Slicing Shared-Memory Concurrent Programs
The Threaded System Dependence Graph Revisited

Carlos Galindo, Marisa Llorens, Sergio Pérez and Josep Silva*
Valencian Research Institute for Artificial Intelligence
Universitat Politècnica de València
Valencia, Spain
{cargaji,mllorens,serperu,jsilva}@dsic.upv.es

***Abstract*—Program slicing is a program analysis technique to identify the parts of a program that can influence the values computed at a given program point. Its application to concurrent programs with shared memory revealed a new program dependence between threads called *interference* and uncovered some problems associated with concurrent executions such as *time travel*. To solve these problems, a new program representation for slicing concurrent programs was proposed: the *threaded System Dependence Graph* (tSDG), which aimed at solving the time travel problem and to provide context-sensitive slices for concurrent programs. In this paper, we show that the problem remains unsolved because the tSDG is not context-sensitive in some situations. We give a counterexample for the tSDG and identify situations where the tSDG is imprecise, generating context-insensitive slices for concurrent programs. To solve these imprecisions, we redesign the tSDG to become context-sensitive in those situations, solving the inaccuracy problem and preserving the solution to time travel. The new program representation is always as precise as the previous tSDG, and sometimes it is more precise. That is, the slices computed with the new graph are always smaller or equal than those computed with the previous tSDG.**

***Index Terms*—Program analysis, Program slicing, Concurrency, Threads, System Dependence Graph**

I. Introduction

Program slicing [23] is a program analysis technique that allows us to extract the subset of a program (called a *slice*) that influences or is influenced by a given program point (usually a variable in a specific program statement) called *slicing criterion*. Program slicing was originally thought for debugging [23], [6], [21] (the slicing criterion is often a symptom of a bug, and the slice is the only part of the program that can contain that bug). Besides debugging, the decomposition of programs into data-flow and control-flow analyses used in program slicing has proven useful for many other applications such as software maintenance [9], program comprehension [2], [22], program specialization [19], code obfuscation [16], or information flow control [10], among others.

This work has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/ 10.13039/501100011033, by the *Generalitat Valenciana* under grant CIPROM/2022/6 (FassLow), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Carlos Galindo was partially supported by the Spanish Ministerio de Universidades under grant FPU20/03861.

In order to compute slices for sequential programs, most techniques rely on a graph representation of the program called the *System Dependence Graph* (SDG) [13], where nodes represent program statements and edges represent dependences between them. The SDG was a milestone in the area of program slicing due to its potential for efficient context-sensitive analysis of programs. An analysis is context-sensitive if it reaches a procedure call, then enters the procedure, makes the proper analysis inside the procedure, and finally, it is able to exit the procedure and return only to the specific call from which it entered, so the context (variables, arguments, dependences, etc.) is preserved during the analysis. In Section VI, we explain the evolution of this area and how different solutions were proposed to slice concurrent programs. Nowadays, the most advanced and accurate approach is the one by Nanda et al. [18] who defined a new graph called *threaded System Dependence Graph* (tSDG) and a new slicing algorithm that produces complete context-sensitive slices. The tSDG effectively and elegantly detects impossible executions (the so-called 'time travel' problem), obtaining more precise slices than their predecessors.

Nevertheless, in this paper we show that the problem has not been solved yet. Specifically, we show that the tSDG produces inaccurate slices in different situations because it is not always context-sensitive. Example 1 shows a counterexample designed to illustrate the inaccuracy of the tSDG.

Example 1 (Inaccurate slice produced by the tSDG): Consider the program in Figure 1a, where a thread `T0` launches two threads `T1` and `T2` that are executed in parallel. Consider also variable `a` in line 7 as the slicing criterion (represented as $\langle 7, a\rangle$). A program slicer can determine what parts of the code can influence the slicing criterion: the value of variable `a` in line 7 (`T1`) may come from the definition of `a` in line 13 executed (i) during the call `f(4)` in line 6 (`T1`); or (ii) during the call `f(5)` in line 9 (`T2`) (depending on which was executed last). Therefore, the slice (in black) does not contain lines 2, 3, and 5 (in gray) because they cannot influence the slicing criterion. This result can only be obtained by a *context-sensitive* analysis. A *context-insensitive* analysis would include lines 3 and 5 in the slice because they can influence lines 12 and 13 and, thus, the slicing criterion.

The slice in Figure 1b has been computed with a tSDG.

In this slice, two calls to procedure `f` (lines 6 and 9) are correctly included in the slice together with the code in procedure `f`; but the calls to procedure `f` in lines 3 and 5 in `T0` and `T1` respectively, are unnecesarily included in the slice. As explained before, even though these calls actually define variable `a`, their values are always overwritten before the execution of the slicing criterion (either by the call in line 6 or the one in line 9) preventing them from influencing it, but the tSDG includes them as a context-insensitive analysis would do.

Example 1 is our first contribution: a counterexample that reveals the imprecision of the tSDG. It is not always context-sensitive and may unnecessarily include in the slice procedure calls that cannot influence the slicing criterion. In this work we propose an alternative graph model (a new tSDG) that solves this imprecision. It is worth to remark that, despite receiving the same name, our tSDG is not a modification over Nanda et al.'s tSDG, but a new graph built from the SDG of a program. To build our new representation, we take advantage of the good properties and principles of the tSDG, but we change the way in which *interference* dependence is handled. In the new tSDG, interference dependences are represented using the same principle of summary edges in the SDG: the idea is to represent all interference dependences that connect statements in different procedures through their call statements, thus making all interference dependences intraprocedural (as it happens with flow dependence). This transformation allows us to use the standard two-phase SDGs algorithm by Horwitz et al. [13], enhanced with Nanda et al.'s approach to avoid time travel. The main contributions of this paper are the following:

- The rationale behind the slicing inaccuracy in Nanda et al.'s tSDG (Section III).
- A new tSDG where interferences are only intraprocedural together with its construction algorithm (Section IV).
- A new backward static slicing algorithm that unifies the context-sensitive properties of the original slicing algorithm proposed by Horwitz et al. and the time travel management of Nanda et al.'s algorithm to obtain context-sensitive slices of concurrent programs in the situations where the old tSDG was not able to (Section V).

```
1  // Thread T0
2  a = 1;
3  call f(2);
4  cobegin{ // Thread T1
5    call f(3);
6    call f(4);
7    print(a);
8  }{ // Thread T2
9    call f(5);
10 }
11
12 procedure f(x)
13   a = x;
```

(a) Expected slice

```
1  // Thread T0
2  a = 1;
3  call f(2);
4  cobegin{ // Thread T1
5    call f(3);
6    call f(4);
7    print(a);
8  }{ // Thread T2
9    call f(5);
10 }
11
12 procedure f(x)
13   a = x;
```

(b) Old tSDG slice

Fig. 1: Minimal slice (left) and old tSDG slice (right) w.r.t. slicing criterion $\langle 7, a \rangle$.

II. Background

This section recalls the fundamentals of program slicing, presenting the standard program representations used to slice both sequential and concurrent programs. In this paper, we consider concurrent programs with shared memory. Threads interact accessing common variables and there is no other synchronization mechanism between threads. Threads are created via the classical `cobegin` language construct (see Figure 1).

Since it was defined in 1988 by Horwitz et al. [12], the *System Dependence Graph* (SDG) is the program representation used in most program slicing techniques. The SDG is the result of an iterative refinement over an initial program representation resource, the *Control-Flow Graph* (CFG). We explain it through its incremental evolution:

$$CFG \rightarrow PDG \rightarrow SDG$$

CFG [1]. It is a directed graph that represents all possible execution paths of a program procedure. In the CFG, statements are represented with nodes, and a node m is connected to a node n if there is a possible execution where n is immediately executed after m. The CFG additionally includes two extra nodes, *Enter* and *Exit*, that represent the initial and final nodes of the procedure execution.

The threaded version of the CFG is called *threaded Control-Flow Graph* (tCFG). In the tCFG the beginning and end of each thread in `cobegin` blocks are modelled with two extra nodes labelled as *Start* and *End*, analogously to *Enter* and *Exit* nodes in CFGs, embedded in the surrounding CFG of the procedure that contains them.

PDG [7]. The *Program Dependence Graph* (PDG) contains the same nodes as the CFG after removing the *Exit* node. The edges of the PDG represent two different types of dependences between program statements: *control dependence* and *flow dependence*, both derived from the CFG, and defined hereunder.

Definition 1 (Control Dependence): Let G be a CFG. Let m and n be nodes in G. A node m is post-dominated by a node n in G if every path from m to the *Exit* node passes through n. Node n is *control dependent* on node m if and only if n post-dominates one but not all of m's CFG successors.

Definition 2 (Flow Dependence): A node n is *flow dependent* on a preceding node m if:

(i) m defines a variable v,
(ii) n uses v, and
(iii) there exists a control-flow path from m to n where v is not redefined.

The concurrent counterpart of the PDG is the *threaded Program Dependence Graph* (tPDG). The tPDG contains the same nodes as the tCFG after removing not only the *Exit* node, but also the *End* nodes of all threads. As the PDG, the tPDG includes control and data edges, but it also includes additional edges that represent a new type of dependence exclusive to concurrent programs called *interference dependence*.

Definition 3 (Interference Dependence [18]): A node n is *interference dependent* on a node m if m defines a variable v, n uses the variable v, and n and m can execute in parallel.

Interference dependence differs from control and flow dependences because it is intransitive. If a node n_3 is interference dependent on a node n_2 and, in turn, n_2 is interference dependent on a node n_1, n_3 is interference dependent on n_1 only if there is a possible execution sequence $\langle n_1, n_2, n_3 \rangle$. This is detected by using the tCFG. All feasible execution sequences are called *threaded witnesses*.

Definition 4 (Threaded witness [18]): Given a tCFG G of a concurrent program, a threaded witness is an ordered sequence of nodes $\langle n_1, n_2, \ldots, n_k \rangle$ belonging to G such that any subsequence of nodes $n_{i_1}, n_{i_2}, \ldots, n_{i_j}$, $j \leq k$, belonging to the same thread, T_i, forms a realizable path in T_i.

Intuitively, a threaded witness can be interpreted as a sequence of tCFG nodes that form a valid execution chronology.

SDG [12]. The PDG is suitable to represent isolated procedures (it is an intraprocedural representation). The SDG is an interprocedural representation. It generalized the PDG by connecting all the procedures of a program in a single graph. A SDG is compositionally constructed by connecting the PDGs of a program's procedures to model parameter passing between procedure calls and procedure definitions. The information transfer between calls and definitions is done through temporary variables.

- Procedure call nodes are augmented with a set of new nodes called *actual-in* and *actual-out*, which are control dependent on the call node. For each argument at the call site, an actual-in node contains an assignment that copies the value from the argument to a temporary variable, and for each argument re-defined during the procedure execution, an actual-out node contains another assignment that copies the temporary variable returned by the procedure back to the caller.
- In a similar way, procedure definition nodes are also augmented with a set of nodes called *formal-in* and *formal-out*, control dependent on the *Enter* node. Formal-in nodes contain an assignment to copy the value from the temporary variable defined at the call site to the corresponding parameter, while formal-out nodes contain an assignment to copy the value of redefined parameters to a new temporary variable.

In this model, flow dependences between different procedures are transmitted throughout *actual* and *formal* nodes in the procedure call. To connect PDGs, three new kinds of edges are defined: (i) *call edges* connect each call site to the corresponding *Enter* node of the procedure, (ii) *input edges* connect actual-in nodes to the corresponding formal-in nodes, and (iii) *output edges* connect formal-out nodes to associated actual-out nodes. It is worth mentioning that, at the end of the construction of this structure, all the flow dependences associated to the values of the arguments in every procedure call are moved to the corresponding actual node increasing the precision of the representation. Finally, a new kind of edge called *summary edge* is added to the SDG to represent the dependences between arguments in procedure calls. A summary edge connects an actual-in node to an actual-out node if the value provided by the actual-in node is used inside the procedure (or one of its transitive callees) to compute the value of the actual-out node. An illustration of an SDG can be seen in Figure 2.

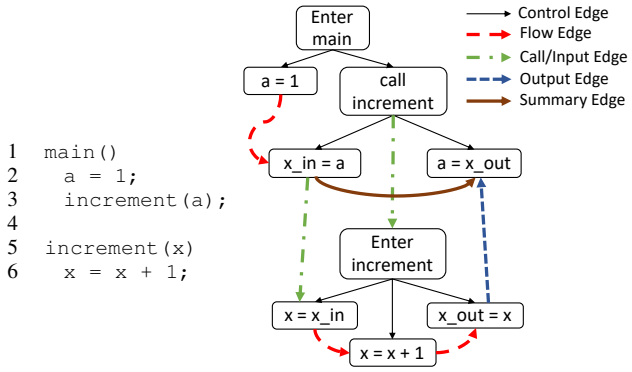

Fig. 2: Procedure call and SDG representation

The standard algorithm [12] to slice a SDG computes program slices by traversing the SDG in two differentiated phases. Both phases traverse all the control and flow edges backwards to reach all possible nodes connected by these dependences, but each phase ignores a specific kind of edge with a particular purpose:

- **Phase 1** ignores output edges. This phase identifies nodes that can reach the slicing criterion, and are either in the same procedure (P) as the slicing criterion itself, or in a procedure that directly or transitively calls P.
- **Phase 2** ignores call and input edges. This phase locates nodes in procedures that are called by (i) P or by (ii) procedures that directly or transitively call P. Since call and input edges are not traversed in this phase, Phase 2 includes in the slice the procedures called during Phase 1, but prevents the inclusion of new calls. This fact makes the algorithm aware of the calling contexts, producing context-sensitive slices.

In the rest of the paper we explain the problems of the current formulation of the threaded version of the SDG, the *threaded System Dependence Graph* (tSDG) proposed by Nanda et al. [18], and we reformulate the model to make it context-sensitive and a natural extension of the SDG with threads.

III. Limitations of the current tSDG

The slices computed with the SDG for concurrent programs may be (i) incomplete or may (ii) travel in time, including statements that, e.g., cannot be executed before the slicing criterion, and obtaining an execution sequence that is not an interprocedural threaded witness (a threaded witness computed using an interprocedural tCFG). To solve these problems, Nanda et al. [18] defined the tSDG, which includes two main changes over the SDG:

1) **The slicing algorithm switches to Phase 1 when an interference edge is traversed**. This solves the

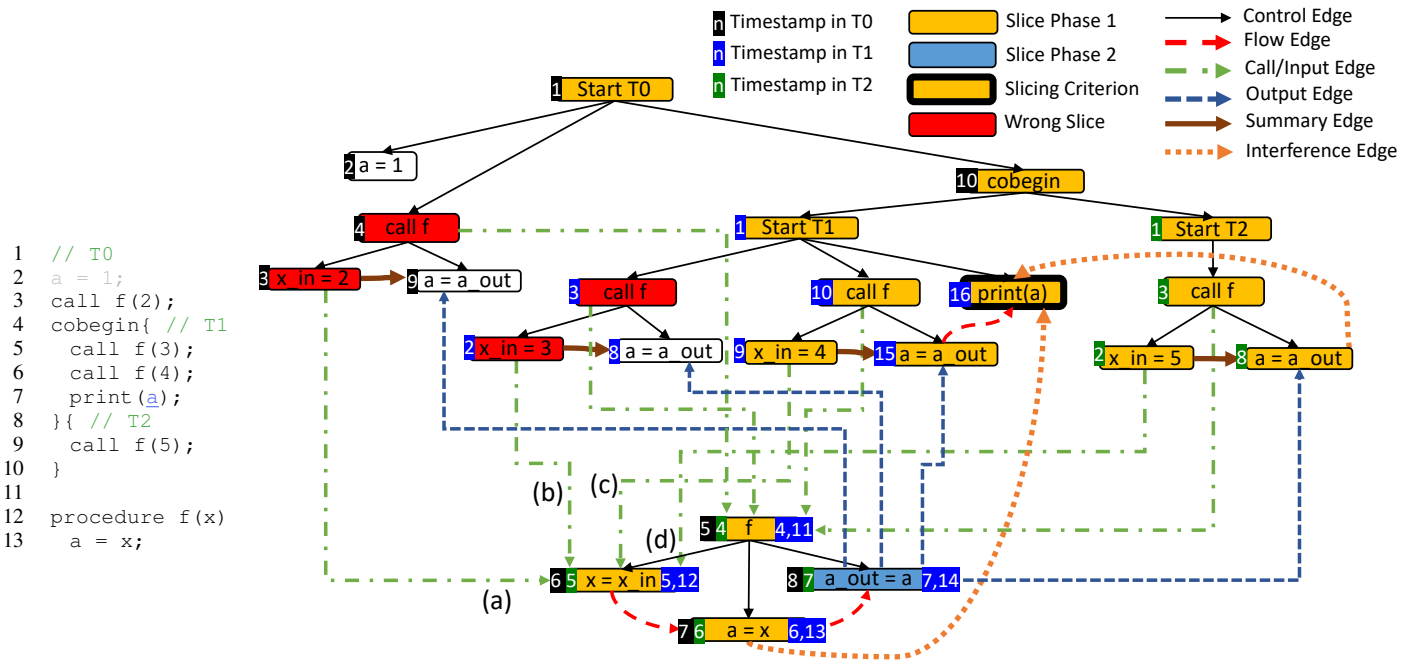

Fig. 3: Nanda et al.'s tSDG of the program in Figure 1

incompleteness problem (cf. Nanda et al. [18, Section 3.1]) because it ensures that all the dependences of the statement that generate an interference are included in the slice when we reach a different procedure throughout an interference edge.

2) **The introduction of timestamps inside tSDG nodes**. These timestamps allow us to solve the time travel problem by detecting realizable paths, thus, ensuring that the traversal of nodes is done using an interprocedural threaded witness.

Unfortunately, the first modification solves the incompleteness problem but introduces a problem of imprecision: the tSDG can generate context-insensitive slices, as shown in Example 1. Example 2 explains, for the code in Figure 1, how the tSDG unnecessarily includes in the slice the calls to procedure `f` in lines 3 and 5.

Example 2: Consider the program in Figure 1 and its associated tSDG shown in Figure 3. The tSDG uses different colours to show the slice computed with Nanda et al.'s algorithm with respect to the slicing criterion $\langle 7, a \rangle$ (the node in bold). The colour of the nodes indicates the phase in which they are included, white nodes are not part of the slice, and red nodes are unnecessarily included in the slice. It can be seen that the algorithm unnecessarily includes irrelevant calls to procedure `f` and their corresponding actual-in nodes `x_in = 2` and `x_in = 3` in the slice. Table I shows the path followed by the slicing algorithm to reach formal-in node `x = x_in` during Phase 1 (which is the cause of including the irrelevant calls to `f`). Each step (or row) represents the traversal of an edge.

In the tSDG, each node is assigned a timestamp that indicates the order in which each node is executed in each thread. The `Traversal State` represents, with timestamps for each thread, what is the chronological state (last nodes executed in each thread).

Step 0 represents the start of the traversal. In this step the slicing criterion is the statement in thread `T1` with timestamp 16. The `Traversal State` is updated by annotating `T1` with the set of timestamps $\{16\}$. Step 1 represents the traversal of the interference edge from the `print(a)` node to statement `a = x` in procedure `f`. This node has four timestamps that are updated in the state. Note that the traversal state is a countdown timer for each thread because the statements are traversed backwards. For this reason, `T1` had timestamp 16 and now it could be 6 or 13 (because there are two calls to `f` in `T1`). Step 2, represents the traversal, inside procedure `f`, of the flow edge to the formal-in node `x = x_in`. After reaching the formal-in, the algorithm finds four possible input edges to traverse labelled with letters from `a` to `d`. In the four cases, the timestamp of the node reached is smaller than the timestamp in the traversal state and there is an interprocedural threaded witness (in the tCFG) from the statement in `New Node` to the one in `Last Node` (`x = x_in`) (recall that we traverse the execution backwards). Therefore, all four nodes could be executed before the last node (in different executions) and, thus, they are (unnecessarily) added to the slice, losing context-sensitivity. Observe in Step 1 that traversing the interference edge leads us to procedure `f`. But, the algorithm exits procedure `f` to all calls to `f` (in any thread) and not to the call from which we entered.

In the next section we solve the imprecision problems of the tSDG. We propose an alternative approach that takes

Step	Last Node	Edge Type	Phase	New Node		Traversal State		
				Statement	Timestamps	T0	T1	T2
0	-	-	1	print(a)	{(T1,{16})}	$\perp$	{16}	$\perp$
1	print(a)	Interference	1	a = x	{(T0,{7}),(T1,{6,13}),(T2,{6})}	{7}	{6,13}	{6}
2	a = x	Flow	1	x = x_in	{(T0,{6}),(T1,{5,12}),(T2,{5})}	{6}	{5,12}	{5}
3a	x = x_in	Input	1	x_in = 2	{(T0,{3})}	{3}	{5,12}	{5}
3b	x = x_in	Input	1	x_in = 3	{(T1,{2})}	{6}	{2}	{5}
3c	x = x_in	Input	1	x_in = 4	{(T1,{9})}	{6}	{9}	{5}
3d	x = x_in	Input	1	x_in = 5	{(T2,{2})}	{6}	{5,12}	{2}

TABLE I: Slicing traversal of the program in Figure 1 from the slicing criterion $\langle 7, a\rangle$ to all the actual-in nodes `x_in = x` in the calls to procedure `f`

advantage of the timestamp mechanism proposed by Nanda et al. to solve time travel; but preserving the standard two-phase slicing algorithm proposed by Horwitz et al. and solving the interference dependence problem when transforming the set of tPDGs in the tSDG, thus being context-sensitive in the situations Nanda et al.'s tSDG was not.

IV. The tSDG revisited

In this section we propose a redefinition of the tSDG in such a way that: (i) It includes a new treatment for interference dependence so that the graph is context-sensitive in situations the old tSDG is not. (ii) It keeps the timestamp design to solve the time travel problem. (iii) It can work with the standard two-phase algorithm, thus, reducing complexity and improving performance.

Interference dependence (see Definition 3) connects definitions and uses of concurrent threads. In some cases, the nodes connected may belong to the body of different procedures (see, e.g., the interference edge in Figure 3). This means that interference can be an interprocedural data dependence, which breaks the principle of the SDG in which all data dependences enter to and exit from procedures throughout formal and actual nodes. This is the fundamental limitation of Nanda et al.'s tSDG. The key point of the new tSDG is to prevent interference edges from freely jumping from the body of one procedure to the body of another one. That is, *interprocedural interference dependence* is not permitted.

One of the main problems of previous approaches is that formal and actual nodes, originally thought to represent dependences in the sequential execution of programs, are also used as source or target of interference edges which represent flow dependences of concurrent executions. This fact results in a mix of sequential and concurrent information managed by the same node, overloading it with conflicting responsibilities and leading to incomplete slices in some cases.

For this reason, the proposal of the new tSDG transfers this knowledge to a new group of nodes that only represent information about the concurrent execution: the formal and actual *interference nodes*. These nodes are analogous to the standard formal and actual nodes in sequential parameter passing, but they model concurrent parameter passing, i.e., they represent the value of a variable used/defined inside a procedure that has been defined/used in a thread executed in parallel:

- Procedure definition nodes are augmented with new *formal-interference-in* (*formal-ifin*) and *formal-interference-out* (*formal-ifout*) nodes, control dependent on the *Enter* node. For each variable v target of an interference dependence whose source is in another procedure, a formal-ifin node is generated with an assignment to copy the value of v (e.g., $v = v\_ifin$). On the other hand, for each variable v source of an interference dependence with target in another procedure, a formal-ifout node contains an assignment to copy any value of v (e.g., $v\_ifout = v$) defined during the execution of the procedure.
- In a similar way, procedure calls are augmented with new *actual-interference-in* (*actual-ifin*) and *actual-interference-out* (*actual-ifout*) nodes, which are control dependent on the call node. At the call site, an actual-ifin node contains an assignment that copies the value of a shared variable v to a temporary variable (e.g., $v\_ifin = v$). Complementarily, an actual-ifout node contains an assignment to copy the value of a shared variable v (e.g., $v = v\_ifout$), defined inside the procedure.

In this model, as it happened with formal and actual nodes, formal and actual interference nodes associated to concurrent parameter passing are also connected by input edges (from an actual-ifin to a formal-ifin) and output edges (from a formal-ifout to an actual-ifout).

The addition of this new structure in procedure calls allows us to separate flow and interference dependences, connecting them to different actual nodes according to their sequential or concurrent nature. In our model, flow dependences are connected to actual-in/out nodes while interference dependences are connected to actual-interference-in/out nodes. We formalize these restrictions over flow and interference dependence as follows: (1) flow dependence ignores variable definitions and uses that occur in interference nodes, and (2) interference dependence ignores definitions and uses in formal/actual nodes, and is exclusively intraprocedural.

Definition 5 (tSDG-Flow Dependence): A node n is *tSDG-flow dependent* on a node m if (i) n is flow dependent on m (see Definition 2) and (ii) neither m nor n are interference nodes.

Definition 6 (tSDG-Interference Dependence): A node n is *tSDG-interference dependent* on a node m if (i) n is interference dependent on m (see Definition 3), (ii) m and

n are in the same procedure, and (iii) neither m nor n are *actual-in / actual-out* nodes.

A. Flow dependences generated by interference nodes

Although interference formal nodes are similar to formal nodes used in sequential parameter passing, the information represented by both is fundamentally different: while sequential formal nodes (formal-in) represent a definition of a variable v that always occur before executing the procedure's code or a use that always occur after executing the whole body of the procedure (formal-out), formal interference nodes represent definitions (formal-ifin) or uses (formal-ifout) of v that *may happen in any timestamp during the execution of the procedure's code*. For this reason, in order to represent this specialized behaviour, we define a new dependence associated to formal interference nodes called *concurrent flow dependence*.

Definition 7 (Concurrent flow dependence): A node n is *concurrent flow dependent* on a preceding node m if:

(i) m or n are formal interference nodes,
(ii) m defines a variable v,
(iii) n uses v, and
(iv) there exists a control-flow path from m to n.

This definition is really interesting because it allows v to be redefined in the control-flow path from m to n. Even in that case, n depends on m. The rationale comes from a fundamental property of interference formal nodes: they represent an interference that could happen at any instant during the execution of the procedure.

B. Timestamps in interference nodes

The addition of interference nodes produces a new drawback when using the timestamp-based method to solve time travel: which timestamps should be assigned to interference nodes? In order to establish a correct node chronology when numbering interference nodes we need to consider all moments in which they could be defined or used. We start considering formal interference nodes:

- Formal-interference-out nodes in a procedure represent all possible definitions of a shared variable. For this reason, the set of timestamps of these nodes includes the union of the timestamps of all the definitions of the variable in the procedure.
- Formal-interference-in nodes in a procedure represent a definition of a shared variable that occurred in a parallel thread and was used at some point in the procedure. Therefore, the set of timestamps associated to formal-ifin nodes is the union of all timestamps of uses of this variable in the procedure.

While formal interference nodes of a procedure contain the timestamps associated to *all* the calls to the procedure, actual interference nodes are only associated to those timestamps that happened during one specific call. They can be computed by selecting the timestamps that happened between the initial and final timestamps of the call.

Algorithm 1 tSDG interference structure creation algorithm

Input: A SDG $G = (N,\ E_c \cup E_f \cup E_{call} \cup E_{in} \cup E_{out} \cup E_s)$
Output: A tSDG $G' = (N', E'_c \cup E_f \cup E_{call} \cup E'_{in} \cup E'_{out} \cup E'_s \cup E'_{if} \cup E'_{cf})$
Initialization:
$N' = N$, $E'_c = E_c$, $E'_{in} = E_{in}$, $E'_{out} = E_{out}$, $E'_{if} = \emptyset$, $E'_{cf} = \emptyset$,
I_{inter} contains all interprocedural interference dependences,
I_{intra} contains all intraprocedural interference dependences

1: **begin**
2: **for** $(n, n') \in I_{inter}$ **do**
3: AddInterferenceInNodes(n')
4: AddInterferenceOutNodes(n)
5: **for** $(n, n') \in I_{intra}$ **do**
6: $E'_{if} = E'_{if} \cup \{(n, n')\}$
7: $E'_s \leftarrow$ ComputeSummaryEdges(G)
8: **return** $G' = (N', E'_c \cup E_f \cup E_{call} \cup E'_{in} \cup E'_{out} \cup E'_s \cup E'_{if} \cup E'_{cf})$
9: **end**

10: **function** AddInterferenceInNodes(n)
11: $v \leftarrow$ GetInterferenceVar(n)
12: **if** RepeatedInterference(n, "in", v) **then return**
13: $root \leftarrow$ GetProcedureRoot(n)
14: $tstamps \leftarrow$ GetTimestamps(n)
15: $newFormal \leftarrow$ AddNode(N', "$v = v\_ifin$", $tstamps$)
16: $E'_c \leftarrow E'_c \cup \{(root, newFormal)\}$
17: $E'_{cf} \leftarrow E'_{cf} \cup \{(newFormal, n)\}$
18: **for** $c \in$ GetCalls($root$) **do**
19: $callTstamps \leftarrow$ FilterTimestamps($c, tstamps$)
20: $newActual \leftarrow$ AddNode(N', "$v\_ifin = v$", $callTstamps$)
21: $E'_c \leftarrow E'_c \cup \{(c, newActual)\}$
22: $E'_{in} \leftarrow E'_{in} \cup \{(newActual, newFormal)\}$
23: AddInterferenceInNodes($newActual$)

24: **function** AddInterferenceOutNodes(n)
25: $v \leftarrow$ GetInterferenceVar(n)
26: **if** RepeatedInterference(n, "out", v) **then return**
27: $root \leftarrow$ GetProcedureRoot(n)
28: $tstamps \leftarrow$ GetTimestamps(n)
29: $newFormal \leftarrow$ AddNode(N', "$v\_ifout = v$", $tstamps$)
30: $E'_c \leftarrow E'_c \cup \{(root, newFormal)\}$
31: $E'_{cf} \leftarrow E'_{cf} \cup \{(n, newFormal)\}$
32: **for** $c \in$ GetCalls($root$) **do**
33: $callTstamps \leftarrow$ FilterTimestamps($c, tstamps$)
34: $newActual \leftarrow$ AddNode(N', "$v = v\_ifout$", $callTstamps$)
35: $E'_c \leftarrow E'_c \cup \{(c, newActual)\}$
36: $E'_{out} \leftarrow E'_{out} \cup \{(newFormal, newActual)\}$
37: AddInterferenceOutNodes($newActual$)

C. An algorithm to generate the tSDG

Algorithm 1 describes how to transform a SDG into its associated tSDG. The SDG is formed from a set of nodes N and a set of control edges (E_c), flow edges (E_f), call edges (E_{call}), input edges (E_{in}), output edges (E_{out}), and summary edges (E_s) (see Section II). The tSDG also includes the interference structure: formal-interference-in, formal-interference-out, actual-interference-in, and actual-interference-out nodes; and the edges needed to connect them, including the interference edges (E'_{if}) (induced by Definition 6) and the concurrent flow edges (E'_{cf}) (induced by Definition 7).

There is a set of functions used in the algorithm that need to be explained beforehand. Given a node n, source or target of an interference, functions GetInterferenceVar(n) and GetTimestamps(n) return the name of the interfered variable in n and the timestamps associated to n, respectively.

Function REPEATEDINTERFERENCE(n,l,v) is the termination condition of the recursion. It checks whether node n has been already processed with the same label l and the same interfered variable v. Function GETPROCEDUREROOT(n) returns the *Enter* node of the procedure where n is contained. Function ADDNODE(N',l,t) creates (and returns) a new node in N' with label l and timestamp t. If the node already exists, t is added to the already existing set of timestamps. Function GETCALLS(n) returns a set with all the call nodes that call the procedure whose *Enter* node is n. Finally, given a call node c and a set of timestamps t, function FILTERTIMESTAMPS(c,t) returns the timestamps that occur inside call c.

The algorithm starts by procesing all interprocedural interference dependences. The structure that represents each interprocedural interference dependence is created by means of functions ADDINTERFERENCEINNODES and ADDINTERFERENCEOUTNODES. While the call to ADDINTERFERENCEINNODES (line 3) creates formal-interference-in and actual-interference-in nodes, the call to ADDINTERFERENCEOUTNODES (line 4) creates formal-interference-out and actual-interference-out nodes.

In each call to functions ADDINTERFERENCEINNODES and ADDINTERFERENCEOUTNODES, we start by extracting the variable involved in the interference and checking whether the input node n has already been processed. In that case, the function returns and nothing is generated (lines 12 and 26). Otherwise, we extract the root (*Enter*) node of the procedure and the set of timestamps associated to n. Then, according to whether it is an input or output node, a different formal interference node is created ($newFormal$) and connected to the *Enter* node with a control edge (lines 15–16 and 29–30), and the corresponding concurrent flow edge adds the connection between the formal node and n (lines 17 and 31).

The next step is to extract all the call nodes to generate the corresponding actual nodes. For each call c, we create and add to the graph the corresponding actual interference node ($newActual$) and connect it to c with a control edge (lines 20–21 and 34–35). Then, actual and formal interference nodes are connected with the corresponding input/output edges (lines 22 and 36).

After processing all interprocedural interference dependences, we convert all intraprocedural interference dependences into interference edges (E'_{if}) (lines 5-6).

Finally, we can generate the summary edges with the standard algorithm (line 7). This completes the final tSDG, which includes the new concurrent parameter passing structure.

Example 3: Consider the code in Figure 4, with two threads `T1` and `T2` that read and write over the shared variable x. Figure 5 shows how the SDG of the program (left) is transformed into the tSDG (right). According to Definition 3, this program contains four interprocedural interference dependences (not drawn because the SDG does not include interferences): two from `x=2` in `T1` to `a=x` and `x=x+1` in `T2`, and other two from `x=x+1` and `x=a` in `T2` to `print(x)` in `T1`. These four dependences stop being interprocedural in the tSDG: they only enter and exit procedure `f` through the corresponding interference nodes (those in grey).

When applying the transformation from SDG to tSDG, the four interprocedural interferences are processed by Algorithm 1 producing: (i) the set of interference nodes (grey nodes), (ii) the input and output edges connecting them, (iii) the set of concurrent flow edges associated to procedure `f` (light blue edges), and (iv) the set of interference edges. As statements 10 and 11 in procedure `f` are target of an interprocedural interference dependence (they both use shared variable `x`), a formal-ifin assignment for `x` is created containing all their associated timestamps (6 and 7). On the other hand, since statements 11 and 12 are source of interprocedural interference dependences (they define shared variable `x`), a formal-ifout assignment for `x` is also included in the tSDG with the timestamp of both definition nodes (7 and 8). The formal-interference structure is replicated in the corresponding call to procedure `f`, generating actual interference nodes with the same timestamps (they are associated to the only call to procedure `f`). Finally, following Definition 7, concurrent flow edges are generated in `f` from formal-ifin nodes (respectively to formal-ifout nodes).

It is interesting to remark that flow edges only connect actual and formal nodes but not interference nodes (according to Definition 5), while concurrent flow edges only connect interference nodes (according to Definition 7), dividing responsibilities.

It is worth mentioning that our tSDG generates new summary edges (see Figure 5) that account for all execution paths induced by concurrent scenarios. This is the key point why our model solves the incompleteness problem (see Section III and cf. Nanda et al. [18, Section 3.1]). Instead of replicating the behaviour of summary edges with interprocedural interference edges and ad-hoc changes to the slicing algorithm, our solution naturally uses the same summary mechanisms and slicing algorithm as the standard SDG.

Finally, we state an important property of the tSDG: all interferences are represented with a chain of dependences. This is especially relevant because the tSDG does not contain interprocedural dependence edges, but thanks to that property they are represented through other dependence chains.

V. SLICING THE TSDG

The slicing algorithm used to slice our tSDG model is the standard two-phase SDG slicing algorithm, but including the timestamp mechanism proposed by Nanda et al. to avoid time travel. The new nodes (interference nodes) introduced in the

```
1       procedure main()
2       x = 1;
3       cobegin{// T1
4         x = 2;
5         print(x);
6       }{// T2
7         call f;
8       }

 9       procedure f
10       a = x;
11       x = x + 1;
12       x = a;
```

Fig. 4: Program that contains two threads with interference

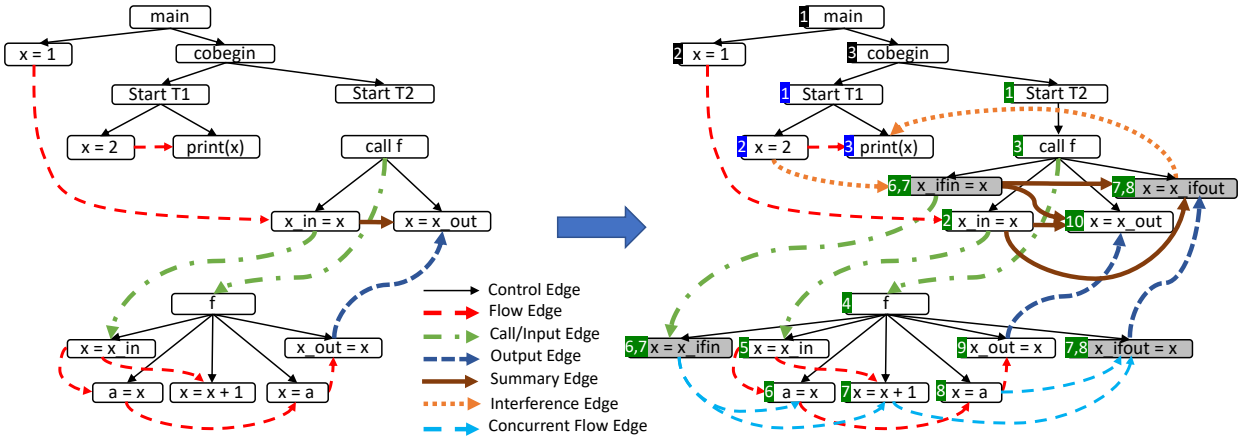

Fig. 5: Transformation from SDG to tSDG of the code in Figure 4

tSDG have been conveniently labelled with timestamps so that they can be directly processed with Nanda et al.'s timestamp algorithm.

Example 4: Consider again the code in Figure 1 and the slicing criterion $\langle 7, a \rangle$. The new tSDG associated to this program is shown in Figure 6. We can compare it with the old tSDG shown in Figure 3. First, the slice computed with the new tSDG is the minimal slice shown in Figure 1a. The (imprecise) slice computed with the old tSDG is shown in Figure 1b.

There are two differences that can be clearly observed: (i) the new tSDG does not allow interprocedural interference edges. In contrast, the old tSDG contains an interprocedural interference edge (it connects a statement inside `f`'s body with a statement outside `f`). Precisely for this reason, (ii) in the old tSDG some nodes in procedure `f` are reached during slicing Phase 1 while in the new tSDG all the nodes in `f` can only be reached during slicing Phase 2. These two facts are the key factors for obtaining the expected slice with the tSDG.

Table II shows how the slicing algorithm prevents the traversal of input edges (because they are reached in Phase 2) and, thus, excludes the calls to `f` in lines 3 and 5. The traversal is divided into five main blocks: the first block (steps 1a and 2aa) represents the path in Phase 1 from the slicing criterion to `x_in = 4`. The second block (steps 1b and 2ba) represents the path in Phase 1 from the slicing criterion to `x_in = 5`. The rest of nodes collected in Phase 1 are not represented in the table because they are trivially reached through control edges. In Phase 2, the third (steps 2ab and 3a) and fourth (steps 2bb and 3b) blocks respectively represent the paths from `a = a_out` and `a = a_ifout` to `a = x`. Finally, the last block traverses a flow edge to reach the formal-in node `x = x_in`. Since we are in Phase 2, the four input edges cannot be traversed. This avoids to unnecessarily collect nodes `x_in = 2`, `x_in = 3`, and, thus, the calls to `f` in lines 3 and 5. Note, however, that nodes `x_in = 4` and `x_in = 5` where already collected by summary edges in Phase 1 during steps 2aa and 2ba, keeping the slice complete, and the slicing traversal context-sensitive.

Complexity

In this section we compute the asymptotic size of the tSDG, the temporal cost of building it, and the complexity of the slicing algorithm. The size of the tSDG can be measured in terms of the following parameters:

- S_{if}: statements that are source or target of an interprocedural interference dependence.
- V_i: number of different variables causing interprocedural interference dependences at statement i.
- C_i: number of calls to procedures that may be in execution while statement i is being executed, i.e., procedures that may directly or transitively execute i.
- P_i: number of procedures that contain at least one of the calls in C_i.
- F_i: number of procedures in P_i not called by any other procedure.

The tSDG increments the number of nodes of the SDG in:

$$\sum_{i \in S_{if}} (C_i + P_i)$$

This formula can be easily obtained from Algorithm 1. In the tSDG, P_i represents formal-ifin/ifout nodes and C_i represents actual-ifin/ifout nodes. On the other hand, the tSDG increments the number of edges in:

$$\sum_{i \in S_{if}} V_i + 2(C_i + P_i) - F_i$$

because for each statement source or target of interprocedural interference dependence, V_i interference-flow edges are generated for each statement in S_{if}. Then, we generate two more edges (one control edge and one interprocedural edge) for all generated formal and actual nodes except for those in F_i (for them, we only generate one control edge).

Constructing a tSDG $G = (N, E)$ has a quadratic cost $\mathcal{O}(N^2)$, exactly the same as the SDG (and the old tSDG). This is due to the fact that the most expensive task is computing

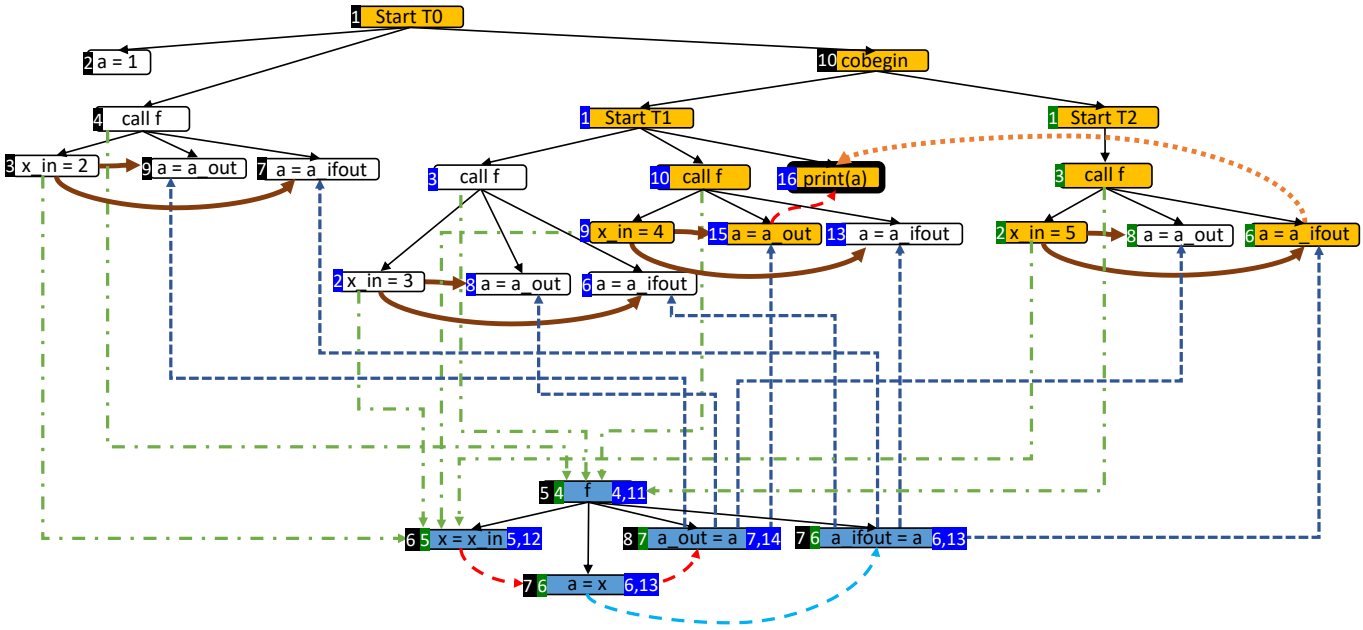

Fig. 6: New tSDG of the program in Figure 1 sliced w.r.t. $\langle 7, a\rangle$

Step	Last Node	Edge Type	New Node		Traversal State			
			Phase	Statement	Number	T0	T1	T2
0	-	-	1	print(a)	{(T1,{16})}	$\bot$	{16}	$\bot$
1a	print(a)	Flow	1	a = a_out	{(T1,{15})}	$\bot$	{15}	$\bot$
2aa	a = a_out	Summary	1	x_in = 4	{(T1,{9})}	$\bot$	{9}	$\bot$
1b	print(a)	Interference	1	a = a_ifout	{(T2,{6})}	$\bot$	{16}	{6}
2ba	a = a_ifout	Summary	1	x_in = 5	{(T2,{2})}	$\bot$	{9}	{2}
2ab	a = a_out	Output	2	a_out = a	{(T0,{8}),(T1,{7,14}),(T2,{7})}	{8}	{7,14}	{7}
3a	a_out = a	Flow	2	a = x	{(T0,{7}),(T1,{6,13}),(T2,{6})}	{7}	{6,13}	{6}
2bb	a = a_ifout	Output	2	a_ifout = a	{(T0,{7}),(T1,{6,13}),(T2,{6})}	{7}	{6,13}	{6}
3b	a_ifout = a	Conc. Flow	2	a = x	{(T0,{7}),(T1,{6,13}),(T2,{6})}	{7}	{6,13}	{6}
4	a = x	Flow	2	x = x_in	{(T0,{6}),(T1,{5,12}),(T2,{5})}	{6}	{5,12}	{5}

TABLE II: New tSDG slicing traversal of the program in Figure 1 (from the slicing criterion $\langle 7, a\rangle$ to all the actual-in nodes `x_in = x` in the calls to procedure `f`)

summary edges (a quadratic operation) that is also done in the standard SDG.

Finally, the complexity of the slicing algorithm is the same as in the old tSDG. The timestamp mechanisms used to compute and validate threaded witnesses makes the complexity of the algorithm $\mathcal{O}(N^{d^t})$, where d is the max depth of the call graph[1] and t the number of threads [18].

While keeping the same asymptotic cost, our algorithm is more efficient than Nanda et al.'s algorithm thanks to the improved treatment of context-sensitivity. Since our slicing algorithm uses the two-phase traversal proposed by Horwitz et al., the maximum number of nodes handled by the algorithm with the same traversal state is N (while in Nanda et al.'s approach is $2N$), being the runtime complexity for managing timestamps at each node $\mathcal{O}(N^3)$ and traversing each node to manage context-sensitivity in our algorithm $\mathcal{O}(N+E)$ (every node and edge is only processed once with the same traversal state).

The termination of the slicing phase is ensured by the fact that whenever a node is visited with the same timestamps, the traversal finishes. Therefore, since the number of nodes to visit is finite and each node is visited at most a number of finite times, termination is ensured. In particular, a node can only be visited as many times as combinations of timestamps exists, and the number of timestamps is also finite.

[1]The maximum number of edges in an acyclic path [8].

VI. Related work

The first concurrency-aware graph representation model was proposed by Cheng [4], [5]. He defined a graph-like representation model called *Process Dependence Net* (PDN), which represented five types of primary program dependences in concurrent ADA programs. Cheng was the first author that reduced the problem of slicing concurrent programs to a graph reachability problem. Zhao [25] introduced a *multi-thread*

dependence graph to represent concurrent Java programs. His approach replicated the classic two-phase slicing algorithm for sequential interprocedural programs, traversing interference and synchronization dependences transitively. Something similar happened with the approach proposed by Hatcliff [11]. Hatcliff specialized extra dependences in concurrent Java programs while still treating interference dependences as transitive. Because they modelled interference dependence as transitive, all three approaches did not solve the time travel problem, producing inaccurate slices.

There are several works that have tried to solve the interference intransitivity for generating accurate slices and work around the time travel problem. Zhang et al. [24] calculated a predecessor set, which ensures that every node in the slice may be executed before the slicing criterion, since all the statements included in a backward slice must be computed before it. On the other hand, Chen et al. [3] discarded the two-phase algorithm and computed what they called dependence sequences. They used these sequences to compute slices as the difference between the sequences reached from the slicing criterion and the sequences formed by nodes that actually cannot be reached due to time travel. Although these two approaches proposed novel ideas, they were not able to completely resolve time travel because both algorithms could not ensure that all nodes in every dependence sequence formed a valid path. The above approaches were the first proposals to deal with concurrent dependences and with the time travel problem.

In 1998, Krinke [14] designed the *threaded Program Dependence Graph* (tPDG), an intraprocedural model based on the PDG that introduced the definition of interference dependence based on threaded witnesses. Krinke's approach was the first to define a precise slicing algorithm for programs with shared memory that completely solved the time travel problem. However, the summary edges and the two-phase algorithm introduced by the SDG were not compatible with Krinke's approach, generating context-insensitive slices and inaccuracies in the presence of nested threads and threads nested within loops. Five years later, the same Krinke proposed in [15] a context-sensitive version of his approach for program slicing. His approach defined a new program representation model called the *threaded Interprocedural Program Dependence Graph* (tIPDG), built upon the *Interprocedural Threaded Control-Flow Graph* (ITCFG). The tIPDG was the first context-sensitive approach to slice concurrent programs. In order to obtain accurate context-sensitive slices, the tIPDG labels interprocedural edges to avoid visiting an actual-in node of a call if its corresponding actual-out node has not been previously included in the slice. Although this approach was context-sensitive, it was much slower than either a context-insensitive concurrent slicer or the SDG. The main difference between Krinke's approach and ours is the method used during the traversal to reach context-sensitivity. Krinke implements a single-phase algorithm that does not require summary edges and collects labels during the traversal of interprocedural edges, reaching context-sensitivity with an exponential cost. In contrast, our model uses the linear-cost two-phase slicing algorithm proposed by Horwitz et al., which manages traversal restrictions according to two differentiated slicing phases. In our approach, summary edges computed during the tSDG construction process are essential to ensure completeness because they represent the information flow of interprocedural interference dependences, which are explicitly represented in Krinke's tIPDG.

Nanda et al. [17] adapted Krinke's traversal algorithm and refined it, defining flow dependences induced by threads generated within loops and nested threads, and providing an alternative way of computing threaded witnesses. Their slicing algorithm was more efficient and remained precise in the presence of the mentioned thread combinations. In a later publication [18] they introduced a more efficient quasi-context-sensitive approach. This approach solved some completeness and correctness issues of their previous work, but left room for improvement in its context-sensitivity as shown in Section III. We use the same system to solve the time travel problem as Nanda et al. The main difference between their approach and ours are (i) the way interferences edges are computed and (ii) the modifications in the slicing algorithm required to ensure completeness. While they allow interprocedural interference edges, we force all interference edges to be intraprocedural and we create new nodes to propagate interferences from one procedure ot another through input and output edges. This fact allows us to use the standard Horwitz et al.'s slicing algorithm, while the old tSDG needs to perform phase changes during the traversal to compute complete slices.

Later approaches have inherited either Krinke's or Nanda et al.'s approaches to context-sensitivity. A novel approach to computing threaded witnesses is provided in [20], where the authors produced a graph that combines the tSDG and the possible states of the traversal proposed by Nanda et al., effectively encoding threaded witnesses into the graph and simplifying the traversal algorithm. This proposal is orthogonal to our technique. Thus, it could be also applied to our graph, speeding up the time required to slice our model.

VII. Conclusions

Concurrent program slicing poses additional challenges in comparison to sequential program slicing. In the shared memory concurrency model, the existence of interference dependences generate two different slicing problems: time travel and incompleteness. The scientific community has proposed different representation models based on the SDG to deal with these problems. In particular, Nanda et al.'s tSDG includes mechanisms to accurately slice `cobegin` blocks dealing with both problems through some changes in the slicing algorithm.

However, we have shown in this paper that even the tSDG may generate context-insensitive slices when a procedure with interference dependences is called from different program points. We have designed a counterexample that reveals that the tSDG is inaccurate in some situations. Furthermore, we have identified the source of this imprecision and explained the rationale behind it.

To solve the imprecision *problem*, we have proposed an alternative representation model of the tSDG. Our graph is based on the SDG model used to represent parameter passing in the SDG and extends it to concurrent parameter passing in the presence of interprocedural interference dependences. For modelling interprocedural interference dependences, our tSDG replaces interference edges with two new components: (i) all procedures with statements that are source or target of interprocedural interference dependences include new formal nodes called formal interference nodes that model concurrent parameter passing. Thus, all the calls to these procedures are augmented with their corresponding actual interference nodes analogously to parameter passing in the SDG; (ii) a new type of flow dependence (concurrent flow dependence) is used in procedure definitions to represent how the value of an interference affects the statements inside a procedure.

These changes produce a new model where concurrent and sequential dependences are separated at procedure calls (thanks to (i)). In this model, interference edges only connect statements executed in the same procedure, since their behaviour inside procedures is represented by other flow dependences (thanks to (ii)). The addition of (i) and (ii) may generate new program dependence paths from formal in to formal out nodes (both sequential and concurrent) in procedure definitions that produce new summary edges at procedure calls that solve the incompleteness problem without modifying the traversal phase.

Finally, given the nature of our representation where concurrent and sequential dependences are split, the slicing algorithm used to slice the tSDG also differs from the one used by the previous tSDG. Our tSDG is capable of solving both the time travel and summary incompleteness problems when slicing concurrent programs in a more precise way by combining the two-phase traditional SDG slicing algorithm (instead of employing forced phase changes during the traversal) with the use of the timestamps to detect time travel during the slicing traversal. The result is a new tSDG that obtains context-sensitive slices for programs with `cobegin` blocks in those cases the previous tSDG could not.

REFERENCES

[1] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[2] D. Binkley, M. Harman, L. R. Raszewski, and C. Smith. An empirical study of amorphous slicing as a program comprehension support tool. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 161–170. IEEE, 2000.

[3] Z. Chen and B. Xu. Slicing concurrent Java programs. *SIGPLAN Not.*, 36(4):41–47, Apr. 2001.

[4] J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, AADEBUG '93, pages 223–240, Berlin, Heidelberg, 1993. Springer-Verlag.

[5] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*, APDC '97, page 370, USA, 1997. IEEE Computer Society.

[6] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. *SIGSOFT Softw. Eng. Notes*, 21(3):121–134, May 1996.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[8] A. C. Fong and J. D. Ullman. Finding the depth of a flow graph.

[9] A. Hajnal and I. Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance*, 24(1):67–82, 2012.

[10] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[11] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Static Analysis: 6th International Symposium, SAS'99 Venice, Italy, September 22–24, 1999 Proceedings*, pages 1–18. Springer, 1999.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.

[14] J. Krinke. Static slicing of threaded programs. *SIGPLAN Not.*, 33(7):35–42, July 1998.

[15] J. Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, September 2003.

[16] A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: Design, correctness, and evaluation. In *Proceedings of the 2007 ACM Workshop on Digital Rights Management*, DRM '07, pages 70–81, New York, NY, USA, 2007. ACM.

[17] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, page 180–190, New York, NY, USA, 2000. Association for Computing Machinery.

[18] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, nov 2006.

[19] C. Ochoa, J. Silva, and G. Vidal. Lightweight program specialization via Dynamic Slicing. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, WCFLP '05, pages 1–7, New York, NY, USA, 2005. ACM.

[20] X. Qi and Z. Jiang. Precise slicing of interprocedural concurrent programs. *Front. Comput. Sci.*, 11(6):971–986, dec 2017.

[21] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, jun 2007.

[22] T. Takada, F. Ohata, and K. Inoue. Dependence-cache slicing: A program slicing method using lightweight dynamic information. In *Proceedings 10th International Workshop on Program Comprehension*, pages 169–177. IEEE, 2002.

[23] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[24] J. Zhang and C. Jin. Control-flow-based static slicing algorithm of threaded programs. *Journal of Jilin University*, 41(4):481–486, 2003.

[25] J. Zhao. Multithreaded dependence graphs for concurrent java program. In *Software Engineering for Parallel and Distributed Systems, International Symposium on*, pages 13–13. IEEE Computer Society, 1999.