Context-Sensitive Slicing of Concurrent Programs

Jens Krinke
Universität Passau
Passau, Germany
krinke@fmi.uni-passau.de

ABSTRACT

Program slicing is a technique to identify statements that may influence the computations at other statements. Precise slicing has been shown to be undecidable for concurrent programs. This work presents the first context-sensitive approach to slice concurrent programs accurately. It extends the well known structures of the control flow graph and the (interprocedural) program dependence graph for concurrent programs with interference. This new technique does not require serialization or inlining.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

program analysis, program slicing, context-sensitive, concurrency, parallelism

General Terms

Algorithms, Performance

1. INTRODUCTION

Today, even small programs use parallelism and languages like Ada or Java have language features for concurrent execution built-in. The analysis of programs where some statements may explicitly be executed concurrently is not new. The *static* analysis of these programs is complicated because the execution order of parallel executed statements is *dynamic*. Testing and debugging of concurrent programs have increased complexity: They may produce different behavior even with the same input. The nondeterministic behavior of a program is hard to understand and finding harmful nondeterministic behavior is even harder. Therefore, supporting tools are required. Unfortunately, most tools for sequential programs are not applicable to concurrent programs as they cannot cope with the nondeterministic execution order of statements. One simple way to circumvent these problems is to simulate these programs through *sequentialized* or *serialized* programs [30]. These are programs in which every possible execution order of statements is modeled through a path where the statements are executed sequentially. This may lead to exponential code explosion, which is unacceptable for analysis. Therefore, special representations of parallel programs have been developed.

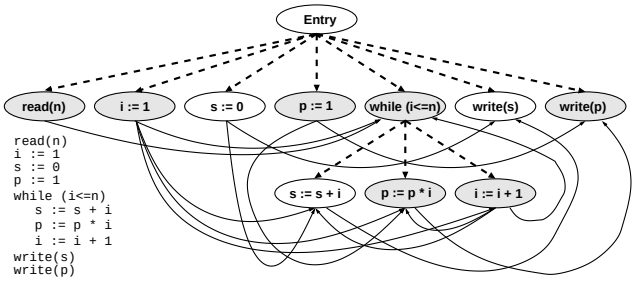

Figure 1: A program dependence graph

In the following a new notation of concurrent programs is introduced by extending the control flow graph (CFG) and program dependence graph (PDG) to their *threaded* counterparts tCFG and tPDG. Based on these graphs a more precise slicing algorithm is presented and it is shown how the basic model of concurrency can be extended to allow synchronization and communication.

The rest of this paper is structured as follows: The next section gives an example-based introduction to slicing of sequential and concurrent programs. Section three defines slicing of concurrent programs more formally and presents a high-precision approach based on a simple model of concurrency which is extended in Section four. Related work is discussed in Section five, followed by conclusions.

2. SLICING

A slice extracts those statements from a program, that potentially have an influence on a specific statement of interest which is the slicing criterion. Originally, slicing was defined by Weiser in 1979; he presented an approach to compute slices based on iterative data flow analysis [31, 32]. The other main approach to slicing uses reachability analysis in program dependence graphs [8]. Program dependence graphs mainly consist of nodes representing the statements of a program and control and data dependence edges:

Control dependence between two statement nodes exists if one statement controls the execution of the other.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

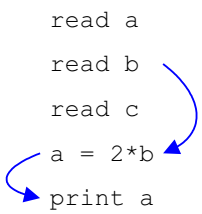

Figure 2: A procedure-less program

Data dependence between two statement nodes exists if a definition of a variable at one statement might reach the usage of the same variable at another statement.

An example PDG is shown in figure 1, where control dependence is drawn in dashed lines and data dependence in solid ones. Weiser used program slicing in debugging, nowadays slicing is used in various other fields as base technology: e.g. testing, differencing, reengineering or model checking. A broad overview on slicing techniques and applications is presented in [29].

2.1 Slicing of Sequential Programs

Example: Slicing without Procedures

Figure 2 shows a first example where we want to slice a program without procedures. To compute the slice for the statement `print a`, we just have to follow the shown dependences backwards. In this example there are two data dependences and the slice includes the assignment to `a` and the read statement for `b`. In all examples we will ignore control dependence and just focus on data dependence for simplicity of presentation. Also, we will always slice backwards from the `print a` statement.

Slicing without procedures is trivial: Just find reachable nodes in the PDG [8]. The underlying assumption is that all paths are *realizable*. This means that a possible execution of the program exists for any path that executes the statements in the same order.

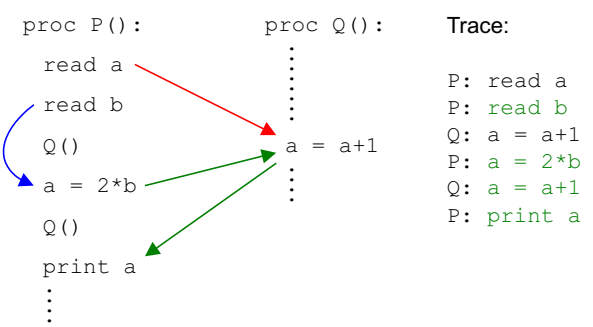

Figure 3: A program with two procedures

Example: Slicing with Procedures

Now, in the extended example in figure 3, procedures are added. If we ignore the calling context and just do a traversal of the data dependences, we would add the `read a` statement into the slice for `print a`. This is wrong because this statement has clearly no influence on the `print a` statement. The `read a` statement just has an influence on the first call of procedure `Q` but `a` is killed before the second call to procedure `Q` through the assignment `a=2*b` in procedure `P`. Such an analysis is called context-insensitive because the calling context is ignored. Paths are now considered realizable only if they obey the calling context. Thus, slicing is context-sensitive if only realizable paths are traversed. Context-sensitive slicing is solvable efficiently—one has to generate summary edges at call sites [11]: Summary edges represent the transitive dependences of called procedures at call sites. During slicing, the summary edges are used and the traversal will consider edges entering or leaving procedures in two separate passes. Thus, the computed slice will be context-sensitive despite that the calling-context is not handled explicitly.

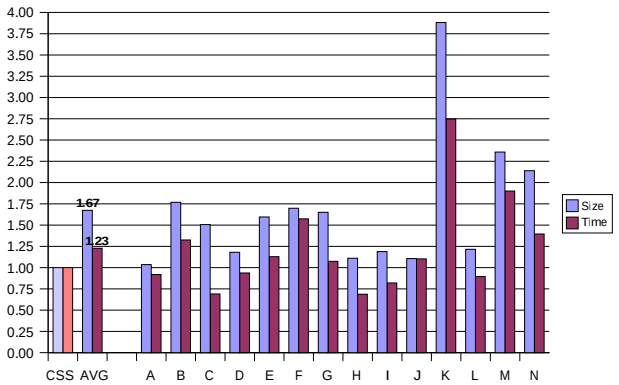

Figure 4: Context-insensitive vs. -sensitive Slicing

Context-Sensitive vs. Context-Insensitive Slicing

To evaluate the effect of context-sensitivity onto precision, we have implemented an infrastructure to compute PDGs for ANSI C programs. Within that infrastructure we have implemented and evaluated various slicing algorithms. The complete evaluation can be found in [16]; here we will only compare the context-sensitive slicing algorithm that uses summary edges with a context-insensitive one. The context-insensitive slicing algorithm is just a simple graph traversal of linear complexity. The context-sensitive algorithm is a little bit more complex: It needs once a preprocessing stage of cubic complexity to compute the summary edges, though the algorithm itself has linear complexity because it is a two-pass graph traversal. The test data were several programs of limited size: For each program 200 to 10.000 slices have been computed depending on the size of the program.

The evaluation in figure 4 shows that context-insensitive slicing is very imprecise in comparison with context-sensitive slicing (CSS). The lighter bars show how much larger the slices computed by context-insensitive slicing on average are (for test program A-N). The darker bars show how much longer it takes to compute context-insensitive slices (ignoring the time needed to compute summary edges). On average (AVG), the computed context-insensitive slices are 67% larger than the context-sensitive slices. This shows that context-sensitive slicing is highly preferable, because the loss of precision is not acceptable. A surprising result is that the simple context-insensitive slicing is *slower* than the more complex context-sensitive slicing: On average, context-insensitive slicing needs 23% more time than context-sensitive slicing. The reason is that the context-sensitive algorithm has to visit much fewer nodes during traversal due to its higher precision.

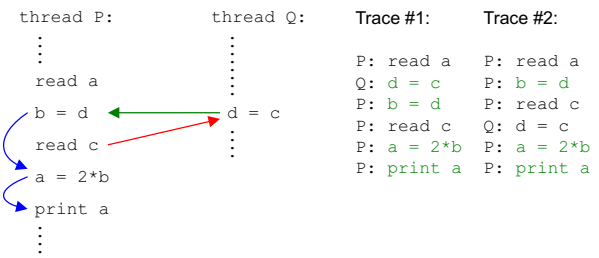

Figure 5: A program with two threads

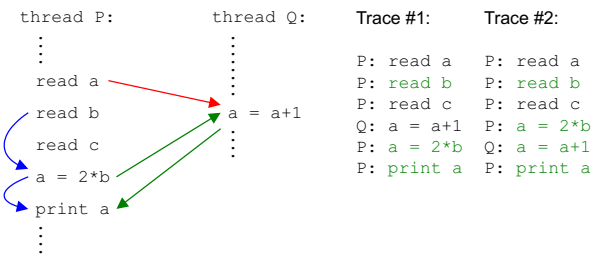

Figure 6: Another program with two threads

2.2 Slicing Concurrent Programs

Example: Slicing Concurrent Programs

Now, lets return to the example and move on to concurrent programs. In the example in figure 5 we have two threads `P` and `Q` that execute in parallel. In concurrent programs that share variables another type of dependence arises: *interference* [15]. Interference occurs when a variable is defined in one thread and used in a parallel executing thread. In this example, there are two interference dependences: One is due to a definition and a usage of variable `d`, the other is due to accesses to variable `c`. A simple traversal of interference during slicing will make the slice imprecise because interference may lead to unrealizable paths again. In the example in figure 5, a simple traversal will include the `read c` statement into the slice. But there is no possible execution where the `read c` statement has an influence on the assignment `b=d`. A matching execution would require time travel because the assignment `b=d` is always executed before the `read c` statement. A path through multiple threads is now realizable if it contains a valid execution chronology. We call slicing *state-sensitive* when the traversal through the PDG only considers realizable paths. At any point, slicing will then only consider reachable states of the program. However, even when only realizable paths are considered, the slice will not be as precise as possible.

Slicing may be imprecise

The reason for this imprecision is that parallel executing threads may kill definitions of other threads. In the example in figure 6, the `read a` statement is reachable from the `print a` statement via a realizable path. But there is no possible execution where the `read` statement has an influence on the `print` statement when assuming that statements are atomic. Either the `read` statement reaches the usage in the thread `Q` but is killed afterwards through the assignment `a=2*b` in thread `P`, or the `read` statement is immediately killed by the assignment `a=2*b` before it can reach the usage in thread `Q`. For procedure-less concurrent programs, we could find an algorithm which obeys such killing and computes precise slices. Anyhow, this is impossible for concurrent programs with procedures: Müller-Olm has shown that precise context-sensitive slicing of concurrent programs is undecidable in general [20]. He used a reduction onto two-counter-state-machines. Another important result has been proven by Ramalingam [24]. He showed that context-sensitive analysis of concurrent programs with procedures and synchronization is undecidable in general. This applies not only to slicing but also to any data flow analysis. Therefore, we have to use conservative approximations to analyze concurrent programs. First of all, synchronization can be ignored. In that case, the results are still correct, only imprecise because unrealizable paths are now allowed. This only leads to additional interference dependences which cannot happen in real executions. If we use more precise *may-happen-in-parallel (MHP)* information (eg. from [22, 23]), some of those interference dependences can probably be removed again making slicing a little bit more precise. Another simple approximation is state-insensitive analysis allowing time travel. However, when we do state-insensitive slicing, we cannot use summary edges to be context-sensitive. Summary edges would ignore the effects of parallel executing threads. So, we have to do context-insensitive slicing and accept the much lower precision or we have to inline called procedures.

Precise Slicing without Summary Edges

To be able to provide precise slicing without summary edges, we have developed a new slicing algorithm [16] which is based on capturing the calling context through *call strings* [28]. Call strings can be seen as a representation of a virtual machine's call stack. Call strings can also be seen as virtual inlining. They are used frequently for context-sensitive program analysis, e.g. pointer analysis. In presence of recursion we have an infinite set of possible call strings which makes analysis infeasible. To make the call strings finite, we fold cycles in PDGs and replace them by a single node. This has no effect on precision: If one node of a cycle is contained in a slice, all other nodes have to be contained, too. A side effect is the size reduction of program dependence graphs: On average 40% of the nodes are removed, making slicing much faster. Because the graph is acyclic after folding, only finite call strings can be generated. The call strings are then propagated along the edges of PDG: At edges that connect procedures the call string is used to check that a call always returns to the right call site. Thus, call strings are never propagated along unrealizable paths.

We have implemented the call string based slicing algorithm for sequential ANSI C and compared it to the other slicing algorithms. The result was not unexpected: This approach suffers from combinatorial explosion of the call strings, just like other call string based data flow analysis algorithms. It is only usable if the length of the call strings is limited to 2 or 3 elements—but length limitation of call strings decreases the precision.

Interprocedural Slicing Concurrent Programs

The rest of this section will present the main idea of our approach to slice concurrent programs. It is presented more formally in the next section. The basic idea is the adaption of the call string approach to concurrent programs. We assume that a concurrent program consists of separate threads that do not share code and communicate via shared variables. The context is now captured through one call string for every thread. The context is then a tuple of call strings which is propagated along the edges in PDGs. Again, the traversal of intraprocedural edges does not change the call string of the enclosing thread. The context can simply be propagated. During traversal of interprocedural edges the call string of the enclosing

thread is used to check that a call always returns to the right call site. This may generate a single new context.

The traversal of interference edges is much more complicated: The call string of the newly reached thread is used to check that the reached node is reachable from a node with the old (saved) call string. To do that, we have to check every call string that the reached node can possibly have. This can generate a set of new call strings that have to be propagated.

To avoid combinatorial explosion of call strings, we pursue a combined approach: using summary edges to compute the slice within threads. Additionally, we only generate and propagate call strings along interference edges if the slice crosses threads. With this approach much fewer contexts are propagated and the length limitation can be increased for higher precision. This only outlines the idea of our approach. The next section presents a detailed description.

3. A CONTEXT-SENSITIVE APPROACH

For presentation purposes a simple model of concurrency will be used. A concurrent program is assumed to consist of a set of *threads* $\Theta = \{\theta_1, \ldots, \theta_n\}$. Threads may be executed in parallel on different processors or interleaved on a single processor. All threads are started immediately after the program's start and the program exits after all threads have finished. The threads do not share any code, communication is done via global variables, and every statement is assumed to be atomic and synchronized properly. Every thread consists of a series of procedures which may call each other but may not call a procedure from a different thread. One of the procedures for every thread is the main procedure, which is called from the runtime system after the corresponding thread has started. The corresponding thread stops after the main procedure returns. Synchronization is ignored for now (but will be discussed later).

The individual procedures of a program are represented in control flow graphs $G_p^{\text{CFG}} = (N_p, E_p, n_p^s, n_p^e)$ for each procedure p with node set N_p and edge set E_p. The statements and predicates are represented by nodes $n \in N_p$ and the control flow between statements is represented by *control flow edges* $(n,m) \in E_p$, written as $n \rightharpoonup m$. Two special nodes n_p^s and n_p^e are distinguished: The START node n_p^s and the EXIT node n_p^e which represent begin and end of procedure p. The variables which are referenced at node[1] n are denoted by $\mathsf{ref}(n)$, the variables which are defined (or assigned) at n are denoted by $\mathsf{def}(n)$.

An *interprocedural control flow graph* (ICFG) is a directed graph $G^{\text{ICFG}} = (N^\star, E^\star, n_0^s, n_0^e)$, where $N^\star = \bigcup_p N_p$ and $E^\star = E^C \cup \bigcup_p E_p$. The calls are represented by *call* and *return edges* in E^C: A call edge $e \in E^C$ is going from a *call node* $n \in N_p$ to the START node n_q^s of the called procedure q. A return edge $e \in E^C$ is going from the EXIT node n_q^e of the called procedure q back to the immediate successor of call node $n \in N_p$.[2] Nodes n_0^s and n_0^e are the START and EXIT node of the main procedure.

3.1 The Threaded Interprocedural CFG

Every thread $t \in \Theta$ can be represented with its interprocedural control flow graph $G_t^{\text{ICFG}} = (N_t^\star, E_t^\star, n_t^s, n_t^e)$. Because all threads are independent, no edges exist between the control flow graphs of two different threads. The *threaded interprocedural CFG (tICFG)* $G^{\text{tICFG}} = (N^\Pi, E^\Pi, S^\Pi, X^\Pi)$ is simply the union of all ICFGs for the different threads:

$$\begin{aligned} N^\Pi &= \bigcup_{t\in\Theta} N_t^\star \\ E^\Pi &= \bigcup_{t\in\Theta} E_t^\star \\ S^\Pi &= \{n_t^s \mid t \in \Theta\} \\ X^\Pi &= \{n_t^e \mid t \in \Theta\} \end{aligned}$$

We make the following assumption: $\Pi(t)$ exists and returns the set of threads which may execute in parallel to thread t. In this simple model, this is trivial: $\Pi(t) = \{t' \in \Theta \mid t' \neq t\}$[3]. However, the following will not rely on that to make more complex models possible later on. The function $\theta(n)$ returns for every node n its enclosing thread. This function is statically decidable and can already be generated during parsing and constructing the tICFG.

The main problem is context-sensitivity: This is handled with explicit context through virtual inlining. It is assumed that the execution state of a thread is encoded by a context in the form of a (possibly infinite) call string. The call string is represented as a stack of nodes, where the topmost node represents the currently executing statement.

We first define when a context *reaches* another context:

Definition 1. The execution state of thread t is a *context* $c = n_0 \ldots n_k$, representing an execution stack of nodes $n_i \in N_t^\star$ with the topmost node $T(c) = n_0$ (context c belongs to the current node n_0 and nodes $n_1 \ldots n_k$ represent call nodes). The 'pop' function P is defined as $P(c) = n_1 \ldots n_k$. Let $\theta(c) = \theta(T(c))$.

A context c *directly reaches* another context c': $c \rightharpoonup_{\mathsf{R}} c'$, iff one of the following alternatives holds:

1. an edge $n \rightharpoonup n' \in E_t^\star$ exists and $n \rightharpoonup n' \notin E_t^C \wedge T(c) = n \wedge T(c') = n' \wedge P(c) = P(c')$
   (corresponding edge in the CFG of a procedure exists),

2. a call edge $n \rightharpoonup n' \in E_t^C$ exists and $T(c) = n \wedge T(c') = n' \wedge c = P(c')$
   (corresponding call edge exists), or

3. a return edge $n \rightharpoonup n' \in E_t^C$ exists and

   (a) $T(c) = n \wedge T(c') = n'$,

   (b) $P(P(c)) = P(c')$, and

   (c) $T(P(c)) \rightharpoonup n' \in (E_t^\star - E_t^C)$

   (corresponding return edge exists which returns to an immediate successor of the last call node).

A context c *reaches* another context c': $c \rightharpoonup_{\mathsf{R}}^+ c'$, iff a series of contexts $c_1, \ldots, c_n$ exists, with $c = c_1 \wedge c' = c_n \wedge \forall_{1 \le i < n} c_i \rightharpoonup_{\mathsf{R}} c_{i+1}$. The set of possible contexts for ICFG $G_t = (N_t^\star, E_t^\star, n_t^s, n_t^e)$ is $C_t = \{c' \mid n_t^s \rightharpoonup_{\mathsf{R}}^+ c' \vee c' = n_t^s\}$.

Note that $c \rightharpoonup_{\mathsf{R}}^+ c'$ implies the existence of an interprocedural realizable path from $T(c)$ to $T(c')$. Also, $\theta(c) \neq \theta(c') \Rightarrow c \not\rightharpoonup_{\mathsf{R}}^+ c'$ because the ICFGs are disjunct.

The *threaded interprocedural witness* is now defined in terms of contexts:

[1] In the rest of this work we will use "node" and "statement" interchangeable, as they are bijectively mapped.

[2] There are two common variants: First, the immediate successor of a call node is an explicitly defined return node. Second, the return edge is going from the EXIT node to the call node itself.

[3] As the parallel execution relation is symmetric, $t' \in \Pi(t) \iff t \in \Pi(t')$ holds. Note that the relation is neither reflexive nor transitive.

Definition 2. A sequence $l = \langle c_1, \ldots, c_k \rangle$ of contexts (execution stacks) is a *threaded interprocedural witness* in a tICFG, iff

$$\forall 1 \leq j < i \leq k : \theta(c_j) \in \Pi(\theta(c_i)) \vee c_j \rightharpoonup_{\mathsf{R}}^{+} c_i$$

Basically this means that all contexts in a thread must be reachable from its predecessors if they cannot execute in parallel.

Intuitively, a threaded interprocedural witness can be interpreted as a sequence of contexts that form a valid execution chronology. Having a threaded witness $l = \langle c_1, \ldots, c_k \rangle$ and a context c, it can be decided whether $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded witness without checking the threaded witness properties of l':

OBSERVATION 1 (PREPENDING). *Let sequence $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness. Then $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded interprocedural witness, iff*

$$\forall 1 \leq i \leq k : \theta(c_i) \in \Pi(\theta(c)) \vee c \rightharpoonup_{\mathsf{R}}^{+} c_i$$

This follows from definition 2.

Definition 3. Let $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness and thread $t \in \Theta$. $F(l,t)$ is defined as:

$$F(l,t) \quad = \quad \begin{cases} c_i & \exists i : \theta(c_i) \notin \Pi(t) \wedge \forall 1 \leq j < i : \theta(c_j) \in \Pi(t) \\ \bot & \text{otherwise} \end{cases}$$

The result is basically the first context of witness l relevant for the execution of thread t (if such a context exists).

THEOREM 1 (SIMPLIFIED PREPENDING). *Let sequence $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness. Then sequence $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded interprocedural witness, iff*

$$F(l, \theta(c)) = \bot \vee c \rightharpoonup_{\mathsf{R}}^{+} F(l, \theta(c))$$

PROOF. Proof by contradiction. First assume that l and l' are witnesses:

$$\begin{aligned} & F(l, \theta(c)) \neq \bot \wedge c \not\rightharpoonup_{\mathsf{R}}^{+} F(l, \theta(c)) \\ \Longleftrightarrow\ & \exists 1 \leq i \leq k : c_i = F(l, \theta(c)) \wedge c \not\rightharpoonup_{\mathsf{R}}^{+} c_i \end{aligned}$$

From definition 3 follows $\theta(c_i) \notin \Pi(\theta(c))$. Altogether:

$$\exists 1 \leq i \leq k : \theta(c_i) \notin \Pi(\theta(c)) \wedge c \not\rightharpoonup_{\mathsf{R}}^{+} c_i$$

However, this is a contradiction to observation 1 because l and l' are threaded witnesses.

Second, assume that l is a witness while l' is not. $F(l, \theta(c)) = \bot$ cannot hold, otherwise l' would be a threaded witness:

$$F(l, \theta(c)) = \bot \Rightarrow \forall 1 \leq i \leq k : \theta(c_i) \in \Pi(\theta(c))$$

Therefore assume that $c \rightharpoonup_{\mathsf{R}}^{+} F(l, \theta(c))$ holds. Because l' is not a witness, from observation 1 follows

$$\exists 1 \leq i \leq k : \theta(c_i) \notin \Pi(\theta(c)) \wedge c \not\rightharpoonup_{\mathsf{R}}^{+} c_i$$

This contradicts $c \rightharpoonup_{\mathsf{R}}^{+} F(l, \theta(c))$ and therefore theorem 1 holds. □

Having a threaded witness $l = \langle c_1, \ldots, c_k \rangle$ and an edge $n \rightharpoonup n'$ with $T(c_1) = n'$, $T(c) = n$ and $c \rightharpoonup_{\mathsf{R}} c_1$, it can be decided whether $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded witness without checking the threaded witness properties of l':

THEOREM 2 (PREPENDING AN EDGE). *Let $l = \langle c_1, \ldots, c_k \rangle$ be a threaded interprocedural witness. If an edge $T(c) \rightharpoonup T(c_1)$ exists, then $l' = \langle c, c_1, \ldots, c_k \rangle$ is a threaded interprocedural witness.*

PROOF. Three possibilities for $T(c) \rightharpoonup T(c_1)$ exist: traditional control flow, call or return edges. Let $n = T(c)$ and $n_1 = T(c_1)$. From $n \rightharpoonup n_1 \in E_t^*$ follows $\theta(n) = \theta(n_1)$ and $\theta(c) = \theta(c_1)$. Using observation 1:

$$\begin{aligned} & \big(\forall 1 < i \leq k : \theta(c_i) \in \Pi(\theta(c_1)) \vee c_1 \rightharpoonup_{\mathsf{R}}^{+} c_i\big) \wedge c \rightharpoonup_{\mathsf{R}}^{+} c_1 \\ \Longrightarrow\ & \big(\forall 1 < i \leq k : \theta(c_i) \in \Pi(\theta(c)) \vee c_1 \rightharpoonup_{\mathsf{R}}^{+} c_i\big) \wedge c \rightharpoonup_{\mathsf{R}}^{+} c_1 \\ \Longrightarrow\ & \big(\forall 1 < i \leq k : \theta(c_i) \in \Pi(\theta(c)) \vee c \rightharpoonup_{\mathsf{R}}^{+} c_i\big) \wedge c \rightharpoonup_{\mathsf{R}}^{+} c_1 \\ \Longrightarrow\ & \forall 1 \leq i \leq k : \theta(c_i) \in \Pi(\theta(c)) \vee c \rightharpoonup_{\mathsf{R}}^{+} c_i \end{aligned}$$

which is observation 1 itself. □

The concept of threaded interprocedural witnesses is needed to define slices in concurrent interprocedural programs based on the threaded interprocedural PDG which is presented next.

3.2 The Threaded Interprocedural PDG

As threaded programs have a special representation in the control flow graph, they also need special representation in the program dependence graph to enable accurate slicing.

The extensions of the ICFG to the tICFG do not influence domination and control dependence. Data dependence in traditional CFGs is based on reaching definitions. However, this is inadequate for tICFGs as reaching definitions include definitions of parallel executing threads. For slicing purposes it is desirable to separate reaching definitions from parallel threads, which makes the data dependence in non parallel threads computable by standard techniques for sequential programs.

3.2.1 Interference Dependence

When a variable is defined in one thread and referenced in another parallel executing thread, *interference* occurs, which must be represented explicitly.

Definition 4. A node n is called *interference dependent* on m, if

1. there is a variable v, such that $v \in \mathsf{def}(m)$ and $v \in \mathsf{ref}(n)$, and
2. $\theta(m) \in \Pi(\theta(n))$
   (n and m may potentially be executed in parallel).

The dependences introduced by interference cannot be handled with normal data dependence as normal dependence is transitive: The transitivity of data and control dependence results from their definitions where a sequential path between the dependent nodes is demanded. The composition of paths in the CFG always results in a path again. Interference dependence is not transitive: If a statement x is interference dependent on a statement y, which is interference dependent on z, then x is only dependent on z iff there is a possible execution where these three statements are executed one by one: The sequence $\langle z, y, x \rangle$ of the three statements has to be a threaded witness in the tICFG.

Control and data dependence is computed for the different threads, assuming that every thread is independent from all the others. This results in a set of non-connected interprocedural program dependence graphs—one IPDG for each thread. The next step is to compute the interference dependence between threads. If a variable is defined in one thread and referenced in another parallel executing thread, an interference dependence edge is generated between the corresponding nodes.

The *threaded interprocedural PDG (tIPDG)* is the union of the IPDGs for each thread, connected by the interference dependence edges. The control flow, call and return edges from the ICFG are also present in the tIPDG (this is necessary for the later algorithm). The usual call edges in IPDGs have to be distinguished from the

control flow call edges. Therefore, control flow edges will be denoted with $\rightharpoonup$ and dependence edges with $\rightarrow$. The different types of dependence edges are distinguished by a label d in $\xrightarrow{d}$, e.g. dd for a data dependence edge $\xrightarrow{\text{dd}}$. An interference dependence edge $n \xrightarrow{\text{id}} m$ will be inserted for all (n,m) if there is a variable v which is defined at n, referenced at m and $\theta(n) \in \Pi(\theta(m))$ holds. Interference can be computed more precisely by using a refined Π according to [17] or [22, 23].

Definition 5. A path $P = \langle n_1, \ldots, n_k \rangle$ in a tIPDG is a *threaded interprocedural realizable path*, iff

1. the path contains no edge from the control flow graph (control flow edges, control flow call or return edges):
$$n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_k,$$
2. every sub-path without an interference edge is an interprocedural realizable path in the containing IPDG:
$$\forall 1 \leq i < k, i < j \leq k : (\forall i \leq m < j : d_m \neq \text{id}) \Rightarrow \langle n_i, \ldots, n_j \rangle \text{ is interprocedural realizable},$$
and
3. a threaded interprocedural witness W exists and corresponds to path P:
$$\exists W = \langle c_1, \ldots, c_k \rangle : P = \langle T(c_1), \ldots, T(c_k) \rangle$$

If a node n is reachable from n' via a threaded interprocedural realizable path, it is denoted as $n' \rightarrow^\star_{\mathsf{R}} n$.

Slicing on the PDG of sequential programs is a simple graph reachability problem because control and data dependence are transitive. As interference dependence is not transitive, this definition of a slice for PDGs is not valid for tIPDGs and hence the standard algorithms are not adequate.[4] Therefore, slicing is now defined in terms of threaded interprocedural realizable paths:

Definition 6. The (*backward*) *slice* $S_\theta(n)$ of a tIPDG at a node n consists of all nodes m on which n depends transitively via a threaded interprocedural realizable path:
$$S_\theta(n) = \{m \mid m \rightarrow^\star_{\mathsf{R}} n\}$$

Definition 7. The *slice* $S_\theta(c)$ in a tIPDG for a context c consists of all contexts c' for which $T(c)$ depends transitively on $T(c')$ via a threaded interprocedural realizable path:
$$S_\theta(c) = \{c' \mid \exists P = \langle T(c'), \ldots, T(c) \rangle : T(c') \rightarrow^\star_{\mathsf{R}} T(c)\}$$
with the additional constraint that P's threaded interprocedural witness W is $W = \langle c', \ldots, c \rangle$. This definition stays the same for a slice $S(c)$ restricted to an IPDG.

These definitions of threaded interprocedural paths and slices are not computable because the set C of possible contexts is infinite when recursion exists in the analyzed program. However, the next section will show a way to make the set of possible contexts finite and thus slices computable.

[4] The "classical" definition of a slice is any subset of a program that does not change the behavior in respect to the criterion: a program is a correct slice of itself. Therefore, if interference is modeled with normal data dependence, the resulting slices are correct but imprecise.

3.3 Slicing the tIPDG

For reachability the amount of recursive calls is irrelevant and cycles in the ICFG can be folded into a single node representing the strongly connected region. There are two sources for cycles:

1. Loops in the program cause cycles only in the intraprocedural part of the CFG and have no 'stacking' influence on the amount of possible contexts.
2. Recursive calls cause cycles over call and return edges. If such cycles are replaced by a single node, reachability is not realizable interprocedurally because call and return edges are not matched correctly.

The cycles are replaced by a two pass approach to keep the matching of call and return edges intact. The first pass finds and folds strongly connected components in the ICFGs consisting of control flow and call edges but ignores return edges. The second pass finds and folds cycles in the resulting graph consisting of control flow and return edges (now ignoring call edges). This replacement generates a function ρ that maps nodes from the original ICFG $G = (N^\star, E^\star, n_0^s, n_0^e)$ to the new graph $\overline{G} = (\overline{N}^\star, \overline{E}^\star, n_0^s, n_0^e)$:

$$\rho(n \in N^\star) = \begin{cases} n & \text{if } \forall n \xrightarrow{d_1} \ldots \xrightarrow{d_k} n : \\ & \quad \exists i, j : d_i = \text{call} \wedge d_j = \text{return} \\ n' \notin N^\star & \text{otherwise} \end{cases}$$

$$\rho(n) \neq n \Leftrightarrow \forall n \xrightarrow{d_1} n_1 \ldots n_{k-1} \xrightarrow{d_k} n : \big(\forall i, j : d_i \neq \text{call} \wedge d_j \neq \text{return}\big) \Rightarrow \big(\forall i : \rho(n_i) = \rho(n)\big)$$

$$\overline{N}^\star = \{\rho(n) \mid n \in N^\star\}$$

Every interprocedural realizable path in the resulting graph has a corresponding realizable path in the original graph. Due to unrealizable matchings of call and return edges, there are still cycles in the graph.

Based on the newly created graph $\overline{G}$ the set of contexts and the 'reaches' relation between contexts are redefined: The execution state $c \in \overline{C}_t$ of thread t is a stack $c = n_0 \ldots n_k$ of nodes $n_i \in \overline{N}_t^\star$ with the topmost node $T(c) = n_0$. The 'pop' function P is defined as before: $P(c) = n_1 \ldots n_k$.

Definition 8. A context $c_1 \in \overline{C}_t$ *reaches directly* another context $c_2 \in \overline{C}_t$: $c_1 \rightharpoonup_{\mathsf{R}} c_2$, iff one of the following alternatives holds:

1. an edge $n_1 \rightharpoonup n_2 \in E_t^\star$ exists with $n_1 \rightharpoonup n_2 \notin E_t^C$ and where $T(c_1) = \rho(n_1) \wedge T(c_2) = \rho(n_2) \wedge P(c_1) = P(c_2)$ (corresponding edge in the CFG of a procedure exists),
2. a call edge $n_1 \rightharpoonup n_2 \in E_t^C$ exists and
   (a) $T(c_1) = \rho(n_1) \wedge T(c_2) = \rho(n_2)$,
   (b) $T(c_1) \neq T(c_2) \rightarrow c_1 = P(c_2)$, and
   (c) $T(c_1) = T(c_2) \rightarrow c_1 = c_2$

   (corresponding call edge exists), or
3. a return edge $n_1 \rightharpoonup n_2 \in E_t^C$ exists and
   (a) $T(c_1) = \rho(n_1) \wedge T(c_2) = \rho(n_2)$,
   (b) $T(c_1) \neq T(c_2) \rightarrow P(P(c_1)) = P(c_2) \wedge \exists n_3 \rightharpoonup n_2 \in (E_t^\star - E_t^C) : \rho(n_3) = T(P(c_1))$, and
   (c) $T(c_1) = T(c_2) \rightarrow c_1 = c_2$

(corresponding return edge exists, leading to a return node which matches the last call node).

A context c *reaches* another context c': $c \rightharpoonup_{\mathsf{R}}^{+} c'$, iff a series of contexts $c_1,\ldots,c_n$ exists with $c = c_1 \wedge c' = c_n \wedge \forall_{1\le i<n} c_i \rightharpoonup_{\mathsf{R}} c_{i+1}$. The set of possible contexts for ICFG $G_t = (N_t^\star, E_t^\star, n_t^s, n_t^e)$ is $\overline{C}_t = \{c' \mid \rho(n_t^s) \rightharpoonup^{+} c' \vee c' = \rho(n_t^s)\}$.

With this definition $\overline{C}_t$ must be finite because traversing call edges does not 'stack' call nodes inside recursive cycles.

Now, a context $c_1 = n_0 \ldots n_k$ reaches another context $c_2 = m_0 \ldots m_l$ ($c_1, c_2 \in C_t$) in terms of definition 1, iff $\overline{c}_1 = \overline{n}_0 \ldots \overline{n}_{k'}$ reaches $\overline{c}_2 = \overline{m}_0 \ldots \overline{m}_{l'}$ in terms of definition 8 ($\overline{c}_1, \overline{c}_2 \in \overline{C}_t$), where $\rho(n_0)\ldots\rho(n_k) = \overline{n}_0{}^{i_1} \ldots \overline{n}_{k'}{}^{i_{k'}} \;\wedge\; \rho(m_0)\ldots\rho(m_l) = \overline{m}_0{}^{j_1} \ldots \overline{m}_{l'}{}^{j_{l'}}$. Based on this observation, definition 8 can now be used in the definitions and theorems about threaded interprocedural witnesses. Hence it is decidable and computable whether a path in a tIPDG is threaded interprocedural realizable. Thus, slices $S_\theta(c)$ can be computed using definition 8, denoted as $\overline{S}_\theta(c)$ and $\overline{S}(c)$.

A naive implementation would enumerate the possible paths to the slicing criterion node and check them to be threaded interprocedural realizable paths. This way is too expensive and various approaches are combined to a more efficient approach for slicing based on tIPDGs, shown in algorithm 1. There, the extraction of the ith element c_i in a tuple $\Gamma = (c_1,\ldots,c_n)$ is denoted by $\Gamma[i]$. The substitution of the ith element c_i in a tuple $\Gamma = (c_1,\ldots,c_n)$ with a value x will be denoted as $[x/_i]\Gamma$. Its basic idea is the coding of possible execution states of all threads in tuples $(c_1,\ldots,c_{|\Theta|-1})$, where the c_i are contexts (in the tIPDG). The value c_i represents that it is still possible to reach context c_i in thread θ_i (a value of $\perp$ does not restrict the state of execution). This is used to keep track of the context c_i where thread θ_i has been left by following an interference edge. If another interference edge is followed back into the thread at node q, the reachability of c from the contexts c' of q ($q = T(c')$) can be checked. It assures that paths over interference edges are always threaded witnesses in the tICFG. This is the reason why the control flow edges have to be kept in the tIPDG. Reachability can be computed without checking threaded witness properties, because the simplified prepending theorem 1 allows checking based on the last reached context in a thread (the first element of a threaded witness relevant to the thread).

The algorithm keeps a worklist of pairs of contexts and state tuples which have to be examined. For computation of the slice $\overline{S}(c)$ inside a thread (not shown in algorithm 1), every edge reaching the top node of the current context is examined and handled dependent on its type. A new pair consisting of the new context and the modified state tuple is inserted into the worklist. According to theorem 2 this is done without checking the threaded witness property.

Interference edges are ignored while computing $\overline{S}(c)$, they are handled explicitly in algorithm 1. An interference dependence edge may only be considered if the (old) context relevant to the source node thread is reachable from a context at the source node in the tICFG (all examined paths are still threaded witnesses). Then, the new pair with the updated state tuple is inserted into the worklist. The resulting slice is the set of nodes which is constructed out of the first elements of the inserted pairs.

Algorithm 1 contains two bottlenecks which are eased in the following: the computation of $\overline{C}_t(n)$ and $\overline{S}(c)$. The idea of subsuming call strings is used to build subsuming contexts: With subsuming contexts, $\overline{C}_t(n_1)$ just contains one element $n_1\diamond$ which subsumes all contexts $n_1\ldots n_k \in \overline{C}_t(n_1)$. Now, the third constraint of definition 8 has an alternative:

> A return edge $n_1 \rightharpoonup n_2 \in E_t^C$ exists and either constraint 3 of definition 8 holds or
>
> 1. $c_1 = \rho(n_1)\diamond \wedge\; T(c_2) = \rho(n_2)$ and
> 2. $T(c_1) = T(c_2) \;\rightarrow\; c_1 = c_2$
>
> (return edge exists which returns to a node matching $\diamond$ automatically).

In particular, traversal of a return edge $n_1 \rightharpoonup n_2$ from a context $\rho(n_1)\diamond$ leads to a context $\rho(n_2)\diamond$.

The second bottleneck is the computation of slice $\overline{S}(c)$. Section 2.2 showed that the computation based on explicit context may suffer from combinatorial explosion. Because the computation is restricted to an IPDG, summary edges can be generated and used for more efficient slicing. Instead of computing an expensive slice $S(c)$, a traditional slice using summary edges is computed. In this slice, all nodes are identified with at least one incoming interference dependence edge. For each of these nodes, a chop is computed between the node and the original slicing criterion. This chop is truncated non-same-level which can be computed efficiently [25]. Now, only along nodes in this chop the more expensive slice $S(c)$ is computed. This is much more efficient because a far smaller set of nodes is visited. In algorithm 2, only the modifications to algorithm 1 implementing this improvement are shown.

Algorithm 1 Slicing Algorithm for tIPDGs, $\overline{S}_\theta$

Input: The tIPDG $G = (N^\Pi, E^\Pi, S^\Pi, X^\Pi)$
The slicing criterion $s \in N^\Pi$
Output: The slice S, a set of nodes of the tIPDG

Let $\overline{C}_t(n)$ be a function which returns the set of possible contexts for a node n in IPDG G_t.

Initialize the worklist with an initial state tuple:
$\Gamma = (n^e_{\theta_1},\ldots,n^e_{\theta_{|\Theta|}})$, *every thread is at its end node*
$W = \{(c,\Gamma') \mid t = \theta(s) \wedge c \in \overline{C}_t(s) \wedge \Gamma' = [c/_t]\Gamma\}$
$M = W$, *Mark the contents of the worklist*

repeat
  Remove the next element $w = (c,\Gamma)$ *from* W*:*
  $W = W - \{w\}$
  $S = S \cup \{n \mid \rho(n) = T(c)\}$

  Compute a slice $\overline{S}(c)$ for c in the IPDG G_t.
  Compute the set of node and contexts with $\geq$*1 incoming interference dependence edges:*
  $I = \{c' \in \overline{S}(c) \mid \exists n \in N^\Pi : n \xrightarrow{\text{id}} T(c')\}$

  foreach $i \in I$ **do**
    foreach edge $n \xrightarrow{\text{id}} n'$, $\rho(n') = T(i)$**do**
      $t = \theta(T(i))$ *current thread*
      $t' = \theta(n)$ *reached thread*
      $\Gamma' = [i/_t]\Gamma$
      Compute valid contexts in the reached thread:
      $C' = \{c' \mid c' \in \overline{C}_{t'}(n) \wedge c' \rightharpoonup_{\mathsf{R}}^{+} \Gamma[t']\}$
      Fill the worklist:
      foreach $w' \in \{(c'', [c''/_{t'}]\Gamma') \mid c'' \in C'\}$ **do**
        if $w' \notin M$ **then**
          $M = M \cup \{w'\}$
          $W = W \cup \{w'\}$
until $W = \emptyset$
return S

Algorithm 2 Improved Slicing Algorithm for tIPDGs

$\vdots$
foreach $t \in \Theta$ **do**
  Generate summary edges and transform IPDG G_t into a SDG
$\vdots$
repeat
  Remove the next element $w = (c, \Gamma)$ *from* W*:*
  $W = W - \{w\}$.
  $N_c = \{n \mid \rho(n) = T(c)\}$
  $S = S \cup N_c$

  Compute a slice $S(n')$ for one $n' \in N_c$ in the SDG G_t.
  $I_N = \{n_1 \in S(n') \mid \exists n_2 \in N^{\Pi} : n_1 \xrightarrow{\text{id}} n_1\}$
  Nodes in the slice with ≥ 1 *incoming interference dep. edges.*

  foreach $i_N \in I_N$ **do**
    Compute a truncated non-same-level chop $C^{\text{TN}}(i_N, n')$
    Compute a slice $\overline{S}(c)$ only along nodes in $C^{\text{TN}}(i_N, n')$.
    foreach $i \in \overline{S}(c) \mid T(c) = \rho(i_N)$ **do**
      foreach incoming interference dep. edge $n \xrightarrow{\text{id}} T(i)$**do**
        $t = \theta(T(i))$ *current thread*
        $t' = \theta(n)$ *reached thread*
        $\vdots$
until $W = \emptyset$
return S

4. EXTENSIONS

For simplicity, additional features of threaded programs have not been looked at so far. Most models of parallel execution include some methods of synchronization. Two such methods are synchronized blocks and send/receive communication, which will be discussed in the following, including a look at other models of concurrency.

4.1 Synchronized Blocks

Synchronized blocks are blocks of statements that are executed atomically: Interference cannot arise inside such a block. An example for an instance are monitors:

```
     ...
5    synchronized {
6      x = y;
7      if (z > 0)
8        x = x + z;
9    }
     ...
```

In this example, interference cannot happen to the variables `x`, `y` and `z` while the synchronized block is executing: The usage of `x` in line 8 can only be data dependent on line 6. However, both definitions in line 6 and line 8 can interfere with any usage or definition of the same variable in other threads.

One possibility is to ignore the synchronization statement and treat synchronized blocks as normal blocks. This is a conservative approximation and will only add unrealizable interference. The precise solution is to compute the set of definitions reaching the end of the synchronized block and the set of usages reaching the entry.

Definition 9. A node m is *interference dependent* on node n, if

1. there is a variable v, such that $v \in \mathsf{def}(n)$ and $v \in \mathsf{ref}(m)$,
2. n is not embedded in a synchronized block or the definition at n reaches the exit of the synchronized block,
3. m is not embedded in a synchronized block or the usage at m reaches the entry of the synchronized block, and
4. $\theta(n) \in \Pi(\theta(m))$ (n and m may potentially be executed in parallel) or the synchronized blocks of n and m would potentially execute in parallel without synchronization.

To improve the precision of the interference dependence edges, the MHP information in [22, 23] can be (and should be) used.

4.2 Communication via Send/Receive

If two threads exchange information via send and receive communication, the execution of the receiving thread may block until the sending thread has sent some information. This has three effects on control and data dependence:

1. The exchange of information creates a data dependence between the sending and receiving statement. To distinguish it from normal data dependence, such dependence may be called communication dependence. In order to omit time travel, communication dependence must be treated like interference dependence during slicing.
2. Because the execution at a receive may be blocked until some other thread sends some data, the computation of Π becomes more complex. However, a conservative approximation is to ignore such blocking as the Π function will still return an (imprecise) superset of the realizable relations.
3. A receiving statement that may block (together with its successors) can be seen as control dependent on the sending statement. This control dependence can be computed simply by inserting the communication dependence into the control flow graph and treat it as a control flow edge.

4.3 Other Models of Concurrency

The presented simple model of concurrency is similar to the Ada concurrency mechanism, except for synchronization. To model the Ada-style rendezvous, the send/receive-style communication can be used.

To allow code sharing between threads, the duplication of the shared code is sufficient. Every thread will then have its own instance of the shared code.

The synchronization extensions can also be used to represent a concurrency model where the different threads are allowed to be started and stopped from other threads. This is similar to the concurrency model of Java. However, in Java, threads are generated dynamically which cannot be represented in the simple concurrency model. Therefore, data flow analysis is needed to compute a conservative approximation of the set of possible threads. The (now) static set can be represented in the simple concurrency model with synchronization extensions, enabling more precise slicing of concurrent Java programs.

The send/receive-style synchronization can also be used to simulate a cobegin/coend parallelism within the presented model: The branches of the cobegin/coend statement are transformed into single threads. At cobegin, synchronization statements that start the newly created threads are introduced, and at coend, synchronization statements are introduced to wait until the newly created threads

have finished. This requires a modified Π-function—the earlier trivial definition of Π has never been exploited in a proof and thus, the presented theorems are not weakened.

5. RELATED WORK

There are many variations of the program dependence graph for threaded programs like parallel program graphs [26, 4, 3, 6, 2]. Most approaches to static or dynamic slicing of threaded programs are based on such dependence graphs.

Dynamic slicing of threaded or *concurrent* programs has been approached by different authors [18, 5, 6, 14, 12, 9] and is surveyed in [29]. Probably the first approach for *static* slicing of threaded programs was the work of Cheng [3, 35, 4]. He introduced some dependences which are more specialized than the previously presented dependences. These are needed for a variant of the PDG, the *program dependence net* (PDN). His *selection* dependence is a special kind of control dependence; his *synchronization* dependence is basically control dependence resulting from the previously presented communication dependence. Cheng's *communication dependence* is a combination of data dependence and the presented communication dependence. Although the tIPDG is not mappable to his PDN and vice versa, both graphs are similar in the number of nodes and edges. Cheng defines slices simply based on graph reachability. The resulting slices are not precise, as they do not take into account that dependences between parallel executed statements are not transitive. Therefore, the integration of his technique of slicing threaded programs into slicing threaded object oriented programs [35, 36, 33, 34] has the same problem.

After the first publication of our previous work on slicing of concurrent programs [15] more work on *precise* static slicing has been done: [21] improves the earlier version of the presented work and [1] is a different approach but also based on dependence graphs.

There is a series of works which use static slicing of concurrent programs but treat interference transitive and accept the imprecision: [10, 7] present the semantics of a simple multi-threaded language that contains synchronization statements similar to the Java virtual machine. Based on this statements, they introduce and define additional types of dependence: divergence dependence, synchronization dependence and ready dependence. [19] applies Cheng's approach to slice Promela for model checking purposes.

Data flow analysis frameworks exist also for multi-threaded programs: [13] uses a cobegin/coend model of parallelism. Seidl [27] presents a framework for the problems of strong copy constant propagation and (ordinary) liveness of variables in concurrent programs. He proves that these problems have the same complexity in both sequential and parallel cases. Slicing is a harder problem than reaching definitions. Proofs for lower bounds can be found in [20, 24]. Both show that precise slicing of concurrent programs is undecidable in the interprocedural case.

6. CONCLUSIONS

All previous approaches known to the author to slice concurrent programs precisely rely on the inlining of called procedures ([21, 1]). The presented approach is the first which is able to slice concurrent *recursive* programs accurately.

The presented approach is precise up to threaded interprocedural realizable paths. The undecidability result in [24] does not apply to the simple model as it does not contain synchronization. Our approach is not optimal in terms of [20]—their undecidability results apply to the model used here: It is possible that a thread kills a definition in a different thread. Within the presented approach we have ignored such killing. To explore how much precision is lost through this approach, we have done another experiment: We have modified the underlying data flow analysis for sequential ANSI C to ignore killing. A definition is now never killed by another definition. This adds more data dependences to the PDG. The results were quite surprising: On average, the generated slices are only 5% larger than before. With this result we argue that ignoring the killing effects of parallel executing threads has only a small influence on precision.

Acknowledgements: Silvia Breu, Maximilian Störzer and Christian Hammer provided valuable comments on an earlier version of this paper.

7. REFERENCES

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

[2] Z. Chen, B. Xu, J. Zhao, and H. Yang. Static dependency analysis for concurrent ada 95 programs. In *7th Ada-Europe International Conference on Reliable Software Technologies*, pages 219–230, 2002.

[3] J. Cheng. Slicing concurrent programs. In *Automated and Algorithmic Debugging, 1st International Workshop, AADEBUG'93*, pages 223–240, 1993.

[4] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *International Conference on Advances in Parallel and Distributed Computing*, 1997.

[5] J.-D. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Prog. Lang. Syst.*, 13(4):491–530, 1991.

[6] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 497–511, 1992.

[7] M. B. Dwyer, J. C. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng. Slicing multi-threaded java programs: A case study. Technical Report KSU CIS TR 99-7, Department of Computing and Information Sciences, Kansas State University, 1999.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, 1987.

[9] D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In *High Performance Computing - HiPC 2000, 7th International Conference*, pages 15–26, 2000.

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

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

[12] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *International Conference on Software Maintenance*, pages 222–231, 1995.

[13] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Prog. Lang. Syst.*, 18(3):268–299, 1996.

[14] B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.

[15] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for*

Software Tools and Engineering (PASTE'98), pages 35–42. ACM Press, 1998. ACM SIGPLAN Notices 33(7).

[16] J. Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, pages 22–31, 2002.

[17] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, 1989.

[18] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 135–144, 1988.

[19] L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking. In *Proceedings of the 4th International SPIN Workshop*, 1998.

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

[21] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *International Conference on Software Testing and Analysis (ISSTA 2000)*, pages 180–190, 2000.

[22] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of 6th International Symposium on the Foundations of Software Engineering*, pages 24–34, 1998.

[23] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In O. Nierstrasz and M. Lemoine, editors, *7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 338–354. Springer, 1999.

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

[25] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 41–52, 1995.

[26] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, pages 633–655, 1993.

[27] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. In *Proceedings of ESOP'00, 9th European Symposium on Programming*, volume 1782 of *LNCS*, 2000.

[28] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[29] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3), 1995.

[30] N. Uchihira, S. Honiden, and T. Seki. Hypersequential programming. *IEEE Concurrency*, pages 44–54, 1997.

[31] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[32] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.

[33] J. Zhao. Multithreaded dependence graphs for concurrent java programs. In *Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23, 1999.

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

[35] J. Zhao, J. Cheng, and K. Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pages 312–320, 1996.

[36] J. Zhao, J. Cheng, and K. Ushijima. A dependence-based representation for concurrent object-oriented software maintenance. In *Procedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 60–66, 1998.