# Chopping Concurrent Programs

Dennis Giffhorn
Universität Karlsruhe (TH)
Karlsruhe, Germany
giffhorn@ipd.info.uni-karlsruhe.de

## 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 five 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 can reduce chop sizes significantly.*

## 1 Introduction

A chop $chop(s, t)$ for a source statement $s$ and a target statement $t$ in a program $p$ contains all statements that are used to convey 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 [1], path conditions [19], input validation [13], reducing programs for model checking [18] and for witnesses for illicit information flow [5]. 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. Further, the costs for the main analysis are reduced, which can outweigh the increased costs for a more precise chopping algorithm. For example, a path condition [19] 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 [17] 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.

```
1 void main()          | 1 int x,y;
2   int m = foo();     |
3   int n = foo();     | 2 thread_1()
                       | 3   int p = x;
4 int foo()            | 4   y = p;
5   return 1;          |
                       | 5 thread_2()
                       | 6   int a = y;
                       | 7   x = a;
```

**Figure 1. Examples for imprecise chopping**

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 also bear a new kind of imprecision, so-called *time travels* [11]. Consider the program on the right side in Fig. 1, consisting of two concurrent threads that communicate via two shared variables, `x` and

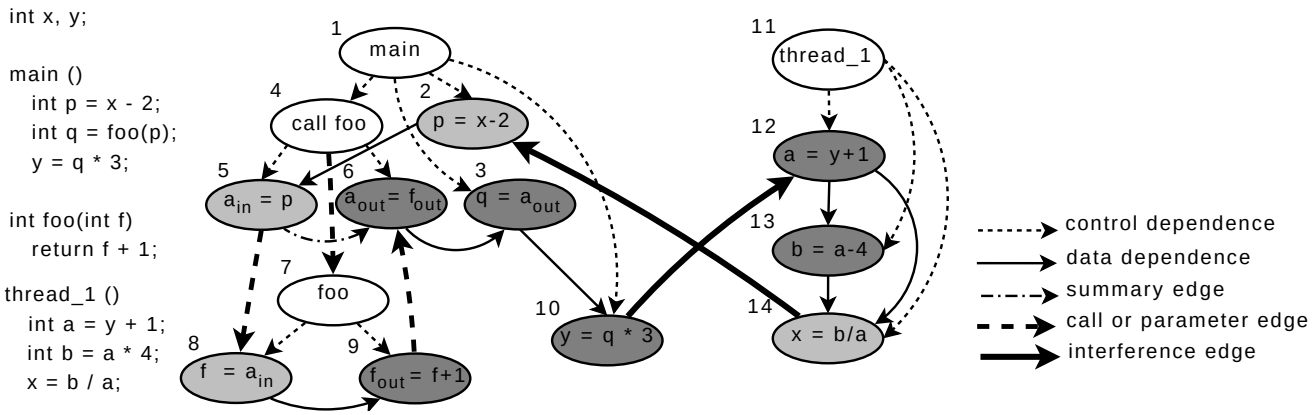


**Figure 2. A concurrent system dependence graph**

y. Clearly, the chop $chop(7, 6)$ 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 statement 6 is influenced by statements {3,4,6}. We show how to avoid such time travels in a chop, resulting in *time-sensitive* chops.

Overall, we present five 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 78% –, but at the expense of considerably increased execution times.

The paper is structured as follows: Chopping algorithms are based on *slicing* [20], another program analysis technique. Section 2 introduces slicing and chopping of sequential programs. Section 3 describes our chopping algorithms for concurrent programs, section 4 presents our evaluation. Section 5 discusses related work and section 6 concludes. Due to space reasons, correctness proofs for our algorithms are located in an extended version.

## 2 Slicing and Chopping

Slicing 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 $c$. Intuitively, a chop from $s$ to $t$ can be computed by intersecting the backward slice of $t$ with the forward slice of $s$ [8].

Slices are often computed based on *system dependence graphs* (SDG) [7]. A SDG $G = (Nodes, Edges)$ for program $p$ is a directed graph, where the nodes in $Nodes$ represent $p$'s statements and predicates, and the edges in $Edges$ 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 via a *call edge* $c \rightarrow_c e$. Parameter passing and result returning is realized using synthetic *parameter nodes* and *edges*. *Summary edges* represent transitive flow between parameter-passing parameter nodes and result-receiving parameter nodes of one call site. Figure 2 shows an example SDG for a program with procedure calls (ignore `thread_1` and interference edges for now).

Summary edges enable an efficient computing of context-sensitive backward slices in two phases [7]. 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. This *two-phase slicer* is the standard slicing algorithm for sequential programs[1].

### 2.1 Slicing Concurrent Programs

Concurrent programs exhibit a special kind of data dependence called *interference dependence* [12]: 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. We call such extended SDGs *concurrent system dependence graphs* (cSDG) [3]. Figure 2 shows an example cSDG. Concurrency causes several other new kinds of dependences, e.g. *fork-* and *join edges* to model thread invocation and -termination, or *synchronization dependences* to

[1] A two-phase slicer for forward slices works accordingly.

model synchronization [2, 3, 6]. For brevity, we will not discuss these kinds of dependences and their effects on precise slicing [2, 3, 14] and chopping in this paper. We conservatively assume that all threads execute entirely in parallel.

The two-phase slicing algorithm for sequential programs cannot be used to slice cSDGs, because summary edges do not capture interprocedural effects of interference dependences [14]. But a simple modification 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 visits an interference 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 [14] and can be implemented to yield context-sensitive slices in $\mathcal{O}(|Edges|)$ (our complexity specifications exclude the dependence graph generation, i.e. they only describe the complexity of the graph traversal).

cSDGs give rise to a new kind of imprecision, so-called *time travels* [12]. Dependences in sequential programs require valid control flow, i.e. if $b$ depends on $a$, $b$ must be reachable from $a$ via control flow. 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 in a slicing algorithm can result in infeasible execution orders. Consider the example in Fig. 2. Assume we are interested in the backward slice for node 13. The I2P slicer visits every node of the graph, but according to the program's control flow, it is impossible for node 14 to influence node 13.

To avoid interference edge traversals that correspond to time travels, Krinke [11] as well as Nanda and Ramesh [14] present slicing algorithms based on symbolic execution of the program which takes all possible interleaving orders into account. They detect and avoid time travels by memorizing the thread execution states: When an interference edge is traversed, they check whether the reached statement in thread $t$ can be executed before the memorized thread state of $t$. If not, the traversal would create a time travel and is rejected. According to our recent evaluation [3], 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.

### 2.2 Chopping sequential programs

As stated before, $chop(s, t)$ can be computed by intersecting the backward slice of $t$ with the forward slice of $s$ [8]. 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 is $\{2, 4, 5\}$ and the context-sensitive backward slice for statement 3 is $\{3, 4, 5\}$, thus the intersection results in chop $\{4, 5\}$.

Reps and Rosay [17] developed a sophisticated algorithm that chops programs context-sensitively. It exploits a well-formedness property of SDGs: all inter-procedural effects are propagated via call sites. First, the RRC determines the common callers of $s$ and $t$, i.e. the procedures which (transitively) call both the procedures of $s$ and $t$. This is achieved by computing a forward slice for $s$ and a backward slice for $t$ that only ascend procedure calls. Intersecting them reveals the common callers and the set of nodes $W$ in these procedures that belong to the chop. In a second step, the RRC collects the nodes in the procedures leading from the common callers to $s$ or $t$ and 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 $W$, where the forward slice only ascends to calling procedures and the backward slice only descends into called procedures. For the procedures leading to $t$ this works analogously. This step ignores the procedures that are called by the visited nodes, but do not (transitively) call the procedures of $s$ or $t$. In a third step, these omitted procedures are analyzed by *same-level choppers* [9, 17]. The resulting chop consists of the nodes visited in steps 2 and 3. By using $W$, $s$ 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 $\mathcal{O}(|Edges| * MaxFormalIns)$, where $MaxFormalIns$ is the maximum number of formal-in nodes in any procedure's PDG.

Though not explicitly stated by Reps and Rosay [17], the RRC can also be used 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. The such extended algorithm retains the same asymptotic running time.

## 3 Chopping Concurrent Programs

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. We will show how to extend the RRC to handle interference dependence. Another source of imprecision in cSDGs are time travels, as

described in section 2.1. We will eliminate time travels by employing the time-sensitive slicing techniques developed by Krinke [11], and Nanda and Ramesh [14]. Since time travel detection is expensive and difficult to implement, we present five different algorithms. These algorithms range from context- and time-insensitive over context-sensitive to context- and time-sensitive and thus offer different degrees of precision, runtime costs and implementation effort.

Our first algorithm, abbreviated with IC (*intersection chopper*), is neither context-sensitive nor time-sensitive. It 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 concurrent programs.

Our second algorithm, the *iterated two-phase chopper* (I2PC) employs a well-known optimization. It computes a backward slice for $t$ with the I2P slicer and then a forward slice for $s$, which only visits the nodes already visited during the backward slice. The resulting forward slice is already the chop, eliminating the intersection done in the IC. Its runtime complexity is in $\mathcal{O}(|Edges|)$, like that of the underlying I2P 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 2 is not in the backward slice of statement 3.

### 3.1 Context-sensitive chopping

In order to develop a context-sensitive algorithm, we first have to define context-sensitivity in the presence of interference dependence. Reps and Rosay define context-sensitive paths in sequential programs via a language of matching parentheses:

**Definition 1.** *(Context-sensitive paths in SDGs [17]) For each call site $c$, label the outgoing call- and parameter edges with a symbol $(_c^e$, where $e$ is the entry of the called procedure, and the incoming parameter edges with a symbol $)_c^e$. Label all other edges with $l$.*

*A path in the SDG of a sequential program is context-sensitive, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal $realizable$ by the following context-free grammar $H$:*

$matched \rightarrow matched\ matched \mid (_c^e\ matched\ )_c^e \mid l \mid \epsilon$
$unbal\_right \rightarrow unbal\_right\ )_c^e\ matched \mid matched$
$unbal\_left \rightarrow unbal\_left\ (_c^e\ matched \mid matched$
$realizable \rightarrow unbal\_right\quad unbal\_left$

We extend that definition to cSDGs and interference edges. Intuitively, if a path traverses an interference 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. Further, 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 an interference edge.

**Definition 2.** *(Context-sensitive paths in cSDGs) In addition to definition 1, label interference edges with $id$. 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 with the following rule:*

$conc\_realizable \rightarrow (realizable\ id)^*\ realizable$

The new rule allows concatenating sequential, context-sensitive paths via interference edges.

**Definition 3.** *(Context-sensitive chop) A context-sensitive chop for a chopping criterion $(s,t)$ in a cSDG $G$ consists of the set of nodes*

$\{n \mid \exists$ *a context-sensitive path* $s \rightarrow^* n \rightarrow^* t$ *in* $G\}$

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 for a concurrent program can be divided into a set of sequential chops. Figure 3 presents an example: It shows four threads that communicate via shared variables. The chop from statement 2 to statement 5 in `main` is highlighted in 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 $RRC(\{2,3\},\{4,5\}) = \{2,3,4,5\}$, $RRC(\{7,11\},\{9,11\}) = \{7,9,11\}$, $RRC(\{13,14\},\{13\}) = \{13,14\}$, 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 interference 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 interference edges and of the original target criterion, if it lies in the thread, e.g. $\{4,5\}$ in `main`. So if we know the interference edges that belong to the whole chop, we are able to compute the single sequential chops context-sensitively using the RRC. The CSC employs the I2PC to determine these interference edges, using a modified I2PC that collects the interference edges $I$ that lie in its chop. Then, for every thread $T$, it picks the interference edges $E \subseteq I$ that enter $T$ and the interference 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

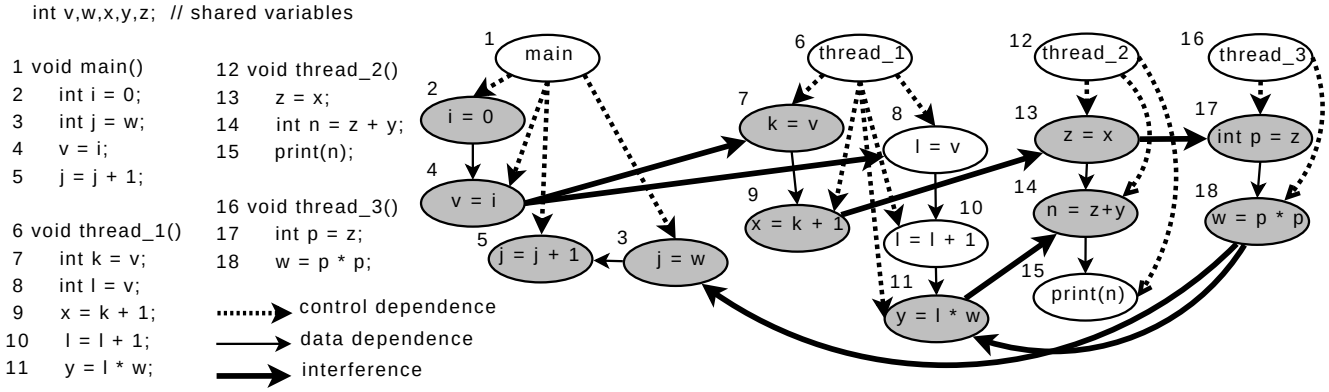


**Figure 3. A chop in a cSDG for criterion (2, 5).**

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

*// collect the interference 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` $C$

**Figure 4. CSC: Context-sensitive chopper**

the whole program consists of the union of these chops for all threads. This algorithm has the same asymptotic runtime behaviour as the original RRC: The worst-case runtime complexity of the I2PC is in $\mathcal{O}(|Edges|)$. As the sub-graphs for the single threads in a cSDG are disjoint, the computation of the sequential chops using the RRC is in $\mathcal{O}(|Edges| * MaxFormalIns)$. Thus CSC's worst-case runtime complexity is in $\mathcal{O}(|Edges| * MaxFormalIns)$.

Figure 4 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 interference edges: The source criterion is formed by the sink nodes $T_I$ of all interference edges in $I$ plus the original source criterion, and the target criterion is formed by the source nodes $S_I$ of all interference edges in $I$ plus the original target criterion. In our example, the interference 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 interference edge in $I$ belongs to the context-sensitive chop. Intuitively, an interference edge traversal towards node $n$ reaches every possible instance of $n$. If an interference edge $m \rightarrow_{id} n$ 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 2, there exists a context-sensitive path from $s$ to $t$ via interference edge $m \rightarrow_{id} n$.

**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. 4. For every node $n$ in $G$ the following holds:*
$n \in CSC(s, t) \Leftrightarrow \exists$ *a context-sensitive path* $s \rightarrow^* n \rightarrow^* t$

### 3.2 Time-sensitive 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. Consider the example in Fig. 2. The grey shaded nodes are the context-sensitive chop $CSC(9, 13)$. That chop contains two time travels: As stated in section 2.1, node 14 cannot influence node 13. Similarly, node 9 cannot influence nodes 2, 5 and 8, 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. 2, this technique would determine the dark grey shaded nodes as the chop for $(9, 13)$, which is context- and time-sensitive. However, this technique does not always compute time-sensitive chops, for which Fig. 5 provides an example. The depicted graph shows the chop from node 2 to node 3, if computed as described above (for better readability, some control dependences irrelevant to the chop are not shown). 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,

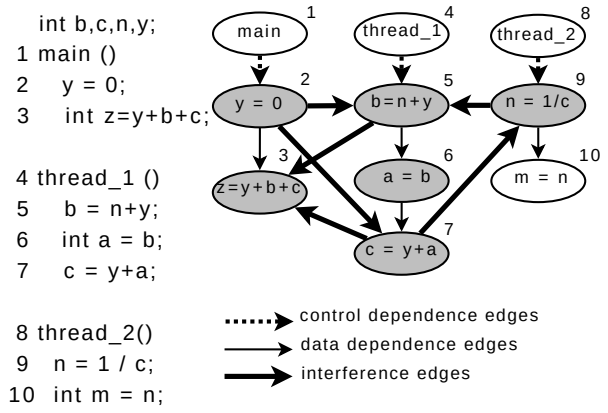


**Figure 5. Intersection is time-insensitive**

computed on these nodes[2], 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 $7 \rightarrow 9 \rightarrow 5$, 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 straight-forward solution is to inspect every possible path in the chop for time travels. Fortunately, there is an easier and more efficient solution. For that purpose, we have to explain how time travel detection works. We start by defining time-sensitive paths, following the work of Krinke [11].

A *context* $c$ consists of a node $n$, annotated with a *call string* of $n$. The call string represents the call stack for a certain invocation of $n$'s procedure. A context $c$ *reaches* another context $c'$, if $c'$ can be executed after $c$, according to control flow[3]. The 'reaches' relation identifies sequences of contexts that can be executed without time travels.

**Definition 4.** *(Threaded witness [11]) 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. Contexts can further be used to traverse SDGs in a context-sensitive manner, by increasing or decreasing the call stack when entering or leaving a procedure. A path $c_1 \rightarrow^* c_k$ of contexts, traversed by that technique, is context-sensitive [9]. Nanda and Ramesh [14] as well as Krinke [11] employ that technique to determine and propagate contexts during the slice. A path of contexts in a cSDG is *time-sensitive* if it is context-sensitive and there exists an adequate threaded witness:

[2]The same optimization as used in I2PC, to omit the intersection.

[3]In the control flow graph, conditional branching is treated as nondeterministic branching to make static analysis of the 'reaches' relation decidable. Algorithms for computing 'reaches' are described in [3, 14].

**Definition 5.** *(Time-sensitive paths [11]) 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* $\langle c_1, ..., c_k \rangle$*.*

We define a time-sensitive chop as follows:

**Definition 6.** *(Time-sensitive chop) A time-sensitive chop in a cSDG* $G$ *for a chopping criterion* $(s, t)$ *consists of the set of nodes*

$$\left\{ n \;\middle|\; \begin{array}{l} \exists \text{ a time-sensitive path } c_s \rightarrow^* c_n \rightarrow^* c_t \text{ in } G: \\ c_n \text{ is context of } n, c_s \text{ is context of } s, c_t \text{ is context of } t \end{array} \right\}$$

Let us go back to the *almost time-sensitive chopper* and examine why it 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 interference edges. For example, a backward slice creates a time travel if it reenters a thread such that the reentering context $c_j$ cannot reach the context $c_i$, where the thread was left. This happens in Fig. 2 during the backward slice for node 13, at the traversal from node 2 to node 14: Thread 1 has been left towards `main` at node 12, but node 14 cannot execute before 12, thus it cannot influence node 12 and node 13. To detect these time travels, Krinke's as well as Nanda and Ramesh's slicing algorithms memorize and propagate *thread execution states* during the slice. A thread execution state maps each thread to a context, which represents the point where that thread was visited last. Fig. 6 shows pseudo code for a time-sensitive backward slicer, which performs the following steps: In an initial thread execution state $e_0$, all threads are mapped to an initial state $\perp$, which is reachable from all contexts by definition. Then, all possible contexts $C_s$ of $s$ are determined. For every $c_s \in C_s$, $s$ is annotated with $c_s$ and a thread execution state, where the state of $s$'s thread is mapped to $c_s$. These annotated nodes are inserted into a worklist $W$, and the slicing algorithm iterates over that worklist, until it is empty. If the slicing algorithm traverses an edge, which is not an interference edge, towards a node $m$, it computes the thread execution state of $m$ by copying $n$'s thread execution state and mapping $m$'s thread to the according context of $m$. The annotated node is inserted into $W$. If the slicing algorithm is about to traverse an interference edge $m \rightarrow_{id} n$, it checks every possible context of $m$ if it can reach the context for $m$'s thread in the thread execution state of $n$. If so, $m$ is annotated with that context and the according thread execution state and inserted into $W$. Else, that traversal would result in a time travel and is omitted. Note that a node can be visited multiple times, as long as the annotations differ. Time-sensitive forward slicing works accordingly.

Let us take a look at the thread execution states computed by the chop $ATSC(2, 3)$ in Fig. 5. Every node

**Input:** A slicing criterion s.
**Output:** The slice for s.

*let* $\theta(n)$ *return the thread of node* $n$
*let* $\texttt{NewState}(e, c)$ *return a thread execution state* $e'$ *by mapping the thread of context* $c$ *in state* $e$ *to* $c$

$e_0 = [\bot, ..., \bot]$ *// every thread is in a nonrestrictive state*
$W = \{(s, c, e) \mid c$ *is a context of* $s \wedge e = \texttt{NewState}(e_0, c)\}$
$M = \{w \mid w \in W\}$ *// a list for marking the contents of W*

```
while W ≠ ∅
  W = W \ {(n, c, e)} // remove next element (n, c, e) from W
  // backward traversal
  foreach m →_f n
    if f ∈ {id} // interference edge
      // Let C_m contain all possible contexts of m
      foreach c_m ∈ C_m
        if c_m reaches e[θ(m)] // detect time travels
          w = (m, c_m, NewState(e, c_m))
          if w ∉ M
            W = W ∪ {w}
            M = M ∪ {w}
    else
      // Let c_m be the reached context of m
      w = (m, c_m, NewState(e, c_m))
      if w ∉ M
        W = W ∪ {w}
        M = M ∪ {w}
return {n' | ∃(n', c', e') ∈ M}
```

**Figure 6. A time-sensitive backward slicer**

has only one context, so we represent it simply by the node itself. The initial thread execution state 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 grey highlighted nodes[4] with thread execution states $\{(3, [3, \bot, \bot]),\ (2, [2, \bot, \bot]),\ (7, [3, 7, \bot]),\ (6, [3, 6, \bot]),\ (5, [3, 5, \bot]),\ (2, [2, 7, \bot]),\ (2, [2, 5, \bot]),\ (9, [3, 5, 9])\}$. The traversal from $(9, [3, 5, 9])$ to node 7 is rejected, because node 5 is not reachable from node 7. The initial thread execution state 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 grey shaded nodes with thread execution states $\{(2, [2, \top, \top]),\ (3, [3, \top, \top]),\ (5, [2, 5, \top]),\ (6, [2, 5, \top]),\ (7, [2, 7, \top]),\ (3, [3, 5, \top]),\ (3, [3, 7, \top]),\ (9, [2, 7, 9])\}$. The traversal from $(9, [2, 7, 9])$ to node 5 is rejected, because node 5 is not reachable from node 7.

We observe the following property of thread execution states: Let $E_{back}(c, t)$ be the set of thread execution states in which the time-sensitive backward slice for a node $t$ visits a context $c$. These thread execution states indicate if in a certain program execution $c$ may influence $t$: If the program execution reaches $c$ with a thread execution state $e$, then $E_{back}(c, t)$ must contain a thread execution state $e_{back}$,

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

**Input:** A chopping criterion $(s, t)$.
**Output:** The chop for $(s, t)$.

*let* $\theta(n)$, $\texttt{NewState}(e, c)$ *be defined as in Fig. 6*
*// call the slicer in Fig. 6 and retrieve its computed thread execution states*
$E_{back} =$ *the set* $M$ *in Fig. 6 after computation of* $\texttt{slice}(t)$
$e_0 = [\top, ..., \top]$ *// every thread is in a restrictive state*
$W = \{(s, c, e) \mid c$ *is a context of* $s \wedge e = \texttt{NewState}(e_0, c)\}$
$M = \{w \mid w \in W\}$ *// a list for marking the contents of W*

```
while W ≠ ∅
  W = W \ {(n, c, e)} // remove next element (n, c, e) from W
  // forward traversal
  foreach n →_f m
    if f ∈ {id} // interference edge
      // Let C_m contain all possible contexts of m
      foreach c_m ∈ C_m
        if c_m reaches e[θ(m)] // detect time travels
          w = (m, c_m, NewState(e, c_m))
          // check if the traversal results in a time-sensitive path
          if w ∉ M∧ restrictive(w, E_back)
            W = W ∪ {w}
            M = M ∪ {w}
    else
      // Let c_m be the reached context of m
      w = (m, c_m, NewState(e, c_m))
      // check if that traversal results in a time-sensitive path
      if w ∉ M∧ restrictive(w, E_back)
        W = W ∪ {w}
        M = M ∪ {w}
return {n' | ∃(n', c', e') ∈ M}

procedure restrictive((n, c, e), E) :
  foreach (n, c, e') ∈ E
    if e is restrictive to e'
      return true
  return false
```

**Figure 7. TSC: A time-sensitive chopper**

such that no thread has executed further in $e$ than in $e_{back}$. Otherwise, $c$ cannot influence $t$ in this state of execution. Assume that progam execution reaches node 9 with thread execution state $[3, 6, 9]$, then it is impossible for node 9 to influence node 3 in that program run. The program execution must not exceed the thread execution state $[3, 5, 9]$ before reaching node 9. Formally, we require that $e$ is *restrictive* to at least one thread execution state in $E_{back}(c, t)$.

**Definition 7.** *(Restrictive thread execution states [14]) Let* $e = [c_1, .., c_k]$ *and* $e' = [c'_1, ..., c'_k]$ *be two thread execution states.* $e$ *is* restrictive *to* $e'$*, iff* $\forall 1 \leq i \leq k : c_i$ *reaches* $c'_i$*.*

Similarly, let $E_{forw}(c, s)$ be the set of thread execution states in which the time-sensitive forward slice for a node $s$ visits a context $c$. These thread execution states indicate if in a certain program execution $s$ may influence $c$: If the program execution reaches $c$ with a thread execution state $e$, then $E_{forw}(c, s)$ must contain an element $e_{forw}$, such that every thread in $e$ has executed at least as far as in $e_{forw}$. Otherwise, $c$ cannot be influenced by $s$ in this state of execution. Assume again that progam execution reaches node 9

with thread execution state $[3, 6, 9]$, then it is impossible for node 2 to influence node 9 in that program run – `thread_1` must have reached node 7 to create a possible influence.

We transfer this observation to chopping. Assume that the chopping algorithm ATSC visits context $c$ with thread execution states $E_{back}(c, t)$ during the backward slice and thread execution states $E_{forw}(c, s)$ during the forward slice. There must exist thread execution states $e_{forw} \in E_{forw}(c, t)$, $e_{back} \in E_{back}(c, s)$, 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 thread execution state $e$, 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. 7. Called for a chopping criterion $(s, t)$, it first calls the backward slicer of Fig. 6 for $t$ and retrieves the visited thread execution states. Then it performs a forward slice for $s$, which is basically dual to the backward slice algorithm, except for the two calls of procedure `restrictive`. That procedure checks if a given thread execution state of a context is restrictive to at least one thread execution state, in which the backward slicer visited that context. If that is not the case, it returns 'false' and the forward slicer does not traverse towards that context.

**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 $G$, where $c_n$ is a context of $n$, $c_s$ is a context of $s$ and $c_t$ is a context of $t$.*

## 4 Implementation and evaluation

We have implemented all five chopping algorithms in Java. We realized the algorithms ATSC and TSC using Nanda and Ramesh's slicing algorithm [14], enriched with several recently developed optimizations [3], which currently seems to be the fastest time-sensitive slicer for concurrent programs [3]. We further employ another optimization for ATSC and TSC, which first computes 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. All implemented algorithms work on cSDGs computed by Hammer's dataflow analysis for Java programs [4]. We used a 2.2Ghz Dual-Core AMD workstation with 32GB of memory running Ubuntu 7.10 (Linux version 2.6.22) and Java 1.7.0.

Our benchmark consists of 12 programs shown in Table 1. 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 are real JavaME[5] applications taken from the SourceForge repository[6]. These programs have graphical user interfaces running as separate threads. Table 1 reports the number of nodes, edges and procedures (Proc.) in the respective cSDGs. Column 'Threads' indicates how many different thread classes a program contains (subclasses of `java.lang.Thread`, plus the main thread). The values in brackets arranged behind denote the number of instances of these threads 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 (how to handle such threads is described in previous work [3]). For example, ProdCons consists of its main thread, which has one instance, and a second thread, of which an arbitrary number of instances may exist at runtime.

| Name | Nodes | Edges | Proc. | Threads (Inst.) |
|---|---|---|---|---|
| Example | 1687 | 6148 | 41 | 2 (1, 1) |
| ProdCons | 2217 | 8775 | 39 | 2 (1, ∞) |
| DiningPhils | 2973 | 11331 | 43 | 2 (1, ∞) |
| AlarmClock | 4085 | 13842 | 74 | 3 (1, 2, 1) |
| LaplaceGrid | 10022 | 100730 | 95 | 2 (1, ∞) |
| SharedQueue | 17998 | 139480 | 122 | 2 (1, ∞) |
| Logger | 9576 | 50800 | 225 | 2 (1, 1) |
| Maza | 10590 | 60021 | 261 | 2 (1, 1) |
| Barcode | 11025 | 67849 | 229 | 2 (1, 1) |
| Guitar | 13459 | 89724 | 307 | 2 (1, 1) |
| J2MESafe | 15666 | 127922 | 256 | 2 (1, 1) |
| Podcast | 23399 | 191849 | 404 | 3 (1, 1, 1) |

**Table 1. The benchmark programs**

In our evaluation, we measured the average chop sizes and performance of our chopping algorithms, the results are shown in Table 2. We determined the number of computed chops as follows: For the first three programs in Table 2, we generated every chopping criterion consisting of one source node and one target node. Then, in a preprocessing step, we removed every criterion for which IC determined an empty chop. As the time-sensitive algorithms would process these chops very fast due to the optimization employing the I2PC chopper mentioned at the beginning of this section, including these chopping criterions would skew the runtime measurements in favor of the time-sensitive algorithms. We only generated every 100th chopping criterion for the next six programs in Table 2, because otherwise the number of chopping criteria would get too big. As the performance of the time-sensitive algorithms declined, we only generated every 100,000th chopping criterion for the last three

[5]The Java Mobile Edition for mobile devices.

[6]http://sourceforge.net/

| Name | chops | IC | I2PC | CSC | ATSC | TSC | IC | I2PC | CSC | ATSC | TSC |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Example | 608496 | 255 | 212 | 208 | 140 | 138 | 2.5 | .8 | 4.5 | 4.1 | 4.4 |
| ProdCons | 536341 | 148 | 146 | 145 | 123 | 120 | 2.7 | .9 | 3.8 | 2.7 | 2.7 |
| DiningPhils | 1029379 | 270 | 268 | 267 | 239 | 230 | 3.7 | 1.5 | 6.3 | 4.5 | 4.8 |
| AlarmClock | 65448 | 1516 | 1516 | 1515 | 1047 | 950 | 12.1 | 8.0 | 35.8 | 3422.6 | 1376.4 |
| Logger | 148837 | 985 | 971 | 967 | 796 | 796 | 14.5 | 9.2 | 31.6 | 73.2 | 77.9 |
| Maza | 259804 | 1543 | 1500 | 1153 | 1021 | 798 | 25.9 | 15.1 | 53.6 | 2783.4 | 2568.0 |
| LaplaceGrid | 278942 | 2470 | 2469 | 2469 | 1783 | 967 | 42.9 | 21.0 | 104.4 | 249.6 | 221.7 |
| Barcode | 498849 | 711 | 565 | 541 | 474 | 469 | 14.8 | 7.1 | 16.6 | 100.9 | 88.2 |
| Guitar | 786728 | 1734 | 1624 | 1606 | 1485 | 1476 | 37.7 | 21.6 | 59.9 | 554.0 | 551.2 |
| SharedQueue | 1697 | 8462 | 8462 | 8462 | 8068 | 7969 | 59.2 | 45.4 | 1003.8 | 8846.5 | 9639.7 |
| J2MESafe | 1323 | 4027 | 3752 | 3611 | – | 2423 | 60.4 | 39.1 | 180.0 | – | 7637.8 |
| Podcast | 3677 | 10423 | 10402 | 10400 | – | 2310 | 56.1 | 44.5 | 283.7 | – | 9039.2 |

**Table 2. Left side: avg. size per chop (number of nodes), right side: avg. time per chop (milliseconds)**

programs. ATSC has no entry for J2MESafe and Podcast, because it could not finish these test runs in reasonable time.

**Context-sensitive chopping** The context-sensitive chops computed by the CSC were on average 10% smaller than the imprecise ones computed by IC, and about 25% smaller in the best case (for Barcode and Maza). The CSC also performed well compared to IC and was usually 1 to 3 times slower than IC; only for SharedQueue it was about 17 times slower. The I2PC was surprisingly precise: Its computed chops were almost as small as the context-sensitive ones, the only considerable difference appeared in the Maza program, where the CSC chops were 20% smaller. As I2PC is additionally very fast – about twice as fast as IC in our evaluation – and easy to implement, it seems to be a good choice for a quick deployment.

**Time-sensitive chopping** For several programs, time-sensitive chopping reduced the chop sizes significantly – about 78% for Podcast and about 60% for LaplaceGrid. On average, the ATSC chops were 24% smaller than the IC chops, the TSC chops were even 35% smaller. The evaluation shows that ATSC can miss a considerable number of time travels: For Maza, the TSC chops were 20% smaller than the ATSC chops; for LaplaceGrid, they were even 45% smaller. Both ATSC and TSC were very fast on our smaller programs; often faster than CSC (e.g. for DiningPhils) and almost as fast as IC. For the bigger programs, performance declined as expected, due to their worst-case exponential runtime behaviour. For several programs – AlarmClock, Maza, SharedQueue, J2MESafe and Podcast – this worst-case behaviour became noticeable. As TSC gains smaller chops than ATSC and is able to outperform ATSC due to its increased precision, we suggest employing TSC for time-sensitive chopping.

**On the practicability of TSC** 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 with different thread execution states. Nanda and Ramesh thus determined a worst-case complexity of $\mathcal{O}(|Nodes|^{p^t})$ [14], where $p$ is the calling depth of the call graph, $|Nodes|^p$ is an upper bound for the contexts, and $t$ is the number of thread instances (threads with possibly infinite instances can be approximated conservatively by one representative in the thread execution states [3]). Since TSC basically computes one forward and one backward slice, it bears the same worst-case runtime complexity. The present evaluation and our recent evaluation of time-sensitive slicing [3] indicate that time-sensitive slicing and chopping, using the optimizations developed so far [3, 11, 14], 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.

## 5 Related work

Chopping originates from Jackson and Rollins' work on modularizing SDGs for reverse engineering [8]. 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* [17], because it does not take callers of the initial procedure into account.

Reps and Rosay extend Jackson and Rollins' same-level chopping to unbounded chopping, where source and target can be in different procedures [17]. Their chopping algorithm, the RRC described in section 2.2, is context-sensitive and the state-of-the-art algorithm for sequential programs.

Krinke developed a new same-level chopper called

*summary-merged chopper* [9], which is context-sensitive, as the iterative approach of Jackson and Rollins, but much faster in practice. In a subsequent work, Krinke introduces the concept of *barrier chopping* and *slicing* [10], 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 was the first to address the time travel problem for slicing of concurrent programs [11, 12]. Nanda and Ramesh developed an alternative algorithm containing several optimizations that strongly reduce the combinatorial explosion of thread execution states [14]. 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 [3].

Whereas Krinke assumes that all threads execute entirely in parallel, Nanda uses a more precise model of concurrency on the level of fork and join points of threads. The effects of concurrency models on slicing, in particular for identifying spurious interference edges, are significant, as shown by several authors [2, 3, 14, 16]. More precise models of concurrency, incorporating a synchronization analysis [6] or a full-fledged may-happen-in-parallel analysis [15], could further increase precision.

## 6 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 five 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 78%. 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. Thus, we recommend time-sensitive chopping mainly as a pre-processing step for more expensive analyses.

## References


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

[2] Z. Chen, B. Xu, H. Yang, K. Liu, and J. Zhang. An approach to analyzing dependency of concurrent programs. In *Proc. APAQS'00*, page 39, 2000.

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

[4] C. Hammer and G. Snelting. An improved slicer for Java. In *Proc. PASTE'04*, pages 17–22. ACM Press, 2004.

[5] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Tech. Rep. 2008-16, Universität Karlsruhe (TH), Germany, 2008.

[6] J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM primitives. *Static Analysis Symposium*, pages 1–18, 1999.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

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

[9] J. Krinke. Evaluating context-sensitive slicing and chopping. In *Proc. ICSM'02*, page 22, 2002.

[10] J. Krinke. Barrier slicing and chopping. In *IEEE Workshop on Source Code Analysis and Manipulation*, 2003.

[11] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proc. ESEC/FSE'03*, pages 178–187. ACM Press, 2003.

[12] J. Krinke. Static slicing of threaded programs. In *Proc. PASTE'98*, pages 35–42, June 1998.

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

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

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

[16] V.P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent Java programs. In *Proc. CC'04*, pages 39–56, 2004.

[17] T. Reps and G. Rosay. Precise interprocedural chopping. In Proc. FSE'95, pages 41–52. ACM Press, 1995.

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

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

[20] F. Tip. A survey of program slicing techniques. *Journal of Prog. Lang.*, 1(3):121–189, 1995.

# A APPENDIX

The appendix consists of two parts. The first part provides proofs for theorems 1 and 2. In the second part we show that the RRC can be extended to compute context-sensitive chops for chopping criteria consisting of sets of nodes, by extending the underlying slicers to compute slices for sets of nodes. This extension provides the same asymptotic running time as the original algorithm.

## A.1 Context- and Time-sensitive Chopping

This part provides proofs for theorems 1 and 2. We assume in these proofs that the RRC is context-sensitive for sequential programs (see also appendix A.2), and that the slicing algorithms employed in ATSC and TSC are time-sensitive.

**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. 4. 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.*

'$\Leftarrow$' We have for every node $v \in CSC(s,t)$ that there exist nodes $s'$ and $t'$ 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 there exists a context-sensitive path $q : s \rightarrow^* s' \rightarrow^* v \rightarrow^* t' \rightarrow^* t$ with $s' \rightarrow^* v \rightarrow^* t' = p$. We distinguish four cases:

1. $s = s' \wedge t = t'$
   In this case, $p = q$ and thus is trivially a context-sensitive path.
2. $s = s' \wedge t \neq t'$
   According to the creation of set $T$ of the chopping criterion in Fig. 4, there exists a context-sensitive path $t' \rightarrow t'' \rightarrow^* t$, such that $t' \rightarrow t''$ is an interference edge. Thus $t' \rightarrow t'' \rightarrow^* t$ has the form $id\ (realizable\ id)^*\ realizable$. 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. 4, there exists a context-sensitive path $s \rightarrow^* s'' \rightarrow s'$, where $s'' \rightarrow s'$ is an interference edge. Thus it has the form $(realizable\ id)^*\ id$. 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.

'$\Rightarrow$' 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 an interference edge, and $t'$ is either $t$ or succeeded by an interference edge. We have to show that $s' \in S$ and $t' \in T$. In that case, the algorithm is guaranteeed 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 that $s' \in S$ in the same way.

□

In order to prove theorem 2, we use the following auxiliary lemma.

**Lemma 1.** *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 thread execution states $e \in E_{forw}(c,s), e' \in E_{back}(c,t)$, such that $e$ is restrictive to $e'$, then there exists a time-sensitive path $c_s \rightarrow^* c \rightarrow^* c_t$, where $c_s$ is a context of $s$ and $c_t$ is a context of $t$.*

*Proof.* We have that there exists a context- and time-sensitive path $c_s \rightarrow^* c$, traversed by the forward slicer, such that $c$ is reached with thread execution state $e$, and a context- and time-sensitive path $c \rightarrow^* c_t$, traversed by the backward slicer, such that $c$ is reached with thread execution state $e'$. It follows that the path $c_s \rightarrow^* c \rightarrow^* c_t$ is context-sensitive. 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 iterate over the path $c \rightarrow^* c_t$ without confronting a time travel.

We start our traversal at $c$ and 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 have 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 an intra-thread 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 thus $c_{j+1}$, because 'reaches' is a transitive relation, 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 an interference 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 $c_{j+1}$, because 'reaches' is a transitive relation, 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 $\theta$ of $c_{j+1}$ was not visited yet during our traversal. This means that the current execution state of $\theta$ is the same as the state of $\theta$ 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 $\theta$'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 $\theta$, the last visit of $\theta$ during the backward slice was at context $c_{j+1}$. According to the propagation rules of thread execution states, $e'[\theta] = c_{j+1}$. Thus, $e[\theta]$ reaches $c_{j+1}$.

□

**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 $G$, where $c_n$ is a context of $n$, $c_s$ is a context of $s$ and $c_t$ is a context of $t$.*

*Proof.*

'$\Rightarrow$' $n$ is only in the chop if there exists a context $c$ of $n$ that was visited by the chopper, and $c$ 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 1, there exists a time-sensitive path $c_s \rightarrow^* c_n \rightarrow^* c_t$ in $G$, where $c_n$ is a context of $n$, $c_s$ is a context of $s$ and $c_t$ is a context of $t$.

'$\Leftarrow$' 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 $e'_{back}$.

□

## A.2 The Reps-Rosay Chopper for Sets of Nodes

In this part we show that the RRC can be extended to compute context-sensitive chops for chopping criteria consisting of sets of nodes, by extending the underlying slicers to compute slices for sets of nodes. This extension provides the same asymptotic running time as the original algorithm.

Following grammar $H$ in definition 1 (section 3.1), let $m \rightarrow^*_{unbr} n$ denote a SDG path which is *unbalanced-right*, i.e. it can be generated from nonterminal *unbal_right*. Similarly, let $m \rightarrow^*_{unbl} n$ denote an *unbalanced-left* path, i.e. it can be generated from nonterminal *unbal_left*. Reps and Rosay define the following operations to compute context-sensitive chops in SDGs [17]:

- $f_{unbr}(S) = \{n \mid \exists s \in S : s \rightarrow^*_{unbr} n\}$
- $f_{unbl}(S) = \{n \mid \exists s \in S : s \rightarrow^*_{unbl} n\}$
- $b_{unbr}(T) = \{n \mid \exists t \in T : n \rightarrow^*_{unbr} t\}$
- $b_{unbl}(T) = \{n \mid \exists t \in T : n \rightarrow^*_{unbl} t\}$

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 [17].

The RRC further needs a function $SLC(e_s \rightarrow e_t)$, which receives an edge $e_s \rightarrow e_t$ and computes a same-level chop [9, 17] from $e_s$ to $e_t$. However, its concrete functionality is irrelevant for this proof. The RRC performs the following 3 steps to compute a chop $RRC(s,t)$ [17]:

1. $W = f_{unbr}(\{s\}) \cap b_{unbl}(\{t\})$,
2. $Chop = (f_{unbr}(\{s\}) \cap b_{unbr}(W)) \cup (f_{unbl}(W) \cap b_{unbl}(\{t\}))$,
3. for every summary edge $e$ on unbalanced-right paths from $s$ to $W$ or unbalanced-left paths from $W$ 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. $W = f_{unbr}(S) \cap b_{unbl}(T)$,
2. $Chop = (f_{unbr}(S) \cap b_{unbr}(W)) \cup (f_{unbl}(W) \cap b_{unbl}(T))$,
3. for every summary edge $e$ on unbalanced-right paths from $S$ to $W$ or unbalanced-left paths from $W$ 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 $s \in S, t \in T$. As all employed operations remain the same, this extension has the same asymptotic running time as the original algorithm.

**Lemma 2.** $RRC(S,T) = \bigcup_{\substack{s \in S \\ t \in T}} RRC(s,t)$

*Proof.*

'$\Leftarrow$' 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 $b_{unbl}$.

'$\Rightarrow$' 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 inserted into the chop either in step 2 or in step 3.

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

□