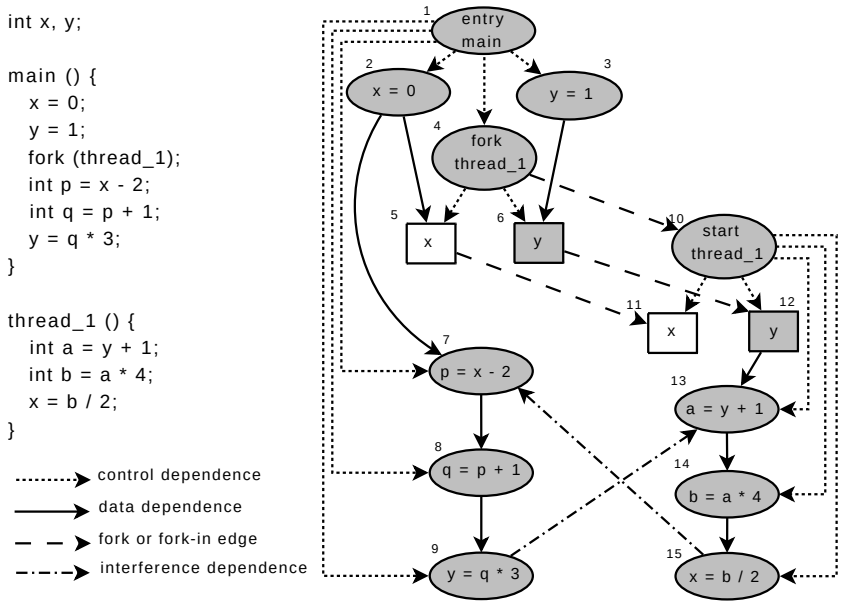

Fig. 12 An example cSDG. The context-sensitive slice for node 14 is highlighted gray

model each potential thread as a separate sub-graph in the cSDG. Instead, threads are modeled similar to procedures: The cSDG contains one sub-graph for each thread class of the program, which is connected with each fork site that invokes that thread. However, a sound interference dependence computation needs a conservative estimation of the number of invocations of each thread class, because there might be interferences between threads of the same thread class. For that purpose, we employ a modification of Ruf's *thread allocation analysis* for Java (Ruf 2000). Our computation of interference dependences and cSDGs is described in detail in Hammer's PhD thesis (Hammer 2009). Several authors provide SDG extensions for concurrent programs similar to the cSDG (Hatcliff et al. 1999; Krinke 2003 (PhD thesis); Nanda and Ramesh 2006; Zhao 1999).

In practice, cSDGs can be constructed in two steps: First, a standard SDG for each thread class in isolation is computed. Note that this can be done with a suitable SDG generator for sequential programs, because data and control dependences remain thread-local and do not have to incorporate inter-thread effects that may arise through thread interleaving. These inter-thread effects are captured by the interference dependences computed in the second step. The second step determines the fork sites and interference edges and uses them to connect the SDGs to a cSDG. For that purpose it employs the thread allocation analysis mentioned above and a *model of concurrency*.

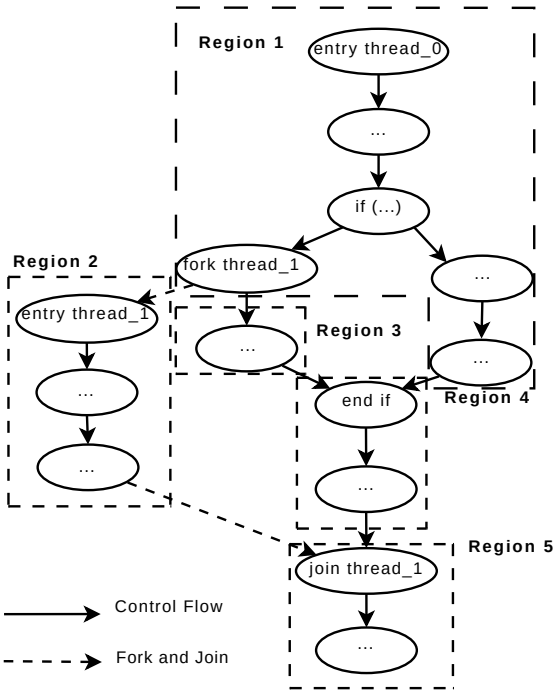

Fig. 13 Thread regions of a threaded program

6.1 Model of Concurrency

Analyses of concurrent programs require a model of concurrency, which identifies the program parts that execute concurrently. A suitable program representation for it are control flow graphs.

A *control flow graph* (CFG) $C = (N, E, s, e)$ for a procedure p is a labeled directed graph, where the nodes in N represent p's statements and predicates. E is a set of edges that represent control flow: Two nodes are connected by an edge if they can be executed back-to-back. s is the *entry node* which has no incoming edge and all nodes in N are reachable from s, e is the *exit node*, which has no outgoing edge and is reachable from every node in N. Control flow graphs for procedural programs, so-called *interprocedural control flow graphs* (ICFG), consist of the CFGs of the single procedures, which are connected via *call* and *return* edges. If a node c_p calls procedure p, then there exists a call edge from c_p to p's entry node. There exists a return edge from p's exit to the direct intra-procedural successor of c'_p, which is termed a *return node*. The tuple (c_p, c'_p) is a *call site* of p. An ICFG has a distinguished procedure *main*, such that every node is reachable from *main*'s entry node, and every node reaches *main*'s exit node. For concurrent programs, ICFGs are extended to *threaded control flow graphs* (TCFG), by connecting the ICFGs of the single threads via *fork* and *join* edges. The graph in Fig. 13 shows an example TCFG.

Concerning slicing and chopping, it is sound to assume too much concurrency, as this may only result in spurious interference edges – assuming too few concurrency in

turn may omit valid interference edges and cause incorrect slices and chops. Therefore, a simple model of concurrency could be to treat all threads as running entirely in parallel. We employ the more elaborate model of Nanda and Ramesh (Nanda and Ramesh 2006), which determines concurrency on the level of fork and join points of threads and is suitable for Java. It splits threads at fork and join points into *thread regions* and determines concurrency on the level of these regions. In summary, a thread region starts after a fork, at a join or at a node where two distinct thread regions meet. A thread region ends where another one begins or at the end of a thread. Concurrent execution of thread regions is determined through the following rule: Two thread regions q and r execute in parallel, if

- q and r belong to different threads,
- there exists a fork node f that reaches the start node of q through a path ϕ_q and the start node of r through a path ϕ_r, such that both paths leave f via different edges and at least one of these paths starts with a fork edge,
- and neither q's or r's start node is dominated by the join node of the other region's thread.

Figure 13 shows the thread regions for an example program whose main thread forks and joins another thread. It can be determined that only regions 2 and 3 as well as 2 and 4 may execute concurrently. This approach could be extended to a full-fledged may-happen-in-parallel (MHP) analysis (Naumovich et al. 1999) including synchronization; however, to date no scalable implementation for full Java has been reported.

We described the impact of concurrency models on slicing in previous work (Giffhorn and Hammer 2009); after introducing slicing of concurrent programs, we will summarize that impact in section 8.2. A more elaborate comparison of concurrency models including cobegin-coend parallelism as in Ada and rendezvous-style synchronization has been published by Chen et al. (Chen et al. 2000).

6.2 Context-sensitive slicing of concurrent programs

The two-phase slicing algorithm for sequential programs cannot be used for slicing cSDGs, because summary edges do not capture interprocedural effects of interference dependences (Nanda and Ramesh 2006). Since these dependences may cross procedure borders arbitrarily and thus violate the well-formedness property of SDGs of propagating all inter-procedural effects through call sites, their effects cannot be summarized by conventional summary edges. Fortunately, a simple extension of the two-phase slicer enables slicing of cSDGs: The two-phase slicer is surrounded by an outer loop, which iterates over a set S of nodes and calls the two-phase slicer for every $s \in S$. Initially, S contains only the slicing criterion. If the two-phase slicer encounters an interference, fork or fork-in edge, it does not traverse the edge but inserts the adjacent node into S. The resulting slice consists of the nodes visited in all iterations of the two-phase slicer. This *iterated two-phase slicer* (I2P slicer) was first described by Nanda and Ramesh (Nanda and Ramesh 2006), and can be implemented to yield context-sensitive slices in $\mathcal{O}(|E|)$: A node that has already been visited in phase 1 during a previous two-phase slice has not to be visited again, because its own slice has already been covered by that two-phase slice. This is not the case for a node that has only been visited in phase 2 yet, because phase 2 omits parameter-in and call edges. It has to be visited again if reached in phase 1 of another two-phase slice or via a concurrency

Input: The cSDG G, a slicing criterion s.
Output: The slice S for s.

```
W = {s}              // a worklist
M = {s ↦ true}       // a map for marking the contents of W
                     // (true represents phase 1, false phase 2)

repeat
   W = W \ {n}       // remove next node n from W

   foreach m →_e n   // handle all incoming edges of n
      // If m wasn't visited yet or it was visited in phase 2 and we are in phase 1 or
      // intend to traverse a concurrency edge
      if m ∉ dom M ∨ (¬M(m) ∧ (M(n) ∨ e ∈ {id, fork, fork_in}))
         // if we are in phase 1 or if e is not a call or param-in edge, add m to W
         if M(n) ∨ e ∉ {pi, c}
            W = W ∪ {m}

            /* Now determine how to mark m: */

            // If we are in phase 1 and e is a param-out edge, mark m with phase 2
            if M(n) ∧ e = po
               M = M ∪ {m ↦ false}
            // If we are in phase 2 and e is a concurrency edge, mark m with phase 1
            elseif ¬M(n) ∧ e ∈ {id, fork, fork_in}
               M = M ∪ {m ↦ true}
            // Else mark m with the same phase as n
            else
               M = M ∪ {m ↦ M(n)}

until W = ∅

return dom M
```

Fig. 14 I2P: The iterated two-phase slicer

edge. This means that each edge has to be traversed at most twice, once in phase 2 and later again in phase 1. As an example, assume that we want to compute the slice for node 14 in the cSDG in Fig. 12. The algorithm first computes a thread-local slice for node 14 using the two-phase slicer and thereby visits the nodes {14, 13, 12, 10}. Nodes 9, 6 and 4 are not visited, but added to set S. The slicer now subsequently computes thread-local slices for these nodes and updates S as needed. The two-phase slice for node 9 visits the nodes {9, 8, 7, 2, 1} and inserts node 15 into S. The two-phase slice for node 6 visits nodes {6, 4, 3} (node 1 has already been visited in phase 1), the one for node 4 can be omitted, because node 4 has already been visited in phase 1. The last two-phase slice for node 15 visits node 15. The resulting slice consists of all visited nodes, which are highlighted gray in Fig. 12.

Figure 14 shows pseudo code for a more compact implementation based on a single map, which maps the visited nodes to the phase in which they have been visited. The condition of the first 'if' inside the foreach-loop checks if the adjacent node m has to be visited. This is the case if m has not been visited yet, or if it has been visited only in phase 2 and we are currently in phase 1 or intend to traverse a concurrency edge. The next conditional realizes the two-phase slicing technique: If the intended traversal happens in phase 2, parameter-in and call edges have to be omitted. The

last conditionals decide how to mark m in the map, according to two-phase slicing. At concurrency edge traversals, the reached node is always treated as visited in phase 1.

6.3 Context-sensitive paths in cSDGs

We have to extend our definition of context-sensitive paths in SDGs to include concurrency edges. Intuitively, if a path traverses a concurrency edge $m \rightarrow n$ towards n, the calling context of m is lost: The thread that has been left is allowed to execute further in parallel, so if the path reenters that thread later, one cannot demand that it reenters the thread at the original calling context. Furthermore, the traversal may reach n in any possible calling context of n, because m interferes with every possible instance of n. Thus a path p in a cSDG is context-sensitive, if it consists of a sequence $p_1, .., p_n$ of sequential, context-sensitive paths, where each pair (p_i, p_{i+1}), $0 < i < n$, is connected via a concurrency edge. Based on this observation we extend Reps and Rosay's definition of context-sensitive paths in SDGs as follows:

Definition 3 (Context-sensitive paths in cSDGs) In addition to definition 1, label interference, fork and fork-in edges with *conc*. A path in the cSDG of a concurrent program is context-sensitive, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal *conc_realizable* by grammar H', which extends grammar H of definition 1 as follows:

$matched \rightarrow matched\ matched \mid (_c^e\ matched\ )_c^e \mid l \mid \epsilon$
$unbalanced\_right \rightarrow unbalanced\_right\ )_c^e\ matched \mid matched$
$unbalanced\_left \rightarrow unbalanced\_left\ (_c^e\ matched \mid matched$
$realizable \rightarrow unbalanced\_right \quad unbalanced\_left$
$conc\_realizable \rightarrow (realizable\ conc)^*\ realizable$

The new rule permits concatenation of sequential, context-sensitive paths via concurrency edges. Extending the definition of context-sensitive slices for SDGs to cSDGs is straightforward.

We will also have to reason about context-sensitive paths in TCFGs later on. Their definition is very similar to definition 3, one basically substitutes parameter-out edges by return edges.

Definition 4 (Context-sensitive paths in TCFGs) For each call site c, label the outgoing call edges with a symbol $(_c^e$, where e is the entry of the called procedure, and the incoming return edges with a symbol $)_c^e$. Label fork and join edges with *conc*. Label all other edges with l.

A path from node m to node n in a TCFG is context-sensitive, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal *conc_realizable* by grammar H' of definition 3.

7 Context-sensitive Chopping of Concurrent Programs

Intersection-based chopping in combination with the iterated two-phase slicer enables fast and simple chopping of concurrent programs. Our first algorithm, abbreviated with IC (*intersection chopper*), intersects the backward slice for t and the forward slice for s computed with the I2P slicer. This algorithm is the easiest chopping algorithm for

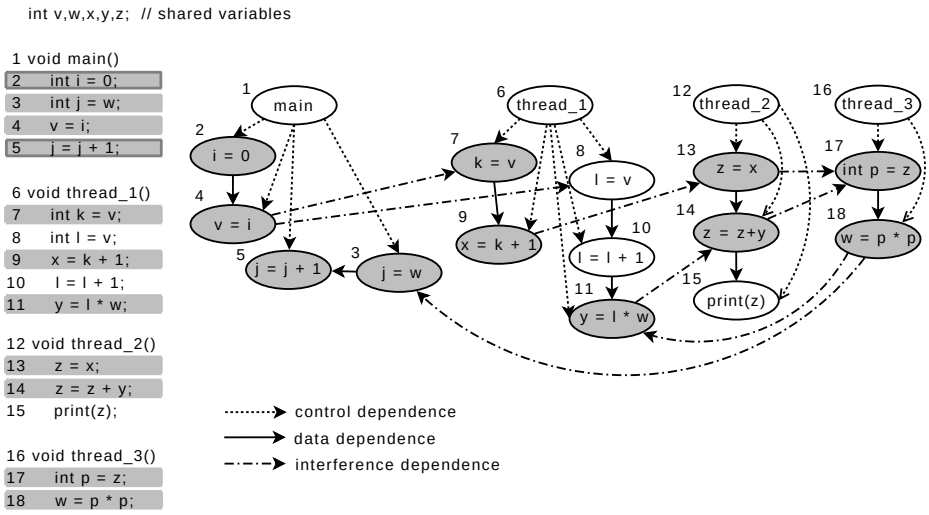

Fig. 15 The context-sensitive chop for chopping criterion $(2, 5)$

concurrent programs. Our second algorithm, the *iterated two-phase chopper* (I2PC) computes a backward slice for t and then a forward slice for s, which only visits the nodes already visited during the backward slice. Its runtime complexity is in $\mathcal{O}(|E|)$, like that of the underlying I2P slicer. Our third algorithm extends the fixed-point chopper from section 4 by substituting the two-phase slicers with iterated two-phase slicers. This extension has the same runtime complexity as the original fixed-point chopper, $\mathcal{O}(|E| * |N|)$.

As in the case of sequential programs, intersection-based chopping of concurrent programs is not context-sensitive. Unfortunately, the RRC cannot be applied to concurrent programs, due to interference dependence. Interference edges cannot be treated as the other kinds of edges, because they cross procedure borders arbitrarily, breaking the well-formedness of SDGs for sequential programs. Our context-sensitive algorithm, the *context-sensitive chopper* (CSC), is an extension of the RRC that is able to handle interference dependence and has the same runtime complexity. The CSC is based on the following observation: A chop in a concurrent program can be divided into a set of sequential chops. Figure 15 presents an example: It shows four threads that communicate via shared variables (for simplicity of presentation, all threads are assumed to run entirely in parallel). The chop from statement 2 to statement 5 in `main` is highlighted gray. It can be partitioned into the thread-local sets $\{2, 3, 4, 5\}$, $\{7, 9, 11\}$, $\{13, 14\}$ and $\{17, 18\}$. As one looks closer, these sets correspond to the sequential chops $RRC(\{2, 3\}, \{4, 5\}) = \{2, 3, 4, 5\}$, $RRC(\{7, 11\}, \{9, 11\}) = \{7, 9, 11\}$, $RRC(\{13, 14\}, \{13, 14\}) = \{13, 14\}$, and $RRC(\{17\}, \{18\}) = \{17, 18\}$. These chopping criteria have the following property: The source criterion consists of every node where the whole chop enters the according thread via concurrency edges, and of the original source criterion, if it lies in that thread, e.g. $\{2, 3\}$ in `main`. The target criterion consists of every node where the whole chop leaves the thread via concurrency edges and of the original target criterion, if it lies in the thread, e.g. $\{4, 5\}$ in `main`. So if we know the concurrency edges that belong to the whole chop, we are able to compute the single

Input: A chopping criterion (s,t).
Output: The chop from s to t.

// collect the concurrency edges I traversed by the I2PC chopper
$I = \texttt{I2PC}(s, t)$
$S = \{s\}$ *// a set for the source criterion*
$T = \{t\}$ *// a set for the target criterion*

// build the chopping criterion
`foreach` $m \rightarrow_{id} n \in I$
  $S = S \cup \{n\}$ *// add sink node n to the source criterion*
  $T = T \cup \{m\}$ *// add source node m to the target criterion*
// compute the chop with the RRC
$C = RRC(S, T)$
`return` C

Fig. 16 CSC: Context-sensitive chopping of concurrent programs.

sequential chops context-sensitively using the RRC. The CSC employs the I2PC to determine these edges, using a modified I2PC that collects the concurrency edges I that lie in its chop. Then, for every thread T, it picks the concurrency edges $E \subseteq I$ that enter T and the concurrency edges $L \subseteq I$ that leave T. Let N_E be the sink nodes of the edges E, i.e. the nodes where T is entered, and let N_L be the source nodes of the edges L, i.e. the nodes where T is left. The chop $RRC(N_E, N_L)$ is the context-sensitive sequential chop from N_E to N_L. The chop for the whole program consists of the union of these chops for all threads. This algorithm has the same asymptotic runtime behavior as the original RRC: The worst-case runtime complexity of the I2PC is in $\mathcal{O}(|E|)$. As the sub-graphs for the single threads in a cSDG are disjoint, the computation of the sequential chops using the RRC is in $\mathcal{O}(|E| * \mathit{MaxFormalIns})$. Thus CSC's worst-case runtime complexity is in $\mathcal{O}(|E| * \mathit{MaxFormalIns})$.

Figure 16 shows pseudo code for the CSC. The second step can be computed by a single call of RRC, because the subgraphs of the threads in a cSDG are disjoint, and RRC ignores concurrency edges: The source criterion is formed by the sink nodes T_I of all concurrency edges in I plus the original source criterion, and the target criterion is formed by the source nodes S_I of all concurrency edges in I plus the original target criterion. In our example, the concurrency edges are $I = \{4 \rightarrow_{id} 7, 9 \rightarrow_{id} 13, 13 \rightarrow_{id} 17, 18 \rightarrow_{id} 3\}$. The source criterion is $S = \{2, 3, 7, 13, 17\}$, the target criterion is $T = \{4, 5, 9, 13, 18\}$, and the chop $CSC(2, 5)$ is computed by $RRC(S, T)$.

At first glance, it is not clear that CSC is context-sensitive, because set I is computed by a context-insensitive technique. However, one can show that each concurrency edge in I belongs to the context-sensitive chop. If we traverse a concurrency edge towards node n in thread θ, then we do not know in which calling context we reach n, since interleaving cannot be forecast in general. We have to assume conservatively that we reach n in every possible context of n. Hence, if a concurrency edge $m \rightarrow n$ is in I, then every possible instance of n is in the context-sensitive forward slice for s, and there must exist at least one instance of n in the context-sensitive backward slice for t. Thus, according to definition 3, there exists a context-sensitive path from s to t via edge $m \rightarrow n$.

Theorem 1 *Let G be a cSDG, and $CSC(s, t)$ be the chop from s to t in G computed by the algorithm in Fig. 16. For every node n in G the following holds:*
$n \in CSC(s, t) \Leftrightarrow \exists$ *a context-sensitive path* $s \rightarrow^* n \rightarrow^* t$

Proof

'⇒' For every node $v \in CSC(s,t)$ there exist nodes $s' \in S$ and $t' \in T$ such that $v \in RRC(s',t')$, thus there exists a context-sensitive sequential path $p : s' \rightarrow^* v \rightarrow^* t'$, which can be generated from nonterminal *realizable* by grammar H'. We are left to show that we can extend p to a context-sensitive path $q : s \rightarrow^* s' \rightarrow^* v \rightarrow^* t' \rightarrow^* t$. We distinguish four cases:

1. $s = s' \wedge t = t'$
   In this case, $q = p$ and is therefore context-sensitive.
2. $s = s' \wedge t \neq t'$
   According to the creation of set T of the chopping criterion in Fig. 16, there exists a context-sensitive path $t' \rightarrow t'' \rightarrow^* t$, such that $t' \rightarrow t''$ is a concurrency edge (because t' has to be visited by the backward slicer after traversing a concurrency edge). Thus $t' \rightarrow t'' \rightarrow^* t$ has the form *conc* (*realizable conc*)* *realizable*. Hence, the concatenation of $s' \rightarrow^* v \rightarrow^* t'$ and $t' \rightarrow t'' \rightarrow^* t$ can be generated from nonterminal *conc_realizable*.
3. $s \neq s' \wedge t = t'$
   According to the creation of set S of the chopping criterion in Fig. 16, there exists a context-sensitive path $s \rightarrow^* s'' \rightarrow s'$, where $s'' \rightarrow s'$ is a concurrency edge (because s' has to be visited by the forward slicer after traversing a concurrency edge). Thus it has the form (*realizable conc*)$^+$. Hence, the concatenation of $s \rightarrow^* s'' \rightarrow s'$ with $s' \rightarrow^* v \rightarrow^* t'$ can be generated from nonterminal *conc_realizable*.
4. $s \neq s' \wedge t \neq t'$
   This is simply the combination of the two previous cases.

'⇐' We can rewrite that path as $s \rightarrow^* s' \rightarrow^* v \rightarrow^* t' \rightarrow^* t$, such that $s' \rightarrow^* v \rightarrow^* t'$ is a context-sensitive sequential path, s' is either s or is preceded by a concurrency edge, and t' is either t or succeeded by a concurrency edge. We have to show that $s' \in S$ and $t' \in T$. In that case, the algorithm is guaranteed to compute the chop $RRC(s',t')$, and then $v \in CRC(s,t)$ holds. For $s = s'$ or $t = t'$, this is trivial.
For $t' \neq t$, we have that $t' \rightarrow^+ t$ is a context-sensitive path, and thus in the backward slice of t, and that $s \rightarrow^* t'$ is a context-sensitive path, too, and thus in the forward slice of s (both paths can be generated from nonterminal *conc_realizable* by grammar H'). Thus $t' \in T$ holds. We can show similarly that $s' \in S$.

□

8 Time travels

cSDGs give rise to a new kind of imprecision, so-called *time travels* (Krinke 1998). An execution order between two sequentially executing statements is a time travel if it contravenes the execution order specified by the program's control flow[6]. Dependences in sequential programs require valid control flow, i.e. if b depends on a, b must be reachable from a in the control flow graph. Interference dependence cannot require such a condition, because thread interleaving cannot be forecast in general. As a result, interference dependence is not transitive. Treating it as being transitive can result in infeasible execution orders containing time travels. Consider the example in Fig. 12.

[6] As usual, conditional branching is treated here as non-deterministic branching.

Assume that we are interested in the backward slice for node 14. The I2P slicer visits the highlighted nodes, but according to the program's control flow, it is impossible for node 15 to influence node 14, because `int b = a * 4` is always executed before `x = b / 2`. Time travels are not limited to slicing, but affect cSDG traversal in general. There exist *time-sensitive* slicing algorithms that eliminate time travels (Krinke 2003 (ESEC/FSE); Nanda and Ramesh 2006), and we will employ these techniques for removing time travels in chops. To this end, this section explains the foundations of time travel detection. The next section introduces Nanda and Ramesh's time-sensitive slicing algorithm, which currently seems to be the most practical one, and serves as the basis of our time-sensitive chopping algorithm.

To avoid interference edge traversals that correspond to time travels, Krinke (Krinke 2003 (ESEC/FSE)) as well as Nanda and Ramesh (Nanda and Ramesh 2006) present slicing algorithms based on symbolic program execution which takes all possible interleaving orders into account. They detect and avoid time travels by keeping track of thread execution states: When an interference edge is traversed, they check whether the reached statement in thread t can be executed before the current execution state of t. If not, the traversal would be a time travel and is rejected. To keep track of the thread execution states, their algorithms annotate every visited node with a *state tuple* Γ containing the last visited node for each thread with respect to the path taken from the slicing criterion to the currently visited node: Initially, the state tuple of the slicing criterion s contains s itself as the state of the thread of s, and all other threads are mapped to an initial (i.e. nonrestrictive) state $\bot$, as they have not been visited yet. Following each backward traversal of an edge $m \rightarrow n$, m is annotated with a copy of n's state tuple, where the entry for m's thread is replaced by m. These annotations allow detection of invalid interference edge traversals: If the slicing algorithm is about to traverse an interference edge $q \rightarrow_{id} m$ towards q in thread t, and q_{old} is the state of t in m's state tuple, then it is compulsory that q may reach q_{old} in the TCFG, or else the traversal forms an invalid execution and is rejected. In our example in Figure 12, this situation arises when the algorithm traverses from node 7 to node 15. But `thread_1` had previously been left via interference dependence from node 13. Hence the algorithm needs to check whether it is possible that node 13 is reachable from node 15 in the TCFG, which is not the case. Thus this traversal would result in an invalid execution order and is rejected. This approach is still imprecise, because the state tuples consist only of nodes and ignore the calling contexts of these nodes. Both Nanda's and Krinke's algorithms improve precision by additionally annotating the nodes with calling contexts. In the remainder, we use the term *context* for a node and its calling context.

In order to give a first insight into how time-sensitive slicing works, Fig. 17 shows the basic structure of both algorithms, which can be viewed as extensions of the iterated two-phase slicer: They iterate a sequential slicing algorithm while determining which encountered concurrency edges are valid for traversal (traversing fork and fork-in edges may lead to time travels, too). A precise concurrent slice is achieved due to keeping track of thread execution states. To this end, their algorithms apply slicing based on contexts instead of nodes. The sequential slicers are called with a context c and its state tuple Γ as slicing criterion and return its sequential slice $\overline{S}(c)$ and the set I of visited pairs of contexts and state tuples with incoming concurrency edges. Similar to the iterated two-phase-slicer, these slicers are called iteratively for every pair of context and state tuple that is reached via a valid concurrency traversal. Besides this basic structure, the algorithms of Krinke and of Nanda and Ramesh differ significantly, particularly in the

Input: The cSDG G, a slicing criterion s.
Output: The slice S for s.

let $C(n)$ *return all possible contexts for node* n
let $\theta(c)$ *return the thread of context* c
let $\Gamma[c/t]$ *return a new state tuple* Γ' *by mapping thread* t *in state tuple* Γ *to context* c
let `SeqSlice`(c, Γ) *return the sequential slice* $\overline{S}$ *for context* c *and state tuple* Γ
and the set I *of visited pairs of contexts and state tuples with incoming concurrency edges*

/ Initialize the worklist* W *with an initial state tuple and mark its contents */*
$\Gamma_0 = (\bot, ..., \bot)$ *// every thread is in an initial state*
$W = \{(c, \Gamma) \mid t = \theta(s) \wedge c \in C(s) \wedge \Gamma = \Gamma_0[c/t]\}$
$M = \{s\}$ *// marks the visited worklist elements*

`repeat`
  $W = W \setminus \{(c, \Gamma)\}$ *// remove next element* (c, Γ) *from* W

  / Compute a sequential slice* $\overline{S}$ *for* (c, Γ) *and the set* I *of visited*
  *pairs of contexts and state tuples with incoming concurrency edges */*
  $(\overline{S}, I) =$ `SeqSlice`(c, Γ)
  $S = S \cup \overline{S}$

  / Compute valid concurrency edges */*
  `foreach` $(i, \Gamma_i) \in I$
    `foreach` $m \rightarrow_e n$: n *is node of context* i, $e \in \{id, fork, fork\_in\}$

      / Compute the valid contexts of* m **/*
      $C_m = \{c_m | c_m \in C(m) \wedge c_m$ *reaches the state of* $\theta(c_m)$ *in* $\Gamma_i\}$

      / Update worklist* W **/*
      `foreach` $w \in \{(c_m, \Gamma_m) | c_m \in C_m \wedge \Gamma_m = \Gamma_i[c_m/\theta(c_m)]\}$
        `if` $w \notin M$
          $W = W \cup \{w\}$
          $M = M \cup \{w\}$

`until` $W = \emptyset$
`return` S

Fig. 17 Slicing concurrent programs precisely

treatment and representation of contexts and in the iterated sequential slicer. Previous work of the author (Giffhorn and Hammer 2009) provides an in-depth comparison of both algorithms.

The depicted algorithm works as follows: Initially, it determines all possible contexts C of the given slicing criterion, node s. Then it annotates each context $c \in C$ with an initial state tuple Γ, where the execution state of c's thread is set to c and the states of the other threads are set to an initial state $\bot$: Every concurrency edge traversal towards a thread in this initial state is valid by definition. The annotated contexts are inserted into a worklist W. The algorithm iterates over every element (c, Γ) of W and computes its sequential slice $\overline{S}$ and the set I of visited pairs (i, Γ_i) of contexts i and state tuples Γ_i with incoming concurrency edges. Then it computes the valid concurrency edge traversals: For each pair $(i, \Gamma_i) \in I$ and each incoming concurrency edge $m \rightarrow n$, where n is the node of context i, the set of *valid contexts* C_m of m are determined. Context c_m of m is considered valid if c_m may reach in the TCFG the context that is saved as the state of c_m's thread in Γ_i. If c_m is valid, it is annotated

with an updated state tuple Γ_m, where c_m's thread t_m is mapped to c_m and the other threads are mapped to the same contexts as in Γ_i, and is inserted into worklist W. The resulting slice is the union of all slices $\overline{S}$.

According to our recent evaluation (Giffhorn and Hammer 2009), these slicers are able to reduce the size of slices significantly: up to 30% in that evaluation. However, due to a worst-case runtime complexity exponential in the number of threads of the target program, these algorithms may run into scalability problems.

8.1 Dynamic thread generation

For the described algorithm to remain sound, the state tuples have to model every thread that may exist at runtime, which requires a way to cope with thread invocation inside loops and recursive procedures. A simple solution is to give an upper bound for the number of invocations. A user of our system can do that by annotating thread classes with the number of instances that exist at runtime. But often such an upper bound is not known. In that case we use the following conservative approximation (Giffhorn and Hammer 2009): As shown in Fig. 17, the slicing algorithm initially gives threads an initial execution state. A thread with an initial execution state is by definition always reachable via a concurrency edge. We assume conservatively that a thread invocation inside a loop or recursion is executed infinitely often, such that there are infinite threads of that thread class. Hence every traversal of a concurrency edge towards such a thread class is able to find an instance that is in the initial execution state: The slicer simply omits the reachability analysis. Furthermore, all threads of that thread class can be represented by a single entry in the state tuples.

Note that the iterated two-phase slicer uses this conservative approximation implicitly for all threads, because it treats every concurrency edge as valid, and is thus able to handle dynamic thread invocation as well.

8.2 The impact of the model of concurrency

For simplicity, the algorithm sketched in Fig. 17 assumes that all threads execute entirely in parallel: The thread execution states in the state tuples are updated independently of each other when the slicer switches to another thread. A different model of concurrency is adopted by modifying the treatment of state tuples. In our case, concurrency is determined via thread regions, thus the state tuples work on the level of thread regions and contain one element per thread region. If the entry of a thread region r is to be updated to context c, then all entries of thread regions that are guaranteed to execute sequentially to r are assigned the same value c. Note that this may result in a thread region r' being mapped to a context of a different thread, if r' and r belong to different threads but execute sequentially to each other. In that case state c abbreviates that the thread to which r' belongs either has not been started yet, or it already has finished execution (because concurrency is computed on the level of fork and join points). We show at the example in Figure 18 that this information allows detection of more time travel situations. The set of shaded nodes marks the slice for node 14 if all threads are deemed to execute entirely in parallel, the dark gray nodes mark the slice if the thread region model is employed. In the latter case, the interference edge traversal $20 \rightarrow_{id} 9$ towards 20 can be identified as invalid: To influence the

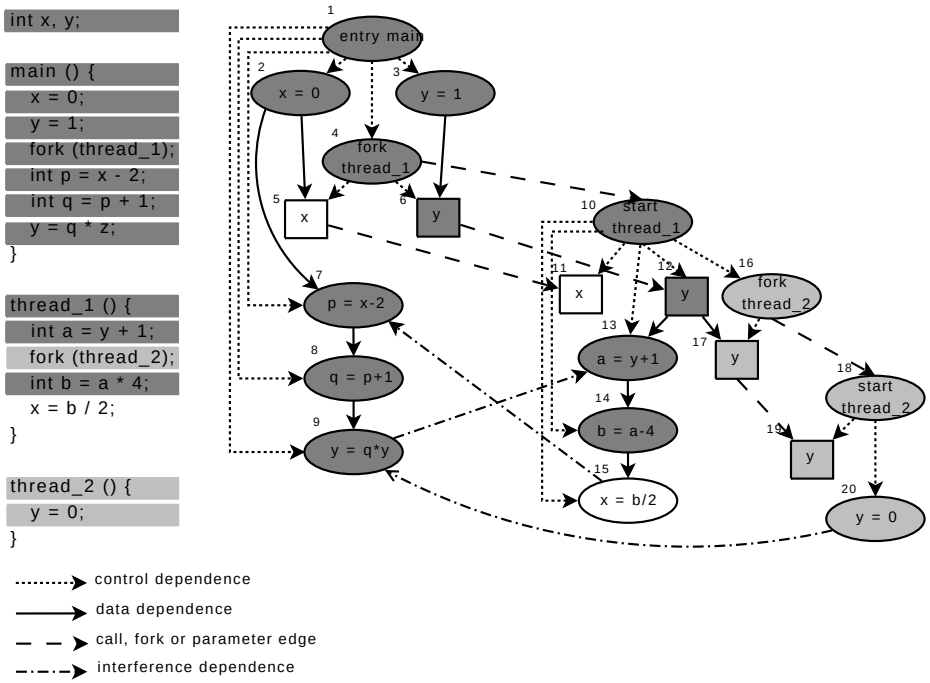

Fig. 18 Thread regions enable a more precise analysis of time travels

slicing criterion node 14, node 20 must be executed before nodes 9 and 13. Since thread 2 is started after node 13, this would require time travel. If all threads are assumed to execute entirely in parallel, one cannot detect that time travel. Note, however, that the interference edge $20 \rightarrow_{id} 9$ cannot be removed from the cSDG! For slicing criteria other than node 14 its traversal might be valid.

8.3 Time-sensitive paths in cSDGs

Intuitively, a slice for criterion s is time-sensitive, if it only contains the nodes of all cSDG paths leading to s that are free of time travels. Since we aim to develop a time-sensitive chopping algorithm, we need a formal definition as well. We define time-sensitive paths following the work of Krinke (Krinke 2003 (ESEC/FSE)).

Definition 5 (Contexts) A *context* c consists of a node n, annotated with a *call string*, which represents the call stack of a certain invocation of n's procedure. The call string encodes the call stack in ascending order: The first symbol represents the bottom of the call stack, the last symbol represents the topmost element. $Node(c)$ returns the node represented by c.

Contexts can be used to traverse cSDGs in a context-sensitive manner, by increasing or decreasing the call stack when entering or leaving a procedure. A path $c \rightarrow^* c'$ of contexts, traversed by that technique, is context-sensitive (Krinke 2002). Nanda and

Ramesh (Nanda and Ramesh 2006) as well as Krinke (Krinke 2003 (ESEC/FSE)) employ that technique to determine and propagate contexts during the slice.

Definition 6 (Context paths) A *context path* $c \rightarrow^* d$ denotes a context-sensitive path from $Node(c)$ to $Node(d)$ in a cSDG or TCFG, such that the path preserves the call stacks of c and d: If the path ascends from c's procedure it decomposes c's call stack, and the procedure calls towards d's procedure taken by the path follow d's call stack. Of course, the path may call other procedures and return from them underway. Or more formally, let w be the word generated from grammar H' for $c \rightarrow^* d$. The sequence of unmatched closing parentheses in w have to correspond to a suffix of c's call string. Similarly, the sequence of unmatched opening parentheses in w have to correspond to a suffix of d's call string.

It can be shown that the concatenation of two context paths $c \rightarrow^* d$, $d \rightarrow^* c'$ results in a new context path $c \rightarrow^* c'$ (Krinke 2003 (PhD thesis)).

A context c *reaches* another context c', if c' can be executed after c, according to control flow.

Definition 7 (Context reaching) A context c reaches another context $c' \Leftrightarrow \exists$ a context path $c \rightarrow^* c'$ in the TCFG.

The 'reaches' relation identifies sequences of contexts that can be executed without time travels:

Definition 8 (Threaded witness (Krinke 2003 (ESEC/FSE))) A sequence $\langle c_1, ..., c_k \rangle$ of contexts is a *threaded witness*, iff $\forall 1 \leq j < i \leq k$, c_i and c_j can execute concurrently, or c_i reaches c_j.

If a sequence of contexts is a threaded witness, then the contexts can be executed in that order without creating a time travel. A path of contexts in a cSDG is *time-sensitive* if it is context-sensitive (i.e. it is a context path) and there exists an adequate threaded witness:

Definition 9 (Time-sensitive paths (Krinke 2003 (ESEC/FSE))) A path $c_1 \rightarrow^* c_k$ of contexts in a cSDG is *time-sensitive*, iff it is context-sensitive and the sequence of its contexts form a threaded witness $\langle c_1, ..., c_k \rangle$.

In the remainder, we will often use indices to make clear to which node a context belongs to, i.e. context c_n is a context of node n. Time-sensitive slices are defined as follows:

Definition 10 (Time-sensitive slice) A time-sensitive backward slice in a cSDG G for a slicing criterion s consists of the set of nodes

$$\{n \mid \exists \text{ a time-sensitive path } c_n \rightarrow^* c_s \text{ in } G\}$$

A time-sensitive forward slice in a cSDG G for a slicing criterion s consists of the set of nodes

$$\{n \mid \exists \text{ a time-sensitive path } c_s \rightarrow^* c_n \text{ in } G\}$$

9 Nanda and Ramesh's time-sensitive slicing algorithm

According to our recent evaluation (Giffhorn and Hammer 2009), the time-sensitive slicer of Nanda and Ramesh currently seems to be the most practical one. Our time-sensitive chopper is based on that algorithm, therefore we explain it in detail in this section. Nanda and Ramesh describe two versions of their algorithm in (Nanda and Ramesh 2006), a version for cobegin-coend parallelism and a version for fork-join parallelism suitable for Java. Since our work focuses on Java, we only refer to the latter. We further include several recently developed optimizations and improvements (Giffhorn and Hammer 2009).

9.1 Context representation and reachability analysis

Since time-sensitive slicing utilizes contexts in the thread execution states, an efficient technique for working with contexts is mandatory. An intuitive representation of contexts are call strings, which can be propagated and modified during the slice (Krinke 2002). Nanda and Ramesh argue that this technique is impractical, because call strings grow with the size of the target program and thus lead to poor runtime performance and high memory consumption. Their slicer represents contexts by single integers instead. For that purpose, a preprocessing step collects and enumerates all possible contexts. This preprocessing step in turn employs call strings, but has to be executed only once.

The preprocessing step starts with a special folding method for cycles in interprocedural control flow graphs and creates an *interprocedural strongly connected regions* (ISCR) graph, which permits a topological enumeration of the remaining contexts in reverse postorder. This enumeration enables a very efficient reachability analysis, as we will see later. The challenge in folding ICFGs is to do so in a context-sensitive manner, such that no precision is lost. Therefore, a simple analysis of strongly connected components is not sufficient. Figure 19 shows an example. In the depicted ICFG, node 6 reaches node 3, but only via context-insensitive paths. A context-sensitive analysis is able to detect and reject these paths. This is not possible anymore in the resulting folded graph.

Nanda and Ramesh describe their folding algorithm in (Nanda and Ramesh 2006). We present an alternative, less complex algorithm. It consists roughly of three steps:

1. Folding context-sensitive cycles.
2. Inlining procedures.
3. Enumerate contexts in reverse postorder.

Folding context-sensitive cycles This step scans the ICFG for cycles that are context-sensitive. For that purpose, cycles containing both call and return edges are ignored. Krinke describes a suitable algorithm working in two phases (Krinke 2003 (PhD thesis)):

- Phase 1 removes all return edges and folds the remaining strongly connected components. After this, the return edges are put back.
- Working on the graph resulting from phase 1, phase 2 removes all call edges and folds the remaining strongly connected components. Afterwards, the call edges are put back.

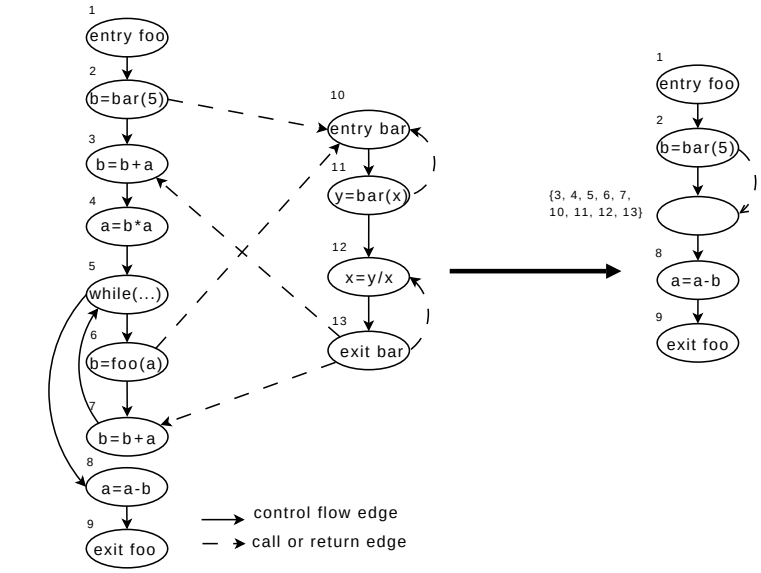

Fig. 19 Naïve folding of strongly connected components removes context information

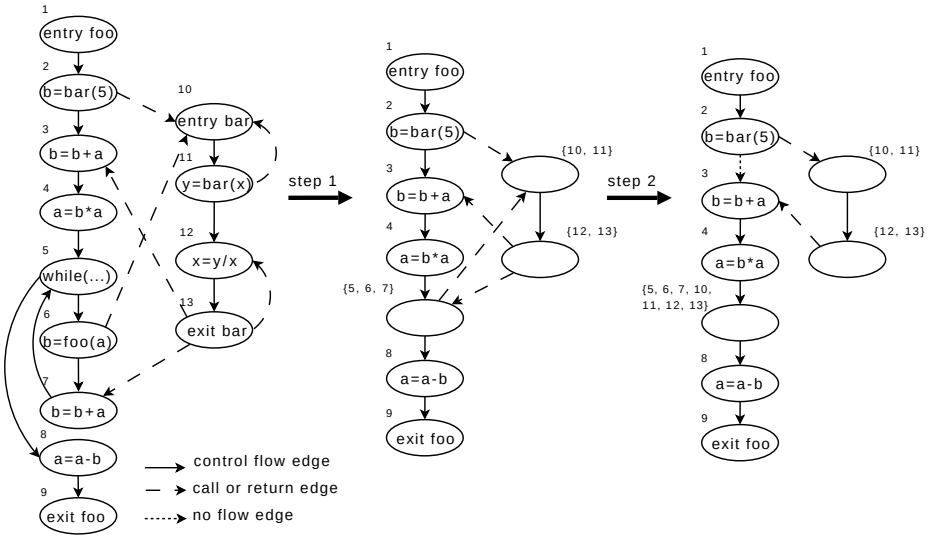

Fig. 20 ISCR graph creation: Folding of context-sensitive cycles (step 1) and procedure inlining (step 2)

The graph in the middle of Fig. 20 shows the result of this phase for our example. The while-loop is only folded intra-procedurally, the recursive call- and return-cycles are folded separately.

Procedure inlining The next step inlines the remaining procedures. All nodes of procedures that are called by a fold node are added to the fold node and the call and return edges between the fold node and these procedures are deleted. Furthermore, the control flow edges from procedure calls to their direct intra-procedural successors are replaced by *no flow* edges, which are needed for the context enumeration described in the next paragraph. The result for our example is shown in Fig. 20.

Context enumeration In order to enumerate the contexts, a *context graph* is generated from the ISCR graph. In a context graph, each node represents a single context, hence paths in that graph are always context-sensitive. The existing contexts are collected by a call string driven traversal of the ISCR graph, edges are added accordingly. Figure 21 shows pseudo code for such a traversal. It employs call strings to remain context-sensitive, by checking at return edges whether that edge returns to the call site stored at the top of the current call string.

Finally, the contexts in the context graph are enumerated in reverse postorder. The finished context graph for our example is shown in Fig. 22. A reverse postorder enumeration has the property that a node n has a smaller number than a node m, if it reaches m. Moreover, in context graphs it enables a quick check whether a procedure can be bypassed by a reachability analysis. Assume that we want to know if in the context graph in Fig. 22 node I reaches node VI. If the reachability analysis is committed by a backward traversal and arrives node V, it decides whether to traverse the incoming return edge by looking at the associated call node II. As II $\not<$ I, node I cannot lie in that procedure[7] and the traversal jumps directly to node II.

In general, employment of context graphs should be considered carefully, because their sizes, compared with the original ICFGs, do not scale well. However, we found that for programs that are within reach of time-sensitive slicers[8], context graphs are actually manageable, if effective cycle folding techniques are applied. Table 4 shows the size of the ICFGs and of the context graphs of our benchmark programs. The last column shows the runtime needed for the context graph generation. In the worst cases, the context graphs had about 8 times more nodes and edges than the ICFGs (for `DayTime` and `CellSafe`).

Reaching analysis in concurrent programs The mindful reader may have noticed that this subsection did not mention any treatment of concurrency-related edges. This is so because fork and join edges may cause cycles, whose treatment is not clear. As a consequence of this, the ICFG of each thread is processed by the above technique separately, resulting in a set of disjoint context graphs. A reachability analysis for contexts in different threads thus needs special treatment. A context s may reach a context t in another thread via two different ways: (1) s reaches in its context graph a context f that forks a thread in which t is executed directly or indirectly in subsequently invoked threads, or (2) in t's context graph, t is reached by a context of a join point that directly or indirectly joins s' thread. In summary, we get the reachability algorithm depicted in Fig. 23. If the given contexts s and t belong to different threads, the algorithm checks the two options described above. Otherwise, it commits a backward traversal starting at t and returns `true` if it visits s in that process.

[7] It cannot lie in the procedure, because all nodes in a procedure are reachable by a call of that procedure.

[8] Time-sensitive slicing currently seems to be practical for programs with about 10.000 LOC (Giffhorn and Hammer 2009).

Input: An ISCR graph G.
Output: The context graph for G.

```
let push(σ, n) add a node n to call string σ
let pop(σ) remove the most recently added node from σ
let top(σ) return the most recently added node

/* Initialize worklist W with the root node and an empty call string*/
W = {(r,"")}  // r is G's root, "" is the empty call string
M = {(r,"")}  // the visited contexts
E = {}  // a set of edges

repeat
   W = W \ {(n, σ)}  // remove next element from W

   /* Traverse to all directly reachable contexts */
   foreach n →e m
      if e ∈ {cf, nf}  // a control flow or no flow edge
         // propagate the call string
         if (m, σ) ∉ M
            W = W ∪ {(m, σ)}
            M = M ∪ {(m, σ)}
            E = E ∪ {(n, σ) →e (m, σ)}

      else if e ∈ {c}  // a call edge
         // extend the call string with call node n
         σ' = push(σ, n)
         if (m, σ') ∉ M
            W = W ∪ {(m, σ')}
            M = M ∪ {(m, σ')}
            E = E ∪ {(n, σ) →c (m, σ')}

      else if e ∈ {r}  // a return edge
         // only traverse to m if the top of the call string matches the reached call site
         let c be the call node connected with m through a no flow edge
         if top(σ) = c
            σ' = pop(σ)
            if (m, σ') ∉ M
               W = W ∪ {(m, σ')}
            M = M ∪ {(m, σ')}
               E = E ∪ {(n, σ) →r (m, σ')}

until W = ∅

return (M, E)
```

Fig. 21 Context graph creation

9.2 The slicing algorithm

Nanda's time-sensitive slicing algorithm is a modified iterated two-phase slicer based on contexts instead of nodes. It queries the context graphs to retrieve the context of a node, which basically works as follows: After traversing a dependence edge $m \rightarrow n$ towards m, where c_n is the current context of n, all contexts C_m of m are retrieved from the context graph. Then a reachability analysis on the context graph determines every context $c_m \in C_m$ that reaches c_n. The slicer proceeds with these contexts. This reachability analysis is applied upon each edge traversal, which can be a bottleneck in

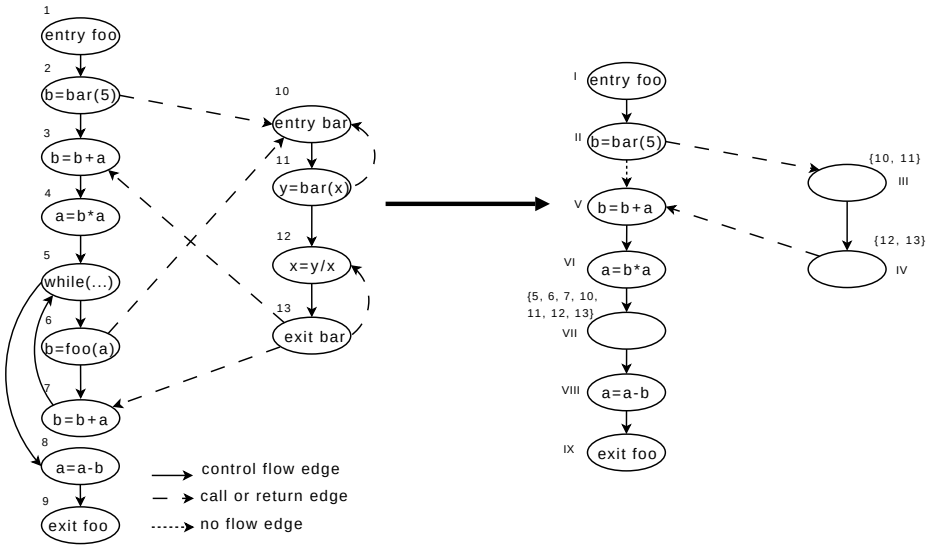

Fig. 22 The enumerated context graph

Table 4 Size of ICFGs and context graphs of our benchmark programs

Name	ICFGs		context graphs		computation
	nodes	edges	nodes	edges	time (in sec.)
Example	1,688	1,918	571	635	.5
ProdCons	2,218	2,472	311	350	.2
DiningPhils	2,974	3,305	314	357	.2
AlarmClock	4,433	5,114	4,860	5,461	.4
LaplaceGrid	10,023	11,252	3,424	3,644	1.0
SharedQueue	17,999	19,063	2,498	2,739	2.7
Daisy	45,603	54,448	225,351	284,234	91.8
DayTime	62,594	75,562	510,583	653,461	73.5
KnockKnock	34,667	42,315	145,048	182,254	15.2
DiskSched	4,387	5,860	18,065	24,438	.6
EnvDriver	19,129	26,734	72,074	99,083	9.4
Logger	9,577	10,782	14,061	16,181	1.2
Maza	10,591	11,790	59,760	69,359	2.1
Barcode	11,026	14,405	21,605	28,540	2.2
Guitar	13,460	17,414	24,382	31,203	2.1
J2MESafe	17,851	22,434	65,284	85,700	5.4
Podcast	25,366	31,467	94,254	119,429	8.5
CellSafe	40,709	50,686	250,367	318,425	38.7
GoldenSMS	26,445	32,429	156,639	203,924	12.9
HyperM	17,847	21,827	51,103	65,918	4.1

programs where statements have many different contexts (Giffhorn and Hammer 2009). We developed several optimizations relieving that bottleneck and describe our version of the algorithm, whose pseudo code is shown in Figures 24 and 25. Some of these optimizations have already been described in previous work (Giffhorn and Hammer 2009).

Input: Two contexts s, t.
Output: true, if s reaches t, else false.

```
let θ(c) denote the thread of context c

if θ(s) ≠ θ(t) // different threads
   // check if s reaches a context where t's thread is forked directly or indirectly
   let F be the set of contexts in θ(s) which directly or indirectly fork thread θ(t)
   foreach f ∈ F
      if reaches(s, f) return true

   // check if s' thread is joined such that the join point reaches t
   let B be the set of contexts in θ(t) which directly or indirectly join thread θ(s)
   foreach b ∈ B
      if reaches(b, t) return true

   return false // s does not reach t

else // θ(s) = θ(t)
   // traverse the context graph of θ(t) backwards, starting from t
   W = {t} // a worklist
   M = {t} // mark the visited contexts

   repeat
      W = W \ {c} // remove the next element c

      if c = s return true // s reaches t
      if c < s continue // c cannot be reached by s ⇒ skip it

      foreach d →_e c // traverse backwards all incoming edges
         if e ∈ {r} // a return edge
            let d' be the context connected with c through a no flow edge
            if d' ≮ s
               // s cannot be in the procedure called by d' ⇒ skip procedure
               if d' ∉ M
                  W = W ∪ {d'}
                  M = M ∪ {d'}
            else
               // enter procedure
               if d ∉ M
                  W = W ∪ {d}
                  M = M ∪ {d}
         else if e ∈ {cf, c} // a call or control flow edge
            if d ∉ M
               W = W ∪ {d}
               M = M ∪ {d}
   until W = ∅

   return false
```

Fig. 23 reaches: Reaching analysis in context graphs

In order to speed up traversal of intra-procedural dependence edges, we annotate contexts with procedure IDs, which are determined through a traversal of the context graphs in the preprocessing phase. If the slicer traverses an intra-procedural dependence edge $m \to n$ towards m, where c is the current context of n, only those contexts C'_m

of m are considered that have the same procedure ID as c[9]. The slicer proceeds with all contexts $c_m \in C'_m$ that reach c_n. For interprocedural edges, the slicer simply takes the adjacent context from the context graph[10].

We proceed by describing the algorithm depicted in Fig. 24. It consists of a main loop which iterates a two-phase slicer working with contexts. The two-phase slicer employs the worklists W_1 and W_2, the main loop uses worklist W. Furthermore, the algorithm employs two sets for marking visited contexts. The worklist for the main loop is initialized by all contexts of slicing criterion s, annotated with state tuples. The state tuples are computed by calling procedure `up` with the current context and an initial state tuple, where each entry is set to a nonrestrictive state $\perp$. Procedure `up`, shown in Fig. 25, works as follows: It copies the given state tuple and sets the states of all thread regions that execute sequentially to the thread region of context c to c.

The two-phase slicer inside the main loop treats intra- and interprocedural edges as described further above. If the two-phase slicer encounters a concurrency edge, it determines all contexts of the adjacent node m whose visiting would not cause a time travel. To this end, a context c_m of m has to reach the context stored as the state of c_m's thread region in the current state tuple. Via procedure `insert`, the valid contexts are annotated with updated state tuples and inserted into worklist W.

It remains to explain the auxiliary procedures `insert`, `insert2` and `restrictive` in Fig. 25, which are responsible for updating the state tuples and the worklists. Nanda and Ramesh identify combinatorial explosion of state tuples to be a major performance problem and define *restrictive state tuples* as a remedy.

Definition 11 (Restrictive state tuples (Nanda and Ramesh 2006)) Let $e = [c_1, .., c_k]$ and $e' = [c'_1, ..., c'_k]$ be two state tuples. e is *restrictive* to e', iff $\forall 1 \leq i \leq k : c_i$ reaches c'_i.

If c is a context, t and t' are state tuples and t' is restrictive compared to t, then a slice for the slicing criterion (c, t') is a subset of the slice for slicing criterion (c, t), because t' imposes more restrictions on the set of valid interference edges than t does. This property allows identification of redundant context pairs and state tuples: When a dependence edge e is traversed towards context c, the associated state tuple t' is computed. Then t' is compared with all state tuples T of earlier visits of c. If t' is restrictive compared to a tuple $t \in T$, the traversal of e towards c is discarded. This optimization is realized by procedures `insert` and `restrictive`. We have shown in previous work that this optimization must not be applied to elements that are to be inserted into the worklist for phase 2, otherwise it may cause incorrect slices (Giffhorn and Hammer 2009). In these cases procedure `insert2` is used instead, which simply checks whether the new element in question has been visited before.

9.3 Runtime complexity

The runtime complexity of time-sensitive slicing is dominated by a possible combinatorial explosion in the thread execution states, because a node can be visited repeatedly

[9] Due to graph folding and procedure inlining, C'_m may indeed contain more than one context.

[10] For that purpose, the SDG specific parameter-passing nodes are mapped to the associated call sites. Actual-in nodes are mapped to the call node, actual-out nodes to the return node, formal-in nodes to the entry node, and formal-out nodes to the exit node.

Input: A cSDG G, a slicing criterion node s.
Output: The slice S for s.

let $C(n)$ *return a sorted set of all contexts of node* n
let $\theta(c)$ *denote the thread of context* c
let $r(c)$ *denote the thread region of context* c

$\Gamma_0 = [\bot, ..., \bot]$ *// an initial context*
$W = \{(s, c_s, \Gamma) \mid c_s \in C(s) \wedge \Gamma = up(\Gamma_0, c_s)\}$ *// initial worklist*
$W_1 = \emptyset, W_2 = \emptyset$ *// two empty worklists*
$M_1 = \emptyset, M_2 = \emptyset$ *// sets for marking visited contexts*

// iterate over a modified 2-phase-slicer until W *is empty*
`repeat`
  // initialize the next iteration
  $W = W \setminus \{(n, c_n, \Gamma_n)\}$ *// remove next element*
  $W_1 = W_1 \cup \{(n, c_n, \Gamma_n)\}$ *// add it to* W_1

  `while` $W_1 \neq \emptyset$ *// phase 1, only ascend to calling procedures*
    $W_1 = W_1 \setminus \{(n, c_n, \Gamma_n)\}$ *// remove next element*
    $S = S \cup \{n\}$ *// add node* n *to the slice*
    `foreach` $m \rightarrow_e n$
      `if` $e \in \{id, fork, fork\_in\}$ *// edge leaves the thread*
        `foreach` $c_m \in C(m) : \theta(c_m) \neq \theta(c_n)$ *// only contexts of* m *in other threads*
          `if reaches`$(c_m, \Gamma[r(c_m)])$ *// time travel detection*
            `insert`(Γ_n, c_m, W, M_1)
      `else if` $e \in \{pi, call\}$
        let c_m *be the direct predecessor of* c_n *in* c_n*'s context graph*
        `insert`$(\Gamma_n, c_m, W_1, M_1)$
      `else if` $e \in \{po\}$
        let c_m *be the direct predecessor of* c_n *in* c_n*'s context graph*
        `insert2`$(\Gamma_n, c_m, W_2, M_2)$
      `else` *// e is an intra-procedural edge*
        `foreach` $c_m \in C(m) : c_m.proc = c_n.proc$ *// contexts in the same procedure*
          `insert`$(\Gamma_n, c_m, W_1, M_1)$

  `while` $W_2 \neq \emptyset$ *// phase 2, only descend to called procedures*
    $W_2 = W_2 \setminus \{(n, c_n, \Gamma_n)\}$
    $S = S \cup \{n\}$
    `foreach` $m \rightarrow_e n$
      `if` $e \in \{id, fork, fork\_in\}$
        `foreach` $c_m \in C(m) : \theta(c_m) \neq \theta(c_n)$
          `if reaches`$(c_m, \Gamma[r(c_m)])$
            `insert`(Γ_n, c_m, W, M_1)
      `else if` $e \in \{po\}$
        let c_m *be the direct predecessor of* c_n *in* c_n*'s context graph*
        `insert2`$(\Gamma_n, c_m, W_2, M_2)$
      `else if` $e \notin \{pi, call\}$
        `foreach` $c_m \in C(m) : c_m.proc = c_n.proc$
          `insert2`$(\Gamma_n, c_m, W_2, M_2)$
`until` $W = \emptyset$

`return` S

Fig. 24 Our version of Nanda and Ramesh's time-sensitive backward slicer

```
procedure insert(Γ_old, m, c_m, W, M)
   Γ_m = up(Γ_old, c_m) // create the new state tuple
   // run the restrictive state tuple optimization
   if ∃ (m, c_m, Γ') ∈ M : restrictive(Γ_m, Γ')
      return // the new element is redundant
   else
      W = W ∪ {(m, c_m, Γ_m)}
      M = M ∪ {(m, c_m, Γ_m)}
end insert

procedure insert2(Γ_old, m, c_m, W, M)
   Γ_m = up(Γ_old, c_m) // create the new state tuple
   if (m, c_m, Γ_m) ∉ M // insert the new element if it wasn't visited before
      W = W ∪ {(m, c_m, Γ_m)}
      M = M ∪ {(m, c_m, Γ_m)}
end insert2

procedure restrictive(Γ, Γ')
   foreach thread region r
      if !reaches(Γ[r], Γ'[r])
         return false
   return true
end restrictive

procedure up(c, Γ)
   Γ' = [c/r(c)]Γ  // create a copy of Γ and set r(c)'s state to c
   for all thread regions r that do not execute in parallel to r(c)
      Γ' = [c/r]Γ'  // set r's state to c
   return Γ'
end up
```

Fig. 25 Auxiliary procedures for the slicer in Fig. 24

with different thread execution states. Nanda and Ramesh thus determined a worst-case complexity of $\mathcal{O}(|N|^{pt})$, where p is the calling depth of the call graph, $|N|^p$ is an upper bound for the contexts, and t is the number of thread instances modeled in the state tuples (Nanda and Ramesh 2006).

10 Time-sensitive chopping

In this section, we transfer the time travel detection techniques employed in time-sensitive slicing to chopping. Intuitively, a chop $chop(s,t)$ is time-sensitive, if it only contains the nodes of all cSDG paths $s \rightarrow^* t$ that are free of time travels. Or more formally:

Definition 12 (Time-sensitive chop) A time-sensitive chop in a cSDG G for a chopping criterion (s,t) consists of the set of nodes

$\{n \mid \exists$ a time-sensitive path $c_s \rightarrow^* c_n \rightarrow^* c_t$ in $G\}$

Since context-sensitive chopping treats interference dependence as being transitive, its computed chops may contain time travels. Consider the example in Fig. 26. The gray shaded nodes are the context-sensitive chop $CSC(8,13)$. That chop contains two time travels: Node 14 cannot influence node 13, because it cannot be executed before

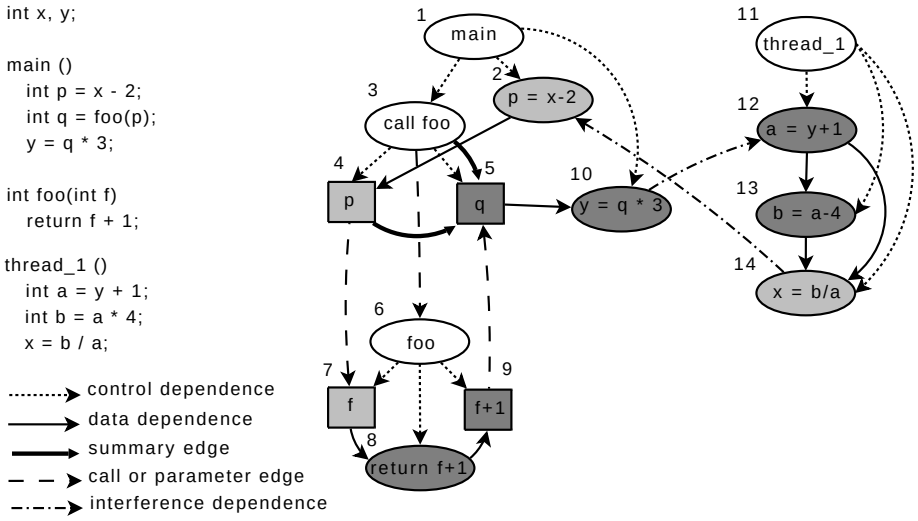

Fig. 26 Chops for chopping criterion $(8, 13)$. The gray shaded nodes mark the context-sensitive chop, the dark gray shaded nodes mark the time-sensitive chop

node 13. Similarly, node 8 cannot influence nodes 2, 4 and 7, so all these nodes should be removed from the chop. The first intuitive idea is to employ time-sensitive slicing algorithms to remove these time travels. One first computes the context-sensitive chop $CSC(s, t)$, and then removes every node which is not in both the time-sensitive backward slice for t and the time-sensitive forward slice for s. In Fig. 26, this technique would determine the dark gray shaded nodes as the chop for $(8, 13)$, which is context- and time-sensitive. However, this technique does not always compute time-sensitive chops, for which Fig. 27 provides an example. The depicted graph shows the chop from node 2 to node 3, if computed as described above (for simplicity, all three threads execute entirely in parallel). The time-sensitive backward slice for 3 consists of the nodes {1, 2, 3, 4, 5, 6, 7, 8, 9}. The time-sensitive forward slice for 2, computed on these nodes[11], visits the nodes {2, 3, 5, 6, 7, 9}, which also form the resulting chop. Unfortunately, node 2 cannot influence node 3 via node 9: All paths from node 2 to node 3 via node 9 contain the sequence $7 \to 9 \to 5$, which is a time travel, as it leaves `thread_1` at node 7 and reenters it later at node 5. We therefore call this algorithm the *almost time-sensitive chopper* (ATSC). A straightforward solution is to inspect every possible path in the chop for time travels. Fortunately, there is an easier and more efficient solution.

Let us examine why ATSC is not time-sensitive. If a slicing algorithm for cSDGs is context-sensitive, the paths it traverses in a cSDG can only become time-insensitive due to leaving and reentering a thread via concurrency edges. For example, a backward slicer creates a time travel if it reenters a thread such that the reentering context c cannot reach the context c', where the thread was left. As explained in section 8, propagating thread execution states during the slice by annotating nodes with state tuples enables time travel detection. Let us take a look at the state tuples computed

[11] The well-known optimization, to omit the intersection.

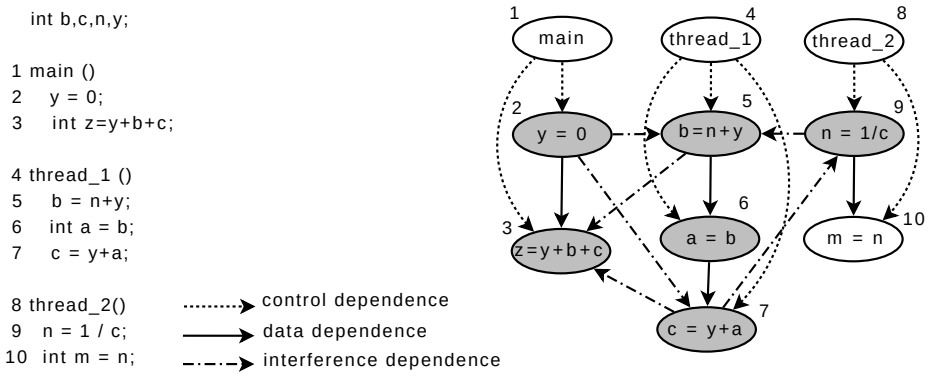

Fig. 27 Intersecting time-sensitive slices does not yield time-sensitive chops. The suchlike computed chop for chopping criterion $(2, 3)$ contains time travels

Table 5 The state tuples for chop(2,3) in Fig. 27

Node	Backward slice state tuples	Forward slice state tuples
2	$[2, \bot, \bot], [2, 7, \bot], [2, 5, \bot]$	$[2, \top, \top]$
3	$[3, \bot, \bot]$	$[3, \top, \top], [3, 5, \top], [3, 7, \top]$
5	$[3, 5, \bot]$	$[2, 5, \top]$
6	$[3, 6, \bot]$	$[2, 6, \top]$
7	$[3, 7, \bot]$	$[2, 7, \top]$
9	$[3, 5, 9]$	$[2, 7, 9]$

by the chop $ATSC(2, 3)$ in Fig. 27. Every node has only one context, so we represent it simply by the node itself. The initial state tuple for the backward slice for node 5 is $[\bot, \bot, \bot]$, where the first entry denotes `main`'s state, the second `thread_1`'s state and the third `thread_2`'s state. The slicer visits the gray highlighted nodes[12] with state tuples $\{(3, [3, \bot, \bot]), (2, [2, \bot, \bot]), (7, [3, 7, \bot]), (6, [3, 6, \bot]), (5, [3, 5, \bot]), (2, [2, 7, \bot]), (2, [2, 5, \bot]), (9, [3, 5, 9])\}$. The traversal from $(9, [3, 5, 9])$ to node 7 is rejected, because node 5 is not reachable from node 7. The initial state tuples for the forward slice for node 2 is $[\top, \top, \top]$ (dual to '$\bot$', '$\top$' represents a state which reaches every context). The slicer visits the gray shaded nodes with state tuples $\{(2, [2, \top, \top]), (3, [3, \top, \top]), (5, [2, 5, \top]), (6, [2, 6, \top]), (7, [2, 7, \top]), (3, [3, 5, \top]), (3, [3, 7, \top]), (9, [2, 7, 9])\}$. The traversal from $(9, [2, 7, 9])$ to node 5 is rejected, because node 5 is not reachable from node 7. Table 5 summarizes these state tuples.

We observe the following property of state tuples: A state tuple for a context c, computed by the time-sensitive backward slicer for a slicing criterion t, represents a sequence of interference dependences over which c may influence t. For example, state tuple $[3, 5, 9]$ of node 9 describes the sequence $9 \rightarrow_{id} 5 \rightarrow_{id} 3$, through which node 9 may influence node 3. State tuple $[2, 7, \bot]$ of node 2 describes the sequence $2 \rightarrow_{id} 7$, through which node 2 may influence node 3, provided that node 3 is executed after node 7. A program execution may only trigger such a sequence if its threads have not executed further than the associated state tuple. Assume that a program execution reaches node 9 in state $[3, 6, 9]$, then it is impossible for node 9 to influence node 3 via

[12] We ignore the visited nodes that lie outside the chop.

$9 \rightarrow_{id} 5 \rightarrow_{id} 3$ in that program run, because node 5, which is needed to transfer the effects of node 9 to node 3, has already been executed. The program execution must not exceed the states in state tuple $[3, 5, 9]$ before reaching node 9. Formally, the state tuple describing its thread execution states has to be *restrictive* compared to $[3, 5, 9]$. This observation can be generalized as follows: Let $E_{back}(c, t)$ be the set of state tuples in which the time-sensitive backward slice for a node t visited a context c. If a program execution reaches context c with state tuple e, then $E_{back}(c, t)$ must contain at least one state tuple e_{back}, such that e is restrictive compared to e_{back}. Otherwise, c cannot influence t in this program execution.

The state tuples computed by a time-sensitive forward slicer have a similar property. They indicate if in a certain program execution slicing criterion s may influence a context c. In contrast to the state tuples computed by the backward slicer, these state tuples mean that a program execution has to execute at least as far in order to trigger the associated sequence of interference dependences. Consider the state tuple $[2, 7, 9]$ of node 9. Assume that a program execution reaches node 9 in state $[1, 7, 9]$, then it is impossible for node 2 to influence node 9 via $2 \rightarrow_{id} 7 \rightarrow_{id} 9$ in that program run – `main` must have reached node 2 to do so. Or more formally, state tuple $[2, 7, 9]$ has to be restrictive compared to the state tuple in which the program execution reaches node 9.

We transfer this observation to chopping. Assume that the chopping algorithm ATSC visits context c with state tuples $E_{back}(c, t)$ during the backward slice and state tuples $E_{forw}(c, s)$ during the forward slice. There must exist state tuples $e_{forw} \in E_{forw}(c, t)$, $e_{back} \in E_{back}(c, s)$, such that e_{forw} is restrictive to e_{back}, else c cannot be in the time-sensitive chop for s and t, because no program execution is able to satisfy both conditions. In our example, $E_{forw}(9, 2)$ for node 9 is $\{[2, 7, 9]\}$, and $E_{back}(9, 3)$ is $\{[3, 5, 9]\}$. There is no possible program execution where node 2 influences node 3 via node 9, because a state tuple e, such that $[2, 7, 9]$ is restrictive to e, cannot be restrictive to $[3, 5, 9]$. Our last algorithm, the *time-sensitive chopper* (TSC), exploits that property to compute time-sensitive chops. Its pseudo code is shown in Fig. 28. Called for a chopping criterion (s, t), it first calls the backward slicer of Fig. 17 for t and retrieves the visited state tuples E_{back}. Then it performs a forward slice for s, which is basically dual to the backward slice algorithm, except for the modified procedures `insertChop` and `insertChop2` in Fig. 29. These are extensions of `insert` and `insert2` in Fig. 25 which additionally check whether the determined state tuple of context c_m is restrictive to any state tuple of C_m stored in E_{back}.

10.1 Optimizations

A very effective optimization for ATSC and TSC is to compute first a chop with the I2PC algorithm to detect if the chop is empty. In that case, they do not not need to execute the expensive time-sensitive slicers and simply return the empty set.

10.2 Correctness of TSC

We state that TSC computes time-sensitive chops and aim to prove that claim in this section. The following auxiliary lemma aids us in proving that claim:

Input: A cSDG G, a chopping criterion (s,t).
Output: The chop S for s.

// call the slicer in Fig. 24 and retrieve its collected state tuples
$E_{back} = M_1 \cup M_2$ *// M_1, M_2 are the sets in Fig. 24 after computation of the slice for t*
$\Gamma_0 = [\top, ..., \top]$ *// an initial context*
$W = \{(s, c_s, \Gamma) \mid c_s \in C(s) \wedge \Gamma = up(\Gamma_0, c_s)\}$ *// initial worklist*
$W_1 = \emptyset, W_2 = \emptyset$ *// two empty worklists*
$M_1 = \emptyset, M_2 = \emptyset$ *// sets for marking visited contexts*

```
repeat
   // initialize the next iteration
   W = W \ {(n, c_n, Γ_n)} // remove next element
   W_1 = W_1 ∪ {(n, c_n, Γ_n)} // add it to W_1
   while W_1 ≠ ∅ // phase 1, only ascend to calling procedures
      W_1 = W_1 \ {(n, c_n, Γ_n)} // remove next element
      S = S ∪ {n} // add node n to the slice
      foreach n →_e m
         if e ∈ {id, fork, fork_in} // edge leaves the thread
            foreach c_m ∈ C(m) : θ(c_m) ≠ θ(c_n) // only contexts of m in other threads
               if reaches(Γ[r(c_m)], c_m) // time travel detection for forward slices
                  insertChop(Γ_n, c_m, W, M_1, E_back)
         else if e ∈ {po}
            let c_m be the direct successor of c_n in c_n's context graph
            insertChop(Γ_n, c_m, W_1, M_1, E_back)
         else if e ∈ {pi, call}
            let c_m be the direct successor of c_n in c_n's context graph
            insertChop2(Γ_n, c_m, W_2, M_2, E_back)
         else // e is an intra-procedural edge
            foreach c_m ∈ C(m) : c_m.proc = c_n.proc // contexts in the same procedure
               insertChop(Γ_n, c_m, W_1, M_1, E_back)
   while W_2 ≠ ∅ // phase 2, only descend to called procedures
      W_2 = W_2 \ {(n, c_n, Γ_n)}
      S = S ∪ {n}
      foreach m →_e n
         if e ∈ {id, fork, fork_in}
            foreach c_m ∈ C(m) : θ(c_m) ≠ θ(c_n)
               if reaches(Γ[r(c_m)], c_m)
                  insertChop(Γ_n, c_m, W, M_1, E_back)
         else if e ∈ {pi, call}
            let c_m be the direct successor of c_n in c_n's context graph
            insertChop2(Γ_n, c_m, W_2, M_2, E_back)
         else if e ∉ {po}
            foreach c_m ∈ C(m) : c_m.proc = c_n.proc
               insertChop2(Γ_n, c_m, W_2, M_2, E_back)
until W = ∅
return S
```

Fig. 28 TSC: A time-sensitive chopper

Lemma 2 *Let c be a context visited by a time-sensitive backward slice for node t and by a time-sensitive forward slice for node s. If there exist state tuples $e \in E_{forw}(c, s), e' \in E_{back}(c, t)$, such that e is restrictive compared to e', then there exists a time-sensitive path $c_s \rightarrow^* c \rightarrow^* c_t$.*

Proof According to the definition of time-sensitive paths, we have to show that there exists a context-sensitive path $c_s \rightarrow^* c \rightarrow^* c_t$, whose sequence of contexts forms a threaded witness.

```
procedure insertChop(Γ_old, m, c_m, W, M, E_back)
   Γ_m = up(Γ_old, c_m) // create the new state tuple
   // run the restrictive state tuple optimization
   if ∃ (m, c_m, Γ') ∈ M : restrictive(Γ_m, Γ')
      return // the new element is redundant
   else if ∃ (m, c_m, Γ') ∈ E_back : restrictive(Γ_m, Γ')
       // remove time travels from the chop
      W = W ∪ {(m, c_m, Γ_m)}
      M = M ∪ {(m, c_m, Γ_m)}
end insertChop

procedure insertChop2(Γ_old, m, c_m, W, M, E_back)
   Γ_m = up(Γ_old, c_m) // create the new state tuple
   if (m, c_m, Γ_m) ∉ M // insert the new element if it wasn't visited before
      if ∃ (m, c_m, Γ') ∈ E_back : restrictive(Γ_m, Γ') // remove time travels from the chop
         W = W ∪ {(m, c_m, Γ_m)}
         M = M ∪ {(m, c_m, Γ_m)}
end insertChop2
```

Fig. 29 Auxiliary procedures for TSC

We know that there exists a time-sensitive path $p_1 = c_s \rightarrow^* c$, traversed by the forward slicer, such that c is reached with state tuple e, and a time-sensitive path $p_2 = c \rightarrow^* c_t$, traversed by the backward slicer, such that c is reached with state tuple e'. It follows that the path $p_1.p_2 = c_s \rightarrow^* c \rightarrow^* c_t$ is context-sensitive, because p_1 and p_2 are context paths and thus can be concatenated to a new context path. It remains to show that it is time-sensitive, i.e. that the sequence of contexts in that path also forms a threaded witness. To this end, we show that the time-sensitive forward slicer can traverse the path $p_1.p_2$ without confronting a time travel.

The proof goes by induction. We already know that the forward slicer can traverse p_1 without confronting a time travel, thus we go directly to context c and proceed with the traversal of p_2. We denote the current context of our traversal with c_j. If we want to traverse to the next element c_{j+1} of the path, we know that this traversal is context-sensitive and that the already traversed path $c_s \rightarrow^* c \rightarrow^* c_j$ forms a threaded witness. A case analysis over the kind of edge from c_j to c_{j+1} shows that $c_s \rightarrow^* c \rightarrow^* c_j \rightarrow c_{j+1}$ forms a threaded witness, too:

- $c_j \rightarrow c_{j+1}$ is not a concurrency edge
  Since $c \rightarrow^* c_j \rightarrow c_{j+1} \rightarrow^* c_t$ is a time-sensitive path traversed by the backward slicer, it follows that c_j can reach c_{j+1}. Further, $\forall c_i$ in $c_s \rightarrow^* c \rightarrow^* c_j$ we have that either c_i reaches c_j and therefore c_{j+1} (because there must be two context paths $c_i \rightarrow c_j$ and $c_j \rightarrow C_{j+1}$, which can be concatenated to a new context path $c_i \rightarrow c_{j+1}$), or c_i executes in parallel to c_{j+1}. Thus the sequence of contexts in $c_s \rightarrow^* c \rightarrow^* c_j \rightarrow c_{j+1}$ forms a threaded witness.
- $c_j \rightarrow c_{j+1}$ is a concurrency edge
  We distinguish two cases:
  1. There exists another context c_k of the same thread as c_{j+1} in the already traversed sub-path $c \rightarrow^* c_j$.
     In that case we know that c_k reaches c_{j+1}, because the contexts in path $c \rightarrow^* c_j \rightarrow c_{j+1} \rightarrow^* c_t$, traversed by the backward slicer, form a threaded witness. Thus, $\forall c_i$ in $c_s \rightarrow^* c \rightarrow^* c_j$ we have that either c_i reaches c_j and thus c_{j+1}, or c_i

executes in parallel to c_{j+1}. The sequence of contexts in $c_s \rightarrow^* c \rightarrow^* c_j \rightarrow c_{j+1}$ forms a threaded witness.

2. Else, the thread θ of c_{j+1} was not visited yet during our traversal. This means that the current execution state of θ is the same as the state of θ in thread execution state e of c, which we call $e[\theta]$. We are left to show that $e[\theta]$ reaches c_{j+1}. We know that $e[\theta]$ reaches θ's state in e', called $e'[\theta]$. Again, we examine the path $c \rightarrow^* c_j \rightarrow c_{j+1} \rightarrow^* c_t$ traversed by the time-sensitive backward slice. Since the sub-path $c \rightarrow c_j$ does not visit θ, the last visit of θ during the backward slice was at context c_{j+1}. According to the propagation rules of thread state tuples, $e'[\theta] = c_{j+1}$. Thus, $e[\theta]$ reaches c_{j+1}.

□

Now we are ready for our main theorem:

Theorem 2 *Let G be a cSDG, and $TSC(s,t)$ be a chop from s to t in G. For every node n in G the following holds:*
$n \in TSC(s,t) \Leftrightarrow \exists$ *time-sensitive path* $c_s \rightarrow^* c_n \rightarrow^* c_t$ *in* G.

Proof

'⇒' n is only in the chop if there exists a context c_n of n that was visited by the chopper, and c_n is only visited if there exist thread execution states $e_{forw} \in E_{forw}(c,s), e_{back} \in E_{back}(c,t)$, such that e_{forw} is restrictive to e_{back}. Thus, according to lemma 2, there exists a time-sensitive path $c_s \rightarrow^* c_n \rightarrow^* c_t$ in G.

'⇐' We have to show that both slicing algorithms visit context c_n. This is clear for the backward slicer, because the path is time-sensitive. The forward slicer only visits c_n, if it can visit every context c in the sub-path $c_s \rightarrow^* c_n$ in a thread execution state e_{forw} that is restrictive to a thread execution state e_{back} determined by the backward slicer. We show that by induction over the sub-path $c_s \rightarrow^* c_n$.

- Induction start
  We start at c_s, which is initially annotated with a thread execution state e_{forw}, where the state of c_s's thread is c_s, and the other states are the entry nodes. Thus, e_{forw} is restrictive to any state e_{back} in which the backward slicer visits c_s.
- Induction step
  We traverse from the current context c_i to the successor c_{i+1}. Let e_{forw} and e_{back} be thread execution states of c_i, such that e_{forw} is restrictive to e_{back}. According to the propagation rules for thread execution states, c_{i+1} is visited by forward and backward slicer in the thread execution states e'_{forw} and e'_{back}, respectively, where the state of c_{i+1}'s thread is c_{i+1}, and the states of the other threads are the same as in e_{forw} and e_{back}, respectively. Thus, e'_{forw} reaches e'_{back}.

□

11 Evaluation

We have implemented the presented algorithms for chopping concurrent programs in Java. The implemented algorithms work on SDGs computed by Graf's and Hammer's

Table 6 The benchmark programs

Name	Nodes	Edges	Procedures	Thread Classes	Thread Instances	Chops
Example	1687	6148	41	2	(1, 1)	100,000
ProdCons	2217	8775	39	2	(1, ∞)	100,000
DiningPhils	2973	11331	43	2	(1, ∞)	100,000
AlarmClock	4085	13842	74	3	(1, 2, 1)	100,000
LaplaceGrid	10022	100730	95	2	(1, ∞)	100,000
SharedQueue	17998	139480	122	2	(1, ∞)	10,000
Daisy	45603	458502	555	2	(1, 1)	10,000
DayTime	62594	644400	734	2	(1, 1)	10,000
KnockKnock	34667	288736	493	4	(1, 2, ∞, 1)	10,000
DiskSched	4378	44546	131	2	(1, ∞)	10,000
EnvDriver	19129	184149	169	2	(1, 1)	1,000
Logger	9576	50800	225	2	(1, 1)	100,000
Maza	10590	60221	261	2	(1, 1)	100,000
Barcode	11025	67849	229	2	(1, 1)	100,000
Guitar	13459	87724	307	2	(1, 1)	100,000
J2MESafe	17851	125221	309	2	(1, 1)	1,000
Podcast	25366	162102	504	3	(1, 1, 1)	1,000
HyperM	17847	93068	277	3	(1, 1, 7)	1,000
CellSafe	40709	845931	524	2	(1, 1)	100
GoldenSMS	26445	212832	414	3	(1, 2, 1)	100

data flow analysis for Java programs (Graf 2009; Hammer and Snelting 2004). For the evaluation we used a 2.2Ghz Dual-Core AMD workstation with 32GB of memory running Ubuntu 8.04 (Linux version 2.6.24) and Java 1.6.0. Our benchmark consists of 20 programs shown in Table 6. These are the same programs used by the evaluation in section 5. The programs in the upper part are small to medium-sized programs which solve a certain task in a concurrent manner (e.g. LaplaceGrid solves Laplace's equation over a rectangular grid). The other programs have graphical user interfaces running as separate threads. Table 6 reports the number of nodes, edges and procedures in their cSDGs. Column 'Thread Classes' indicates how many different thread classes a program contains (subclasses of `java.lang.Thread`, plus the main thread). The entries in column 'Thread Instances' denote the number of instances of each thread class that may exist at runtime. Most of the programs have, besides the main thread, one additional thread of which only one instance exists at runtime. Several programs may create an arbitrary number of thread instances at runtime, which happens if threads are spawned inside loops or recursion. For example, KnockKnock consists of its main thread, which has one instance at runtime, a second thread with 2 instances, a third thread, of which an arbitrary number of instances may exist, and a fourth thread with 1 instance. Column 'Chops' shows the number of chops we computed for each program. We had to decrease the numbers from 100,000 chops for our smaller programs to 100 chops for the bigger ones, because time-sensitive chopping did not scale well.

In our evaluation, we measured the average chop size and runtime performance of our chopping algorithms. Furthermore, we investigated whether the more precise algorithms are able to detect more empty chops than the imprecise ones. This is valuable information for many analyses that employ chopping as a preprocessing step, because if a chop $chop(s, t)$ is empty, it is guaranteed that s cannot influence t in any possible program run. In that case, many applications, e.g. taint analyses, may omit the actual main analysis.

Table 7 Average size per chop (number of nodes)

Program	IC	I2PC	FPC	CSC	ATSC	TSC
Example	55.37	45.94	45.11	45.09	45.09	45.09
ProdCons	16.31	16.08	15.98	15.97	13.52	13.52
DiningPhils	31.06	30.85	30.71	30.68	27.59	27.59
AlarmClock	755.59	755.40	754.65	754.63	608.38	336.61
LaplaceGrid	693.88	693.80	693.79	693.78	326.54	273.00
SharedQueue	3957.87	3957.72	3957.66	3957.66	2841.06	2410.47
Daisy	25662.51	25662.37	25662.37	25662.37	–	–
DayTime	37636.24	37635.25	37595.92	37595.90	–	–
KnockKnock	13933.72	13931.04	13908.21	13908.18	–	–
DiskSched	846.50	846.01	845.22	845.21	583.80	568.17
EnvDriver	3647.10	3587.03	3583.06	3583.06	–	3561.33
Logger	152.00	149.09	148.41	148.40	136.67	136.67
Maza	371.17	361.83	290.59	272.98	237.11	200.72
Barcode	284.05	223.86	216.90	213.97	197.15	196.62
Guitar	815.28	766.03	758.28	757.26	733.69	731.56
J2MESafe	2370.25	2205.25	2165.16	2158.03	2135.45	2131.46
Podcast	5811.47	5797.30	5796.85	5796.85	2234.66	2197.35
HyperM	5742.82	5739.57	5739.41	5739.41	2512.10	2224.51
CellSafe	12257.68	11729.95	11581.70	11535.53	–	11500.76
GoldenSMS	8900.32	8900.20	8899.66	8899.66	–	2369.31

In summary, we evaluated the following algorithms:

- IC, intersects the forward slice for s with the backward slice for t, which are computed by the I2P slicer.
- I2PC, computes a backward slice for t on the forward slice for s, using the I2P slicer.
- FPC, basically the fixed-point chopper shown in Fig. 10, but employing the I2P slicer.
- CSC, our context-sensitive chopper shown in Fig. 16.
- ATSC, the almost time-sensitive chopper that intersects time-sensitive slices.
- TSC, our time-sensitive chopper shown in Fig. 28.

11.1 Precision

Table 7 shows the average chop size for each chopping algorithm and program. Many of the randomly generated chopping criteria led to empty chops (cf. section 11.3). These chops were not removed beforehand, since the different algorithms detect a different number of empty chops, which prevents their removal[13]. Thus, the average chop sizes include these empty chops. For Daisy, DayTime and KnockKnock, the time-sensitive choppers were not able to compute the chops in reasonable time, thus their entries are missing. ATSC has no entries for EnvDriver, CellSafe and GoldenSMS as well, because it could not finish these chops in reasonable time.

Context-sensitive chopping The context-sensitive chops computed by CSC were on average 5% smaller than the imprecise ones computed by IC, and about 25% smaller in

[13] One could exclude the empty chops determined by the most imprecise algorithm, but this would result in handpicked chopping criteria, reducing the expressiveness of our evaluation.

Table 8 Average runtime per chop (in seconds)

Program	IC	I2PC	FPC	CSC	ATSC	TSC
Example	.001	.001	.001	.001	.006	.006
ProdCons	.001	.001	.001	.001	.001	.001
DiningPhils	.001	.001	.001	.001	.003	.003
AlarmClock	.009	.006	.014	.021	.173	.160
LaplaceGrid	.017	.010	.014	.030	.443	.239
SharedQueue	.088	.067	.118	.930	16.136	12.899
Daisy	.525	.495	.934	6.031	–	–
DayTime	.631	.582	1.645	7.994	–	–
KnockKnock	.222	.188	.497	2.087	–	–
DiskSched	.015	.011	.023	.044	8.651	7.798
EnvDriver	.082	.063	.136	1.292	–	671.548
Logger	.007	.004	.004	.007	.062	.062
Maza	.012	.007	.011	.015	10.227	13.143
Barcode	.011	.006	.007	.010	.292	.249
Guitar	.022	.013	.019	.028	2.318	2.168
J2MESafe	.049	.031	.065	.126	195.997	231.883
Podcast	.085	.062	.110	.225	213.355	128.618
HyperM	.076	.060	.109	.412	356.021	291.548
CellSafe	.452	.344	.959	1.478	–	8299.311
GoldenSMS	.139	.101	.281	.796	–	8942.593

the best case (for Barcode and Maza). Similar to their pendants for chopping sequential programs, the intersection-based algorithms I2PC and FPC were almost as precise as the context-sensitive CSC. The only considerable differences appeared in program Maza, where the CSC chops were 24% smaller than the I2PC chops and 6% smaller than the FPC chops. On average, the CSC chops were only 2% smaller than the I2PC chops and less than 1% smaller than the FPC chops.

It is interesting that context-sensitivity has not the same impact on chopping concurrent programs as it has on chopping sequential programs. The same programs were used in the evaluation in section 5, where the context-sensitive chops were considerably smaller than the naïve intersection-based ones. The cause of that effect seems to be that context-sensitive traversal of a cSDG drops the current context when it switches threads via concurrency edges.

Time-sensitive chopping For several programs, time-sensitive chopping reduced the chop sizes significantly – about 73% for GoldenSMS and about 60% for Podcast, HyperM and LaplaceGrid. On average, the ATSC chops were 28% smaller than the IC chops and 22% smaller than the CSC chops. The TSC chops were even 32% smaller than the IC chops and 27% smaller than the CSC chops. The evaluation shows that ATSC can miss a considerable number of time travels: For LaplaceGrid, the TSC chops were 45% smaller than the ATSC chops. On average, the TSC chops were 8% smaller than the ATSC chops.

11.2 Runtime

Table 8 shows the average runtime in seconds needed for one chop.

Table 9 Percentage rate of empty chops within our chopping criteria

Program	IC	FPC	CSC	TSC
Example	78.2	83.1	83.1	83.1
ProdCons	89.1	89.6	89.6	90.1
DiningPhils	88.4	88.9	88.9	89.2
AlarmClock	60.5	60.9	60.9	62.0
LaplaceGrid	72.0	72.2	72.2	77.4
SharedQueue	51.2	53.3	53.3	55.4
Daisy	24.6	25.6	25.6	–
DayTime	20.0	22.8	22.8	–
KnockKnock	29.8	36.6	36.6	–
DiskSched	50.7	55.1	55.1	62.0
EnvDriver	43.8	54.6	54.6	54.6
Logger	83.7	86.7	86.7	87.2
Maza	76.3	81.5	81.5	82.8
Barcode	56.9	83.4	83.4	83.9
Guitar	54.4	74.3	74.3	74.4
J2MESafe	45.1	63.9	63.9	64.3
Podcast	42.1	53.4	53.4	67.2
HyperM	36.3	44.1	44.1	56.0
CellSafe	31.0	49.0	49.0	49.0
GoldenSMS	43.0	44.0	44.0	63.0
Total	53.9	61.1	61.1	70.7

Context-sensitive chopping Since context-sensitive chopping of concurrent programs gains less precision compared to intersection-based chopping than in the sequential case, it is also somewhat slower than intersection-based chopping. Here, algorithms IC and even FPC are always clearly faster than the context-sensitive CSC. In the best case, for EnvDriver, IC was about 16 times faster than CSC, FPC was 9 times faster. By far the most performant chopper in our evaluation was I2PC.

Time-sensitive chopping The runtime evaluation shows that the high precision of time-sensitive chopping is at the expense of runtime costs. Only for the smaller programs ATSC and TSC could keep up with the other algorithms. For the other programs, performance declined as expected, due to their worst-case exponential runtime behavior. In the worst case, for GoldenSMS, a TSC chop needed almost 2.5 hours on average to compute a chop. Noticeably, due to its increased precision, TSC is able to outperform ATSC on most benchmark programs.

11.3 Empty Chops Detection

Table 8 shows how many empty chops our algorithms determined for our chopping criteria. The Table shows that the ratio of empty chops varied strongly from program to program. For DayTime, roughly every 5th chop was empty, for ProdCons 9 out of 10 chops were empty. Even though context-sensitive chopping increased precision only about 5% on average, it was more effective in finding additional empty chops. On average, it determined 61.1% of the chops to be empty, compared to 53.9% empty chops found by algorithm IC. Interestingly, FPC and CSC found exactly the same number of empty chops. The time-sensitive TSC found even more empty chops. On average 70.7% of its computed chops were empty.

11.4 Study Summary

Finally, we want to summarize the results of our evaluation.

Context-sensitive chopping Compared to naïve intersection-based chopping, context-sensitive chopping reduced chop sizes about 5% on average, and about 25% in the best cases. Since this gain of precision is smaller than in the case of sequential programs, CSC's runtime performance is not as competitive as that of its pendant for sequential programs, the RRC. Nevertheless, CSC seems to be practical for practical programs. As in the sequential case, there are no arguments in favor of algorithm IC. Algorithms I2PC and FPC have a similar implementation effort, are faster and more precise. Since both are also almost as precise as CSC, with respect to both chop sizes and finding empty chops, they are genuine alternatives to CSC.

Time-sensitive chopping Time-sensitive chopping increases precision drastically – up to 73% in the best case and about 32% on average. It also detects a significant number of empty chops, which are considered non-empty by the intersection-based and context-sensitive choppers. However, one has to pay a price for that precision. The employed time travel detection techniques do not scale well, because they have a worst-case exponential runtime behavior. For several of our benchmark programs, the time-sensitive choppers were not able to finish computation in reasonable time. Overall, TSC gains smaller chops than ATSC and is able to outperform ATSC due to its increased precision, thus we suggest employing TSC for time-sensitive chopping.

Threats to validity Since evaluations depend on the quality of the benchmark, we want to discuss possible flaws of our program selection.

Our benchmark consists only of 20 programs. Table 8 shows that the execution times of time-sensitive chopping may vary greatly between programs of similar size (e.g. SharedQueue and EnvDriver). In order to make a robust statement about the practicability of time-sensitive chopping, an extended runtime evaluation on a much bigger and more differentiated benchmark is needed.

Since our chopping criteria were created randomly without any filtering technique eliminating 'nonsensical' chopping criteria, our results should be verified for concrete applications of chopping, whose settings may a priori exclude some kinds of chopping criteria.

Further threats to validity are possible bugs in our implementations, because the time-sensitive algorithms are extremely complicated.

12 Discussion

TSC basically computes one forward and one backward slice, therefore it bears the same worst-case runtime complexity as time-sensitive slicing, being $\mathcal{O}(|N|^{pt})$ (see section 9.3). The present evaluation and our recent evaluation of time-sensitive slicing (Giffhorn and Hammer 2009) indicate that time-sensitive slicing and chopping, using the optimizations developed so far (Giffhorn and Hammer 2009; Krinke 2003 (ESEC/FSE); Nanda and Ramesh 2006), can handle programs with about 10 kLOC in reasonable time. New optimizations to further relieve the combinatorial explosion remain an important issue for future work. Binkley et al. (Binkley et al. 2007) conduct an empirical

study on how to improve the performance of graph-based slicing for large sequential programs via graph folding, and yield a maximum reduction of 71% in runtime for a certain combination of folding techniques. It would be interesting for future work to evaluate if these results carry over to slicing and chopping of concurrent programs. An interesting property of time-sensitive chopping is its ability to detect empty chops, which are deemed non-empty by more imprecise techniques. Future work could explore if it is possible to detect these empty chops without computing the complete time-sensitive forward and backward slices. This could result in a more practical time-sensitive scanner for empty chops.

It is difficult to discuss pros and cons of time-sensitive chopping in this work, because we do not provide a concrete application. For applications that require fast response times, like debugging, time-sensitive analyses are probably too slow; for applications that require high precision, like security analyses, non time-sensitive analyses are maybe too imprecise. In the end this has to be determined by the developer of a concrete application. For that purpose, this work provides various algorithms offering different degrees of precision and speed and gives an insight into their performance characteristics. Furthermore, we were able to show that naïve intersection-based chopping is dispensable, because it is too imprecise and too slow and has no advantage in being easy to implement.

13 Related Work

Chopping Chopping originates from Jackson and Rollins' work on modularizing SDGs for reverse engineering (Jackson and Rollins 1994). They define chops to be confined to a single procedure. The source and the target of a chop must be within the same procedure, and only that procedure's code is analyzed. They suggest an iterative approach to extend such an intra-procedural chop to procedures called within that chop: If the chop contains a call to another procedure, another intra-procedural chop is computed, where the parameter variables of the called procedure form the source criterion, and the return variables of the called procedure form the target criterion. This kind of chopping is called *same level chopping*, because it does not take callers of the initial procedure into account.

Reps and Rosay extend Jackson and Rollins' same-level chopping to unbound chopping, where source and target can be in different procedures (Reps and Rosay 1995). Their chopping algorithm, the RRC described in section 3.2, is context-sensitive and the state-of-the-art algorithm for sequential programs. The authors integrated their algorithm in the Wisconsin Program-Slicing Tool for C, which was the foundation of CodeSurfer (Anderson et al. 2003), a commercial program analysis tool for C.

Krinke developed a new same-level chopper called *summary-merged chopper* (Krinke 2002), which is context-sensitive, as the iterative approach of Jackson and Rollins, and much faster in practice. In a subsequent work, Krinke introduces the concept of *barrier chopping* and *slicing* (Krinke 2003 (SCAM)), where a user can specify a barrier which must not be crossed by the chopping algorithm. This permits to exclude program parts from the analysis one is not interested in, e.g. library calls. Krinke implemented all these algorithms in the VALSOFT system (Krinke 2003 (PhD thesis)).

Slicing concurrent programs Probably the first author who addressed slicing of concurrent programs was Cheng (Cheng 1993, 1997). He uses a *Program Dependence Net*

(PDN) to represent dependences in parallel or distributed programs without procedures, where the concurrent tasks communicate via channels. Slicing on PDNs is performed using simple graph reachability.

Krinke was the first to address the time travel problem for slicing of concurrent programs (Krinke 1998, 2003 (ESEC/FSE)). Nanda and Ramesh developed an alternative algorithm containing several optimizations that strongly reduce the combinatorial explosion of thread execution state tuples (Nanda and Ramesh 2000, 2006). These techniques seem to be mandatory for an application of time-sensitive slicing. Previous work of the author evaluated both algorithms and contains several new optimizations and extensions (Giffhorn and Hammer 2009).

Chen presents a different approach to handle the intransitivity of interference dependence (Chen and Xu 2001). He uses execution orders, MHP analysis and synchronization information to detect time-travel situations during slicing. Since his approach needs to inline methods that use synchronization, it cannot completely handle recursion.

Qi and Xu (Qi and Xu 2005) present the *task communication reachability graph* (TCRG) to eliminate time travels in Ada programs. The TCRG is basically a control flow graph that unrolls the symbolic execution committed by Krinke's and Nanda and Ramesh's algorithms and describes all possible execution orders of a program. For that purpose, its nodes are pairs of contexts and state tuples and the edges model control flow between these elements. By using an optimization similar to restrictive state tuples, it is possible to subsume many state tuples by one representative, so it is not necessary to include all possible state tuples in the graph. From the TCRG, a data flow analysis creates as special dependence graph, in which all dependences are transitive. Context- and time-sensitive slices are then computed via simple graph reachability. The authors do not present an evaluation of their technique, hence it is not clear whether their approach is practical.

Zhao (Zhao 1999) introduced the *Multi-threaded Dependence Graph* (MDG) for Java which is similar to the cSDG and additionally contains *synchronization dependences* arising from Java's operations for synchronization. To slice MDGs, he adapts the two-phase slicer such that it additionally traverses interference and synchronization dependences in both phases. Nanda and Ramesh have shown that such a simple inclusion of interference dependence results in incorrect slices (Nanda and Ramesh 2006).

Hatcliff et al. (Hatcliff et al. 1999) use slicing in their Bandera project, a tool set for compiling Java programs into inputs of several existing model-checkers, to analyze and omit program parts that are unrelated to a given specification. They use dependences similar to that of the cSDG and define further dependences to represent synchronization and infinite delays of execution. Their *synchronization dependence* captures dependences between a statement and its innermost-enclosing acquisition and release of a monitor. The *divergence dependence* represents the situation where an infinite loop may infinitely delay the further execution, *ready dependence* similarly represents the situation where a statement may block the further execution of a thread. They treat interference dependence as being transitive.

Ramalingam has shown that synchronization-sensitive context-sensitive slicing of concurrent programs is undecidable (Ramalingam 2000). The proof consists of reducing Post's Correspondence Problem to the synchronization-sensitive context-sensitive reachability problem.

Müller-Olm and Seidl have shown that precise slicing of concurrent interprocedural programs is undecidable (Müller-Olm and Seidl 2001). Basically, if two nodes n and m are interference dependent $n \rightarrow_{id} m$ due to some variable v, then it is not decidable whether another statement s that redefines v may execute between n and m.

Concurrency analysis for Java Naumovich et al. (Naumovich et al. 1999) present a may-happen-in-parallel (MHP) analysis for Java programs that computes for every pair (s, s') of statements whether s and s' may execute concurrently. Their approach considers fork and join points as well as synchronization and is more precise than the technique we used. The analysis works on a *parallel execution graph* (PEG) which is derived from the control flow graph of the input program. PEG creation imposes several restrictions on the input program, most notably absence of recursion, because procedures have to be inlined. The MHP analysis on the PEG has a runtime complexity cubic to the number of PEG nodes and seems to be practical only for PEGs with < 2000 nodes (Li and Verbrugge 2008).

Li and Verbrugge (Li and Verbrugge 2008) implemented Naumovich et al's MHP analysis based on the Soot framework and developed several PEG simplification techniques, which may reduce PEG sizes drastically. These techniques basically identify areas in the PEG which can be merged to a single node. Their evaluation results suggest that MHP analysis for Java can be made practical for programs of reasonable size. However, restrictions like absence of recursion still persist.

Nanda and Ramesh (Nanda and Ramesh 2006) segment Java threads at fork and join points into *thread regions* and determine concurrency in Java programs on the level of these regions. Their analysis is not as precise as Naumovich et al's MHP analysis, because it ignores synchronization, but on the other hand, it can handle full Java including recursion and dynamic thread generation. We used their technique in this work, which turned out to be very performant in terms of execution times and memory consumption.

Barik (Barik 2005) introduces the *thread creation graph* (TCG) for an efficient MHP analysis for Java. The TCG consists of fork and join points of threads and models the thread invocation structure of a program. The author uses the TCG to determine which threads may execute concurrently. This works similar to Nanda and Ramesh's analysis, but is more coarse-grained, as Barik treats threads only as a whole: If a thread t spawns another thread t', they are deemed to execute sequentially, whereas Nanda and Ramesh find that the part of t behind the fork point executes concurrently to t'. He presents an alternative MHP model on the level of single statements which is also computed on the TCG.

Ruf (Ruf 2000) investigated synchronization removal techniques for Java programs and developed a *thread allocation analysis* for determining the number of instances a thread class may have at runtime. Basically, the analysis collects the allocations of thread objects in the program and determines conservatively how often such an allocation may be executed. Such an analysis is mandatory for treatment of dynamic thread generation.

14 Conclusion

This work examines precise chopping of concurrent programs, which has not been investigated before. It shows how two dimensions of precision, context-sensitivity and

time-sensitivity, affect chopping in concurrent programs, and how these degrees of precision can be achieved. To this end, it presents six different chopping algorithms, ranging from imprecise to context- and time-sensitive, whose gain of precision and runtime performance has been evaluated on a benchmark of Java programs. Context-sensitive chopping reduced the chop sizes up to 25%, while moderately raising execution times. Thus its employment seems to be promising. Time-sensitive chopping emerged as a powerful technique that reduced the chop sizes up to 73%. However, as the underlying approach has a worst-case runtime complexity exponential in the number of threads of the target program, these algorithms may run into scalability problems. It seems that the high costs require a selective employment of time-sensitive chopping. A pragmatic approach e.g. for taint analysis would employ a less precise algorithm first, and examine cases of possibly illicit flow further, if that flow can be excluded with one of the precise algorithms. Similar ideas are applicable for other areas of application. That way, one can greatly reduce analysis overhead, and still benefit from the precision of time-sensitive chopping.

References

Anderson, P., Reps, T. and Teitelbaum, T. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Softw. Eng.*, 29(8):721–733, 2003.

Barik, R. Efficient computation of May-Happen-in-Parallel information for concurrent Java programs. In *Proc. 18th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, pp. 152–169, 2005.

Binkley, D., Harman M. and Krinke, J. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.

Brumley, D., Newsome, J., Song, D., Wang, H. and Jha, S. Towards automatic generation of vulnerability-based signatures. In *Proc. SP'06*, pp. 2–16, 2006.

Chen, Z. and Xu, B. Slicing concurrent Java programs. *ACM SIGPLAN Notices*, 36(4):41–47, 2001.

Chen, Z., Xu, B., Yang, H., Liu, K. and Zhang, J. An approach to analyzing dependency of concurrent programs. In *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software*, pp. 39–43, 2000.

Cheng, J. Slicing concurrent programs. *Automated and Algorithmic Debugging, LNCS*, 749, pp. 223–240, Springer, 1993.

Cheng, J. Dependence analysis of parallel and distributed programs and its applications. *International Conference on Advances in Parallel and Distributed Computing*, pp. 395–404, 1997.

Giffhorn, D. Chopping concurrent programs. In *9th IEEE Int. Work. Conf. on Source Code Analysis and Manipulation*, pp. 13–22, 2009.

Giffhorn, D. and Hammer, C. Precise slicing of concurrent programs - An evaluation of static slicing algorithms for concurrent programs. *Springer JASE*, 16(2):197-234, 2009.

Graf, J. Improving and evaluating the scalability of precise system dependence graphs for objectoriented languages. Tech. Rep., Universität Karlsruhe (TH), Germany, 2009.

Hammer, C. Information flow control for Java - A comprehensive approach based on path conditions in dependence graphs. PhD thesis, Universität Karlsruhe (TH), 2009.

Hammer, C. and Snelting, G. An improved slicer for Java. *Workshop on Program analysis for software tools and engineering (PASTE'04)*, pp. 17–22, 2004.

Hammer, C. and Snelting, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Springer IJIS*, 8(6):399-422, 2009.

Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S. and Zheng, H. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. *Static Analysis Symposium, LNCS*, 1694, pp. 1–18, Springer, 1999.

Horwitz, S.B., Reps, T.W. and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, 1990.

Jackson, D. and Rollins, E.J. A new model of program dependences for reverse engineering. In *Proc. FSE*, pp. 2–10, 1994.

Krinke, J. Static slicing of threaded programs. *PASTE '98*, pp. 35–42, 1998.

Krinke, J. Evaluating context-sensitive slicing and chopping. *International Conference on Software Maintenance*, pp. 22–31, 2002.

Krinke, J. Barrier slicing and chopping. In *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

Krinke, J. Context-sensitive slicing of concurrent programs. *Proc. ESEC/FSE'03*, pp. 178–187, 2003.

Krinke, J. Advanced Slicing of Sequential and Concurrent Programs. PhD thesis, Universität Passau, 2003.

Li, L. and Verbrugge, C. A practical MHP information analysis for concurrent Java programs. In *Proc. 17th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, pp. 194–208, 2004.

Liu, H. and Kuan Tan, H.B. An approach for the maintenance of input validation. *Inf. Softw. Technol.*, 50(5):449–461, 2008.

Müller-Olm, M. and Seidl, H. On optimal slicing of parallel programs. *STOC 2001 (33th ACM Symposium on Theory of Computing)*, pp. 647–656, 2001.

Nanda, M.G. and Ramesh, S. Slicing concurrent programs. *ISSTA 2000*, pp. 180–190, 2000.

Nanda, M.G. and Ramesh, S. *Interprocedural slicing of multithreaded programs with applications to Java*. *ACM TOPLAS.*, 28(6):1088–1144, 2006.

Naumovich, G., Avrunin, G.S. and Clarke, L.A. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proc. ESEC/FSE '99*, pp. 338–354. Springer, 1999.

Qi, X. and Xu, B. An approach to slicing concurrent Ada programs based on program reachability graphs. *IJCSNS*, 6(1):29–37, 2005.

Ramalingam, G. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22(2):416–430, 2000.

Reps, T. and Rosay, G. Precise interprocedural chopping. In *Proc. FSE'95*, pp. 41–52. ACM Press, 1995.

Ruf, E. Effective synchronization removal for Java. *Programming Language Design and Implementation (PLDI)*, pp. 208–218, 2000.

Shacham, O., Sagiv, M. and Schuster, A. Scaling model checking of dataraces using dynamic information. *J. Parallel Distrib. Comput.*, 67(5):536–550, 2007.

Snelting, G., Robschink, T. and Krinke, J. Efficient path conditions in dependence graphs for software safety analysis. *ACM TOSEM*, 15(4):410–457, 2006.

Zhao, J. Slicing concurrent Java programs. *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pp. 126–133, 1999.