Slicing of Concurrent Programs and its Application to Information Flow Control

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften /
Doktors der Naturwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

vorgelegte

Dissertation

von

Dennis Giffhorn

aus Wolfsburg

Tag der mündlichen Prüfung:

Erster Gutachter: Prof. Dr.-Ing. Gregor Snelting

Zweiter Gutachter: Prof. Dr. Markus Müller-Olm

Acknowledgment

I am thankful to my adviser Prof. Gregor Snelting for providing the support, freedom and protection to do research without pressure. I thank Prof. Martin Müller-Olm for contributing the second review of this thesis.

I would also like to thank my colleagues Matthias Braun, Sebastian Buchwald, Jürgen Graf, Christian Hammer, Martin Hecker, Andreas Lochbihler, Denis Lohner, Manuel Mohr, Maximilian Störzer, Mirko Streckenbach, Daniel Wasserrab, and Andreas Zwinkau for fruitful discussions and cooperations. Christian Hammer and Jürgen Graf were the other members in the Joana team, who developed and provided innumerable algorithms without which my thesis would not have been possible. Andreas Lochbihler's comprehension of information flow control resulted in several important improvements of my information flow control technique. The formal definitions and proofs of slicing and information flow control developed by Daniel Wasserrab and Denis Lohner provided valuable inspirations.

I would like to give a special mention of my students who helped us to implement, extend and maintain the Joana framework. Bo Li maintained and extended the Graphviewer, a tool that graphically visualizes program dependence graphs. Martin Seidel was a great help in improving the Joana Eclipse plugin, which integrates the Joana framework into the Eclipse framework.

A big 'thank you' goes to Alexandra Schmidt, Eva Reichhart, Eva Veitweber, Katja Weisshaupt, Sonja Klinger, Brigitte Sehan, Adolf Stockinger and Bernd Traub, whose skillful administrative work made it possible to pass all kinds of bureaucratic obstacles.

The Valsoft/Joana project was funded in part by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-1 and Sn11/9-2).

Karlsruhe, May 2011 *Dennis Giffhorn*

Zusammenfassung

Informationsflusskontrolle beschäftigt sich mit der Sicherheit von vertraulichen Daten während ihrer Verarbeitung durch Software. Sie stellt sicher, dass die Software die vertraulichen Daten weder unautorisierten Empfängern preisgibt, noch mit Daten aus unautorisierten Quellen vermischt. Es handelt sich damit um ein Konzept, mit dem traditionelle Sicherheitstechniken wie Verschlüsselung und Zugriffskontrolle ergänzt werden können, um die Sicherheit vertraulicher Daten zusätzlich zu erhöhen.

Diese Dissertation präsentiert ein Verfahren zur Informationsflusskontrolle für nebenläufige Programme mit Fäden und gemeinsamen Speicher. Das Verfahren garantiert die Sicherheit vertraulicher Daten hinsichtlich eines angemessenen Angreifermodells und basiert auf *Programmabängigkeitsgraphen* (PAGs), einer sprachunabhängigen Darstellung des Informationsflusses innerhalb eines Programms. Durch die Verwendung von PAGs kann das Verfahren auf alle Programme angewandt werden, die in PAGs übersetzt werden können.

Die Dissertation ist Teil des Projekts *ValSoft/Joana* am Lehrstuhl für Programmierparadigmen des Karlsruher Instituts für Technologie. Das Ziel dieses Projekts ist die Entwicklung von hochpräzisen Techniken zur Informationsflusskontrolle für sicherheitsrelevante Software in gängigen Sprachen wie Java oder C. Diese Techniken sollen Sicherheit garantieren, also keine Lecks übersehen, und durch ihre hohe Präzision so wenig Fehlalarme wie möglich generieren. Das Herz des Projekts ist *Joana*, ein Rahmenwerk zur Programmanalyse mittels PAGs, welche den Informationsfluss innerhalb eines Programms mit sehr hoher Präzision wiedergeben. Joanas Zielsprache ist Java, eine objektorientierte Sprache mit Fäden, gemeinsamen Speicher und einer hohen Verbreitung, und damit auch einer hohen Relevanz. Joana beherrscht vollen Java Bytecode, inklusive Objekten, dynamischer Bindung, Seiteneffekten von Prozeduraufrufen und Fäden, und ist derzeit in der Lage, PAGs für Java-Programme mit bis zu 30.000 Zeilen Code zu generieren.

Zu Beginn dieser Dissertation zielte ValSoft/Joana auf die Analyse von Java-Programmen für Chipkarten. Derartige Programme sind verhältnismäßig klein, und die dafür angebotene Java-Variante, JavaCard, enthielt zu jenem Zeitpunkt keine Fäden. Das Joana-Rahmenwerk war bereits in der Lage, PAGs für Java-Programme mit Fäden erzeugen, machte aber zu diesem Zweck sehr konservative Annahmen, die zu sehr ungenauen Resultaten führten. Desweiteren konnte das bereits vorhandene Verfahren zur Informationsflusskontrolle nur mit sequentiellen Java-Programmen umgehen. Da Fäden ein integraler Bestandteil von Java sind, war es daher notwendig, Joana mit entsprechenden Analysen für Fäden zu erweitern, um Joana auf beliebige

Java-Programme anwenden zu können. Dies war das Thema dieser Dissertation. Sie erbringt dabei die folgenden Beiträge:

- Joana wurde durch eine Parallelitätsanalyse erweitert, welche diejenigen Teile eines Programms identifiziert, die nebenläufig zueinander ausgeführt werden können. Mithilfe dieser Analyse kann eine große Anzahl von Programmabhängigkeiten, die bisher von Joana für nebenläufige Programme berechnet wurden, als redundant erkannt und verworfen werden.

- PAGs werden üblicherweise mittels Graphtraversierung weiterverarbeitet und untersucht. Ein PAG eines nebenläufigen Programms berücksichtigt alle möglichen Verschränkungen der Fäden des Programms, wodurch der PAG Abhängigkeiten enthalten kann, die auf zueinander inkompatiblen Verschränkungen beruhen, und somit auch Pfade, die in der Realität von keiner Programmausführung verursacht werden können. Es bleibt den Traversionsalgorithmen überlassen, diese unerfüllbaren Pfade zu erkennen und auszuschliessen. Zu diesem Zweck wurden in der Dissertation hochpräzise Traversionsalgorithmen, die dieses Problem in Angriff nehmen, untersucht, verbessert, neu entwickelt und miteinander verglichen.

- Die vorgenommenen Erweiterungen von Joana wurden schließlich zur Entwicklung eines auf PAGs und ihrer Traversion basierenden Verfahrens zur Informationsflusskontrolle für nebenläufige Programme verwendet. Das Verfahren garantiert die Vertraulichkeit des untersuchten Programms unabhängig vom Scheduler und macht dabei im Verhältnis zu anderen Ansätzen wenig Einschränkungen an das Programm.

Besonderes Augenmerk richtete der Autor dabei auf die Fähigkeit der entwickelten Algorithmen, vollen Java Bytecode zu behandeln, sodass die zu analysierenden Programme keinen sprachlichen Einschränkungen unterliegen. Zum anderen werden für viele behandelte Teilprobleme mehrere Algorithmen mit unterschiedlicher Präzision, Laufzeitkosten und Implementierungsaufwand präsentiert, sodass Anwendern die Auswahl des geeignetsten Algorithmus ermöglicht wird. Alle vorgestellten Algorithmen wurden vollständig ins Joana-Rahmenwerk integriert. Es ist damit eines der wenigen, die PAG-basierte Analysen von nebenläufigen Java-Programmen erlauben. Durch die Integration des entwickelten Verfahrens zur Informationsflusskontrolle beinhaltet Joana unseres Wissens nach die erste implementierte Informationsflusskontrolle für nebenläufige Programme, die in einer echten, weitverbreiteten Sprache verfasst sind.

Abstract

Information flow control is concerned with the security of sensitive information being processed by a software. It aims to ensure that software does not leak rightfully accessed sensitive information to unauthorized sinks or taints it with data from unauthorized sources during its computations. It can be used supplementary to established security techniques like access control or encryption to enhance the protection of sensitive information.

This thesis presents a practical technique for information flow control for concurrent programs with threads and shared-memory communication. The technique guarantees confidentiality of information with respect to a reasonable attacker model and utilizes *program dependence graphs* (PDGs), a language-independent representation of information flow in a program, which allows to apply it to all programs that can be translated into a PDG.

The thesis is embedded in the *ValSoft/Joana* project of the Programming Paradigms Group at the Karlsruhe Institute of Technology. The project aims to develop IFC techniques that analyze security relevant software written in established languages, like C or Java, with maximal precision, so that no information leaks are missed and very few false alarms are generated. Its core is the *Joana* framework, a general-purpose program analysis framework based on PDGs, highly precise representations of the possible information flows in a program. Joana focuses on Java and is currently able to build PDGs for Java programs with up to 30.000 lines of code. It is capable of full Java bytecode, including features like objects, dynamic dispatch, side effects of procedure calls, exceptions and threads.

At the time this thesis started, ValSoft/Joana focused on analyzing Java programs for smart cards, written in JavaCard. Such programs are comparatively small, and JavaCard did not support threads at that time. The Joana framework was already able to create PDGs for Java programs with threads, but made very conservative approximations in order to do so, leading to very imprecise results. Furthermore, the already existing information flow control technique was limited to sequential Java programs. Threads are an integral component of the Java language, so full support of Java threads had to be integrated into Joana in order to analyze general Java programs. This was the task of this thesis. It accomplishes that task by making the following major contributions:

- It extends Joana with a *may-happen-in-parallel* analysis, which identifies those parts of a Java program that may execute concurrently. Using this analysis, a large number of dependences computed by the previous versions of Joana can be identified as redundant.

- PDGs are usually examined via graph traversal. A PDG of a concurrent program comprises all possible ways of interleaving of its threads, wherefore it may contain dependences based on incompatible ways of interleaving and thus, paths that cannot be realized by any program execution. It is left to the graph traversal algorithm to avoid these unrealizable paths. To this end, the thesis investigates, develops, improves and compares highly precise algorithms for the traversal of PDGs that address that problem.

- The above enhancements of Joana are used to develop an information flow control technique for concurrent programs based on PDGs and their traversal. The technique guarantees the confidentiality of successfully verified programs independent from the concrete scheduler and is permissive compared with other approaches.

A crucial objective of the author was to develop algorithms capable of full Java bytecode, to avoid restrictions on the syntax of the programs to be analyzed. Furthermore, for many investigated subproblems, alternative algorithms are presented, which solve the subproblem with different degrees of precision, runtime costs and implementation effort. This enables a user to choose the algorithm that suits best. All presented algorithms have been integrated into Joana, resulting in a PDG-based, general-purpose program analysis framework for full Java bytecode, containing, to the best of our knowledge, the first reported implementation of an information flow control technique for a contemporary language with threads and shared memory.

Contents

1. Introduction

Information flow control is concerned with the security of sensitive information processed by software. Whereas security of information is predominantly understood as an *accessing* problem, tackled by techniques such as access control or encryption, information flow control sees it as a *processing* problem: Software that rightfully accesses sensitive information might intentionally or unintentionally leak it to unauthorized sinks, violating its *confidentiality*, or taint it with data from unauthorized sources, violating its *integrity*. It is complementary to access control, encryption and likewise techniques and can be combined with them in order to achieve *end-to-end* security, protecting sensitive information during its whole lifetime.

Figure 1.1 shows on the left side typical confidentiality violations in sequential programs. The depicted program reads a secret PIN (we assume that it has the right to do so), copies it to variable `y` and prints it directly to the screen. A somewhat more subtle leak happens inside the `if` conditional. The output of 0 reveals that the PIN is smaller than 1234. Information flow control is supposed to detect and prevent these information leaks. This can be done dynamically, during the execution of the program, or statically, at compile-time. This thesis is concerned with static information flow control.

Due to the complexity of modern software it is not realistic to find and eliminate information leaks by manual inspection of program code. Static information flow control requires automated techniques, which in turn have to be built on a strong theoretical foundation of what 'security' really means. Whereas static information flow control for sequential programs has been studied for about 35 years and has already given birth to practical tools [58, 103, 132], it has not got that far yet for concurrent programs. Concurrency is generally hard to reason about. Its semantics is difficult to grab with formal methods, which makes it difficult to define a suitable security concept. The existence of many different flavors of concurrency (e.g. threads vs distributed systems, message passing vs shared memory) complicates that matter even more. There also

```
void main():
  x = inputPIN();
  if (x < 1234)
    print(0);
  y = x;
  print(y);
```

```
void thread_1():
  print(1);
```

```
void thread_2():
  x = inputPIN();
  while (x > 0)
    x--;
  print(2);
```

Figure 1.1.: Typical information leaks in sequential (left side) and concurrent programs (right side).

```
void thread_1():          void thread_2():
  skip;                     while (y > 0)
  skip;                       y--;
  print(1);                 print(2);
```

Figure 1.2.: Assume that the depicted program is scheduled by a Round-Robin scheduler switching threads after each statement. If variable `y` is smaller than 1, 1 is always printed behind 2, otherwise, 2 is always printed behind 1.

exist numerous kinds of schedulers, of which at least the most common should be taken into account. Last but not least, concurrency gives rise to several special kinds of information leaks, which have to be dealt with. The program on the right side of Fig. 1.1 provides an example for those subtleties. It consists of two concurrently executing threads. The program always prints 1 and 2 to the screen, but their order depends on the interleaving. Assume that the scheduler schedules after every executed statement and picks a thread with a certain probability. Then PIN's value alters the probabilities of the two possible output orders, because it defines the number of iterations of the `while` loop. The higher the PIN, the higher is the probability that 1 is printed first. An attacker being aware of the concrete probability distribution and being able to observe multiple program runs with the same PIN can deduce information about the PIN. This kind of leak is called a *probabilistic channel*.

At first glance, it seems that probabilistic channels do not pose a real threat, because in general a statistical evaluation of a number of program executions is needed to exploit them. But if the scheduler in charge is deterministic, say, a Round-Robin, then probabilistic channels behave similar to leaks resulting from conditional branching. Consider the example program in Fig. 1.2, where the `skip` statements serve as placeholder for serious code and `y` is a global variable, and assume that the scheduler switches threads after each statement. Then, if `y` is smaller than 1, 1 is always printed behind 2, otherwise, 2 is always printed behind 1. The threads could be put inside a loop that repeatedly forks and joins them and uses them to gradually leak sensitive information via variable `y`.

This thesis presents a technique for static information flow control for concurrent programs with threads and shared-memory communication that guarantees confidentiality of information with respect to a reasonable attacker model. The technique utilizes *program dependence graphs*, a language-independent representation of information flow in a program, which allows to apply it to all programs that can be translated into a program dependence graph. In the context of this thesis, the whole technique has been implemented for Java, an object-oriented language that offers concurrency via threads and shared memory. To the best of the author's knowledge, this is the first existing realization of static information flow control for a mature, contemporary and broadly used programming language with concurrency.

1.1. About This Thesis

This thesis is embedded in the ValSoft/Joana project [6, 45], whose goal is the development of practical information flow control techniques for realistic languages and programs. The project aims to develop analyses that [6]

- guarantee security, i.e. miss no information leak,
- analyze established languages, like C or Java,
- process larger software systems with acceptable effort,
- offer maximal precision, so that very few false alarms are generated.

ValSoft/Joana consists of a general-purpose program analysis framework for Java, called Joana [45], which has been developed by Christian Hammer [52], Jürgen Graf [49] and the author. Today, Joana can handle Java programs with up to 30,000 lines of code and full Java bytecode, including features like pointers, objects, dynamic dispatch, side effects of procedures, recursion, exception handling and threads.

At the time this thesis started, Joana had only rudimentary support for threads and was intended to analyze JavaCard [2] programs. JavaCard is a Java subset for smart cards, which at that time (version 2.x) did not support threads. On top of Joana, Hammer [52] developed an information flow control technique for full sequential Java bytecode. In order to use Joana for analyzing general Java programs, support of Java threads had to be integrated into Joana, because threads are an integral component of the language.

1.2. Information Flow Control

Programs contain various kinds of information flow, which may intentionally or unintentionally unveil confidential data. The most intuitive kinds of information flow are *explicit* and *implicit flows*, the former resulting from assignments and the latter from conditional branching. The program on the left side of Fig. 1.1 leaks the input PIN via implicit and explicit flow. Statement `print(0)` leaks information about the PIN being smaller than 1234, which is an illicit implicit flow. The program also directly prints the PIN copied to variable `y`, which is an illicit explicit flow. Concurrent programs may additionally contain *probabilistic channels*, an example being the leak described in the program on the right side of Fig. 1.1. Information flow control (IFC) aims to detect these and other kinds of information leaks. For that purpose, it needs a concrete definition of when a program is secure. A widespread security policy for sequential programs is *noninterference* [47].

1.2.1. Noninterference

Intuitively, noninterference demands from a program that all possible program runs working on the same public data, but on possibly different confidential data, must expose the same behavior observable for an attacker. As a consequence, the attacker cannot distinguish the program runs and thus cannot draw conclusions about the confidential data. Noninterference is defined for deterministic programs only, because it assumes that one input leads to one observable behavior. It thus cannot be applied to concurrent programs, where interleaving may cause several possible observable behaviors for the same input, but there exist several convenient enhancements. One of them is *observational determinism* [98, 121].

1.2.2. Observational Determinism

A major problem of IFC for concurrent programs is the existence of numerous different scheduling strategies. Many existing approaches to IFC for concurrent programs do only account for one or a couple of scheduling strategies and fail in case another scheduler is applied. Therefore, *scheduler independence* should be a primary objective. Observational determinism offers an elegant solution to that problem. It requires that the interleaving of statements contributing to the observable behavior of a program has to be deterministic. From the point of view of an attacker, an observational deterministic concurrent program behaves like a deterministic program, because one input can only cause one observable behavior. Observational deterministic programs are therefore free of probabilistic channels. Combining observational determinism and noninterference results in a promising, scheduler-independent security policy for concurrent programs.

Several existing IFC techniques based on observational determinism [64, 145, 160] are *flow-insensitive* and do not take the relative execution order of program statements into account. These techniques impose drastic restrictions on programs for being classified as observational deterministic, rendering them impractical for real programs (see sect. 5.7.3 for a comparison). For a practical employment of observational determinism, flow-sensitivity seems to be compulsory. We therefore base our IFC technique on *program dependence graphs* and *slicing*.

1.3. Slicing

A *slice* of a program consists of all statements that may influence a given program point of interest, the so-called *slicing criterion*. Consider the program on the left side of Fig. 1.3 and assume we are interested in the value of variable c in statement 5. Its value is influenced by statements 1, 3, 4 and 5. These statements either influence the value of variable c by defining or modifying contributing variables or they decide whether statement 5 is actually executed. They are a *slice* for variable c in statement 5.

```
1 void main():
2   a = input + 5;
3   b = input - 5;
4   if b > 0
5     c = input * b;
6   else
7     c = input * a;
```

```
1 void main():
2
3   b = input - 5;
4   if b > 0
5     c = input * b;
```

Figure 1.3.: A simple program and its slice for variable c in statement 5.

Slicing is a general-purpose technique used in numerous kinds of program analyses, such as:

- Debugging, where slicing is used to exclude program parts that cannot have influenced erroneous program behavior [69, 140].
- Testing, where slicing aids in determining which components of a modified program can be tested with tests from the old test suite, and for which components new tests have to be written [20].
- Software complexity, where slicing is used for measurement of software complexity metrics, for example [155]: (1) coverage, a measure of the length of slices versus the length of the program; (2) overlap, a measure of the number of statements in a slice which belong to no other slice; (3) parallelism, the number of slices with few statements in common.
- Model checking, where slicing serves as a preprocessing step for the identification of program parts unrelated to a given specification [61]. The model building process excludes these parts and yields significantly smaller models.
- Clone detection, where slicing enables a very accurate detection of code duplicates [71].

The first formal definition of a slice stems from Mark Weiser [155, 156], who observed that programmers debugging a program mentally create slices to exclude program parts that cannot have caused the error. He suggested employing automatic slicing tools to support and alleviate debugging. According to Weiser, a slice is defined as follows:

Definition 1.1. *Let P be a program and (c,V) be a slicing criterion, consisting of a statement c and a set of variables V. A sub-program S of P is a slice of P for (c,V) if*

1. *S is a valid program, and*
2. *whenever P halts for a given input, S also halts for that input, showing the same values for the variables in V whenever c is executed.*

Of course, this definition always allows a trivial slice – the program itself. In order to benefit from slicing, slices should be as small as possible. A *statement-minimal* slice for a slicing criterion (c,V) is a slice that only contains statements that are guaranteed to influence (c,V) in some execution. Unfortunately, the computation of statement-minimal slices is due to conditional branching undecidable [156], which is in practice usually abstracted away by treating it as non-deterministic branching. This leads to the issue of the *precision* of a slice and of different slicing algorithms. In this thesis, the precision of a slice is understood as follows: First and foremost, slices should be *correct*. A slice is correct if it contains all statements that can influence the slicing criterion in some execution, a slice failing to do so is called *incorrect*. A slice S is *more precise* than another slice S' for the same slicing criterion if both slices are correct and S is smaller than S'. If additionally S is a subset of S', then S is *strictly more precise* than S'.

1.3.1. Slicing and Information Flow Control

It is known for some time that slicing allows to enforce noninterference in sequential programs: If the slice for the statements creating observable behavior is free of statements that process confidential data, the program is noninterferent [7, 21, 137]. However, the technical challenges concerning slicing of full-fledged languages and real-world applications have prevented implementations of slicing-based IFC until only recently. The first reported implementations stem from Yokomori et al. [159], for Pascal, and from Hammer et al. [58], for sequential Java bytecode. This thesis extends slicing-based IFC to concurrent programs with threads and shared memory communication. For that purpose, it provides an in-depth investigation of slicing of concurrent programs.

1.3.2. Slicing as a Graph Reachability Problem

Weiser computed slices via an iterative data flow algorithm. A few years later, Ottenstein and Ottenstein [109] suggested computing slices by dint of a reaching analysis in *program dependence graphs* (PDGs). A PDG $P = (N,E)$ for program p is a directed graph, whose nodes in N represent p's statements[1] and whose edges in E represent *data dependences* and *control dependences* between them. Node b is data dependent on node a if it may use a value computed by a, and it is control dependent on a if a's evaluation controls the execution of b. Figure 1.4 shows the PDG of the program in Fig. 1.3. The start node is defined to control the mere execution of the program, which explains the outgoing control dependences. A formal description of PDGs and their construction is given in chapter 2.

Slicing based on PDGs is somewhat limited compared with slicing based on data flow analysis: One has to choose a PDG node c as the slicing criterion and thereby restricts the set V

[1] In the remainder, the terms 'node' and 'statement' will be used interchangeably.

```
1 void main():
2   a = input + 5;
3   b = input − 5;
4   if b > 0
5     c = input ∗ b;
6   else
7     c = input ∗ a;
```

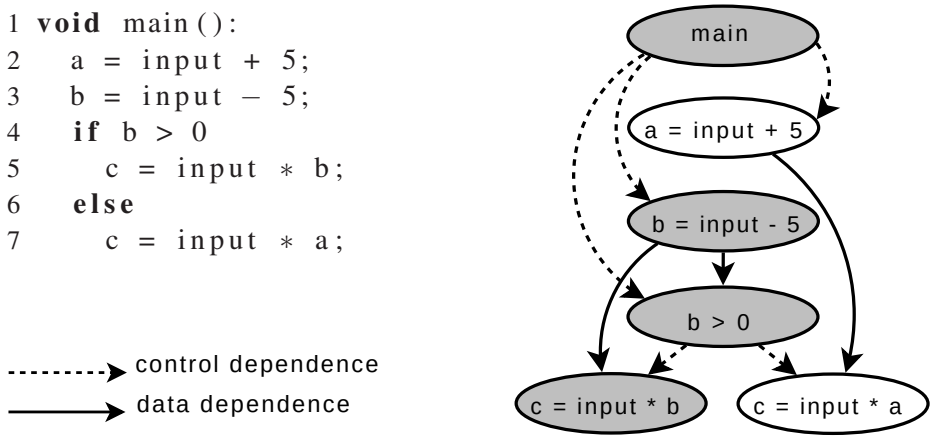


Figure 1.4.: The PDG for the program in Fig. 1.3. The highlighted nodes form a slice for statement 5.

of variables of the slicing criterion (c,V) to the set of variables accessed by node c. A slice of a PDG for a node n consists of all nodes that reach n. As an example, Fig. 1.4 highlights the slice for statement 5. The advantage of PDG-based slicing is that the structure of PDGs is not bound to a specific programming language, wherefore PDG-based slicing and applications based thereon are language-independent. Therefore, PDG-based slicing established next to slicing based on data flow analysis.

We decided to use PDG-based slicing because of its language independence. The algorithms presented in this thesis are much more complex than simple graph reachability and can thereby be reused for all languages and programs for which PDGs can be generated.

1.3.3. Slicing of Concurrent Programs

PDG-based slicing of sequential programs is well-studied, and there exist sophisticated algorithms for creating and slicing PDGs for contemporary sequential languages, including features like pointers, objects, dynamic dispatch, side effects of procedures, recursion, exception handling and many more [52, 152]. But many modern languages, like Java or C♯, have built-in support for concurrent execution. Applications that use slicing to analyze such languages need algorithms suitable for concurrent programs. Unfortunately, the precise and efficient algorithms known for sequential programs cannot be applied. Concurrent programs with shared memory exhibit a new kind of dependence, *interference dependence* [73], which is basically a data dependence between concurrently executing statements. Figure 1.5 extends the program of Fig. 1.4 with a second thread, which assigns `b` to `x`, and modifies statement 2 to add `x` to `input`. The resulting PDG contains two interference dependences.

The computation of interference dependence has to take all possible ways of interleaving into account. Since different ways of interleaving may exclude each other, different interference dependences may exclude each other, too. The highlighted nodes in Fig. 1.5 show the slice

```
1 void thread_1():
2   a = input + x;
3   b = input - 5;
4   if b > 0
5     c = input * b;
6   else
7     c = input * a;

8 void thread_2():
9   x = b;
```

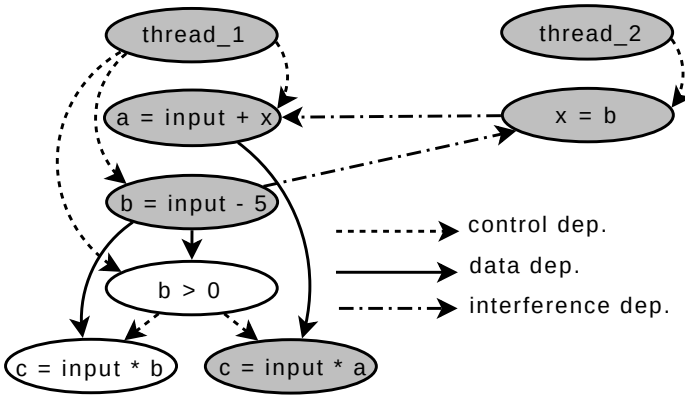


Figure 1.5.: A concurrent program and its PDG. The highlighted nodes form a slice for statement 7.

for statement 7 computed by collecting all reaching nodes. The slice includes statement 3, even though it cannot influence statement 7 – it would have to be executed before statement 2 in order to do so. This kind of imprecision is called *timing-insensitivity*. To make matters worse, it may depend on the slicing criterion whether two interference dependences may exclude each other or not. The detection of mutually exclusive interference dependences is therefore left to the slicing algorithm. Krinke [73] discovered that by consulting the control flow during the slicing process a slicing algorithm is able to avoid timing-insensitive results. If the slicing algorithm computing the slice for statement 7 in Fig. 1.5 remembers that it has left thread 1 at statement 2 when it traversed to statement 9, it is able to detect that reentering thread 1 at node 3 is timing-insensitive, because node 3 cannot execute before node 2.

1.4. Contributions

The following paragraphs summarize the contributions of this thesis.

MHP analysis An analysis of concurrent programs requires to identify those parts of a program that may execute in parallel, which is done by a *may-happen-in-parallel* (MHP) analysis. Prior to this work, Joana assumed that all threads of a program may execute entirely in parallel, which led to a huge number of spurious interference dependences. We integrated a MHP analysis into the framework, a combination of Nanda's [106] and Barik's [18] MHP analyses, which complement one another very well. The MHP information is used to prune spurious interference dependences. On average, we were able to remove 75% of all interference dependences determined without the MHP analysis.

Timing-sensitive slicing The problem of timing-insensitive paths in PDGs of concurrent programs was first identified by Krinke [73], who also presented the first timing-sensitive slicing algorithm [73, 75, 77]. However, he did not implement his algorithm and could therefore not

investigate its practicability. Nanda [104, 105, 106] improved on his technique by developing several crucial optimizations and reported the first implementation of a timing-sensitive slicer. When we started our thesis, Krinke's algorithm had not been implemented at all, and only one implementation of Nanda's algorithm had been reported [106], giving rise to several questions:

- Which algorithm is more precise?
- Which of them is more time efficient?
- Are the algorithms correct?
- Are the algorithms practical on practical programs?

As shown by Nanda, the algorithms have a worst case runtime complexity exponential in the number of the threads of the analyzed program. However, as this is only the worst case, the algorithms might behave well in most cases; this had not been investigated either. It was one goal of this thesis to thoroughly examine and compare both algorithms.

Our investigation revealed that Nanda's slicer is an important step forward, but still has several shortcomings:

- It applies an optimization too greedily and may therefore omit valid edges and yield incorrect slices. Unfortunately, the optimization is crucial for making timing-sensitive slicing practical, so it cannot be removed. We describe how this problem can be solved.
- It cannot handle thread creation inside recursive procedures, only inside loops (Krinke's technique cannot handle either of both). We developed a mechanism that handles thread creation inside loops and recursive procedures and can be integrated into both algorithms.
- It contains several bottlenecks, which we relieve by several new optimizations, leading to a significant speedup.
- It exploits MHP information to yield more precise slices than Krinke's algorithm. We developed a new way to integrate MHP information into timing-sensitive slicers, which improves precision even more.

We have integrated all these algorithms and improvements into Joana and evaluated them on a large set of concurrent Java programs. The evaluation indicates that timing-sensitive slicing currently scales up to programs with up to 5,000 - 10,000 lines of code and significantly reduces the average size of the slices. For our benchmark, the timing-sensitive slices were on average 22% smaller than the timing-insensitive ones.

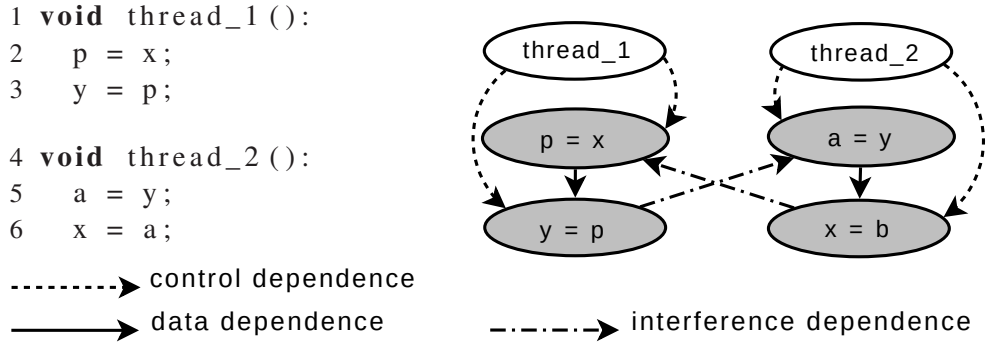


Figure 1.6.: The highlighted chop $chop(6,5)$, computed by collecting all nodes on paths between nodes 5 and 6, is timing-insensitive.

Timing-sensitive chopping Another analysis technique closely related to slicing is *chopping*. Chopping answers the question of which statements are involved in conveying effects from a statement s to another statement t. Chops are usually computed on PDGs, where a chop $chop(s,t)$ from s to t consists of all nodes on paths from s to t.

Chopping also found its way into information flow control: Snelting [137, 138] proposed to use *path conditions* as witnesses for illicit information flow. A path condition between two statements, s and t, is a necessary condition on the program state that a program run has to satisfy in order to reach t, when coming from s. The path condition is composed of all predicates influenced by s and influencing t, which in turn are determined by the chop from s to t. A path condition for an information leak enables to reproduce situations in which that leak occurs and thus possesses evidentiary value.

Prior to our work, no chopping algorithm for concurrent programs had been reported at all. In order to make path conditions for concurrent programs possible, this thesis developed the first chopping algorithms for concurrent programs. Similar to slicing, the precision of chops of concurrent programs may suffer from timing-insensitivity. Consider the program shown in Fig. 1.6, consisting of two concurrent threads which communicate via two shared variables, `x` and `y`. Clearly, $chop(6,5)$ should be empty, because statement 6 is executed after statement 5 and therefore cannot influence it. But if the chop is computed by collecting all nodes on paths between statements 6 and 5, the result is $chop(6,5) = \{2,3,5,6\}$. We transferred the idea behind timing-sensitive slicing to chopping and developed a timing-sensitive chopping algorithm.

Since timing-sensitive chopping is expensive and difficult to implement, we present six different chopping algorithms, which offer different degrees of precision, runtime performance and implementation effort. An extensive evaluation on a set of concurrent Java programs shows the benefits and drawbacks of these algorithms.

IFC for concurrent programs We present a flow-sensitive security property that prevents illicit explicit and implicit flow and probabilistic channels in programs with threads and shared-memory communication. The security property is based on observational determinism and guarantees that a program is free of the these kinds of leaks if, putting it simply,

1. the program statements causing behavior observable for an attacker have a fixed interleaving order and

2. the implicit and explicit flow in the program does not leak sensitive data.

We show that these conditions can be verified by analyzing PDGs. The resulting algorithm has been integrated into Joana and can handle full Java bytecode. A comparison with other security properties and an evaluation on a set of concurrent Java programs provides insights into its practicability and power. To the best of our knowledge, this is the first implementation of an IFC technique for concurrent programs written in a contemporary language.

The novelty of our IFC technique, besides of being implemented for Java, is that it is scheduler-independent and at the same time comparatively permissive. The currently most popular security properties, *strong security* [128], *weak probabilistic noninterference* [135] or variants thereof, are not based on observational determinism, hold only for a class of schedulers and may break for others[2]. Other existing security properties based on observational determinism are confined to message-passing languages and cannot detect probabilistic channels manifesting in the interleaving orders of program output [160], forbid observable behavior behind loops whose guards contain sensitive data [64] or impose restrictions on the number of statements that are allowed to be executed between two observable statements [146]. These restrictions result from the flow-insensitivity of the security properties. We are able to circumvent them by using PDGs and slicing.

1.5. Organization

This thesis is organized as follows: Chapter 2 introduces PDGs and slicing for sequential programs. Chapter 3 presents PDGs for concurrent programs, our MHP analysis and timing-sensitive slicing. Chapter 4 is concerned with chopping of sequential and concurrent programs. Chapter 5 presents our IFC technique for concurrent programs. Chapter 6 provides a brief overview of Joana and chapter 7 concludes.

[2] They have other advantages, such as permitting a modular verification of libraries, which is not possible with ours. A comparison can be found in section 5.7.

2. Slicing Sequential Programs

This chapter serves as an entering guide to slicing and PDGs. It introduces PDGs and slicing algorithms for interprocedural, sequential programs. The chapter is organized as follows: The dependences in a PDG are based on properties of the control flow of the program, which is usually represented by a *control flow graph*. The first section introduces control flow graphs for intra- and interprocedural programs. The second section presents PDGs and slicing for procedure-less programs. Section 2.3 defines PDGs and slicing algorithms for interprocedural programs. Section 2.4 gives a brief overview of the computation of PDGs. Section 2.5 describes a slicing technique that keeps track of the calling context of procedures during the slicing process. Section 2.6 evaluates the presented slicing algorithms on a set of Java programs, partly recapitulating well-known results and partly investigating new optimizations. Finally, section 2.7 presents related work.

2.1. The Control Flow Graph

Control flow graphs are broadly used to represent the control flow of a program. Each statement or predicate of the program is represented by a node in the graph, and two nodes are connected by a control flow edge if they can be executed back-to-back.The graph contains specific start and exit nodes, which mark the start and termination of the program and do not represent any statement or predicate.

Definition 2.1 (Control flow graph (CFG))**.** *A control flow graph $G = (N, E, s, e)$ of a procedure or a procedure-less program p is a directed graph, where*

- *N is the set of nodes and each statement or predicate in p is represented by a node $n \in N$,*
- *E is the set of edges representing the control flow between the nodes,*
- *s is the start node, which has no incoming edges and reaches all nodes in N,*
- *e is the exit node, which has no outgoing edges and is reachable from every node in N.*

Often, definitions of CFGs label the edges with `true`, `false` or the empty word in order to model the branches of conditional constructs. Even more detailed annotations up to the point of transition functions modeling the semantics of the program are possible. However, such annotations of control flow edges are negligible in this thesis and are ignored.

```
1 void main():
2   p = foo(2);
3   q = foo(p);
4   y = q * 3;

5 int foo(f):
6   return f + 1;
```

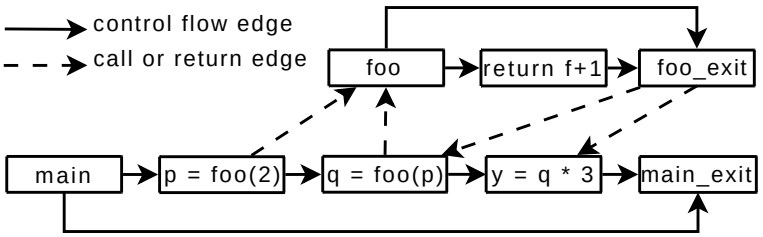


Figure 2.1.: An interprocedural control flow graph.

Interprocedural programs are represented by *interprocedural control flow graphs*, in which the CFGs of the single procedures are connected via *call* and *return* edges.

Definition 2.2 (Interprocedural control flow graph (ICFG))**.** *An interprocedural control flow graph* $G = (CFG_q, main, Call)$ *of a program q consists of the set* CFG_q *of the CFGs of the procedures of q, a distinguished procedure* main *and a set* Call *of* call sites *with the following properties:*

- *For each pair* p_i, p_j *of procedures of q, the node sets* N_{p_i}, N_{p_j} *and the edge sets* E_{p_i}, E_{p_j} *are disjoint.*

- *Let* c_p *be a node in procedure p that represents a call of procedure* p'*. Then there exists a* call site $(c_p \rightarrow_{call} s_{p'}, e_{p'} \rightarrow_{ret} c'_p) \in Call$*, where* $c_p \rightarrow_{call} s_{p'}$ *is a* call edge *from* c_p *to the start node* $s_{p'}$ *of procedure* p' *and* $e_{p'} \rightarrow_{ret} c'_p$ *is a* return edge *from the exit node* $e_{p'}$ *of* p' *to the direct successor* c'_p *of* c_p *in procedure p. We name* c_p *a* call node *and* c'_p *a* return node*. Call site* $(c_p \rightarrow_{call} s_{p'}, e_{p'} \rightarrow_{ret} c'_p)$ *can be represented alternatively by the tuple* $(c_p, s_{p'})$*.*

- *Every node in G is reachable from the start node of* main *and reaches the exit node of* main*.*

In order to simplify the computation of program dependence graphs, each CFG contains a synthetic control flow edge from the start node to the exit node, and at each call site there exists a synthetic control flow edge from the call node to the return node. Figure 2.1 shows the ICFG of a program with two procedures.

2.1.1. Control Flow Paths in ICFGs

A path Φ in an ICFG is a possibly infinite sequence of nodes, $\langle n_1, n_2, \ldots \rangle$, such that for every consecutive pair (n_j, n_{j+1}) in the path the ICFG contains an edge from n_j to n_{j+1}.

Realizable paths A path is *realizable* if there exists an execution of the program that executes the statements in the path in the same order. If a path is not realizable, it can be ignored, which may improve the analysis results. Since evaluation of predicates is in general not statically decidable, every path in a CFG is deemed realizable.[1] In ICFGs we are interested in paths that distinguish different calling contexts of the same procedure, so-called *context-sensitive* paths: A path that enters a procedure and returns from it later should return to the same call site. Since ICFGs represent a procedure by a single CFG and connect every call of that procedure with that CFG, ICFGs may contain context-insensitive paths. For example, all paths in Figure 2.1 entering procedure `foo` at call node `p = foo(2)` and leaving it later on towards node `y = q * 3` are of that ilk. Following Reps and Rosay [120], context-sensitive paths in ICFGs can be described by a language of matching parentheses:

Definition 2.3 (Context-sensitive paths in ICFGs)**.** *For each call site* $(c \rightarrow_{call} s, e \rightarrow_{ret} c')$*, label the call edge with a symbol* $(_c^s$ *and the return edge with a symbol* $)_c^s$*. Label all other edges with* l*. A path from node* m *to node* n *in the ICFG is context-sensitive, abbreviated with* $m \rightarrow_{cs}^* n$*, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal* `realizable` *by the following context-free grammar* H*:*

$$\texttt{matched} \rightarrow \texttt{matched matched} \mid (_c^s\ \texttt{matched}\ )_c^s \mid l \mid \varepsilon$$
$$\texttt{unbalanced-right} \rightarrow \texttt{unbalanced-right}\ )_c^s\ \texttt{matched} \mid \texttt{matched}$$
$$\texttt{unbalanced-left} \rightarrow \texttt{unbalanced-left}\ (_c^s\ \texttt{matched} \mid \texttt{matched}$$
$$\texttt{realizable} \rightarrow \texttt{unbalanced-right unbalanced-left}$$

Nonterminal `matched` describes 'matched' paths: paths that start and end in the same procedure and contain only accomplished procedure calls. 'Unbalanced-right' paths are sequences of matched paths interrupted by unmatched procedure returns. They start in a procedure p and end in a procedure in which p was called[2]. 'Unbalanced-left' paths are sequences of matched paths interrupted by unmatched procedure calls. They start in a procedure p and end in a procedure that is called by p. A 'realizable' path is a concatenation of an unbalanced-right and an unbalanced-left path. It starts in a procedure p, leaves it towards a procedure q in which p was called and ends in a procedure r called by q.

2.1.2. Dominance and Postdominance

Several essential relations between the statements of a program can be derived from its ICFG. Let m and n be two nodes in an ICFG:

- *m dominates n* if every realizable path from the start node to n passes through m.

[1] In certain cases it is possible to statically exclude certain branches or sequences of branches, but we do not investigate that issue.

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

- *n postdominates m* if every realizable path from m to the exit node passes through n.
- *n strictly postdominates m* if *n postdominates m* and $n \neq m$.
- n is the *immediate postdominator* of m if $n \neq m$ and n is the first postdominator on every realizable path from m to the exit node.
- *n strongly postdominates m* if n postdominates m and there is an integer $k \geq 1$ such that every realizable path from m of length $\geq k$ passes through n [110]. In effect, strong postdominance dismisses realizable paths from m to n containing loops or recursion. It is sensitive to the possibility of nontermination along paths from m to n.

2.2. Slicing Intra-Procedural Programs

Program dependence graphs were originally developed by Ferrante et al. [39] as an intra-procedural data structure for the analysis of possible optimizations and parallelism in procedures. A PDG $G = (N, E)$ of a procedure-less program p is a directed graph, whose nodes in N represent statements of p and whose edges in E represent *data dependences* and *control dependences* between them. Data dependence, originally called *flow dependence* by Ferrante et al. [39], is defined as follows:

Definition 2.4 (Data dependence)**.** *Let m, n be two nodes in a CFG. Let $def(n)$ be the variables defined by n and let $use(n)$ be the variables used by n. Node n is data dependent on node m, abbreviated by $m \rightarrow_{dd} n$, iff there exists a variable $v \in def(m) \cap use(n)$ and a path $\Phi = \langle n_1, \ldots, n_k \rangle$ in the CFG, for which $n_1 = m, n_k = n$ and $\forall 1 < i < k : v \notin def(n_i)$ holds.*

The definition of control dependence is based on postdominance [39]:

Definition 2.5 (Control dependence)**.** *Let m, n be two nodes in a CFG. Node n is control dependent on m, abbreviated by $m \rightarrow_{cd} n$, iff*

1. *there exists a path ϕ from m to n in the CFG such that n postdominates every node in ϕ, except for n and m,*
2. *and n does not postdominate m.*

The PDG of a program p is built by computing the program dependences on the CFG of p. Every node being part of a dependence is added to the PDG, and for every dependence one corresponding edge is inserted. Due to the synthetic control flow edge between the start and exit node of the CFG, every node in the CFG is at least involved in one control dependence, except for the exit node. The exit node is usually omitted from the PDG because it is independent of the other nodes. The slice of a PDG consists of all nodes that reach the slicing criterion:

Definition 2.6 (Slice of a PDG)**.** *Let $G = (N, E)$ be a PDG. A slice of G for a slicing criterion $s \in N$ consists of the set of nodes $\{n \mid \exists\, n \to^* s \text{ in } G\}$, where $n \to^* s$ denotes a transitive path from n to s in G.*

Additional dependences A program analysis may require additional kinds of dependences for its purpose. One important example is *weak control dependence* [110, 115], which is used by program analyses that have to account for program termination. Intuitively, a node n is weakly control dependent on nodes that may cause the program to bypass n or to infinitely delay its execution.

Definition 2.7 (Weak control dependence)**.** *Let m, n be two nodes in a CFG. Node n is weakly control dependent on m iff there exist two successors m', m'' of m such that n strongly postdominates m', but not m''.*

2.3. Slicing Interprocedural Programs

Ottenstein und Ottenstein [109] showed how PDGs can be used for slicing, but did not investigate how to extend PDGs to programs with multiple procedures. A suitable extension is the *system dependence graph* (SDG) of Horwitz, Reps and Binkley [63, 119]. The SDG $G = (N, E)$ of a program p is a directed graph, where N is the set of nodes and the edges in E represent the dependences between them. The SDG is composed of the PDGs of the single procedures of p. The PDGs are connected at *call sites*[3], which consist of a call node c and the start node s of the called procedure, connected through a *call edge* $c \to_{call} s$, and of synthetic *parameter nodes* and *-edges*, which model parameter passing and result returning:

- For every passed parameter, there exists an *actual-in node* a_i at the call node and a *formal-in node* f_i at the start node, connected via a *parameter-in edge* $a_i \to_{pi} f_i$.
- For every returned value, there exists an *actual-out node* a_o at the call node and a *formal-out node* f_o at the start node, connected via a *parameter-out edge* $f_o \to_{po} a_o$.
- Formal-in and formal-out nodes are control dependent on the start node, actual-in and actual-out nodes are control dependent on the call node.
- So-called *summary edges* between actual-in and actual-out nodes of one call site represent transitive flow from a parameter to a return value in the called procedure.

Side effects of procedures, such as access to global variables and heap locations or exception handling, are treated as additional parameters and return values [63]. That way, all interprocedural effects are modeled by parameter nodes and edges at the call sites. This is a well-formedness

[3] The call sites in an ICFG can be bijectively mapped to the call sites in the corresponding SDG. We therefore use the term *call site* in both cases.

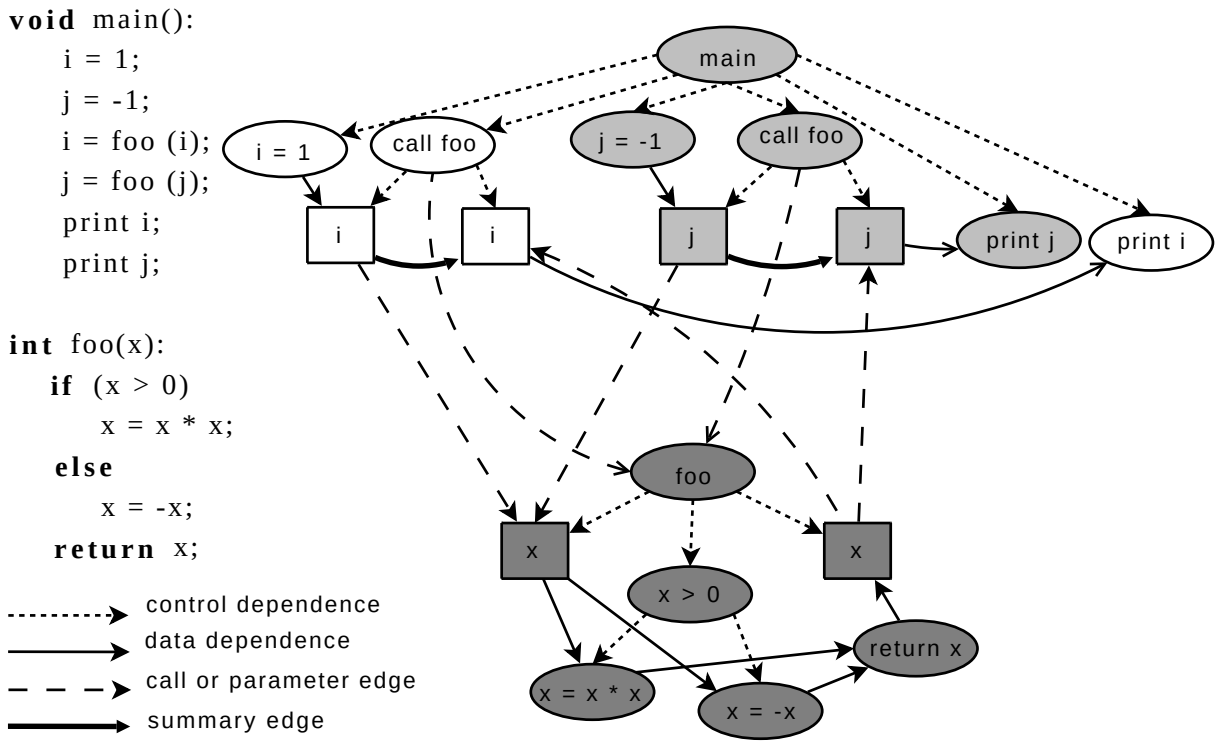


Figure 2.2.: A system dependence graph. Parameter nodes are symbolized by rectangular nodes, where the parameter nodes to the left of a call or start node are the actual-in or formal-in nodes, the ones to the right are the actual-out or formal-out nodes. The highlighted nodes form the context-sensitive slice for `print j`, as computed by the two-phase slicer. The light gray nodes are visited in phase 1, the dark gray nodes in phase 2.

property of SDGs, which is exploited by the upcoming slicing algorithm. Figure 2.2 shows an example program and its SDG.

Relationship between SDGs and ICFGs

The set of nodes of a SDG $G = (N, E)$ can be partitioned into two disjoint sets, $N = N_{PDG} \cup N_{syn}$, where N_{PDG} consists of the nodes of the PDGs and N_{syn} contains the synthetic parameter nodes. Similarly, the set of edges can be partitioned into two disjoint sets, $E = E_{PDG} \cup E_{syn}$, where E_{PDG} consists of the edges of the PDGs and E_{syn} consists of the call-, parameter- and summary edges. It should be clear that the sets of nodes of the SDG and of the corresponding ICFG are in no subset relation. A node in the ICFG is missing in the SDG if it is not involved in any program dependence, and the synthetic SDG nodes are missing in the ICFG. However, on several occasions in this thesis, SDG nodes have to be mapped to ICFG nodes. In these cases, we map actual-in and -out nodes to the associated call or return node and formal-in and -out nodes to the start or exit node of the procedure:

Definition 2.8. *Let $G = (N_{PDG} \cup N_{syn}, \_)$ be a SDG, and let $ICFG = (N_{ICFG}, \_)$ be the corresponding ICFG. Function $map : N_{PDG} \cup N_{syn} \mapsto N_{ICFG}$ is defined as follows:*

$$map(n) = \begin{cases} n & n \in N_{PDG} \\ c & n \text{ is an actual-in node and } c \text{ is the corresponding call node} \\ r & n \text{ is an actual-out node and } r \text{ is the corresponding return node} \\ s & n \text{ is a formal-in node of procedure } p \text{ and } s \text{ is } p\text{'s start node} \\ e & n \text{ is a formal-out node of procedure } p \text{ and } e \text{ is } p\text{'s exit node} \end{cases}$$

We assume that this mapping is done implicitly whenever necessary.

2.3.1. Context-Sensitive Slicing

The major difficulty of slicing interprocedural programs is to maintain context-sensitivity. Similar to ICFGs, SDGs may contain context-insensitive paths. Context-sensitive paths in SDGs can be described by the grammar for context-sensitive ICFG paths in definition 2.3 [120]:

Definition 2.9 (Context-sensitive paths in SDGs)**.** *Let G be a SDG. For each call site (c,s) in G, label the associated call and parameter-in edges with a symbol $(_c^s$, and the associated parameter-out edges with a symbol $)_c^s$. Label all other edges with l.*

A path in G is context-sensitive, symbolized with $\rightarrow_{cs}^*$, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal `realizable` *by the context-free grammar H in definition 2.3 (it will always be clear whether $\rightarrow_{cs}^*$ denotes a context-sensitive path in a SDG or ICFG).*

A context-sensitive slice of a SDG consists of all nodes on context-sensitive paths to the slicing criterion.

Definition 2.10 (Context-sensitive slice)**.** *Let $G = (N,E)$ be a SDG. A context-sensitive slice of G for a slicing criterion $s \in N$ consists of the set of nodes $\{n \mid \exists\, n \rightarrow_{cs}^* s \text{ in } G\}$.*

Two-phase slicing

Computing a slice of a SDG by collecting all nodes reaching the slicing criterion does not yield context-sensitive slices. Fortunately, SDGs are purpose-built for context-sensitive slicing. Their summary edges enable an efficient computation of context-sensitive slices in two phases [63]: Phase 1 slices from the slicing criterion only ascending to calling procedures, where summary edges are used to bypass call sites. If the slicer encounters a parameter-out edge, the adjacent node is stored in a list L. Phase 2 slices from all nodes in L only descending to called procedures. This *two-phase* approach is the most established slicing technique for interprocedural programs.

Algorithm 2.1 The two-phase slicer of Horwitz, Reps and Binkley [63].

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

```
W1 = {s}, W2 = {}, S = {s}  // two worklists and the result set

/* phase 1 */
repeat
  W1 = W1 \ {n}  // process the next node in W1
  for all m →e n  // handle all incoming edges of n
    if m ∉ S  // m has not been visited yet
      S = S ∪ {m}
      // if e is not a param-out edge, add m to W1, otherwise, add m to W2
      if e ∉ {po}
        W1 = W1 ∪ {m}
      else
        W2 = W2 ∪ {m}
until W1 = ∅

/* phase 2 */
repeat
  W2 = W2 \ {n}  // process the next node in W2
  for all m →e n  // handle all incoming edges of n
    if m ∉ S  // m has not been visited yet
      // if e is not a param-in or call edge, add m to W2 and to the slice
      if e ∉ {pi, call}
        W2 = W2 ∪ {m}
        S = S ∪ {m}
until W2 = ∅
return S
```

In Fig. 2.2, the context-sensitive slice for statement `print j` is highlighted gray. The light gray shaded nodes in Fig. 2.2 are visited in phase 1, the dark gray shaded nodes are visited in phase 2. Algorithm 2.1 shows pseudocode of this *two-phase slicer*, which has an asymptotic running time of $\mathcal{O}(|E|)$.

2.4. Computation of System Dependence Graphs

This section briefly describes the computation of SDGs. Since this thesis is not engaged in computation of SDGs in itself, the section is confined to an explanation of the most basic techniques and only sketches the treatment of advanced language features like pointers and objects. For detailed descriptions of SDG construction for contemporary, object-oriented languages we refer to Walkinshaw's PhD thesis [151], Hammer's PhD thesis [52] and Graf's recent publication [49].

2.4.1. Data Flow Analysis Frameworks

The computation of SDGs is based on *data flow analysis*. Data flow analyses usually work on the control flow graph of the program and collect information about variables at program statements. A typical example is the computation of *reaching definitions*, a well-known task in compiler construction. Conveniently, reaching definitions are also used to compute data dependences.

Definition 2.11 (Reaching definitions)**.** *Let m be a node in a CFG G, and let $def(m)$ be the set of variables defined at m. A definition of variable v at node m reaches a (not necessarily different) node n if there exists a realizable path $\langle n_1,\ldots,n_k\rangle$ in G, for which $k > 1$, $n_1 = m$, $n_k = n$ and $\forall 1 < i < k : v \notin def(n_i)$ holds.*

The reaching definitions in a program can be determined by an *iterative data flow analysis*, which computes the reaching definitions for each node by iterating over the control flow graph until reaching a fixed point. This technique uses the below presented functions *gen* and *kill*, where *gen* describes which variables are defined at a node n and *kill* describes which reaching definitions are 'killed', i.e. redefined at n. Let $def(n)$ be the set of definitions at node n and $def(v)$ be the set of all definitions of variable v. Then *gen* and *kill* are defined as follows:

$$gen(n) = def(n)$$
$$kill(n) = \bigcup_{v \in def(n)} def(v)$$

A node n has the following semantics with respect to the set R of reaching definitions: All definitions reaching n and not killed by n leave n together with the definitions generated by n:

$$[\![n]\!](R) = (R \setminus kill(n)) \cup gen(n)$$

This semantics can be extended to capture the effects of a CFG path $\Phi = \langle n_1,\ldots,n_k\rangle$:

$$[\![\Phi]\!](R) = \begin{cases} [\![]\!]_{id} & \Phi = \langle\rangle \\ [\![\langle n_2,\ldots,n_k\rangle]\!] \circ [\![n_1]\!] & otherwise \end{cases}$$

The set of all definitions that may reach a node n is determined by collecting all definitions reaching n via some path from the CFG start node s to n. Since s does not define any variable, we have an empty initial set of reaching definitions in the following equation:

$$Reaching_{MOP}(n) = \bigcup_{\Phi = \langle s,\ldots,n\rangle} [\![\Phi]\!](\emptyset)$$

This formula is a *meet-over-all-paths (MOP)* solution. There exist infinite paths in the presence of loops, which make the computation of the MOP solution impossible. Fortunately, there exists an alternative formula, computing a *minimal-fixed-point (MFP)* solution, which provably coincides with the MOP solution:

$$Reaching_{MFP}(n) = \begin{cases} \emptyset & n = s \\ [\![n]\!](\bigcup_{m \to n} Reaching_{MFP}(m)) & otherwise \end{cases}$$

The computation of the MFP solution is based on the theory of *monotone data flow analysis frameworks*. A monotone data flow analysis framework consists of a lattice representing the data flow facts and a monotone function space defined on the lattice. In our example, the data flow facts are the sets of reaching definitions in program p, the lattice being the power set, and the function space consists of the transfer functions $[\![n]\!]$ of all nodes n in the CFG of p. The monotonicity of the function space guarantees the existence of a fixed point and the termination of its computation.

A monotone data flow analysis framework is a *distributive data flow analysis framework* if the transfer functions are distributive. It is proven that the MFP solution computed by a distributive monotone data flow analysis framework coincides with the correct and precise MOP solution. It is also proven that the equations of the reaching definitions fit in a distributive data flow analysis framework.

2.4.2. Computation of PDGs and SDGs

The computation of a PDG for a procedure-less program or single procedure p works as follows: Initially, the existing control and data dependences are computed on the CFG of p.

The control dependences are traditionally determined via the *post-dominator tree* of the program, which can be extracted from the CFG via the Lengauer-Tarjan algorithm [86] in time $\mathcal{O}(|N| * \alpha(|N|))$ (where α is the inverse Ackermann function). Then, for every control flow edge $m \to n$ where n does not postdominate m, one traverses the post-dominator tree upwards from n to the parent of m. Every visited node, including n, is control dependent on m. A detailed description of this method can be found in Ferrante et al.'s publication [39].

The definition of data dependence can be rephrased in terms of reaching definitions: If the definition of a variable v at node m reaches node n and n references v, then n is data dependent on m. Computing the reaching definitions first and then checking every node for a usage of a variable of a reaching definition retrieves the existing data dependences.

Having the data and control dependences, the PDG is populated with all nodes of the CFG that are involved in at least one dependence and with one edge for each dependence.

From PDGs to SDGs

The computation of a SDG first creates the PDGs of the single procedures and connects them to a SDG. Since procedures are single-entry single-exit regions in ICFGs, control dependences in interprocedural programs can be computed intra-procedurally. The synthetic control flow edge between the start and exit node of a CFG makes sure that no node in a CFG is control dependent on a node outside. All nodes of the CFG lie in a branch of the start node and thus can only postdominate nodes inside the CFG.

Like control dependences, data dependences remain intra-procedural. However, interprocedural effects have to be accounted for. Interprocedural effects arise from accesses to non-local variables, e.g. global variables or heap locations. A *side effect analysis* [17] determines for each procedure p the sets $GMOD(p)$ and $GREF(p)$ of all non-local variables modified ($GMOD(p)$) or referenced ($GREF(p)$) in p or in a procedure called by p. This is done by a fixed-point analysis, which starts by collecting the modified and referenced variables for each procedure in isolation. These results are propagated to calling procedures, and this need for propagation leads to a fixed-point computation in the presence of recursion.

By using the results of the side effect analysis, the data dependences can be determined as in the intra-procedural case on the basis of intra-procedural reaching definitions. For that purpose, the interprocedural effects of procedures are mapped to their call nodes in the ICFG, which serve as proxies for the called procedures during the data dependence computation. Any variable v that may be defined or referenced by a called procedure is assumed to be defined or referenced at the call node itself.

Having the results of the data dependence computation, the PDGs are connected at their call sites. Data dependences involving call, start or exit nodes of the ICFG are deflected to the fitting parameter nodes in the PDGs. Finally, summary edges are added to the evolving SDG. Reps et al. [119] present a suitable algorithm, which commits an intra-procedural backward traversal starting at each formal-out node. If such a traversal reaches a formal-in node, the associated actual-in and -out nodes are connected by a summary edge. Since the insertion of a new summary edge may lead to new intra-procedural paths, the computation is repeated until no new summary edge has been added.

2.4.3. SDGs for Object-Oriented Languages

Data flow analyses for object-oriented languages, and for heap manipulating programming languages in general, need knowledge about variables that point to objects residing on the heap. Heap manipulating languages usually allow different pointers to point to the same heap location, which may cause data dependences that cannot be discovered by the hitherto presented data flow analysis. Applied on the program in Figure 2.3 it would not find that statement 9

```
1 class O:
2   int x;

4 void main():
5   O p = new O();
6   O q = new O();
7   q = p;
8   q.x = 2;
9   print p.x;
```

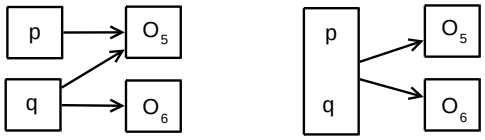


Figure 2.3.: A program fragment and its points-to graphs computed by Andersens's (mid) and Steensgaard's algorithm (right side).

depends on statement 8 because it lacks knowledge about $p.x$ in line 9 being identical with $q.x$ in line 8. A *points-to analysis* reveals which pointers may point to which heap locations.

Points-to analysis

A suitable representation of points-to information is the *points-to graph*, a directed graph whose nodes are the pointers and heap location representatives in the analyzed program[4]. Its edges describe which pointers may point to which other pointers or heap locations.

There exist roughly two classes of points-to algorithms, inclusion-based and unification-based algorithms.

Andersen's algorithm [13] is the origin of inclusion-based points-to analysis. It starts with constructing an initial points-to graph from the assignments in the program and then iterates the following rule until reaching a fixed-point: If the program contains an assignment `p = q`, then `p` has to point to at least everything to which `q` points.

Steensgaard's algorithm [141] in contrast computes equivalence classes of pointers, based on the following rule: If the program contains an assignment `p = q`, then both p and q have to point to the same. To this end, the nodes of p and q in the points-to graph are merged. His algorithm is the origin of unification-based points-to analysis.

Figure 2.3 contains the points-to graphs of the example program resulting from Andersen's algorithm (mid) and from Steensgaard's algorithm (right side). It can be seen that Steensgard's algorithm is less precise than Andersen's. In the example, it gradually merges the pointers to one single node, so that eventually every pointer points to every object. On the other hand, his algorithm offers a near-linear runtime complexity of $\mathcal{O}(n * \alpha(n))$, where n is the number of statements in the program and α is the inverse Ackermann function, whereas the runtime complexity of Andersen's algorithm is in $\mathcal{O}(n^3)$. Therefore, both styles are in use.

[4] In order to avoid name conflicts, renaming techniques like static single assignment [36] are incumbent. In our example, we simply annotate affected pointers and heap locations with the line number of their declaration.

These general algorithms are usually extended to suit individual languages or purposes. For example, Streckenbach and Snelting [142] observed that exploiting the following properties of Java significantly improves the results:

- Java is type correct. This information can be used to avoid spurious edges in the points-to graph.
- Java has no function pointers, only dynamic dispatch. The possible targets of procedure calls can be narrowed by type information.
- Java disallows pointer arithmetics and pointers pointing to other pointers. This means that the paths in points-to graphs of Java programs have a maximal length of 1, which allows to use optimized data structures for the graphs.

Alias analysis An important application of points-to information is the identification of *aliasing* pointers [62]. Two pointers are aliasing if they point to the same heap location. Since aliasing is statically undecidable [113], there are two kinds of approximations. Two pointers are *may-aliasing* if they may point to the same heap location in some program execution. They are *must-aliasing* if they are guaranteed to point to the same location in every program execution.

May-aliasing can be directly derived from the points-to information. Two pointers are said to be may-aliasing if the intersection of their points-to sets is not empty. A sound computation of data dependence in the presence of pointers has to take may-aliasing pointers into account.

It is difficult to determine must-aliasing using points-to information because points-to analyses are usually may-analyses themselves: They collect all heap locations and pointers to which some pointer may point during a program execution. Still, it is possible to define must-aliasing on the basis of points-to information. Two pointers are said to be must-aliasing if their points-to sets are identical and contain only one element.

Dynamic dispatch Points-to information can also be used to improve the static analysis of dynamic dispatch. A simple analysis of dynamic dispatch is to determine the set of possible procedures via static lookup. By additionally consulting points-to information it is possible to exclude those procedures for which a suitable object is missing in the points-to set.

Objects as parameters SDGs for object-oriented languages have to model objects that are passed as parameters at procedure calls. Several authors [57, 88] suggest *object trees* as a suitable representation. The root in such a tree is the object itself, each node represents a field of the parent node and the leaves are the primitive types of which the whole object is eventually composed. The edges in the tree are usually control dependence edges. The paths in the tree mirror the access paths necessary to access a certain field and are traversed by the

```
1 class A:
2   int x;

3 class B:
4   A a;

5 void foo(B b):
6   b.a.x = 1;
```

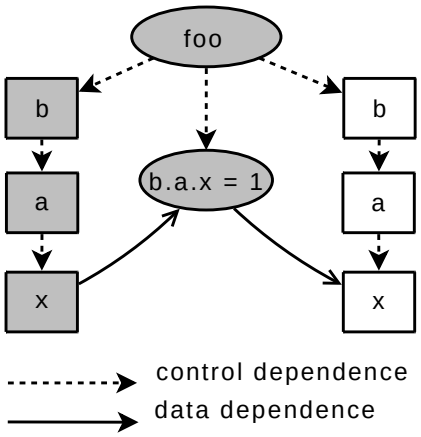


Figure 2.4.: An example for object trees. Procedure `foo` changes field `a.x` of parameter `b`. The slice for `b.a.x = 1` is highlighted gray.

slicing algorithms, so that dependences arising from access paths are correctly accounted for. Figure 2.4 provides a small example, where procedure `foo` receives an object of type `B` and assigns 1 to `b.a.x`. A slice for that assignment (highlighted gray) includes not only `x` but also `b` and `a`, since it has to access them in order to access `x`. Note that object trees are also used as formal-out nodes (the object tree on the right of node `foo`).

A major problem with object trees is the treatment of recursively defined objects, which may result in infinite trees. Hammer [52, 57] developed an *unfolding criterion* for a sound abortion of the unfolding process based on points-to information. In his own words [52, p. 54]:

> "The idea is to stop unfolding, when the same field has been observed earlier in this branch of the object tree and the base pointers of these two locations have the same points-to sets. In this case, both locations are equivalent with respect to all properties used for computing data dependences, so including the second tree would just add the same information again, rendering one of these locations redundant."

Whereas Hammer's unfolding criterion solves that problem, Graf [49] observed another one. If two nodes in an object tree represent different pointers, but with the same may-alias situation, their subtrees represent the same object. But if the nodes lie in different branches of the object tree, their subtrees appear *twice* in the tree. This is a concession to the tree structure, which unfortunately means that object trees grow larger the more imprecise the employed points-to analysis is, because a more imprecise points-to analysis leads to more may-aliasing. Since bigger object trees lead to higher runtime costs of the summary edge computation, this is a performance deadlock: A fast but comparatively imprecise points-to analysis causes a more expensive summary edge computation, a more precise points-to analysis relieving the costs of the summary edge computation is more expensive itself. As a remedy, Graf suggests merging

subtrees of pointers with the same aliasing situation. These *object graphs* shrink with more imprecise points-to analyses and thus resolve the performance problem.

2.5. Context-Sensitive Slicing via Call Strings

Context-sensitive slicing via SDGs and two-phase slicing has one hitch: The computation of summary edges employs a fixed-point search of all matched paths from actual-in to actual-out nodes of each call site, whose runtime complexity is cubic to the maximal number of parameter nodes in a PDG [119]. Evaluations indicate that the summary edge computation becomes a severe bottleneck for huge programs (>100k lines of code)[5]. In order to circumvent this, several authors [9, 25, 74] present techniques that omit summary edges and instead annotate nodes with their current calling contexts during the graph traversal, which are used to check whether the traversal towards a calling procedure preserves context-sensitivity. The most advanced solution has been presented by Krinke [74], thus his approach is presented here. Working with calling contexts during SDG traversal is a prerequisite for timing-sensitive slicing, and the technique presented here is employed in chapter 3.

The data structure used by this alternative slicing technique is the *interprocedural dependence graph* (IPDG), which is simply a SDG without summary edges. In order to compute context-sensitive slices, visited nodes are annotated with *call strings*, a textual representation of calling contexts.

Definition 2.12 (Call strings and contexts)**.** *Let G be an IPDG, and label each call site in G with a unique identifier s_i. A call string [131] is a list of call sites $\langle s_1,\ldots,s_k\rangle$, where for every $1 \leq i < k$, s_i calls the procedure of s_{i+1}. Let n be a node of G and $\sigma = \langle s_1,\ldots,s_k\rangle$ be a call string such that s_k calls the procedure of n. The pair $c = (n,\sigma)$ is a context of n. Indices are often used to make clear to which node a context belongs, i.e. context c_n is a context of node n.*

A context-sensitive traversal of IPDGs is achieved by checking whether the traversal from a procedure to its caller is in accordance with the call string of the current context. Given a context $(n,\langle s_1,\ldots,s_k\rangle)$ and an edge $m \rightarrow_e n$, the following rules guarantee a context-sensitive backward traversal:

- $e \in \{cd, dd\}$ (control or data dependence)
  This is an intra-procedural traversal. The traversal visits context $(m,\langle s_1,\ldots,s_k\rangle)$.
- $e \in \{po\}$ (parameter-out edge)
  The traversal descends to a called procedure, n is an actual-out node and belongs to a call site s, which is appended to the call string. The traversal visits context $(m,\langle s_1,\ldots,s_k,s\rangle)$.

[5] This was stated by Paul Anderson from GrammaTech in his keynote at the SCAM conference in 2008. GrammaTech is the producer of CodeSurfer [35], which is currently the most advanced SDG generator for C/C++.

- $e \in \{pi, call\}$ (parameter-in or call edge)
  The traversal ascends to a calling procedure. Node m is an actual-in or call node and belongs to a call site s, which is compared with the call site s_k at the end of the call string. If they match, the traversal is context-sensitive and visits context $(m, \langle s_1, \ldots, s_{k-1} \rangle)$. Otherwise, the traversal is rejected.

For a context-sensitive forward traversal, we have to switch the implications of the second and third rule.

Handling recursive calls

The usage of call strings bears a major problem: Recursive calls may lead to infinite call strings. A possible solution to that problem is k-limiting call strings: If a call string exceeds length k, the oldest element is removed. However, k-limiting introduces some context-insensitivity, because in case a call string has been decomposed to the empty string, the traversal has to leave the current procedure towards all call sites. Krinke's solution is to represent a recursive cycle of procedure calls by a single synthetic call site. Given a context $(n, \langle s_1, \ldots, s_k \rangle)$ and an edge $m \rightarrow_e n$, a context-sensitive slice is achieved by modifying the second and third rule as follows:

- $e \in \{po\}$ (parameter-out edge)
  The traversal descends to a called procedure. Let s be the corresponding call site. If s does not belong to a recursive cycle of procedure calls, we proceed as before. Otherwise, s is a synthetic call site representing a recursive cycle. If $s_k = s$, then we are already in the recursive cycle. The call string is propagated unchanged, for not to create infinite call strings, and the traversal visits context $(m, \langle s_1, \ldots, s_k \rangle)$. If $s_k \neq s$, we add the call site to the call string, visiting context $(m, \langle s_1, \ldots, s_k, s \rangle)$.

- $e \in \{pi, call\}$ (parameter-in or call edge)
  The traversal ascends to a calling procedure. If the corresponding call site s is not part of a recursive cycle, we proceed as before. Otherwise, we compare the synthetic call site s with the call site at the end of the call string. If they match, we have to propagate *two* different call strings: One that remains in the recursion and leaves the call string untouched, visiting context $(m, \langle s_1, \ldots, s_k \rangle)$, and one that leaves the recursion by removing the last element from the call string, visiting context $(m, \langle s_1, \ldots, s_{k-1} \rangle)$.

Algorithm 2.2 presents Krinke's *IPDG slicer*, which uses the above rules to compute context-sensitive slices of IPDGs. The algorithm is initialized with all possible contexts of the slicing criterion. These contexts can be determined on the *calling context graph* [75] of the program[6].

[6] Not to be confused with the more popular *call graph*.

Algorithm 2.2 Krinke's IPDG slicer [74].

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

```
W = {(s, σ) | σ is a call string of s}  // initialize the worklist with all contexts of s
S = {}  // the result set

repeat
  W = W \ {(n, σ)}  // process the next context in W
  S = S ∪ {n}
  for all m →_e n  // handle all incoming edges of n
    if e ∈ {pi, call}  // ascend to a calling procedure
      Let s be the corresponding call site
      if σ == σ's  // s is the last element of σ; this test guarantees context-sensitivity
        if m has not been marked with σ'
          W = W ∪ {(m, σ')}
          mark m with σ'
        if s is recursive and m has not been marked with σ
          // at recursive calls we additionally have to conserve call string σ
          W = W ∪ {(m, σ)}
          mark m with σ

    else if e ∈ {po}  // descend to a called procedure
      Let s be the corresponding call site
      if s is recursive and σ == σ's and m has not been marked with σ
        // we are in a recursive cycle – preserve the call string
        W = W ∪ {(m, σ)}
        mark m with σ
      else
        σ'' = σs  // append s to σ
        if m has not been marked with σ''
          W = W ∪ {(m, σ'')}
          mark m with σ''

    else  // intra-procedural edge – preserve call string σ
      if m has not been marked with σ
        W = W ∪ {(m, σ)}
        mark m with σ

until W = ∅
return S
```

Definition 2.13 (Calling context graph)**.** *A calling context graph $G = \{N, E\}$ of a program is a directed graph, whose set of nodes N consists of all call nodes of the program. E contains an edge $m \to n$ between $m, n \in N$ if m calls a procedure p and n is a call node inside p.*

The nodes in the calling context graph are used as the representatives of the call sites. Recursive cycles in the calling context graph are folded, yielding the synthetic call sites needed for

programs with recursive procedures. The possible call strings of a node n can be computed by a backward depth-first search starting at the call nodes that call n's procedure.

Optimizations

Let us take a look at the pseudocode of the IPDG slicer. It stands out that it has to check all the time whether the currently reached context has already been visited. The comparison of contexts is a bottleneck in the IPDG slicer because call strings can grow arbitrarily long. Krinke [74] suggests k-limiting of call strings as a possible optimization and shows in his evaluation that a choice of $k = 4$ offers a good trade-off between precision and speed. Since we need to remain context-sensitive in order to compute timing-sensitive slices, we developed an alternative to k-limiting. We suggest caching of call strings, so that contexts with the same call string share the same data structure. This reduces the comparison of call strings to a comparison of pointers. For that purpose, we arrange the set of possible call strings in a modified calling context graph, in which every node represents a complete call string and from which the slicer queries the reached call string when it enters or leaves a procedure. In contrast to k-limiting, this optimization preserves context-sensitivity. Our evaluation in section 2.6 shows that this technique leads to a significant speedup, and, despite the caching of call strings, even to lower memory consumption.

Krinke [74] furthermore observed that if a node n is annotated with two call strings σ, σ' and σ is a prefix of σ', then σ' can be omitted because a context-sensitive traversal starting at context (n, σ') visits a subset of the nodes that it visits for context (n, σ). However, notice that call string σ' describes a recursive call of n's procedure. Thus this situation does not appear if recursive calls are folded and no k-limiting is used. Hence, we do not investigate that optimization any further.

2.5.1. Context-Restricted Slicing

Binkley [23] noticed that the SDG/IPDG slicing approach presented so far is too imprecise in situations like the following: Imagine that a program aborts with an error at a statement n, but only for a certain invocation of n's procedure (e.g. a faulty usage of a library). The two-phase slicer and the IPDG slicer are context-sensitive, but they treat the slicing criterion n in a context-insensitive manner: The computed slice for n is the slice for all possible contexts of n. For the above scenario it would be suitable to have a slicer that computes a slice restricted to the calling context in which n causes the program abortion. To this end, Krinke [78, 79] suggests combining the two-phase slicer with the IPDG slicer. The resulting *context-restricted slicer* receives a context c as the slicing criterion and employs the context-based traversal technique of the IPDG slicer in phase 1, visiting only those procedures that form the calling context of c. Phase 2 is equivalent to phase 2 of the two-phase slicer and collects the nodes in procedures

Algorithm 2.3 Krinke's context-restricted slicer [79].

Input: A SDG G, a slicing criterion c (a context).
Output: The slice of G for c.

```
W1 = {c}  // the worklist for phase 1
W2 = {}  // the worklist for phase 2
S = {}  // the slice

/* phase 1 */
repeat
  W1 = W1 \ {(n, σ)}  // process the next context in W1
  S = S ∪ {n}
  for all m →_e n  // handle all incoming edges of n
    if e ∈ {pi, call}  // ascend to a calling procedure
      Let s be the corresponding call sites
      if σ == σ's  // this test guarantees context-sensitivity
        if m has not been marked with σ'
          W1 = W1 ∪ {(m, σ')}
          mark m with σ'
        if s is recursive and m has not been marked with σ
          // at recursive calls we additionally have to conserve call string σ
          W1 = W1 ∪ {(m, σ)}
          mark m with σ

    else if e ∈ {po}  // descend to a called procedure
      if m ∉ S  // m has not been visited yet
        W2 = W2 ∪ {m}

    else  // intra-procedural or summary edge – preserve call string σ
      if m has not been marked with σ
        W1 = W1 ∪ {(m, σ)}
        mark m with σ
until W1 = ∅

/* phase 2 */
repeat
  W2 = W2 \ {n}  // process the next node in W2
  S = S ∪ {n}
  for all m →_e n  // handle all incoming edges of n
    if m ∉ S ∧ e ∉ {pi, call}  // do not ascend to calling procedures
      W2 = W2 ∪ {m}

until W2 = ∅

return S
```

called underway. The pseudocode of this slicer is shown in Algorithm 2.3. Note that this algorithm requires summary edges.

2.5.2. Forward Slicing

Slicing as introduced so far determines the set of nodes that may influence the slicing criterion. *Forward slicing* [63] collects the set of nodes that may be influenced by the slicing criterion.

Definition 2.14 (Context-sensitive forward slice)**.** *A context-sensitive forward slice for slicing criterion s in a SDG G consists of the set of nodes $\{n \mid \exists\, s \rightarrow^*_{cs} n \text{ in } G\}$.*

In situations where we have to distinguish both kinds of slicing we speak of *backward* and *forward slicing*. Unless otherwise noted, the sole term 'slicing' always refers to backward slicing.

The presented backward slicers are easily converted into forward slicers. The edges have to be traversed forward and the treatment of interprocedural edges has to be inverted: Parameter-out edges are treated by the forward slicers like parameter-in and call edges by the backward slicers and vice versa.

2.6. Evaluation

We have implemented the presented slicing algorithms in Java and evaluated them on a set of 22 Java programs shown in Table 2.1. The programs in the upper part are small to medium-sized Java programs taken from the Bandera [1] benchmark and solve a certain task in a concurrent manner (e.g. LaplaceGrid solves Laplace's equation over a rectangular grid). The other programs are real JavaME [4] applications, a Java variant for mobile devices, taken from the SourceForge repository[7]. All these programs contain threads and were also used to evaluate our algorithms for concurrent programs. The sequential slicing algorithms simply treat threads as normal procedures and ignore all concurrency-related dependences. Our SDGs do not only include the source code of the program, but also parts of the called libraries. Column 'LOC' shows how many lines of code of the source code and of library code are included in the SDGs. The numbers for the library code were retrieved by analyzing the source code information present in Java bytecode.

The implemented algorithms work on SDGs computed by the Joana framework [45]. The evaluation was committed on a 2.2Ghz Dual-Core AMD workstation with 32 GB of memory running Ubuntu 8.04 (Linux version 2.6.24) and Java 1.6.0. Each slicing algorithm had a working memory of 8 GB at his disposal.

The evaluation consists of two parts. The first part compares the execution times and the average slice sizes of the two-phase slicer, the IPDG slicer and a context-insensitive slicer, which simply collects all nodes reaching the slicing criterion. The second part compares two-phase slicing with context-restricted slicing.

[7] `http://sourceforge.net/`

Table 2.1.: Statistics of our benchmark programs.

Name	LOC			Nodes	Edges	Procs
	src		lib			
Example	29	+	313	2509	36386	89
ProdCons	83	+	335	3331	39418	100
DiskScheduler	220	+	457	4389	43286	133
AlarmClock	187	+	366	4781	44328	124
DiningPhils	90	+	519	5115	125131	116
LaplaceGrid	175	+	531	6175	50359	161
SharedQueue	357	+	1138	11284	78022	199
EnvDriver	2677	+	472	19106	181407	180
KnockKnock	592	+	2471	35852	310325	506
DaisyTest	1114	+	2340	43138	433837	527
DayTime	371	+	4407	59708	626742	696
Logger	279	+	1165	10333	51614	227
Maza	921	+	1125	11235	66599	250
Barcode	783	+	1242	12344	63688	271
Guitar	761	+	1256	13257	67580	296
J2MESafe	512	+	1621	17754	124696	309
HyperM	366	+	1332	17768	91422	277
Podcast	2012	+	1965	23576	157303	407
GoldenSMS Key	1139	+	1736	21860	176376	362
GoldenSMS Msg	900	+	1913	26333	210493	414
GoldenSMS Rec	695	+	1796	22088	147465	370
Cellsafe	3024	+	2137	41707	860244	534

2.6.1. Context-Sensitive Slicing

In this part of our evaluation, we measured the average slice sizes and runtime performance of the two-phase slicer, *2P*, of the context-insensitive slicer, *CI*, which collects all nodes reaching the slicing criterion, and of two variants of the IPDG slicer. The first variant, *I*, uses our proposed technique of caching call strings, the second variant, *I-dyn*, builds and decomposes call strings during the slicing process, as originally proposed by Krinke [74]. As caching techniques always bear the risk of exhaustive memory consumption, we additionally compared the memory consumption of I and I-dyn. The algorithms had to compute the slices for each node in the SDGs.

Precision Table 2.2 shows the average size per slice and program in number of nodes of the context-sensitive slices and the context-insensitive algorithm. The three context-sensitive algorithms *2P*, *I* and *I-dyn* computed the same slices and are summarized by columns *cont.-sens*. The context-sensitive slices were always strictly more precise than the context-insensitive

Table 2.2.: Average sizes, in number of nodes, of the context-insensitive (cont.-ins.) and the context-sensitive slices (cont.-sens.). The percentage values denote the ratio of context-sensitive to context-insensitive slice sizes.

Name	cont.-ins.	cont.-sens.		Name	cont.-ins.	cont.-sens.	
Example	998	905	(91%)	Logger	3954	2860	(72%)
ProdCons	1275	1139	(89%)	Maza	4850	3981	(82%)
AlarmClock	2210	1291	(58%)	Barcode	6640	2736	(41%)
DiskScheduler	1668	1154	(69%)	Guitar	7411	3623	(49%)
DiningPhils	2688	1546	(57%)	J2MESafe	10353	6444	(62%)
LaplaceGrid	3335	1774	(53%)	HyperM	9505	3160	(33%)
SharedQueue	7657	3459	(45%)	Podcast	14622	8316	(57%)
EnvDriver	9059	6770	(75%)	GoldenSMS Key	12760	6232	(49%)
KnockKnock	22135	6715	(30%)	GoldenSMS Msg	16013	6491	(41%)
DaisyTest	32258	23000	(71%)	GoldenSMS Rec	13463	5576	(41%)
DayTime	46081	29566	(64%)	Cellsafe	30040	22928	(76%)

ones. On average, they contained about 60% of the nodes in the context-insensitive slices, in the best case, for KnockKnock, even only 30%.

Execution times Table 2.3 shows on the left side the average execution time per slice and program in milliseconds. The middle part shows the average slowdown compared with the fastest algorithm, the two-phase slicer, and the right side shows the slowdown of algorithm *I-dyn* compared with *I*. Several entries are missing for *I* and *I-dyn* because for several programs they ran out of memory and aborted.

One can see that context-insensitive slicing is significantly slower than two-phase slicing, on average about 1.9 times. This well-known effect results from the huge gain of precision: For the programs where the gain of precision was highest, e.g. HyperM and KnockKnock, the runtime differences were highest, too.

The IPDG slicers were clearly slower than the other algorithms. The measured execution times show that slicing based on call strings scales inferior to two-phase slicing. Algorithm *I* was on average 22.1 times slower than *2P*; in the worst case, for KnockKnock, it was even 280 times slower. The right side of Table 2.3 shows that caching of call strings results in a notable speedup: Algorithm *I* was on average more than twice as fast as *I-dyn*.

Memory consumption Table 2.4 compares the memory consumption of our two variants of the IPDG slicer. It shows the maximal used memory in megabytes per program. Several programs are missing because our profiling tool was restricted to 2 megabytes of memory, which were exhausted for these programs. The measured values show that caching and reusing of call strings significantly decreased memory consumption. On average, the dynamic context genera-

Table 2.3.: Average execution time per slice in milliseconds (left side), average slowdown compared with the two-phase slicer (mid), and average slowdown of dynamic context representation compared with static context representation (right side).

Name	CI	2P	I	I-dyn	CI	2P	I	I-dyn	I	I-dyn
Example	7	5	10	22	1.26	1	1.97	4.08	1	2.07
ProdCons	6	5	22	60	1.22	1	4.38	12.21	1	2.79
AlarmClock	6	5	29	66	1.22	1	5.65	12.75	1	2.26
DiskScheduler	6	6	22	53	1.00	1	3.26	8.92	1	2.47
DiningPhils	19	13	40	83	1.45	1	3.06	6.42	1	2.10
LaplaceGrid	14	8	47	104	1.71	1	5.57	12.46	1	2.24
SharedQueue	23	10	159	343	2.33	1	16.21	34.96	1	2.16
EnvDriver	37	28	234	398	1.35	1	8.48	14.42	1	1.70
KnockKnock	94	25	7011	–	3.77	1	280.26	–	1	–
DaisyTest	174	100	–	–	1.74	1	–	–	–	–
DayTime	252	137	–	–	1.84	1	–	–	–	–
Logger	10	8	25	56	1.30	1	3.27	7.23	1	2.21
Maza	13	11	54	129	1.15	1	4.91	11.63	1	2.37
Barcode	17	7	15	27	2.46	1	2.22	3.91	1	1.76
Guitar	19	9	38	68	2.03	1	3.95	7.19	1	1.82
J2MESafe	33	18	166	421	1.85	1	9.23	23.49	1	2.55
HyperM	25	8	28	61	3.25	1	3.60	7.79	1	2.16
Podcast	46	25	127	271	1.85	1	5.13	10.92	1	2.13
GoldenSMS Key	40	18	433	1122	2.23	1	24.10	–	1	–
GoldenSMS Msg	49	20	203	524	2.52	1	10.43	26.86	1	2.58
GoldenSMS Rec	42	16	200	487	2.62	1	12.54	–	1	–
Cellsafe	210	151	5039	5520	1.39	1	33.31	36.49	1	1.10

Table 2.4.: Maximal memory in Mbytes consumed by the algorithms *I-dyn* and *I*. The percentage values denote the ratio of *I-dyn*'s to *I*'s memory consumption.

Name	I	I-dyn		Name	I	I-dyn	
Example	88.21	116.63	(132%)	Logger	114.32	132.62	(116%)
ProdCons	114.65	134.93	(118%)	Maza	122.20	135.51	(111%)
AlarmClock	118.69	133.19	(112%)	Barcode	117.91	138.75	(118%)
DiskScheduler	111.36	135.06	(121%)	Guitar	120.64	144.12	(119%)
DiningPhils	117.40	140.92	(120%)	J2MESafe	253.40	1398.99	(552%)
LaplaceGrid	115.47	216.01	(187%)	HyperM	165.34	213.91	(228%)
SharedQueue	163.50	1335.93	(817%)	Podcast	225.45	380.08	(169%)
EnvDriver	212.81	538.96	(253%)	GoldenSMS Msg	303.32	1424.54	(470%)
GoldenSMS Key	733.60	1518.99	(207%)	GoldenSMS Rec	383.69	729.66	(190%)

tion done by *I-dyn* consumed more than 2.5 times as much memory as algorithm *I*. In the worst case, for SharedQueue, it needed more than 8 times as much memory. However, high memory consumption remains the most important problem of context-sensitive slicing via call strings.

2.6.2. Context-Restricted Slicing

The second part of our evaluation investigates how much more precise context-restricted slicing is in comparison with two-phase slicing. Again, we compared precision and runtime performance of both algorithms. For that purpose, we computed the two-phase slices for all nodes and the context-restricted slices for all contexts in the SDGs of our benchmark. The results are presented in Table 2.5. They show that the slice for a single context is on average 21% and in the best case up to 35% smaller than the two-phase slice for the node of that context. This is an expected result, since the comparison of context-insensitive and context-sensitive slicing shows that accounting for the calling context has a strong impact on precision. But surprisingly, the measured execution times show that context-restricted slicing was also similarly fast. The context-restricted slicer was on average even 1.1 times faster than the two-phase slicer. This competitiveness seems not to be limited to small and simple programs: For the biggest programs enlisted in the Table, Cellsafe and KnockKnock, it was about 1.2 times faster. Unfortunately, context-restricted slicing has one flaw: If the program size gets too big, it struggles with the memory consumption needed for the call strings. It was not able to slice the programs DayTime and DaisyTest due to memory exhaustion. Therefore, these programs are missing in Table 2.5.

A similar gain of precision of context-restricted slicing has been reported by Krinke [79]. Unfortunately, he did not report runtime costs, so it is unclear whether he experienced a similar runtime performance.

2.6.3. Study Summary

Our comparison of context-insensitive and context-sensitive slicing recapitulates a well-known result: Context-sensitive two-phase slicing is significantly more precise and usually faster than context-insensitive slicing. Our evaluation further shows that IPDG slicing with unlimited call strings is not competitive to two-phase slicing. It is much slower for our benchmark programs and consumes too much memory, and its performance will decline even more for bigger programs. At the moment, its application seems to be restricted to cases where summary edges are not available or where one needs to explicitly collect the set of visited contexts for other purposes. However, our caching optimization shows that IPDG slicing still offers opportunities for optimizations.

The second part of our evaluation investigated slicing for individual contexts. It shows that context-restricted slicing is much more precise than two-phase slicing and that it seems to be similarly fast in practice. This suggests that context-restricted slicing is a viable alternative to two-phase slicing if a slice for an individual context is required. This is an interesting result, which should be verified on a bigger benchmark. However, the practicability of context-

Table 2.5.: Two-phase slicing (2P) vs context-restricted slicing (CR): Average slice sizes (left side, number of nodes) and average execution times, (right side, in milliseconds). The values in brackets denote the size ratio (left side) and the speedup (right side) of context-restricted slicing to two-phase slicing.

Name	2P	CR		2P	CR	
Example	905	760	(84%)	6	7	(0.86)
ProdCons	1139	905	(79%)	5	5	(0.97)
AlarmClock	1291	842	(65%)	7	5	(1.24)
DiskScheduler	1154	781	(68%)	5	4	(1.19)
DiningPhils	1546	1023	(66%)	12	9	(1.33)
LaplaceGrid	1774	1242	(70%)	7	6	(1.24)
SharedQueue	3459	2951	(85%)	10	9	(1.06)
EnvDriver	6770	5655	(84%)	28	32	(0.89)
KnockKnock	6715	4589	(68%)	25	20	(1.25)
Logger	2860	1990	(70%)	7	6	(1.15)
Maza	3981	3094	(78%)	12	11	(1.07)
Barcode	2736	2209	(81%)	7	7	(1,04)
Guitar	3623	3297	(91%)	10	11	(0.91)
J2MESafe	6444	5564	(86%)	20	19	(1.04)
HyperM	3160	2592	(82%)	8	7	(1.06)
Podcast	8316	6923	(83%)	24	23	(1.05)
GoldenSMS Key	6232	6011	(96%)	17	18	(0.94)
GoldenSMS Msg	6491	5327	(82%)	21	19	(1.08)
GoldenSMS Rec	5576	4854	(87%)	14	13	(1.08)
Cellsafe	22928	18546	(81%)	174	146	(1.19)
Total			(79%)			(1.09)

restricted slicing with unlimited call strings is currently limited by the high amount of memory needed for the call strings, albeit in a lesser extent than IPDG slicing.

2.7. Related Work

There exists a vast amount of literature about slicing, of which this section presents only the most related. In particular, the description of related work concerning slicing and program dependence graphs for concurrent programs is postponed to chapter 3. *Chopping*, a technique closely related to slicing, is presented in chapter 4. Exhaustive overviews of fundamental slicing techniques can be found in Tip's classic survey [147] and in the recent survey of Xu et al. [158].

Slicing was first introduced by Mark Weiser in 1979, who developed his approach in several subsequent articles [155, 156]. His slicing algorithm consists of a data flow analysis working on the control flow graph of the program and is able to handle interprocedural programs, albeit in a context-insensitive manner. Weiser stated that his algorithm can be refined to be

context-sensitive, but to disproportionately high runtime costs. He also pointed out that finding statement-minimal slices is unsolvable because it is equivalent to solving the halting problem.

Karl and Linda Ottenstein [109] took the program dependence graph developed by Ferrante et al. [39] and founded the methodology of PDG-based slicing. Horwitz, Reps and Binkley [63] extended this intra-procedural approach to interprocedural programs. Their system dependence graph enables context-sensitive slicing in time linear to the number of edges by using the two-phase slicer. However, the computation of summary edges described in that article had a runtime complexity of $\mathcal{O}(TotalSites * Sites^2 * Params^4)$, where *TotalSites* is the total number of call sites in the program, *Sites* is the maximal number of call sites in any procedure and *Params* is the maximal number of formal-in nodes in any procedure's PDG. The final breakthrough came several years later, when Reps et al. [119] developed an asymptotically faster summary edge algorithm. Forgács and Gyimóthy [40] identified the memory consumption of the summary edge computation to be an important problem of the creation of SDGs of large programs. Their solution uses the call graph of the program in question to partition the procedures of the program into strongly connected components (SCCs) and computes the summary edges for each SCC in reverse invocation order. This proceeding relieves the amount of intermediate results that have to be cached during the computation.

Wasserrab and Lohner [153] developed a machine-checked proof formalized in Isabelle\HOL that two-phase slicing is correct in terms of Weiser's definition of slicing (definition 1.1). The proof extends ICFGs to labeled transition systems, by annotating the edges with the semantic effects of the source nodes. A two-phase slice is mapped to the ICFG by replacing the annotations of all edges whose source nodes are not in the slice by no-ops. The proof shows that this *sliced ICFG* weakly simulates the original ICFG with respect to the slicing criterion.

SDGs for object-oriented languages Some of the earliest work on SDGs for object-oriented languages stems from Larsen and Harrold [85], who extended the SDG to a *class dependence graph* (ClDG). The ClDG explicitly models the data members of classes as global variables and passes them as parameters to every method of the class. The ClDGs of a program are connected to a SDG and slicing is performed with the standard two-phase slicer. However, ClDGs are not object-sensitive. Due to the representation of the data members as global variables they cannot distinguish data members of different objects from the same class. Furthermore, the authors do not describe how to handle data members that are objects themselves: data members seem to be restricted to basic types. In order to represent inheritance, the authors suggest an incremental approach: ClDGs of derived classes reuse those ClDG portions of the base class that represent methods being not overwritten by the derived class. A polymorphic method call is modeled by adding one call site for each implementation of that method, grouped under a *polymorphic choice node*. In a subsequent publication, Liang and Harrold [88] explained that the incremental treatment of inheritance may produce incorrect slices. If a new derived class is added, the ClDG

of the base class is not updated. If the derived class introduces a new target for a polymorphic method call inside a method of the base class, the ClDG of the base class fails to account for that new possible target.

Tonella et al. [148] developed an object-sensitive SDG that does not represent data members as global variables but extends the method signatures such that an object can pass its necessary data members as soon as one of its methods is called. Their approach permits objects as parameters (*parameter objects*), but is *field insensitive* because it does not distinguish the fields of a parameter object. A parameter object is always treated as an indivisible entity.

Liang and Harrold [88] picked up and improved the above approaches. They use Tonella et al.'s technique to achieve object-sensitivity, and, in order to increase precision even further, they suggest representing parameter objects as trees, where each node represents an object and the children of a node represent its fields. Using object trees, they are able to identify and omit those fields of a parameter object which do not need to be passed to the called method. In order to cope with objects of recursively defined classes, the authors k-limit the unfolding of object trees, so that a node at the k-limit represents an object and all of its fields and thus leaves some field-insensitivity. In order to meet the shortcomings in [85], the authors describe in which cases the class dependence graph of a base class has to be recomputed after a new derived class is added, so that polymorphic calls are safely approximated.

As already described in section 2.4.3, Hammer and Snelting [57] addressed the problem of parameter objects of recursively defined classes. They propose an unrolling technique based on points-to information, which identifies where the unrolling of an object tree may be aborted without losing precision. Thus, their solution is more precise than Liang and Harrold's k-limiting.

Graf [49] investigated the behavior of object trees in a large-scale evaluation and observed that object trees and points-to analysis hamper each other: A faster but more imprecise points-to analysis may result in larger object trees and thus raises the runtime costs of the summary edge computation. Hence, one has either an expensive points-to analysis or an expensive summary edge computation. Graf suggests merging subtrees of object trees with the same aliasing situation to *object graphs*. This resolves the performance deadlock because the size of object graphs shrinks with the imprecision of the employed points-to analysis.

Allen and Horwitz [12] investigated how to include Java's exception handling into SDGs for Java programs. Their work describes which data and control dependences may arise from exception handling in Java and integrates exception handling into the SDG computation by extending the control flow graph with *exceptional control flow*. In order to yield the additional dependences, `try`, `catch` and `throw` statements are treated as *pseudo-predicates*, which receive additional outgoing control flow edges for the control flow in case of exceptions. Since throwing of exceptions introduces different points of return in a procedure, a CFG receives one

exceptional-exit node for each type of exception that may be thrown by the method. A call site of a procedure that may throw exceptions has multiple return nodes: One for the normal termination of the called procedure and one for each handler of the possible exceptions (a *catch* or *throw node*). These successors are connected with the corresponding exit or exceptional-exit nodes via interprocedural control dependence edges.

Hammer [52] argues that Allen and Horwitz' SDG extension is non-standard because the lack of unique exit nodes in their CFGs leads to interprocedural control dependence edges. Instead, he models the possible exceptions as additional formal-out and actual-out nodes and connects them via standard parameter-out edges. The actual-out nodes have outgoing data dependence edges to the corresponding catch or throw nodes in the PDG. That way, control and data dependences can be computed with the traditional algorithms. His solution results in the same transitive dependences as the one of Allen and Horwitz, albeit in not the same transitive control dependences.

Variants of slicing Since slicing is more or less a generic program analysis technique, time has led to a number of variants and specializations of slicing for special purposes. A detailed formal investigation of the relationship between different forms of slicing has been published by Binkley [24].

Slicing was originally developed to aid programmers during debugging. But since debugging is usually confined to an erroneous program run and slicing comprises every possible program run, slices often contain program parts not of interest for the particular program run. Korel and Lasky [72] introduced *dynamic slicing* as a remedy, which computes a slice only for a particular program run. Agrawal and Horgan [11] introduced the *dynamic dependence graph* (DDG), which represents the statements of a program run and the dependences between them and permits to compute dynamic slices via graph traversal. A major problem of dynamic slicing is to get the size of DDGs into grip. Originally, each point in a program run received it own node in the DDG. Several authors [11, 10, 162, 161] developed effective optimization techniques identifying subgraphs in DDGs that can be reused without sacrificing precision.

Dicing, invented by Lyle and Weiser [91], intends to yield more accurate results for debugging than traditional slicing. A dice is the set difference between the slice for a correct slicing criterion and a slice for an incorrect slicing criterion, where a slicing criterion is called correct if it showed no erroneous behavior during the computation of a given test suite. Dicing removes those parts of a slice that very unlikely contain the fault. However, dicing bears the risk of removing the faulty statement if the given test suite is not reliable and the program contains more than one bug [91].

Canfora et al. [29] introduced *conditioned slicing*. A conditioned slice of program p is computed for a *conditioned slicing criterion* s_{cond}, which is annotated with a first order formula F on a subset of the input variables of p. F reduces the set of possible inputs to the program and

therefore the set of feasible program executions. The conditioned slice for s_{cond} has only to consider these feasible program executions and is therefore smaller than the traditional slice for s. In order to compute a conditioned slice, p is first reduced to a *conditioned program* by a symbolic executor, which identifies and omits infeasible program paths. The result is transferred to the SDG G of p: The nodes absent in the conditioned program and the dependences that arise only from paths identified as infeasible are marked as invalid. Then, a slice of G for s is computed, which has to exclude the marked components. Fox et al. [41] implemented conditioned slicing for a subset of C in the ConSIT system.

Amorphous slicing, invented by Harman and Danicic [59], combines program transformation and slicing to achieve smaller slices than with traditional slicing. Any transformation that simplifies the program while preserving its semantics with respect to the slicing criterion is an amorphous slice. Amorphous slicing has been found useful in applications where preserving the syntax of the original program is not important, for example in program comprehension or re-engineering. Harman et al. [60] present an improved, interprocedural amorphous slicer for programs written in WSL, which has been implemented in the GUSTT system. The authors also present a combination of amorphous and conditioned slicing, *amorphous conditioned slicing*.

Krinke [76] defined the concept of *barrier slicing*, where the slicing algorithm is given a set of nodes or edges that must not be visited or traversed during the slice. The purpose of barrier slicing is to permit the exclusion of program parts one is not interested in, e.g. library code, and can drastically reduce the size of slices. The presented barrier slicer is an extension of the two-phase slicer that additionally checks that the given barrier is not violated. Barrier slicing requires to identify summary edges that are ‘blocked’ by the barrier, for which Krinke presents an efficient algorithm.

Path slicing by Jhala and Majumdar [68] intends to reduce control flow paths before they are fed to a model checker. The technique removes all parts of a control flow path that cannot influence the last statement in the path via control or data dependence. This is achieved by a combination of data flow analysis on the path and control dependence analysis on the whole control flow of the program. In particular, path slicing is able to identify and remove irrelevant loops and procedure calls from a given path, which greatly improves the performance of a subsequent model checking.

Sridharan et al. [140] introduced *thin slicing*, which takes into account only those statements that are part of a chain of assignments computing and propagating a value to the slicing criterion. Thin slicing does not only exclude control dependences, but also data dependences that do not directly contribute to the values used by the slicing criterion. For example, if a value relevant to the slicing criterion is temporarily stored in a container, thin slicing ignores the data dependences resulting from the internal mechanisms of the container.

Empirical studies A good portion of the publications on slicing contain some sort of evaluation. The survey of Binkley and Harman [26] identifies the most reliable and expressive evaluations and bundles their results. It incorporates evaluations on the size of slices (for both static and dynamic slicing), on the impact of points-to analysis, field- and flow sensitivity, on the usability of existing slicing tools, on applications of slicing and on human comprehension.

A question of fundamental interest is the impact of context-sensitivity on the size of the slices. Binkley and Harman's survey cites two studies [25, 74], which compare naïve context-insensitive slicing with two-phase slicing. Both report an increase of precision and runtime performance in favor of the two-phase slicer. In the first study, the context-insensitive slices were 50% bigger and 77% slower than the context-sensitive slices, in the second study, the context-insensitive slices were on average 67% bigger and 30% slower. This win-win situation is confirmed by our evaluation.

Another important question, addressing the usability of slicing, is to which extent a slice reduces the size of the overall program. Here the results of three studies [25, 90, 74] are mentioned in the survey. All three studies were committed on C programs and report that context-sensitive slices for interprocedural C programs contain on average about 30% of the whole program. Again, this result is supported by our evaluation, where the context-sensitive slices contained on average 32% of the whole program.

Krinke [74] evaluated the precision and runtime behavior of his IPDG slicer. In particular, he investigated k-limiting of contexts and evaluated the effects of different k's. His results indicate that a choice of $k = 4$ is a good trade-off between precision and speed. A bigger k led only in a few cases to increased precision, whereas the runtimes grew disproportionately. He concludes that IPDG slicing should only be used in cases where summary edges are not available.

Binkley et al. [27] investigated the effects of several optimization techniques on graph-based slicing, by evaluating these techniques on a benchmark with more than 1,000,000 lines of code. The investigated optimizations consisted of four different kinds of strongly connecting components, memory footprint reduction by bit packing and removal of redundant transitive edges. These optimizations were used in different combinations to preprocess the SDGs before slicing them with the two-phase slicer. The combination of all optimizations led to an average speedup of 4.57. However, as the optimizations themselves require computation time, the authors calculated that these optimizations pay only off if the number of slices computed exceeds 1.7% of the number of nodes in the SDG.

3. Slicing Concurrent Programs

This chapter discusses slicing of concurrent programs. The descriptions focus on Java's concurrency mechanism, which is based on threads and shared-memory communication, and are intended to enable the reader to reproduce and implement the presented techniques. They may have to be adjusted in order to handle languages with differing concurrency mechanisms. Preliminary versions of this chapter have been published together with Christian Hammer [44, 46].

Concurrency in Java

In Java, threads are special objects of the `Thread` class, whose behavior is described by the statements in their `run()` procedure. In order to create a thread, a user instantiates a `Thread` object and calls a special procedure `start()` that forks the thread by executing its `run()` procedure concurrently to the rest of the program. This can be done only once for each thread object, subsequent calls of `start()` on the same thread object lead to exceptions. A call of its `join()` procedure terminates the thread. The joining of threads is not mandatory, threads may in principle run infinitely. Furthermore, the termination of a thread t does not imply the termination of the threads started by t. Hence, threads may survive their parent threads. Threads in Java are totally dynamic. Instantiating, starting and joining threads may happen in loops or recursion, so there is no static bound on the number of threads. Static detection of the number of threads that may exist during a program run is generally undecidable.

According to the Java Language Specification [48], the concurrent execution of threads in a Java program can be achieved by time-slicing a single hardware processor, by using many hardware processors, or by time-slicing many hardware processors. All threads share a single heap for storage of objects and interact via monitor-style synchronization and shared variables. Synchronization is achieved by embedding the code to be synchronized in `synchronized` blocks or procedures, which only one thread is allowed to execute at a time. Each Java object provides a *monitor* that a thread can *lock* or *unlock*. Synchronized blocks receive the object whose monitor shall be used, synchronized procedures use the monitor of the reference object. If a thread enters a synchronized block or procedure, it tries to acquire the lock on the associated monitor. If the monitor is already locked by another thread, the thread blocks until the monitor becomes available. A lock on a monitor is released if the possessing thread leaves the synchronized block or procedure. A thread already holding a lock on a monitor may lock it multiple times, by entering other synchronized blocks that use the same monitor. A monitor counts the number of its

```
class ProdCons {
  public static
  void main(String[] s) {
    LinkedList buf
        = new LinkedList();

    // create the threads
    Producer prod1
        = new Producer(buf);
    Producer prod2
        = new Producer(buf);
    Consumer cons1
        = new Consumer(buf);
    Consumer cons2
        = new Consumer(buf);

    // start the threads
    prod1.start();
    prod2.start();
    cons1.start();
    cons2.start();

    // wait for the
    // threads to finish
    prod1.join();
    prod2.join();
    cons1.join();
    cons2.join();
  }
}
```

```
class Producer extends Thread {
  LinkedList buf;

  public Producer(LinkedList buf) {
    this.buf = buf;
  }

  public void run() {
    while (!enough())
      synchronized (buf) {
        buf.add(1);
        buf.notifyAll();
      }
  }
}

class Consumer extends Thread {
  LinkedList buf;

  public Consumer(LinkedList buf) {
    this.buf = buf;
  }

  public void run() {
    while (!enough())
      synchronized (buf) {
        while (buf.isEmpty())
          buf.wait();
        buf.pop();
      }
  }
}
```

Figure 3.1.: A producer-consumer program written in Java.

current locks and all these locks have to be released before another thread can lock it. Java additionally provides a *wait-notify* mechanism. Each object has a `wait()` procedure a thread may call to release its locks on that object's monitor and become inactive. An object keeps a waitlist containing all threads that are inactive due to calling its `wait()` procedure and provides the procedures `notify()` and `notifyAll()` to reactivate these threads. Calling `notifyAll()` reactivates all waiting threads, `notify()` reactivates an arbitrarily chosen thread.

Example The program in Figure 3.1 shows a simple producer-consumer example written in Java and is taken from Naumovich et al. [107]. The producer and consumer threads share a linked list used as a buffer. The main procedure forks the threads by calling their `start()` procedures and joins them at the end of the program. The threads execute their synchronized blocks repeatedly until the not shown procedure `enough()` returns 'true'. Both synchronized blocks use the buffer's monitor for synchronization. If a producer enters its syn-

chronized block, it adds a value to the buffer and wakes up all threads in the waitlist of the buffer by calling `buf.notifyAll()`. After leaving the synchronized block, the producer releases the lock. If a consumer enters its synchronized block and finds the buffer empty, it calls `buf.wait()` to release the lock and becomes inactive. If it is woken up by a producer's call of `buf.notifyAll()`, it has to lock the buffer's monitor again in order to proceed execution. It may do so after the locking thread has left its synchronized block (depending on the scheduler's choice), and in that case will leave the innermost while-loop, remove one element from the buffer and release the lock by leaving the synchronized block.

Note: The above example shows that the Java syntax is very verbose. We therefore use more compact pseudocode in the forthcoming examples.

The Java memory model

According to the Java Virtual Machine (JVM) specification [89], a Java program has a main memory in which all variables are stored. A thread of the program does not work on the main memory but on a local working memory that contains a copy of each variable known to the thread. The value of a variable in a working memory may be different from its values in the main memory and in the working memories of the other threads because the memories only have to synchronize on special occasions. The concrete rules for that mechanim are defined in the JVM specification and in the Java memory model [93]. Basically, only the execution of a synchronization operation guarantees that the working memories of the involved threads are synchronized. This mechanism may lead to the situation where a thread reading a shared variable `x` may not be aware of another thread having redefined `x` in the meantime if these two actions are not properly synchronized, i.e. if they form a data race. A data flow analysis for concurrent Java programs has to account for that behavior.

Interference dependence

Shared-memory communication gives rise to a special kind of data dependence, so-called *interference dependence* [73]. A statement *n* is interference-dependent on statement *m* if *n* may use a value computed at *m* and *m* and *n* may execute concurrently.

Figure 3.8 shows a program fragment consisting of two threads that communicate through the shared variables `x` and `y`. The SDGs of both threads are connected via two interference edges representing the interference dependences. Interference dependences cross procedure borders arbitrarily and thus are not captured by traditional summary edges. As a consequence of this, the two-phase slicer for sequential programs cannot be applied; it may even compute incorrect slices [106]. An example is given in section 3.5, after all the necessary concepts have been introduced. Nanda [106] extended the two-phase slicer to a context-sensitive slicer for concurrent programs. This extension is presented in section 3.5.2.

```
int x, y;

void thread_1():
  x = y + 1;
  y = 0;
```

```
void thread_2():
  int a = y;
  x = <input>;
  if a > 0
    int b = 0;
  else
    y = 0;
```

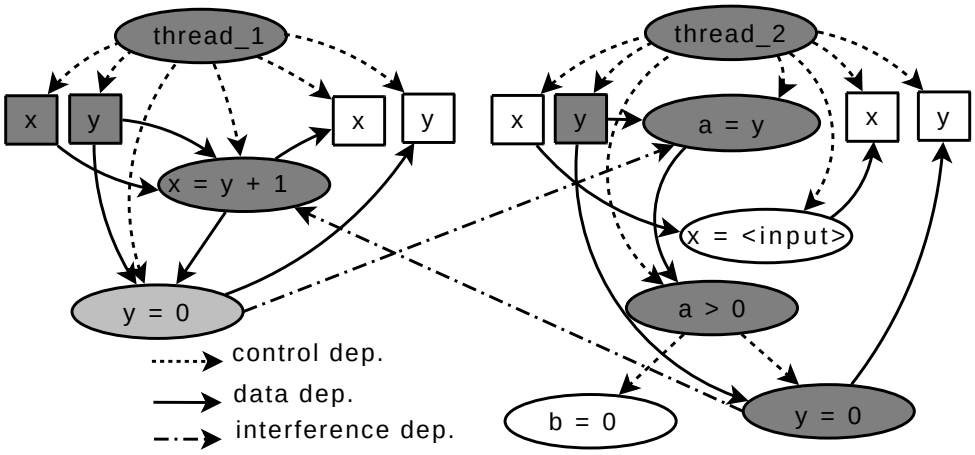


Figure 3.2.: Interference dependences between two threads.

Timing-insensitivity

An interference dependence makes an implicit requirement on the interleaving of the involved threads: The source of the dependence must be able to execute before the sink. Different interference dependences may exclude each other, because their underlying requirements on the interleaving are incompatible. This is actually the case in Fig. 3.2. The two interference dependences in the example exclude each other, because the one basically requires `thread_2` to execute *before* `thread_1` and the other one `thread_2` to execute *after* `thread_1`. If a slicer ignores this incompatibility, it may produce *timing-insensitive* slices. As an example, let us compute the slice for node `x = y + 1` by simply collecting all reaching nodes. The result is highlighted gray. But the light gray node `y = 0` in `thread_1` cannot influence the slicing criterion, because it cannot execute before `x = y + 1`. Krinke [73] coined the term *time travel* for that kind of imprecision, because a program execution would have to journey through time in order to create a suitable execution order.

The problem is complicated even more by procedure calls, because it may be the case that two interference dependences exclude each other only in certain calling contexts of their procedures. In such a case the interference dependences may exclude each other for one slicing criterion *but may be compatible for another one*, which prevents a treatment directly in the SDGs. Figure

```
int x, y;

void thread_1():
  foo();
  print(x);
  foo();
  print(x);

void foo():
  x = y + 1;
  y = 0;
```

```
void thread_2():
  int a = y;
  x = <input>;
  if a > 0
    int b = 0;
  else
    y = 0;
```

Figure 3.3.: Timing-insensitivity is even more complicated for interprocedural programs.

3.3 extends the program in Fig. 3.2 and puts the code of `thread_1` into procedure `foo`. The slice for the first `print(x)` does not contain node `y = 0` because here the timing-insensitive situation described in Fig. 3.2 appears. But the slice for the second `print(x)` does contain `y = 0`, because here the second interference edge can be traversed to node `y = 0` from the first invocation of `foo`. Thus, the detection of mutually exclusive interference dependences has to be done during the slicing process. Krinke [73, 75, 77] and Nanda [104, 105, 106] developed suitable timing-sensitive slicing algorithms, which are inspected in detail in this chapter.

Organization of this chapter

The analysis of concurrent programs requires information about which parts of the program may actually execute in parallel, which is determined by a *may-happen-in parallel* (MHP) analysis. Section 3.1 presents our *thread invocation analysis* for Java, which determines which threads may exist at runtime, section 3.2 introduces the *threaded control flow graph* and section 3.3 describes our MHP analysis.

Since concurrency and shared memory communication introduce new kinds of program dependences, the system dependence graph has to be extended. Section 3.4 presents the *concurrent system dependence graph* (CSDG), and section 3.5 describes Nanda's extension of the two-phase slicer for computing context-sensitive slices of CSDGs [106]. The subsequent sections deal with timing-sensitive slicing: Section 3.6 introduces the foundations, section 3.7 investigates the impact of MHP information on timing-sensitive slicing, and section 3.8 discusses limitations of timing-sensitive slicing. Sections 3.9 and 3.10 describe the timing-sensitive slicing algorithms of Krinke and Nanda as well as our extensions and optimizations. Section 3.11 presents a new slicing algorithm that is not completely timing-sensitive, but significantly faster in practice. Section 3.12 presents an extensive evaluation of the presented algorithms. Section 3.13 discusses several open issues and future work and section 3.14 summarizes related work.

3.1. Thread Invocation Analysis

A thread invocation analysis determines which threads may exist during a program run. For languages like Java, where it is possible to create threads inside loops and conditional structures, this cannot be decided in general by a static analysis, thus a conservative approximation is needed. In the context of slicing it is sound to determine too much existing threads, but unsound to determine too few. Our thread invocation analysis employs contexts (cf. sect. 2.5) for identifying and distinguishing the threads in a program and is inspired by Barik's MHP analysis [18]. Via the calling context graph (def. 2.13) of the analyzed program we collect all contexts of the `run()` procedures of its thread classes. As long as threads are not created inside loops or recursive procedures, each of these *thread contexts* represents exactly one thread of the program.

The topmost example in Fig. 3.4 illustrates that case. The depicted program possesses at runtime at most four threads: the `main` thread, one of kind `thread_1` and two of kind `thread_2`. The graph on the right of the program is its calling context graph, where forks are treated like ordinary procedure calls. The numbers of the nodes correspond to the line numbers of the program. The program possesses the thread contexts $\{(main, \langle\rangle), (thread\_1, \langle 2\rangle), (thread\_2, \langle 3\rangle), (thread\_2, \langle 2,5\rangle)\}$, which exactly represent the four possible threads.

If a thread is created inside a loop or recursive procedure, we assume conservatively that infinitely many threads of the corresponding thread class are created. These threads are represented by a single thread context marked as a *multi-thread*. The second example in Fig. 3.4 displays a program that creates threads inside a loop. Collecting the thread contexts yields the contexts $(main, \langle\rangle)$, $(thread\_1, \langle 2\rangle)$, $(thread\_2, \langle 2,8\rangle)$, $(thread\_1, \langle 4,6\rangle)$ and $(thread\_2, \langle 4,6,8\rangle)$. The first three contexts each represent exactly one thread, the other two contexts originate from the `while`-loop and are marked as multi-threads. Multi-threads originating from loops are identified by screening thread contexts for call sites inside loops, in our example node 4.

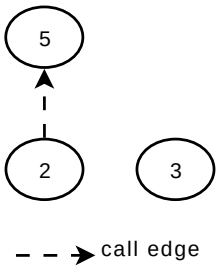
The third example shows a program that creates a thread inside a recursive procedure. It has five thread contexts: $(main, \langle\rangle)$, $(thread\_1, \langle 2\rangle), (thread\_2, \langle 2,8\rangle), (thread\_1, \langle\{3,6\},5\rangle)$ and $(thread\_2, \langle\{3,6\},5,8\rangle)$, where $\{3,6\}$ denotes the synthetic call site representing the recursive cycle formed by call sites 3 and 6. The first three contexts each represent one thread. The other two contexts represent the threads invoked inside the recursive procedure and are marked as multi-threads. These multi-threads can be identified searching for synthetic call sites in the thread contexts.

The fourth example illustrates the case where threads invoke *themselves* recursively, either directly or indirectly. Here we have the contexts $(main, \langle\rangle)$, $(thread\_1, \langle 2\rangle)$, $(thread\_2, \langle 3\rangle)$, $(thread\_1, \langle 2, \{5,7\}\rangle)$, $(thread\_1, \langle 3, \{5,7\}\rangle)$, $(thread\_2, \langle 2, \{5,7\}\rangle)$ and $(thread\_2, \langle 3, \{5,7\}\rangle$. The last four thread contexts result from the recursion and are marked as multi-threads. They can be identified by looking in the thread contexts for synthetic call sites.

```
1 void main():
2   fork thread_1();
3   fork thread_2();

4 void thread_1():
5   fork thread_2();

6 void thread_2():
7   ...
```

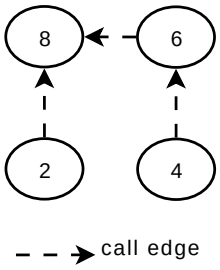



```
1 void main():
2   fork thread_1();
3   while (...):
4     foo();

5 void foo():
6   fork thread_1();

7 void thread_1():
8   fork thread_2();

9 void thread_2():
10  ...
```

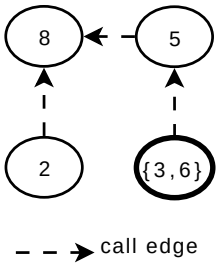



```
1 void main():
2   fork thread_1();
3   foo();

4 void foo():
5   fork thread_1();
6   foo();

7 void thread_1():
8   fork thread_2();

9 void thread_2():
10  ...
```




```
1 void main():
2   fork thread_1();
3   fork thread_2();

4 void thread_1():
5   fork thread_2();

6 void thread_2():
7   fork thread_1();
```

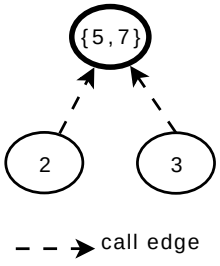


Figure 3.4.: Four cases distinguished by our thread invocation analysis. From top to bottom: (1) Thread creation without loops or recursion. (2) Thread creation inside a loop. (3) Thread creation inside a recursive procedure. (4) Two threads creating each other recursively.

3.2. The Threaded Control Flow Graph

A suitable representation of the control flow in a concurrent program with fork-join style parallelism is the *threaded control flow graph* (TCFG). A TCFG consists of the ICFGs of the single threads of the program, connected via *fork* and *join* edges. A TCFG has to account for the fact that threads in Java and likewise languages are created dynamically so that their concrete number is generally not decidable by a static analysis, and due to thread creation inside loops or recursion even an upper bound may not exist. Thus, it is not possible to model each potential thread as a separate subgraph in the TCFG. Instead, we follow Hammer's [52, chapt. 3] idea of modeling threads in analogy to procedures: A TCFG contains one ICFG per thread class.

Definition 3.1 (Threaded control flow graph (TCFG))**.** *A threaded control flow graph* $G = (ICFG_p, main, F, J)$ *of a program* p *consists of the set* $ICFG_p$ *of ICFGs of the thread classes of* p*, of a distinguished ICFG* main*, and of two sets,* F *and* J*, of* fork *and* join *edges. A TCFG has the following properties:*

- *For each pair* (T, T') *of ICFGs in* $ICFG_p$*, the node sets* $N_T, N_{T'}$ *and the edge sets* $E_T, E_{T'}$ *are disjoint.*

- *The ICFGs are connected via fork and join edges. Set* F *contains a fork edge* $(f_T, s_{T'})$ *if* f_T *is a fork node in ICFG* T *that invokes a thread of the thread class represented by ICFG* T' *and* $s_{T'}$ *is the start node of* T'*. Set* J *contains a join edge* $(e_T, j_{T'})$ *if* $j_{T'}$ *is a join node in ICFG* T' *that joins a thread of the thread class represented by ICFG* T *and* e_T *is the exit node of* T*.*

- *Every node in the TCFG is reachable from the start node of* main*.*

Several properties of that definition merit discussion:

- The definition requires the ICFGs in a TCFG to be disjoint. Since threads may share procedures (e.g. libraries), this means that the CFGs of shared procedures have to be duplicated. We make this requirement to simplify matters; in practice, it is recommended to share the CFGs amongst different ICFGs because CFG duplication may drastically increase the size of TCFGs. Yet, CFG sharing introduces the problem of *thread insensitive* paths: A CFG may be entered coming from one ICFG and may be left towards another one. Fortunately, thread sensitivity can be achieved unproblematically by assigning each thread a unique ID and annotating nodes with the current ID during traversal [52, chapt. 3]. By requiring ICFGs to be disjoint we can omit this treatment in our descriptions and in the pseudocode of our algorithms.

```
int x,y;

void main():
  int p = x - 2;
  fork thread_2;
  int q = foo(p);
  y = q * 3;

int foo(int f):
  return f + 1;

void thread_2():
  int a = y + 1;
  int b = y;
  if (a > 0):
    join main;
  else:
    x = b / a;
```

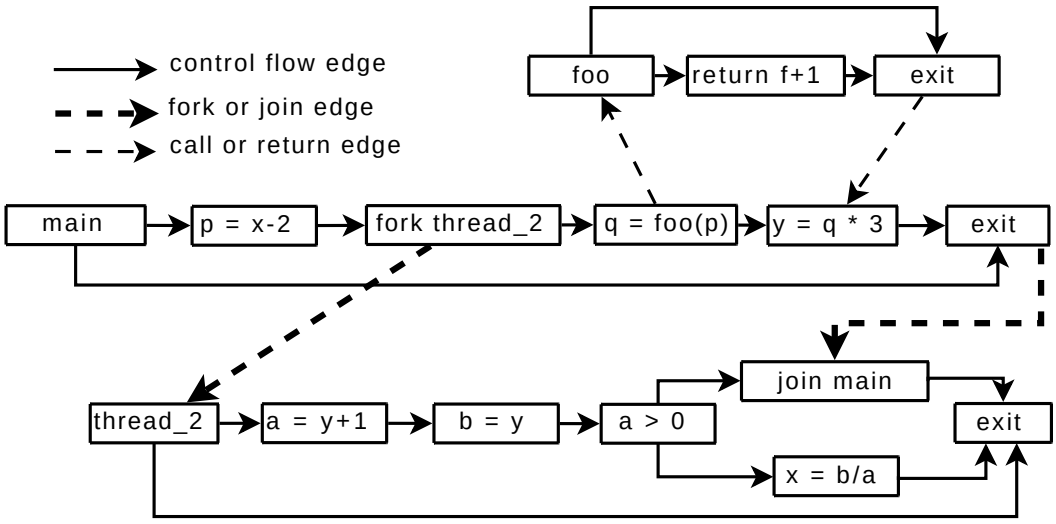


Figure 3.5.: A TCFG of a program with two threads.

- In many languages, including Java, threads do not have to be joined at all and may run infinitely, even after the main thread has terminated. Thus, a TCFG does not possess a unique exit node.

Figure 3.5 shows a TCFG of a program with two threads.

3.2.1. Reachability between contexts

An important concept for analyzing TCFGs is reachability between contexts. Let n be a node in a TCFG, and let G be n's ICFG. The call string of a context of n in the TCFG consists of the call string of the thread context of a thread represented by G, appended with a call string of n in G, computed as described in section 2.5. Thus, a context in a TCFG belongs to a fixed thread.

Krinke [75] defines reachability between contexts as follows: A context (m, σ_m) of a node m *directly reaches* another context (n, σ_n) of a (not necessarily different) node n, written $(m, \sigma_m) \rightsquigarrow (n, \sigma_n)$, if one of the following cases holds:

1. There exist a control flow edge $m \to n$ in the TCFG and $\sigma_m = \sigma_n$.
2. There exist a call edge $m \to n$ in the TCFG such that s symbolizes the call site and
   - $\sigma_m = \sigma$ and $\sigma_n = \sigma s$, or
   - $\sigma_m = \sigma_n = \sigma s$ and s is a synthetic call site
     (the definition treats recursive cycles in analogy to the IPDG slicer in section 2.5).
3. There exist a return edge $m \to n$ in the TCFG such that s symbolizes the call site and
   - $\sigma_m = \sigma s$ and $\sigma_n = \sigma$, or
   - $\sigma_m = \sigma_n = \sigma s$ and s is a synthetic call site.

Krinke's work ignores forking and joining of threads, so we extend that definition with cases for fork and join edges. Let $thread(c)$ return the longest prefix of the call string of context c that forms the call string of a thread context.

4. There exist a fork edge $m \to n$ in the TCFG such that s symbolizes the call site and
   - $\sigma_m = \sigma$ and $\sigma_n = \sigma s$, or
   - $thread((m, \sigma_m)) = \sigma_n$ and s is a synthetic call site
     (the second case treats threads that invoke themselves recursively; such threads share the same thread context).
5. There exist a join edge $m \to n$ in the TCFG
   (since the joining of a thread does not need to take place in the same procedure and not even in the same thread as the forking, join edges impose no restrictions).

According to Krinke, a context c *reaches* another context c, written $c \rightsquigarrow^* c'$, if there exists a sequence $\langle c_1, \ldots, c_k \rangle$ of contexts with $c = c_1, c' = c_k$ and $\forall 1 \leq i < k : c_i \rightsquigarrow c_{i+1}$. It follows directly from this definition that reachability between contexts is transitive.

3.3. May-Happen-In-Parallel Analysis for Java

A *may-happen-in-parallel* (MHP) analysis identifies those program parts that may execute concurrently. Since an optimal static MHP analysis is undecidable, a suitable conservative approximation is required. For many purposes, including slicing, it is sound to assume too much parallelism, but unsound to omit some. Too much parallelism may lead to spurious interference dependences, but too few parallelism may prune valid ones.

The degree of concurrency in a Java program is influenced by the fork and join points of threads and by synchronization. Figure 3.6 shows the control flow graph of a program consisting of two threads, where `main` forks and joins `thread_1`. The statements `lock l` and `unlock l`

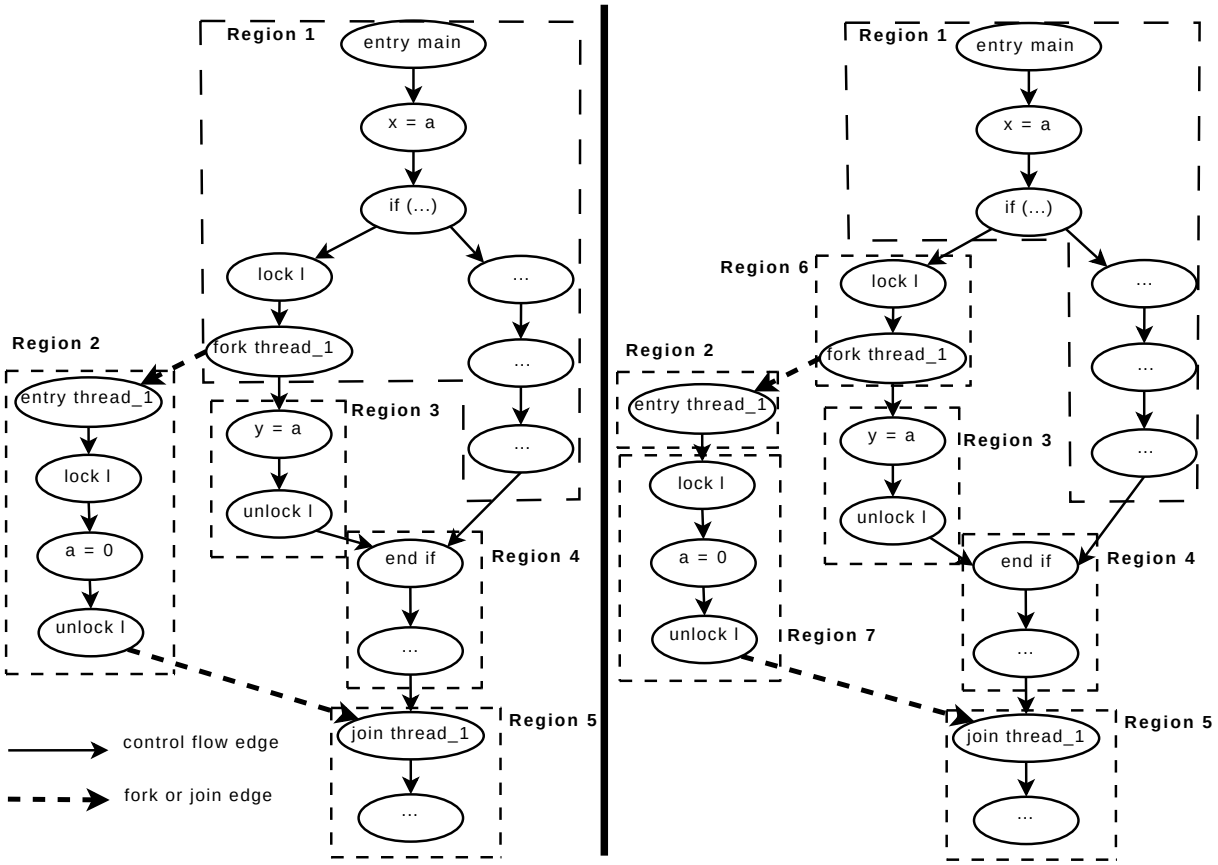


Figure 3.6.: Thread regions of a threaded program. On the left side: thread regions without synchronization. On the right side: thread regions with synchronization.

symbolize monitor-style synchronization, where `lock l` locks monitor `l` and `unlock l` releases it. A simple and sound MHP analysis is to assume that all threads may entirely happen in parallel to each other. In the current example, this assumption ignores forking, joining and synchronization and concludes that `main` and `thread_1` may happen in parallel. However, this assumption leads to a very imprecise computation of interference dependences[1]. In our example, it leads to a spurious interference edge from statement `a = 0` in `thread_1` to statement `x = a` in `main`. We therefore aim for a more precise solution.

3.3.1. Overview of Existing MHP Analyses for Java

To date, there does not exist a scalable MHP analysis for full Java suitable for slicing that includes synchronization. The central problem is that such an analysis must only take synchronization into account that is guaranteed to happen (*must-synchronization*). Since synchronization in Java works by locking monitor objects, the computation of must-synchronization for Java programs in turn requires must-aliasing of monitor objects. Since must-alias analyses have

[1] As we show in our evaluation in section 3.12.1.

been investigated only sparely, a satisfactorily approach to computing must-synchronization remains an open problem.

The most precise solution has been presented by Li and Verbrugge [87], an extension of Naumovich et al.'s MHP analysis [107]. It determines for every pair (s, s') of statements whether s and s' may happen in parallel. The analysis works on a *parallel execution graph* (PEG), which is derived from the control flow graph of the input program and models forking, joining and synchronization operations. Operations that access monitor objects whose aliasing situation is ambiguous are duplicated such that each copy represents a must-alias situation. The PEG imposes several restrictions on the input program. Every possibly existing thread has to be modeled in the PEG as a separate subgraph, which means that the number of threads created inside loops or recursive procedures has to be bounded statically. PEGs also require inlining of procedures containing synchronization operations in order to achieve context-sensitivity, hence such procedures must not be involved in a recursion. The MHP analysis on the PEG has a runtime complexity cubic to the number of PEG nodes and seems to be practical only for PEGs with up to 2000 nodes [87].

Nanda [106] segments Java threads at fork and join points into *thread regions* and determines concurrency on the level of these regions. In summary, a thread region starts behind a fork node, at a join node or at a node where two distinct thread regions meet, and it consists of the nodes dominated by its start node. All nodes in one thread region may happen in parallel to the same set of nodes, thus it suffices to compute and store the MHP information on the level of thread regions. Figure 3.6 shows on the left side the thread regions of the depicted example program. It can be determined that only regions 2 and 3 as well as 2 and 4 may happen in parallel. This analysis is not as precise as the one of Li and Verbrugge / Naumovich et al. because it ignores synchronization, but on the other hand it is able to handle recursion completely and can be extended to handle multi-threads. It is also possible to extend the analysis to include synchronization. In Figure 3.6, the acquisition of monitor `l` before the fork of `thread_1` assures that regions 2 and 3 cannot happen in parallel. The computation of thread regions would have to treat `lock` and successors of `unlock` as additional starting points of thread regions. The result is shown on the right side of Fig. 3.6. Having that information, the analysis would be able to determine that only regions 2 and 3, 2 and 4 as well as 7 and 4 may happen in parallel. However, this would require an analysis of must-synchronization and to date no implementation of such an extension has been reported. A weak point of Nanda's approach is that it lacks a thread invocation analysis and assumes that the number of threads is statically known. However, it can be extended to deal with multi-threads.

Barik's [18] MHP analysis for Java is divided into two phases. The first phase computes initial MHP information on the level of threads, the second phase refines that information on the level of single statements. The first phase is based on the *thread creation tree* (TCT), which

```
void main():
  if (...):
    fork thread_1;
  else:
    fork thread_2;

void main():
  fork thread_1;
  fork thread_2;
```

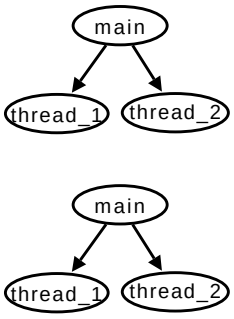


Figure 3.7.: Two programs and their thread creation trees.

represents the thread invocation structure of a program: A node in the tree represents a thread t, its parent represents the thread that forks t and its children represent the threads that t forks. A thread θ is called an *ancestor* of another thread θ' if its representative in the TCT is an ancestor of that of θ'. Figure 3.7 shows two programs, whose main threads create two other threads, and their TCTs. The MHP computation in the first phase precisely determines the MHP information between threads that are *not* in an ancestor relation in the TCT, by using context-sensitive ICFG paths and must-join information. It assumes, however, that each thread may happen in parallel to all of its ancestors in the TCT. For example, it would state that in the two programs in Fig. 3.7 the main threads happen in parallel to threads 1 and 2. This imprecision is removed in the second phase, which computes precise MHP information between statements from threads in an ancestor relation in the TCT. Barik's MHP analysis is capable of full Java, including recursion and multi-threads, but also ignores synchronization.

We chose to use Nanda's analysis and to improve it by adding our thread invocation analysis and by using the first phase of Barik's analysis as an optimization. For our purposes, the analysis of Li and Verbrugge / Naumovic et al. imposes too much restrictions on Java programs. The usage of thread regions is compelling, because it avoids to compute and store MHP information on a per-statement level, which promises a good scalability compared with the other presented approaches. Inspired by Barik's technique, our MHP analysis determines the MHP information on the level of threads whenever possible and uses Nanda's thread region analysis otherwise.

3.3.2. Our MHP Analysis

In order to simplify matters, we assume in this section that a TCFG contains one ICFG per thread distinguished by the thread invocation analysis. The presented technique can be applied to our TCFGs by annotating nodes with thread IDs during the analysis, but using such annotations here would unnecessarily complicate the presentation.

In analogy to thread contexts, a *fork site context* is the context of a fork site. The fork site context of a thread can be trivially derived from the call stack of the thread context. Given a fork site context f, we say that f *indirectly forks* a thread θ if it forks θ or an ancestor of θ.

In order to remain conservative, our MHP analysis only accounts for *must-joins*. A join node of a thread θ is a must-join of θ if the points-to analysis reveals that θ is the only possible thread joined there. In particular, multi-threads have no must-joins.

Given a TCFG G, we assume that two nodes m, n in G cannot happen in parallel if one of the following cases holds:

1. m and n belong to the same thread θ and θ is not a multi-thread.
2. m and n belong to different threads and one node is dominated by a must-join of the thread of the other node.
3. m and n belong to different threads and there exists no fork site context that indirectly forks the thread of one node via an outgoing fork edge and reaches the other node via a different outgoing edge.

The first case deals with the problem of multi-threads. If a loop or recursion may fork multiple threads of the same thread class, we assume conservatively that these threads happen in parallel. The second case treats thread joining conservatively. If, say, node n is dominated by a must-join of m's thread, then it is guaranteed that m's thread has ceased to exist at the time a program execution executes n. The third case captures the necessary prerequisite for a parallel execution of m and n. There must exist a fork site context which actually forks the control flow of the program such that m is executed in the one branch and n in the other. Actually, the first case is comprised by the third, but for clarity this special case is distinguished.

Only in these cases our MHP analysis is allowed to identify two nodes as not happening in parallel.

Thread regions

Nanda [106] defines thread regions as follows:

Definition 3.2 (Thread region)**.** *Let G be a TCFG. Node n starts a thread region R in G if n is a start node of a thread, a join node, a direct successor of a fork node, or a node where two distinct thread regions meet (called a* merge node*). A node m belongs to R if n is the closest start node of a thread region that dominates m.*

Thread regions work on the level of nodes and ignore calling contexts. Precision could be increased by using contexts instead of nodes, but our corresponding experiments exhibited disproportionately high computation times.

Nanda's original MHP analysis lacks a thread invocation analysis and assumes that the number of threads is statically known. In particular, it ignores the issue of multi-threads. Thus, we present an extended MHP relation for thread regions that incorporates multi-threads:

Definition 3.3 (MHP relation $\|_{region}$)**.** *Let G be a TCFG, let Q and R be two thread regions in G, and let θ be R's thread and θ' be Q's thread.*

1. *$\theta = \theta'$ and θ is a multi-thread, or*

2. *$\theta \neq \theta'$, neither start node of one of the two regions is dominated by a must-join of the thread of the other region, and*

   - *there exists a fork site context f in G that indirectly forks the thread of the one region via an outgoing fork edge and reaches the start node of the other region via a different outgoing edge.*

The following relation determines whether two nodes may happen in parallel.

Definition 3.4 (MHP relation $\|_{node}$)**.** *Let G be a TCFG. Two nodes m,n in G may happen in parallel, written $m \| n$, if there exist two thread regions Q,R in G such that $m \in Q, n \in R$ and $Q \|_{region} R$.*

Relation $\|_{node}$ meets our requirements for a sound MHP analysis. Two nodes in the same thread are only reported not to happen in parallel if the thread is not a multi-thread, two nodes in different threads, if one of both is dominated by the must-join of the thread of the other or if there exists no fork site context that indirectly forks the thread of one node via an outgoing fork edge and reaches the other node via a different outgoing edge.

We need a last MHP relation, $\|$, which accounts for our assumption that each thread is represented by a separate ICFG. Relation $\|$ is meant to be used from 'outside', where we have only one ICFG per thread class. It can be applied to pairs of nodes and to pairs of nodes annotated with concrete threads.

Definition 3.5 (MHP relation $\|$)**.** *Let G be a TCFG.*

- *Two nodes m,n in G may happen in parallel, written $m \| n$, if there exist two thread regions Q,R in G such that $m \in Q, n \in R$ and $Q \|_{region} R$.*

- *Two annotated nodes $(m,\theta_m),(n,\theta_n)$ in G may happen in parallel, written $(m,\theta_m) \| (n,\theta_n)$, if there exist two thread regions Q,R in G such that $m \in Q, n \in R$, Q belongs to thread θ_m, R belongs to thread θ_n and $Q \|_{region} R$.*

MHP information on the level of threads

Barik [18] observed that two threads that do not indirectly fork or join each other or synchronize with each other either happen entirely in parallel or not at all. A more fine-grained analysis is only necessary for the other threads. We use that observation to optimize the thread regions approach.

Barik's TCTs are used to identify threads that may entirely happen in parallel. Our proceeding differs from Barik's because his analysis considers threads in an ancestor relation to happen in parallel in order to yield a sound result. We do not follow that approach because we intend to use his technique only to optimize the MHP analysis based on thread regions. Thus, we determine concurrency on the level of threads as follows:

Definition 3.6 (MHP relation $\|_{thread}$)**.** *Let θ_0 and θ_1 be two threads. Let t_0 and t_1 be their representatives in the corresponding TCT and let t_a be their lowest common ancestor. Let θ_a be the thread represented by t_a.*

If $\theta_a \neq \theta_0 \wedge \theta_a \neq \theta_1$, let f_0 be the fork site context in θ_a that indirectly forks θ_0 and let f_1 be the fork site context in θ_a that indirectly forks θ_1.

θ_0 and θ_1 may happen in parallel, written $\theta_0 \|_{thread} \theta_1$, if

1. *θ_a is a multi-thread, or*
2. *$\theta_a \neq \theta_0 \wedge \theta_a \neq \theta_1$ and $\exists i \in \{0,1\} : f_i$ reaches f_{1-i} in θ_a's ICFG via a non-trivial path (length > 0).*

The first rule addresses threads forked by a multi-thread. We conservatively assume that in such a situation an unbound number of threads is created and that these threads may happen in parallel. The second rule deals with the situation where two threads are subsequently forked by the same ancestor.

Let us apply the above rules to the example programs in Fig. 3.7. The analysis of the upper program shows that `thread_1` and `thread_2` cannot happen in parallel, because none of the rules applies: Their last common ancestor, the main thread, is not a multi-thread, and none of the fork sites reaches the other. The analysis of the lower program yields a different result: Here `thread_1` and `thread_2` may happen in parallel, because the fork site of `thread_1` reaches the fork site of `thread_2`.

Relation $\|_{thread}$ is integrated into relation $\|_{region}$. The idea is that the second part of rule 2 of $\|_{region}$ can be partitioned into two cases. In the first case, the two regions R and Q referred to by the rule belong to two different threads and neither of them is an ancestor of the other. In the second case, one of the threads is an ancestor of the other.

In the first case, the regions may only happen in parallel if the lowest common ancestor is able to fork both threads in the same program execution. This is the case if the lowest common

ancestor is a multi-thread, which is covered by rule 1 of relation $\|_{thread}$, or if one of the fork site contexts in the lowest common ancestor that indirectly fork the two threads is able to reach the other one in the ICFG of the lowest common ancestor, which is covered by rule 2 of relation $\|_{thread}$.

In the second case, the thread θ of one of the regions, say R, is the ancestor of the other thread. Here, the nodes may only happen in parallel if R's thread is a multi-thread, which is covered by rule 1 of relation $\|_{thread}$, or if the fork site context in θ which indirectly forks Q's thread is able to reach the start node of R in the ICFG of θ.

This leads to the following redefinition of relation $\|_{region}$:

Definition 3.7 (MHP relation $\|_{region}$ (redefined))**.** *Let G be a TCFG, let R_0 and R_1 be two thread regions in G, and let θ_0 be R_0's thread and θ_1 be R_1's thread.*

R_0 and R_1 may happen in parallel, written $R_0 \|_{region} R_1$, if

1. *$\theta_0 = \theta_1$ and θ_0 is a multi-thread, or*

2. *$\theta_0 \neq \theta_1$, neither start node of one of the two regions is dominated by a must-join of the thread of the other region, and*

   - *$\theta_0 \|_{thread} \theta_1$, or*
   - *$\exists i \in \{0,1\}$ such that θ_i is an ancestor of θ_{1-i} and there exists a fork site context f in θ_i that indirectly forks θ_{1-i} and reaches the start node of R_i in the ICFG of θ_i via a non-trivial path.*

Our MHP algorithm

The computation of MHP information on the level of threads and the thread region analysis can be intertwined to an efficient algorithm based on context-sensitive forward traversal of ICFGs. For that purpose, the context-restricted slicer in Alg. 2.3 is adapted to work on ICFGs and to traverse forward: Phase 1 traverses return and control flow edges, phase 2 traverses call and control flow edges. The depicted algorithm collects the MHP information in four steps. The first step handles multi-threads and threads forked within a multi-thread. The second step determines threads that may happen in parallel due to being forked subsequently by the same ancestor. This is done by implementing the second rule of definition 3.6. The third step accounts for thread regions that lie sequentially behind a fork site. They may happen in parallel to all thread regions of the forked thread and its descendant threads. The fourth step refines the hitherto computed result by analyzing must-joins. All thread regions whose start nodes are dominated by a must-join of a thread θ cannot happen in parallel to the thread regions of θ.

Algorithm 3.1 Computation of MHP information.

Input: A TCFG G, the set T of thread contexts from the thread invocation analysis.
Output: A map $M_{regs} : \mathit{ThreadRegions} \times \mathit{ThreadRegions} \mapsto \{true, false\}$,
and a map $M_{threads} : \mathit{Threads} \times \mathit{Threads} \mapsto \{true, false\}$

Compute the set R of thread regions.
Initialize maps M and T by setting all values to `false`.
Let $\mathit{regions}(\theta)$ be the set of thread regions in thread θ.
Let $\theta(c)$ be the thread of context c.
Let $\mathit{Desc}(\theta)$ be the set of all threads of which θ is an ancestor.

/ Step 1: handle multi-threads */*
for all multi-threads θ
  // collect all descendant threads
  for all $(\theta, \theta') \in (\mathit{Desc}(\theta) \cup \{\theta\}) \times (\mathit{Desc}(\theta) \cup \{\theta\})$ *// all these threads may happen in parallel*
    $M_{threads}(\theta, \theta') = true$
    $M_{threads}(\theta', \theta) = true$

/ Step 2: determine the threads that are forked subsequently by the same ancestor */*
Compute the set F of all fork site contexts (can be easily derived from set T).
Let θ_f be the thread forked by fork site context f.
for all $(f, f') \in F \times F : f \neq f' \wedge \theta(f) == \theta(f')$ *// all pairs of fork site contexts in the same thread*
  if f reaches f' in the ICFG of $\theta(f)$ *// use Alg. 3.6*
    // the threads and their descendant threads may happen in parallel
    for all $(\theta, \theta') \in (\mathit{Desc}(\theta_f) \cup \{\theta_f\}) \times (\mathit{Desc}(\theta_{f'}) \cup \{\theta_{f'}\})$
      $M_{threads}(\theta, \theta') = true$
      $M_{threads}(\theta', \theta) = true$

/ Step 3: all threads forked by a fork site f may happen in parallel to the thread regions behind f */*
for all $f \in F$
  Compute the set S of nodes reachable by f in the ICFG of $\theta(f)$ *// adapt Alg. 2.3 for that task*
  // the regions in θ and in descendant threads may happen in parallel to the regions lying in S
  for all $\theta \in (\mathit{Desc}(\theta_f) \cup \{\theta_f\})$
    for all $q \in \mathit{regions}(\theta)$
      for all thread regions r lying in S
        $M_{regs}(q, r) = true$
        $M_{regs}(r, q) = true$

/ Step 4: refine the result with must-join information */*
Let J be the set of all join nodes in G that are identified as must-joins
for all $j \in J$
  Let θ be the thread joined by j
  for all $r \in R$: j dominates r's start node
    for all thread regions q of θ
      $M_{regs}(q, r) = false$
      $M_{regs}(r, q) = false$

return M

Implementation and runtime complexity The start nodes of the thread regions are identified through a technique that iteratively colors the nodes in the TCFG in order to find the thread regions starting at merge nodes. In each iteration, each node reachable in its ICFG by an already determined start node of a thread region on a context-sensitive path containing no other start nodes and no nodes already labeled with another start node is labeled with that start node. These paths can be determined via a forward traversal analogically to two-phase slicing, because the control flow edges between the call and return nodes of the call sites make up for the summary edges. The traversal can be confined to the ICFG because thread regions are thread-local. If the traversal arrives at a node already labeled with another start node, a merge node starting a new thread region is found. This proceeding is iterated until no new merge node can be found; the coloring resulting from the last iteration represents the thread regions. The technique has an asymptotic running time of $\mathcal{O}(|N|^2 * |E|)$: One iteration has a runtime complexity of $\mathcal{O}(|N| * |E|)$, because for each already determined start node at most all edges have to be traversed once, and in the worst case $|N|$ iterations are necessary. In comparison, algorithms computing the interprocedural dominance relation [143], which would be necessary for determining the thread regions directly via dominance, have a runtime complexity of $\mathcal{O}(|N|^3 * |E|)$. The price paid for this speedup is that the result of our technique is not completely context-sensitive, because bypassing a procedure call via the mentioned control flow edge might miss a start node of a thread region in the bypassed procedure. In effect, the TCFG is partitioned into more thread regions than necessary.

The nodes dominated by the must-joins are determined in a similar way. For each must-join, these are all nodes not reachable by the start node of the TCFG via context-sensitive paths not containing the join node. Thus, for each must-join the edges of the TCFG have to be traversed at most once, leading to a runtime complexity of $\mathcal{O}(|N| * |E|)$. However, the number of must-joins is expected to be very small in practice. Similar to the identification of the thread regions, this proceeding is not completely context-sensitive and may omit nodes that are in fact dominated by a must-join. This means that the result of our implementation is only a conservative approximation of relation $\|_{region}$.

The handling of multi-threads in step 1 requires at most $\mathcal{O}(|T|^2)$ accesses to map $M_{threads}$, where $|T|$ is the number of thread contexts. In step 2, the analysis of whether a fork site context reaches another one in an ICFG requires the traversal of at most all of its edges (a suitable algorithm, Alg. 3.6, is described later in section 3.9.1). Furthermore, for each pair of fork site contexts, $\mathcal{O}(|T|^2)$ accesses to map $M_{threads}$ may be necessary. Thus, step 2 has a complexity of $\mathcal{O}(|T|^2 * |E| + |T|^4)$. Step 3 may require for each fork site context a complete traversal of the according ICFG and $\mathcal{O}(|T| * |R|^2)$ accesses to map M_{regs}, where $|R|$ is the number of thread regions, leading to an overall complexity of $\mathcal{O}(|T| * |E| + |T|^2 * |R|^2)$. Step 4 may require $\mathcal{O}(|N| * |R|^2)$ accesses to map M_{regs}.

This leads to a runtime complexity of the MHP analysis of $\mathcal{O}(|N|^2 * |E| + |T|^2 * |E| + |T|^4 + |T|^2 * |R|^2 + |R|^2 * |N|)$. Since $|N| > |T|$, $|N| > |R|$, $|R| \geq |T|$ and $|E| \geq |N| - 1$, this notation can be simplified to $\mathcal{O}(|N|^2 * |E| + |T|^2 * |R|^2)$. Thus, the costs of the analysis are dominated by the identification of the thread regions and the accesses to map M_{regs} in step 3. In our evaluation presented in section 3.12.1, the number of distinguished threads was always a single-digit, whereas the number of thread regions often exceeded 1,000, and the most time-consuming part was the computation of the thread regions.

3.4. The Concurrent System Dependence Graph

This section presents the *concurrent system dependence graph* (CSDG), our extension of the SDG for modeling concurrent programs, developed by Christian Hammer [52], Jürgen Graf and the author. Similar to TCFGs, a CSDG is composed of the SDGs of the thread classes of the program, which are connected via different kinds of concurrency-related dependences [52, chapt. 3]. To keep matters simple, we assume in the forthcoming descriptions that the SDGs are disjoint, as we did for TCFGs. Several authors provide extended SDGs for concurrent programs, more or less similar to the CSDG [61, 75, 106, 163, 164]. Differences and similarities are summarized in section 3.14.

The most important concurrency-related dependence is interference dependence [73].

Definition 3.8 (Interference dependence)**.** *A node n is interference dependent on node m, abbreviated by $m \rightarrow_{id} n$, if n may use a value computed at m and m and n may happen in parallel.*

Note that in contrast to data dependence interference dependence ignores the issue of reaching definitions. It does not require the existence of a possible interleaving under which the value computed at *m* in fact reaches *n*. As a consequence, the computation of interference dependence is separated from the computation of data dependence, which remains thread-local. According to Hammer [52, chapt. 3], this treatment safely approximates the effects of the Java memory model.

Hammer [52, chapt. 3] followed the work of Zhao et al. [164] to model forking of threads similarly to procedure calls via *fork sites*, where shared variables are passed as parameters. *Fork* and *fork-in edges*, abbreviated by $\rightarrow_{fork}$ and $\rightarrow_{fi}$, are defined in analogy to call and parameter-in edges. Thread joining is modeled similarly via *join sites*, where the final states of the shared variables modified in the joined thread are passed to the join site via *join-out edges*, abbreviated by $\rightarrow_{jo}$.

Example Figure 3.8 shows the CSDG of a program consisting of two threads that communicate via shared variables `x` and `y`. The program finally prints variable `x`, whose value depends on the interleaving of statement `x = <input>` in thread 1 and `x = y + 1` in thread 2. The

```
int x, y;

void main():
  fork thread_1();
  fork thread_2();
  join thread_2();
  join thread_1();
  print x;
```

```
void thread_1():
  int a = y;
  x = <input>;
  if a > 0
    int b = 0;
  else
    y = 0;
```

```
void thread_2():
  x = y + 1;
  y = 0;
```

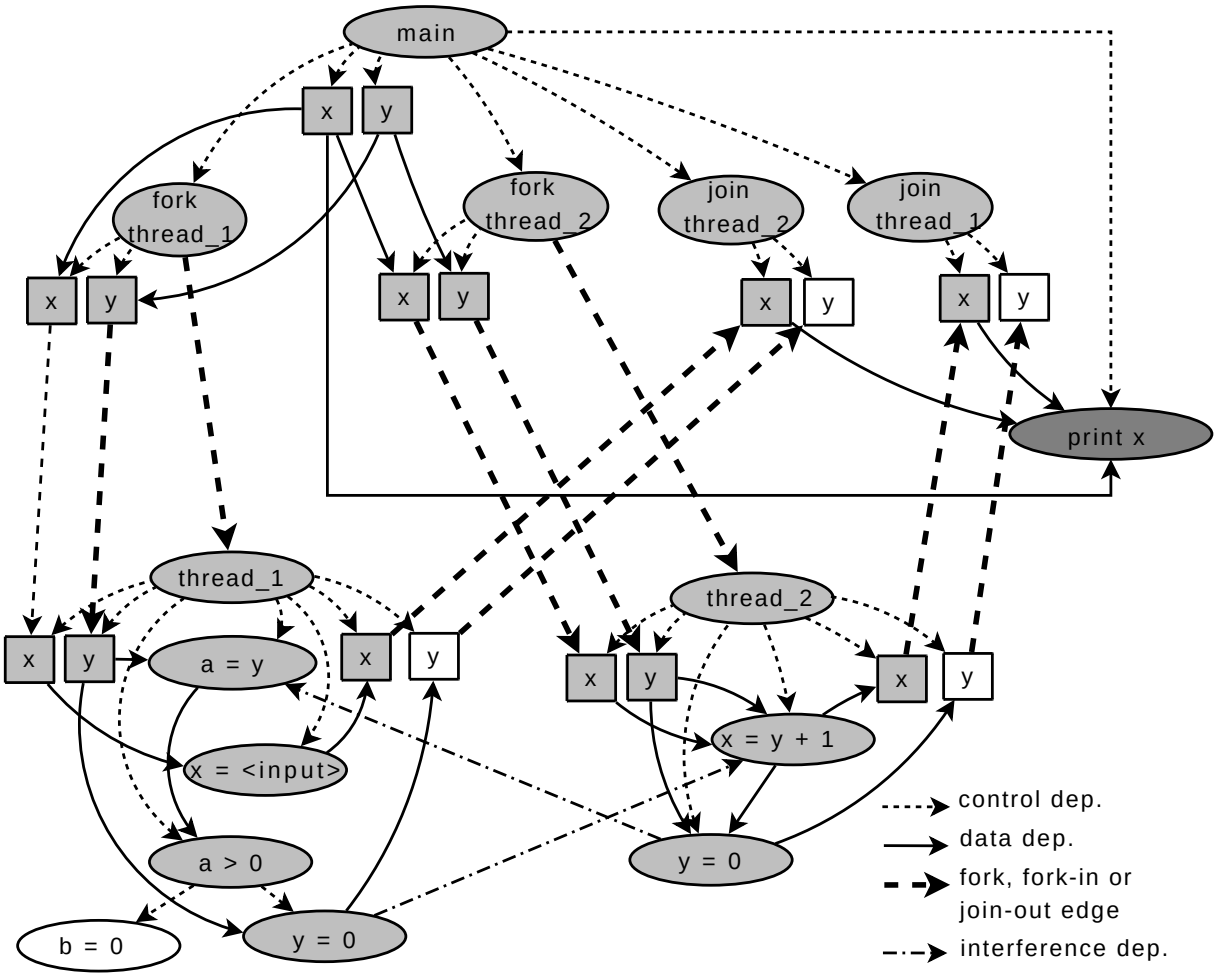


Figure 3.8.: A concurrent program and its CSDG. The highlighted nodes are the slice for `print x` computed by collecting all reaching nodes.

gray highlighted nodes are the slice for `print x` computed by collecting all reaching nodes. Node `print x` is not interference dependent on `x = <input>` and `x = y + 1` because it cannot happen in parallel to them. The CSDG accounts for the dependence of `print x` on these statements by passing the final states of `x` in the threads to their join sites. Node `print x` is data dependent on both actual-out nodes for `x`, since it may use either of these values, depending on the interleaving of the threads. In order to yield both data dependences, the *kill*-sets of join nodes are defined to be empty. Note that this leads to an additional data dependence

```
void thread_1():
  lock l;
  x = x + 1;
  unlock l;
```

```
void thread_2():
  lock l;
  x = x - 1;
  unlock l;
```

Figure 3.9.: Two threads synchronizing their access to shared variable `x`.

from the formal-in node for `x` at the beginning of `main` to `print x` because the analysis of data dependence remains thread-local and the definition is not killed by the join nodes.

Please note that for the sake of simplicity many subsequent example programs and CSDGs consist only of two threads and lack a common `main`-procedure.

Synchronization-related dependences

The presence of synchronization complicates the detection of interference dependences. Consider the program fragment in Figure 3.9. Both threads read and write shared variable `x`, so intuitively there should be interference dependences between `x = x + 1` and `x = x - 1` in both directions. However, these two statements cannot happen in parallel due to synchronization. Instead of introducing a cumbersome parameter-passing mechanism at the beginning and at the end of synchronized blocks, we refine the definition of interference dependence to exclude the effects of synchronization. Note that our MHP analysis delivers exactly the required MHP information.

Definition 3.9 (Interference dependence (redefinition))**.** *A statement n is interference-dependent on statement m, abbreviated by* $m \rightarrow_{id} n$*, if n may use a value computed at m and m and n may happen in parallel up to synchronization.*

Synchronization may also cause additional dependences. Hatcliff et al. [61] investigated dependences induced by synchronization in Java programs and introduced *synchronization-* and *ready dependence*. Synchronization dependence embeds a node *n* into its innermost enclosing synchronization block. The node is defined to be synchronization dependent on the lock and unlock operations of that block. A node *n* is ready dependent on a node *m* if *m* may delay *n*'s execution infinitely by blocking *n*'s thread via synchronization operations before it reaches *n* or completes its execution.

We do not use these dependences for the following reasons: Hatcliff et al. aim for executable slices and thus have to preserve the semantics of synchronization operations and of program termination. For that purpose, a node *n* has to be synchronization dependent on the unlock operation of the innermost enclosing synchronization block. However, this contravenes our concept of slicing, because the lock release does not influence *n* (it executes *behind n*). Ready dependence is defined in the spirit of weak control dependence and captures possible nontermi-

```
void thread_1():
  lock l;
  wait l;
  print 0;
  unlock l;
```

```
void thread_2():
  lock l;
  unlock l;
  notify l;
```

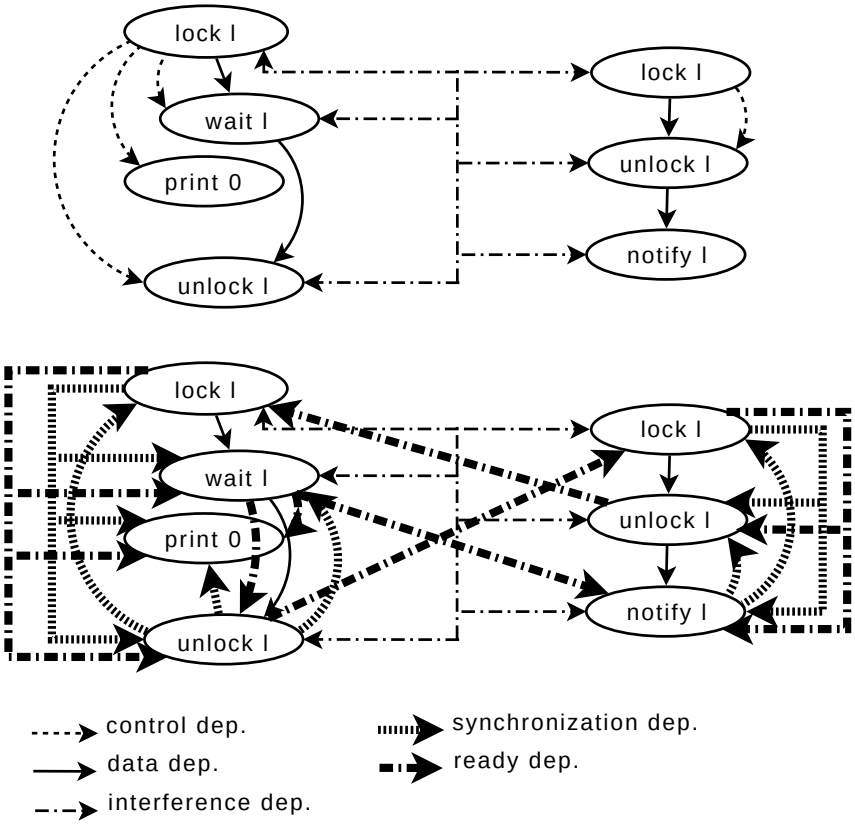


Figure 3.10.: Two concurrent threads and two possible ways of modeling the dependences induced by synchronization. The upper CSDG fragment uses interference- and control dependences to model these dependences, the lower one uses synchronization- and ready dependences.

nation induced by synchronization. Since we do not aim for termination-sensitive slices, we do without ready dependence, too. Instead, we use Nanda's [106] idea to cover the dependences induced by synchronization via control- and interference dependence instead. This works for Java, because Java's locking, waiting and notifying operations reference and modify objects and thus create appropriate interference dependences. Only a single extension is needed: Synthetic control flow edges are inserted from a lock operation to the successor of its associated unlock operation. These synthetic edges render nodes (transitively) control dependent on the lock operation of the innermost enclosing synchronization block.

Example Figure 3.10 shows a program fragment consisting of two threads that synchronize on a shared object `l`. The upper CSDG fragment depicts our modeling of the program, the lower one uses synchronization- and ready dependence. For clarity, the start nodes of the threads and the parameter passing are omitted. Let us explain the depicted dependences. In the upper graph, every node part of a synchronization block is control dependent on the lock operation, due to our synthetic control flow edges. In the lower graph, synchronization dependence accounts for these dependences, but also leads to dependences from the unlock operations to all nodes in the synchronization blocks. Since monitors in Java are associated with objects, each access to a monitor in the program fragment is interference dependent on the three synchronization operations in the other thread. The ready dependences between the locks and unlocks and between `wait l` and `notify l` are safely approximated by these interference dependences. The ready dependences between `wait l` and its thread-local successors are lacking in the upper graph, since we do not aim for termination-sensitive CSDGs.

3.4.1. Computation of CSDGs

For a detailed description of how CSDGs are computed by the Joana framework we refer to Hammer's PhD thesis [52]. In summary, a CSDG is constructed as follows: First, a standard SDG comprising the whole program is computed, treating forking and joining of threads as ordinary procedure calls. Then, a preliminary set of interference dependences is computed. For that purpose, the program is assumed to have two threads of each thread class at runtime, except for the main thread, which is unique. Additionally, all threads of the program are assumed to happen in parallel. These assumptions make sure that all possible interference dependences between threads of different thread classes and between threads of the same thread class are included. The points-to information computed in the first step is used to identify pairs of statements in different threads that read and write the same heap locations, which yields the interference dependences. In a postprocessing step, our MHP analysis is used to identify and remove redundant interference edges. Finally, the call- and parameter edges at the fork- and join sites are converted to fork-, fork-in and join-out edges.

3.5. Context-Sensitive Slicing of Concurrent Programs

The two-phase slicer for sequential programs cannot be used to slice CSDGs, because summary edges do not capture interprocedural effects of interference dependences [106]. Since interference dependences cross procedure borders arbitrarily and violate the well-formedness property of SDGs of propagating all interprocedural effects through call sites, their effects cannot be summarized by conventional summary edges. Figure 3.11, taken from Hammer's PhD thesis [52], shows a minimalist producer-consumer-style example with an interference dependence

```
int x;

void thread_1():
  int a = 2;
  set(a);

void thread_2():
  int b = get();
  print b;

void set(int a):
  x = a;

int get():
  return x;
```

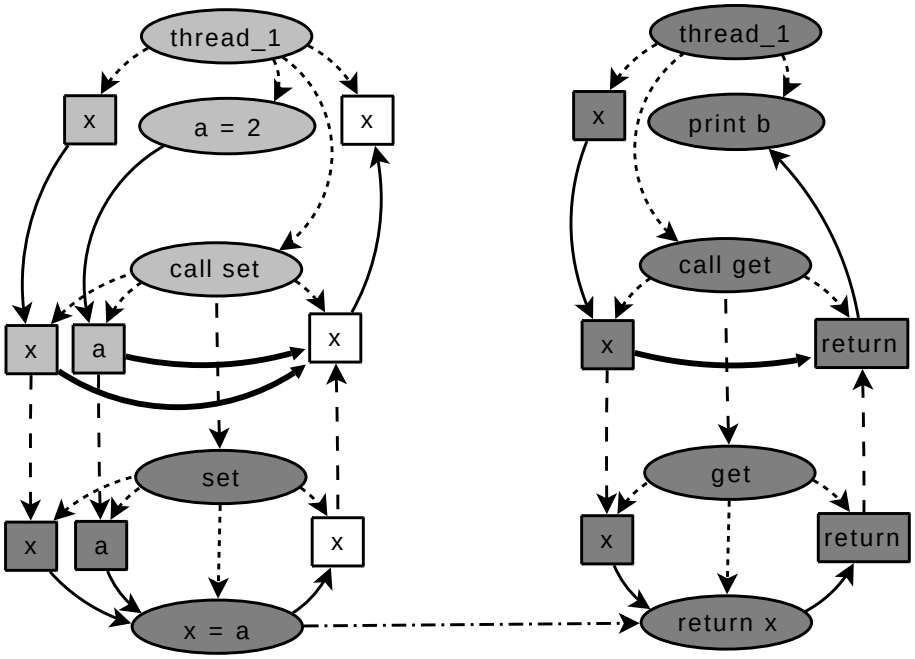


Figure 3.11.: A small producer-consumer program and its CSDG. The two-phase slice for `print b` is highlighted dark gray. It omits the light gray nodes, which belong to a correct slice.

between the producer and the consumer. The shaded nodes highlight the statement-minimal slice for `print b`, the darker nodes mark the slice computed by the two-phase slicer. The slicer visits `thread_1` only in the second phase by traversing the interference edge, which means that the slicer cannot leave procedure `set` towards the main procedure of `thread_1`. Hence, the call of of `set` will not be visited, even though it belongs to the slice.

3.5.1. Context-Sensitive Paths in CSDGs

In order to slice CSDGs context-sensitively, we have to define what context-sensitivity means in the presence of threads. To this end, we extend the definition of context-sensitive paths in SDGs with interference dependence, fork-, fork-in- and join-out edges, summarized as *concurrency edges* in the remainder. Intuitively, if a path traverses a concurrency edge $m \to n$ towards n, the calling context of m is lost: The thread that has been left is allowed to execute further in parallel, so if the path reenters that thread later, one cannot demand that it reenters the thread at the original calling context. Furthermore, the traversal may reach n in any possible calling context of n, because m interferes with every possible instance of n. Thus, a path p in a CSDG is context-sensitive if it consists of a sequence $p_1, \ldots, p_n$ of sequential, context-sensitive paths, where each pair (p_i, p_{i+1}), $0 \leq i < n$, is connected via a concurrency edge.

Definition 3.10 (Context-sensitive paths in CSDGs)**.** *In addition to definition 2.9, label concurrency edges with conc. A path in the CSDG of a concurrent program is context-sensitive, iff the sequence of symbols labeling edges in the path is a word generated from nonterminal conc_realizable by grammar H_{conc}, which extends grammar H of definition 2.3 as follows:*

$matched \to matched\ matched \mid (^s_c\ matched\ )^s_c \mid l \mid \varepsilon$
$unbalanced\_right \to unbalanced\_right\ )^s_c\ matched \mid matched$
$unbalanced\_left \to unbalanced\_left\ (^s_c\ matched \mid matched$
$realizable \to unbalanced\_right\quad unbalanced\_left$
$conc\_realizable \to (realizable\ conc)^*\ realizable$

Since that definition includes its counterpart for sequential programs, we reuse the notation $\to^*_{cs}$ for context-sensitive paths in CSDGs. The definition of context-sensitive slices of CSDGs is a generalization of the one for SDGs:

Definition 3.11 (Context-sensitive slices)**.** *Let $G = (N, E)$ be a CSDG.*

A context-sensitive backward slice of G for a node $s \in N$ consists of the set of nodes

$$\{n \mid \exists\ n \to^*_{cs} s\ in\ G\}.$$

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

$$\{n \mid \exists\ s \to^*_{cs} n\ in\ G\}.$$

3.5.2. The Iterated Two-Phase Slicer

The following extension of the two-phase slicer enables context-sensitive slicing of CSDGs: The two-phase slicer is embedded in a loop that iterates over a list W of nodes and calls the

```
int x, y;

void main():
  x = 0;
  y = 1;
  fork thread_1();
  int p = x - 2;
  int q = p + 1;
  y = q * 3;

void thread_1():
  int a = y + 1;
  int b = a * 4;
  x = b / 2;
```

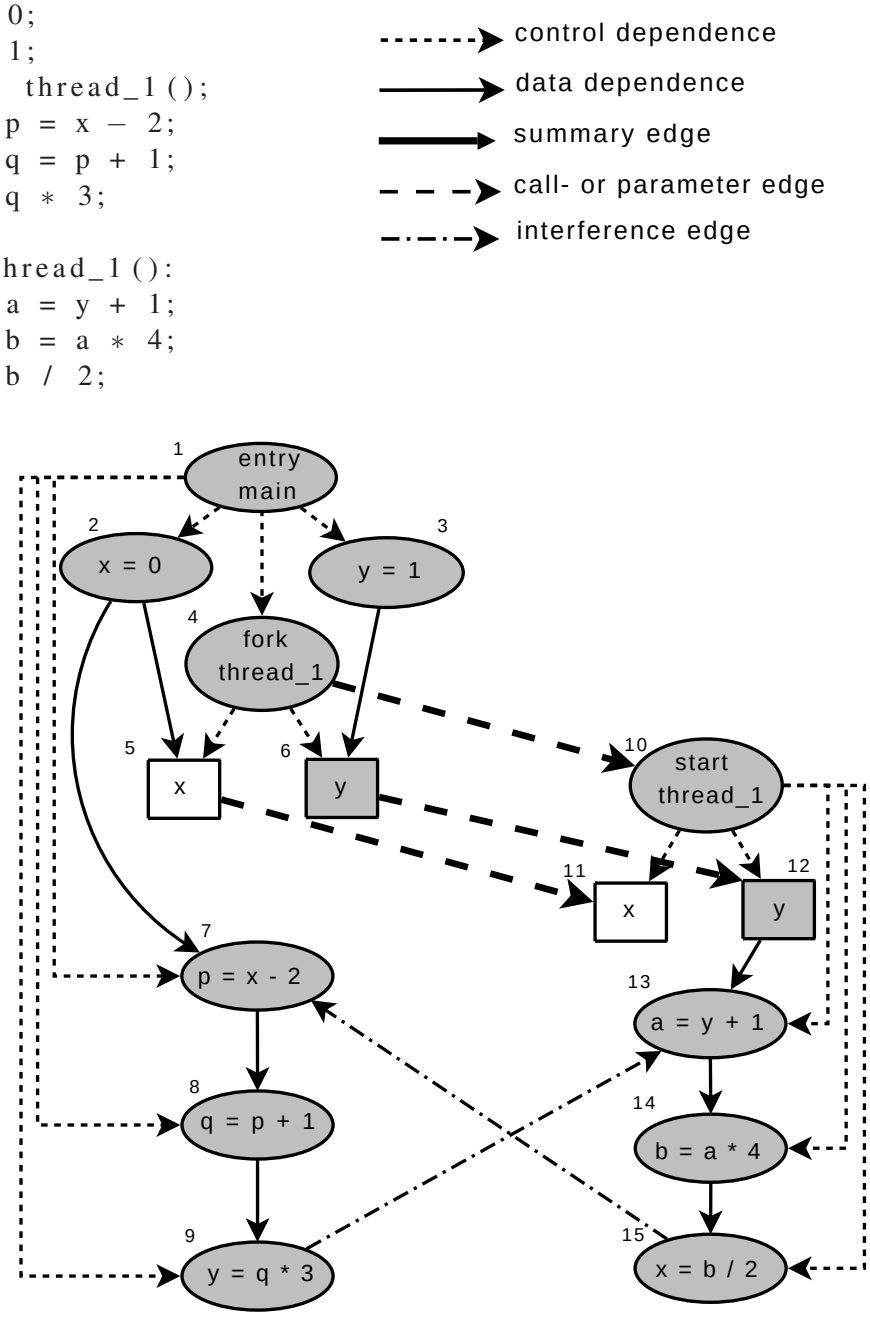


Figure 3.12.: A CSDG of a concurrent program. In order to keep the graph as simple as possible, the formal-in nodes of `main` and the formal-out nodes of `thread_1` for `x` and `y` are not shown. The highlighted nodes form the context-sensitive slice for node 14.

two-phase slicer for every $n \in W$. Initially, W contains only the slicing criterion. If the two-phase slicer encounters a concurrency edge, it does not traverse the edge but inserts the adjacent node into W. The resulting slice consists of the nodes visited in all iterations of the two-phase

Algorithm 3.2 I2P: Hammer's [52] version of Nanda's iterated two-phase slicer [106].

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

```
W = {s}  // a worklist
M = {s ↦ true}  // maps visited nodes to true (phase 1) or false (phase 2)

repeat
  W = W \ {n}  // process the next node in W
  for all m →_e n  // handle all incoming edges of n
    // if m hasn't been visited yet or we are in phase 1 and m has been visited only in phase 2
    if m ∉ dom M ∨ (¬M(m) ∧ (M(n) ∨ e = conc))
      // if we are in phase 1 or if e is not a call or param-in edge, add m to W
      if M(n) ∨ e ∉ {pi, c}
        W = W ∪ {m}
        /* determine how to mark m */
        if M(n) ∧ e = po
          // we are in phase 1 and e is a param-out edge: mark m with phase 2
          M = M ∪ {m ↦ false}
        else if ¬M(n) ∧ e = conc
          // we are in phase 2 and e is a concurrency edge: mark m with phase 1
          M = M ∪ {m ↦ true}
        else
          // mark m with the same phase as n
          M = M ∪ {m ↦ M(n)}
until W = ∅
return dom M
```

slicer. This *iterated two-phase* (I2P) slicer was first described by Nanda [106] and can be implemented to yield context-sensitive slices in $\mathcal{O}(|E|)$: A node already been visited in phase 1 during a previous two-phase slice has not to be visited again, because its own slice has already been covered by that two-phase slice. This is not the case for a node which has only been visited in phase 2 yet, because phase 2 omits parameter-in and call edges. It has to be visited again if reached in phase 1 of another two-phase slice or via a concurrency edge. This means that each edge has to be traversed at most twice, once in phase 2 and later again in phase 1. As an example, let us compute the slice for node 14 in the CSDG in Fig. 3.12. The algorithm first computes a thread-local slice for node 14 using the two-phase slicer and visits the nodes {14, 13, 12, 10}. Nodes 9, 6 and 4 are not visited but added to list W. The slicer now subsequently computes thread-local slices for these nodes and updates W as needed. The two-phase slice for node 9 visits the nodes {9, 8, 7, 2, 1} and inserts node 15 into W. The two-phase slice for node 6 visits nodes {6, 4, 3} (node 1 has already been visited in phase 1), the one for node 4 can be omitted, because node 4 has already been visited in phase 1. The last two-phase slice for node 15 visits node 15. The resulting slice, highlighted in Fig. 3.12, consists of all visited nodes.

Figure 3.2 shows a more compact implementation developed by Hammer [52]. It is based on a single map, which maps the visited nodes to the phase in which they have been visited. The

condition of the first '`if`' inside the `forall`-loop checks if the adjacent node *m* has to be visited. This is the case if *m* has not been visited yet, or if it has been visited only in phase 2 and we are currently in phase 1 or intend to traverse a concurrency edge. The next conditional realizes the two-phase slicing technique: If the intended traversal happens in phase 2, parameter-in and call edges have to be ignored. The last conditionals decide how to mark *m* in the map, according to two-phase slicing. If a concurrency edge has been traversed, the reached node is always treated as being visited in phase 1.

3.6. Timing-Sensitive Slicing

The remainder of this chapter investigates the timing-sensitive slicing algorithms of Krinke [73, 75, 77] and of Nanda [105, 104, 106]. Unless otherwise noted, our descriptions of Krinke's work refer to [75], and our descriptions of Nanda's work refer to [106].

3.6.1. Timing-Sensitive Paths in CSDGs and TCFGs

A slice is timing-sensitive if it contains only those nodes that lie on a *timing-sensitive path* to the slicing criterion. Krinke developed the first definition of timing-sensitive paths, which is based on reachability between contexts. For that purpose, contexts in a CSDG have to be mapped to the corresponding contexts in the TCFG, which is done by mapping the node of the context to its counterpart in the TCFG via the following extension of function *map* (definition 2.8) for fork and join sites. In the remainder, we assume that this mapping is done implicitly whenever the *reaches* relation is applied to contexts in a CSDG.

Definition 3.12. *Let $G = (N_{PDG} \cup N_{syn}, \_)$ be a SDG, and let $ICFG = (N_{ICFG}, \_)$ be the corresponding ICFG. Function $map : N_{PDG} \cup N_{syn} \mapsto N_{ICFG}$ is defined as follows:*

$$map(n) = \begin{cases} n & n \in N_{PDG} \\ c & n \text{ is an actual-in node and } c \text{ is the corresponding call or fork node} \\ r & n \text{ is an actual-out node and } r \text{ is the corresponding return or join node} \\ s & n \text{ is a formal-in node of procedure } p \text{ and } s \text{ is } p\text{'s start node} \\ e & n \text{ is a formal-out node of procedure } p \text{ and } e \text{ is } p\text{'s exit node} \end{cases}$$

The MHP relation $\|$ for contexts is a context-sensitive variant of definition 3.3.

Definition 3.13 (Relation $\|$ for contexts)**.** *Let $(n,\sigma),(m,\sigma')$ be two contexts in a CSDG G, and let $\theta((n,\sigma))$ and $\theta((m,\sigma'))$ be their threads. Then (n,σ) and (m,σ') may happen in parallel, written $(n,\sigma) \parallel (m,\sigma')$, if*

- *$\theta((n,\sigma)) = \theta((m,\sigma'))$ and $\theta((n,\sigma))$ is a multi-thread, or*

- $\theta((n,\sigma)) \neq \theta((m,\sigma'))$, *neither n nor m is dominated by a must-join of the thread of the other context, and*
  - *there exists a fork site context f in G that indirectly forks the thread of the one context via an outgoing fork edge and reaches the other context via a different outgoing edge.*

There exists a *context edge* between two contexts if their nodes are connected via an edge in the CSDG and its traversal is in accordance with the call strings of the contexts. Context edges are defined in analogy to Krinke's *directly reaches* relation presented in section 3.2.1.

Definition 3.14 (Context edge)**.** *Let $(m,\sigma_m),(n,\sigma_n)$ be two contexts in a CSDG G. We say that G contains a context edge from c to d, denoted by $(m,\sigma_m) \to (n,\sigma_n)$, iff one of the following cases holds:*

1. *There exists a control or data dependence $m \to n$ in G and $\sigma_m = \sigma_n$.*
2. *There exist a call or parameter-in edge $m \to n$ in G such that s symbolizes the call site and*
   - *$\sigma_m = \sigma$ and $\sigma_n = \sigma s$, or*
   - *$\sigma_m = \sigma_n = \sigma s$ and s is a synthetic call site.*
3. *There exist a parameter-out edge $m \to n$ in G such that s symbolizes the call site and*
   - *$\sigma_m = \sigma s$ and $\sigma_n = \sigma$, or*
   - *$\sigma_m = \sigma_n = \sigma s$ and s is a synthetic call site.*
4. *There exist a fork or fork-in edge $m \to n$ in G such that s symbolizes the call site and*
   - *$\sigma_m = \sigma$ and $\sigma_n = \sigma s$, or*
   - *$\text{thread}((m,\sigma_m)) = \sigma_n$ and s is a synthetic call site.*
5. *There exist a join-out edge $m \to n$ in G.*
6. *There exist an interference dependence $m \to n$ in G and $(m,\sigma_m) \parallel (n,\sigma_n)$ (interference edges should only be traversed if the contexts may happen in parallel).*

A *context path* describes a path through a CSDG that is in accordance with the call strings of the involved contexts.

Definition 3.15 (Context path)**.** *A sequence of contexts $\langle c_{n_1},\ldots,c_{n_k}\rangle$ in a CSDG G is a context path, denoted by $c_{n_1} \to^* c_{n_k}$, if for every consecutive pair $(c_{n_j},c_{n_{j+1}})$ in the sequence, G contains a context edge from n_j to n_{j+1}.*

It follows directly from that definition that two context paths, $c \rightarrow^* d$ and $d \rightarrow^* c'$, can be connected to a new context path, $c \rightarrow^* d \rightarrow^* c'$. It is also easy to see that the nodes of the contexts in a context path form a context-sensitive path.

It remains to define which context paths are timing-sensitive. We follow the work of Krinke and define which context paths correspond to a valid interleaving of the involved threads. MHP information and the *reaches* relation $\rightsquigarrow^*$ are used to identify whether a context path is timing-sensitive. This is the case if all contexts in the path that cannot happen in parallel can be executed by the program in the relative order in which they appear in the path, up to conditional branching.

Definition 3.16 (Timing-sensitive path)**.** *A context path $c_1 \rightarrow^* c_k$ is a timing-sensitive path, written $c_1 \rightarrow^*_{ts} c_k$, iff $\forall 1 \leq j < i \leq k : c_i \parallel c_j \vee c_i \rightsquigarrow^* c_j$.*

This condition is almost the same as used by Krinke in his definition of *threaded witnesses* [77]. It ensures that a path Φ describes a valid interleaving of the threads involved in Φ. The major difference between Krinke's definition of timing-sensitive paths and ours is that Krinke bases his directly on sequences of contexts and reachability between contexts, whereas ours is based on context paths and context edges as an additional, intermediate layer. We introduced them because we have to require explicitly that a context edge resulting from an interference dependence implies that the two contexts may happen in parallel. Krinke assumes that all threads happen in parallel, which implicitly fulfills that requirement.

Having defined timing-sensitive paths, it remains to define timing-sensitive slices:

Definition 3.17 (Timing-sensitive slices)**.** *Let $G = (N, E)$ be a CSDG.*

A timing-sensitive backward slice of G for a node $s \in N$ consists of the set of nodes

$$\{n \mid \exists c_n \rightarrow^*_{ts} c_s \text{ in } G\}.$$

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

$$\{n \mid \exists c_s \rightarrow^*_{ts} c_n \text{ in } G\}.$$

3.6.2. The Basic Idea of Timing-Sensitive Slicing

It follows an explanation of the basic idea of timing-sensitive slicing. To keep matters simple, we assume that all threads of a program may happen in parallel. The inclusion of more precise MHP information is discussed in section 3.7.

In order to compute timing-sensitive slices, one needs to avoid timing-insensitive paths. A very important observation has been made by Krinke in his *prepending property*, which we redefine in terms of our context paths.

Theorem 3.1 (Prepending property)**.** *Let $\Phi = c_1 \rightarrow^*_{ts} c_k$ be a timing-sensitive path in a CSDG G. Let $e = c_0 \rightarrow c_1$ be a context edge in G. Path $c_0 \rightarrow c_1 \rightarrow^*_{ts} c_k$ is timing-sensitive iff*

- *e is thread-local, or*
- *c_0 reaches the first context c in Φ that cannot happen in parallel to c_0.*

Proof. It follows directly from definition 3.15 that $c_0 \rightarrow c_1 \rightarrow^*_{ts} c_k$ is a context path. It remains to show $\forall 1 \leq i \leq k$, $c_0 \nparallel c_i \Rightarrow c_0 \rightsquigarrow^* c_i$. For that purpose, it suffices to show that c_0 reaches the first context c in Φ that cannot happen in parallel to c_0. Since all threads may happen in parallel and Φ is timing-sensitive, we have $c_0 \nparallel c_i \Rightarrow c \nparallel c_i \wedge c \rightsquigarrow^* c_i$. From $c_0 \rightsquigarrow^* c$ follows $c_0 \rightsquigarrow^* c_i$ because $\rightsquigarrow^*$ is transitive. Thus, the second case is already shown. It remains the case of thread-local edges.

If e is thread-local, then the first context c in Φ that cannot happen in parallel to c_0 is c_1, and e is either an intra-procedural dependence, a call or parameter-in edge or a parameter-out edge. We rewrite $c_0 = (m, \sigma_m)$ and $c_1 = (n, \sigma_n)$.

- e is a data or control dependence
  According to definition 3.14, $\sigma_m = \sigma_n$. It follows from the definitions of data and control dependence that there exists a path of control flow edges from m to n. Thus, $c_0 \rightsquigarrow^* c_1$.
- e is a call or parameter-in edge or a parameter-out edge
  Since the context of a parameter-passing node is mapped to the context of the associated call-, return-, start- or exit node, c_0 directly reaches c_1 via a call edge or return edge.

□

The prepending property says that nothing needs to be done as long as the context-sensitive traversal of a CSDG or TCFG remains thread-local, and in case a concurrency edge is traversed a single reachability check is sufficient. This observation permits to develop slicing algorithms that remain on timing-sensitive paths throughout the traversal. The timing-sensitive slicing algorithms of Krinke and of Nanda work on the level of contexts and annotate every visited context c with a *state tuple* Γ, which maps each thread to the lastly visited context of that thread on the path taken so far.

Definition 3.18 (Configuration)**.** *Let c_n be a context and Γ be a state tuple. The tuple (c_n, Γ) is a configuration of node n.*

Configurations store the information necessary to check the prepending property and thus allow to detect whether the backward traversal of a concurrency edge $c \rightarrow d$ results in a timing-insensitive path: If c belongs to a thread θ and c_{old} is the state of θ in the state tuple with which d is annotated, then it is compulsory that c may reach c_{old} in the TCFG. Otherwise, the traversal

would result in a timing-insensitive path and is rejected. In our example in Figure 3.12, this situation arises when the slicer traverses from node 7 to node 15 (in that example each node has exactly one context, which is simply represented by the node itself). Thread `thread_1` has previously been left at node 13 via the incoming interference edge. Therefore, the slicer needs to check whether it is possible to reach node 13 from node 15 in the TCFG. This is not the case, hence the traversal should be rejected.

The basic algorithm

Algorithm 3.3 gives an overview of the algorithmic details of timing-sensitive slicing. It shows the basic structure of both Krinke's and Nanda's algorithms, which can be viewed as extensions of the iterated two-phase slicer: They iterate a context-sensitive slicing algorithm for sequential programs and determine which encountered concurrency edges can be traversed without creating a timing-insensitive path. For that purpose, each visited context is annotated with a state tuple that maps each thread to its lastly visited context. The iterated slicer for sequential programs works on configurations in order to determine at which configurations the current thread can be left and can be imagined as an extension of the IPDG slicer of section 2.5. It receives a configuration (c,Γ) and returns the thread-local slice for c and the set I of visited configurations of nodes with incoming interference dependences. In order to compute set I, it has to propagate and update state tuples during the slice: Following each backward traversal of an edge, the reached context c is annotated with a copy of the state tuple of the previous context, where the entry of c's thread is set to c. Similar to the iterated two-phase-slicer, the thread-local slicer is called iteratively for every configuration visited via a valid traversal of a interference dependence.

The depicted algorithm works as follows: Initially, it determines all possible contexts $C(s)$ of the given slicing criterion, node s. Then, it annotates each context $c \in C(s)$ with an initial state tuple Γ, where the execution state of c's thread is set to c and the states of the other threads are set to an initial, nonrestrictive state $\bot$: Every traversal of an interference dependence edge towards a thread in this initial state is valid by definition. These configurations are inserted into a worklist W. The algorithm iterates over worklist W and computes for each configuration $(c,\Gamma) \in W$ the thread-local slice $\overline{S}$ for c and the set I of visited configurations of nodes with incoming interference edges. Then, it determines the valid interference dependences: For each configuration $(i_n,\Gamma_{i_n}) \in I$ of a node n and each interference dependence $m \rightarrow_{id} n$, the set of *valid contexts* C_m of node m is determined. A context c_m of m is valid if c_m may reach the context stored as the state of c_m's thread in Γ_i. Every valid context $c_m \in C_m$ is annotated with an updated state tuple Γ_m, where the entry of c_m's thread is set to c_m and the other entries are set to the same values as in Γ_i, the resulting configuration is added to worklist W. The resulting slice is the union of all thread-local slices.

Algorithm 3.3 Timing-sensitive slicing of concurrent programs.

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

Let $C(n)$ be the set of all possible contexts of node n
Let $\theta(c)$ be the thread of context c
Let $\Gamma(\theta)$ be the context stored in state tuple Γ for thread θ
Let $[c/\theta]\Gamma$ return a new state tuple Γ' by mapping thread θ in state tuple Γ to context c
Let `SeqSlice`(c,Γ) return the thread-local slice $\overline{S}$ for context c and and the set I of visited configurations of nodes with incoming interference dependences

```
/* Initialize the slicer */
Γ_0 = (⊥,...,⊥)  // initially, every thread is in a nonrestrictive state
W = {(c,Γ_c) | c ∈ C(s) ∧ Γ_c = [c/θ(s)]Γ_0}
M = {} ∪ W  // mark the visited configurations

repeat
   W = W \ {(c,Γ_c)}  // process the next configuration

   /* Compute a thread-local slice S̄ for c and the set I of visited configurations of nodes with incoming
   interference dependences */
   (S̄,I) = SeqSlice(c,Γ_c)
   S = S ∪ S̄

   /* Compute which interference edges can be validly traversed */
   for all (i,Γ_i) ∈ I
      for all m →_id n : n is node of context i
         /* Compute the valid contexts of m */
         C_m = {c_m ∈ C(m) | Γ_i(θ(c_m)) == ⊥ ∨ c_m reaches Γ_i(θ(c_m))}

         /* Update worklist W */
         for all w ∈ {(c_m,Γ_m) | c_m ∈ C_m ∧ Γ_m = [c_m/θ(c_m)]Γ_i}
            if w ∉ M
               W = W ∪ {w}
               M = M ∪ {w}

until W = ∅
return S
```

3.6.3. Runtime Complexity

The asymptotic running time of timing-sensitive slicing is dominated by a possible combinatorial explosion in the state tuples, because a context can be visited repeatedly with different state tuples. Nanda determined a worst-case complexity of $\mathcal{O}(|N|^{p^t})$, where p is the calling depth of the call graph, $|N|^p$ is an upper bound for the number of contexts, and t is the number of entries in the state tuples.

3.6.4. Restrictive State Tuples

Nanda identified combinatorial explosion in the state tuples to be the major performance problem and introduced *restrictive state tuples* as a remedy.

Definition 3.19 (Restrictive state tuples)**.** *Let* $\Gamma = [c_1, \ldots, c_k]$ *and* $\Gamma' = [d_1, \ldots, d_k]$ *be two state tuples. State tuple* Γ *is restrictive to* Γ' *iff* $\forall 1 \leq i \leq k : c_i$ *reaches* d_i*.*

If c is a context and Γ and Γ' are state tuples such that Γ is restrictive to Γ', then a slice for configuration (c, Γ) is a subset of the slice for configuration (c, Γ'), because Γ imposes more restrictions on the set of valid interference dependences than Γ' does. This property allows to identify and eliminate redundant configurations. According to our evaluation, the usage of this optimization is mandatory for a practical employment of timing-sensitive slicing.

3.6.5. Thread Creation Inside Loops and Recursion

The state tuples in the described algorithm have to model every thread that may exist at runtime, which requires a way to cope with multi-threads. A simple solution is to give an upper bound for the number of threads created there. But often such an upper bound is not estimable, and the incorrectness resulting from an arbitrarily chosen number may be inadmissible. Our solution is based on the following observation: A context from a multi-thread may happen in parallel to all contexts of the same thread. Due to our current assumption that all threads may happen in parallel, this means that a context from a multi-thread may happen in parallel to all contexts of the program. Thus, a traversal towards a context in a multi-thread via an interference dependence always results in a timing-sensitive path. Our solution represents a multi-thread by a single entry in the state tuples and excludes it from the updating mechanism, whereby it remains in the initial state throughout the slicing process.

Note that the iterated two-phase slicer implicitly accounts for multi-threads, because it treats every encountered interference dependence as valid.

3.7. The Impact of MHP Information on Slicing

MHP information about a concurrent program affects the precision of slicing in two different ways. Obviously, more precise MHP information may prune more spurious interference dependences. Consider the example shown in Figure 3.13. The program consists of three threads communicating via shared variables `x` and `y`. The conservative assumption that all threads happen entirely in parallel would result in an interference dependence from node 20 to node 13. It can be pruned if the MHP information factors in that `thread_2` is forked only after node 13.

MHP information affects precision also in a second, more subtle way during the computation of a slice. The example in Figure 3.13 demonstrates that more precise MHP information allows

```
int x, y;

void main():
  x = 0;
  y = 1;
  fork thread_1();
  int p = x - 2;
  int q = p + 1;
  y = q * y;
```

```
void thread_1():
  int a = y + 1;
  fork thread_2();
  int b = a * 4;
  x = b / 2;

void thread_2():
  y = 0;
```

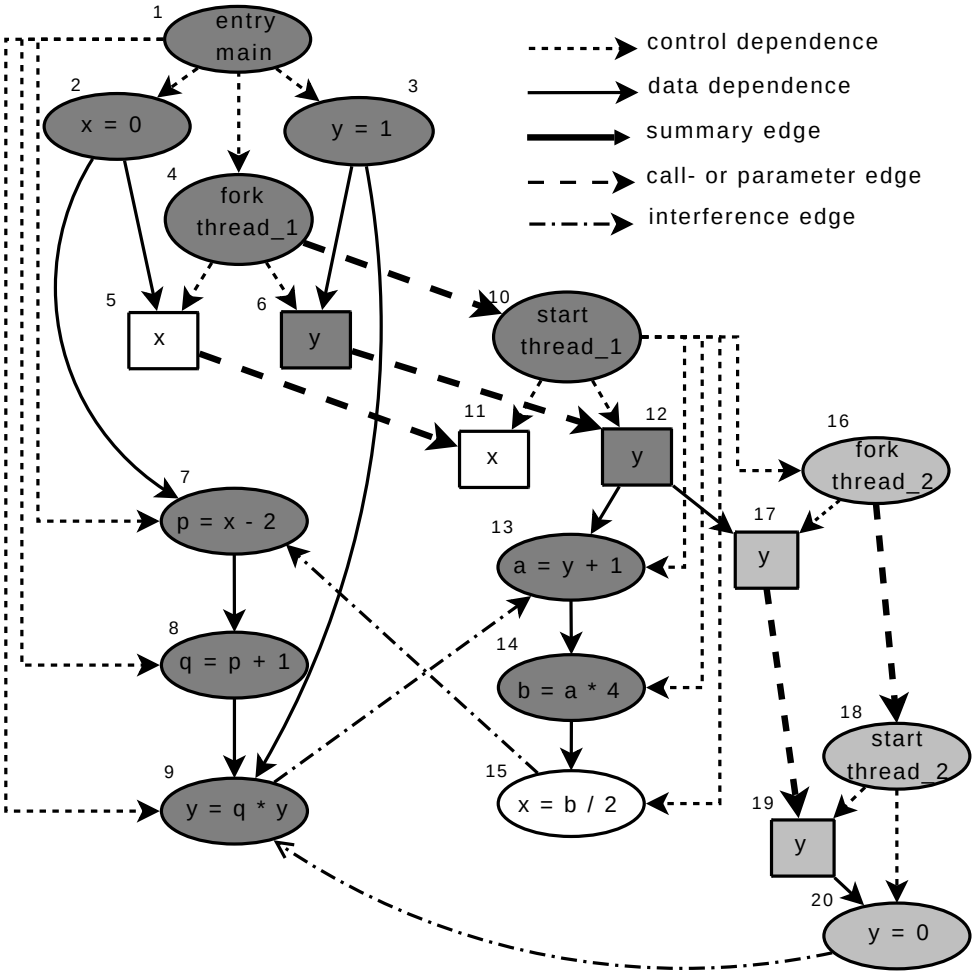


Figure 3.13.: More precise MHP information results in more precise slices: The gray nodes denote the timing-sensitive slice for node 14 in case all threads are deemed to happen in parallel. The dark gray nodes denote the slice if the fork sites of the threads are taken into account.

to detect more timing-insensitivity. The set of shaded nodes marks the timing-sensitive slice for node 14 in case all threads are deemed to happen entirely in parallel. The dark gray nodes mark

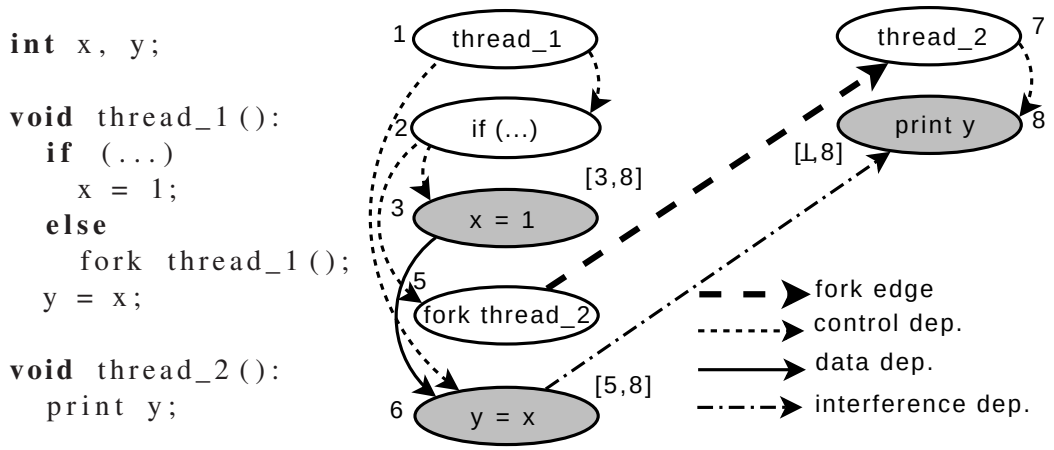


Figure 3.14.: The prepending property allows to traverse path $\Phi = 3 \rightarrow 6 \rightarrow 8$. In case the threads are assumed to happen in parallel, Φ is timing-sensitive. More precise MHP information revealing that node 3 and `thread_2` are exclusive identifies Φ as timing-insensitive.

the slice in case the fork sites of the threads are accounted for. In the latter case, the traversal of edge $20 \rightarrow_{id} 9$ towards node 20 can be identified as invalid: In order to influence the slicing criterion, node 20 has to be executed before nodes 9 and 13. Since `thread_2` is started only *after* node 13, this is impossible. Note that edge $20 \rightarrow_{id} 9$ cannot be removed from the CSDG! For slicing criteria other than node 14 its traversal might be valid.

Integrating more precise MHP information into the timing-sensitive slicer

For simplicity, Alg. 3.3 assumes that all threads of a program may happen in parallel to each other. In order to exploit more elaborate MHP information, it has to be adjusted.

Krinke developed his prepending property under the assumption that all threads happen in parallel to each other. This information is exploited by the property so that is does not hold for general MHP information. Figure 3.14 provides an example. The shown program consists of two threads, where `thread_2` is forked only if the `if`-conditional evaluates to 'false'. The prepending property allows us to traverse from node 8 over node 6 to node 3: The traversal from node 8 to node 6 is valid because the nodes may happen in parallel. The traversal from node 6 to node 3 is valid because the connecting edge is thread-local. If we assume that both threads may happen in parallel, then this path is timing-sensitive: Nodes 6 and 8 may happen in parallel, nodes 3 and 8 may happen in parallel, too, and node 3 reaches node 6. However, if we know more precisely that node 3 and `thread_2` are exclusive, then this path is not timing-sensitive anymore: Nodes 3 and 8 cannot happen in parallel anymore, therefore node 3 must be able to reach node 8 in order to make the path timing-sensitive, which it does not.

Our above observation means that slices computed via the prepending property are not timing-sensitive with respect to general MHP information, the problem being that paths can become timing-insensitive after the traversal of a thread-local edge. A possible way to compute timing-

sensitive slices with respect to general MHP information is to store the complete path taken so far and to check before each edge traversal whether the resulting path would be timing-sensitive. However, this would lead to impractically many reachability checks, therefore it seems that some precision has to be sacrificed.

State tuples on the level of thread regions Nanda uses the following technique to integrate her MHP information into the slicer: State tuples contain one entry per thread region, and a context c about to be visited via an interference edge has to reach the context stored at the position of c's thread region in the current state tuple. After the traversal of an edge towards a context c, the entries of all thread regions that cannot happen in parallel to c's thread region are set to c. That way, situations where parts of two threads cannot happen in parallel are treated more precisely than by Alg. 3.3. In the example of Fig. 3.13, thread 2 would be represented by one thread region, whose state is set to node 13 at the time the slicer visits node 13, because it cannot happen in parallel to it. At the attempted traversal from node 9 to node 20 the state of that region is still node 13, so the traversal can be identified as invalid. On the contrary, the approach significantly enlarges the state tuples: Our evaluation in chapter 3.12 reveals that middle-sized programs with 2 or 3 threads may possess more than thousand thread regions. The approach is also not able to detect the timing-insensitivity in Fig. 3.14.

State tuples accounting for non-existing threads Our proposed solution needs only one entry per thread in the state tuples. It focuses on information about whether a thread cannot exist at all at the currently visited context. For that purpose, our state tuple mechanism uses the following special states:

- $\bot$, the *nonrestrictive state*, is used for existing threads that have not been visited yet and for multi-threads.
- $\top$, the *exclusive state*, is used for threads that cannot exist if the slicer reaches the current context, because their fork site context cannot have been executed before the current context.

Relation *may-exist* determines whether a given thread θ may have been forked before a given context c.

Definition 3.20 (May-exist)**.** *Let θ be a thread and c be a context, and let s be the context of θ's start node. Thread θ may exist at context c, written may-exist(c, θ), if*

- *there exists a fork site context f that reaches s via one outgoing edge and reaches c via a different outgoing edge, or*
- *s reaches c.*

Algorithm 3.4 Procedure `update`: Updating state tuples.

Input: A context c and a state tuple Γ.
Output: A state tuple Γ'.

```
Γ' = Γ  // create a copy of Γ
Let θ(c) be the thread of c
if θ(c) is not a multi-thread   // leave multi-threads alone
   Γ' = [c/θ(c)]Γ'  // set θ(c)'s state to c
for all threads θ   // deactivate threads
   if ¬ may-exist(c, θ)   // check whether θ cannot exist anymore
      Γ' = [⊤/θ]Γ'  // set θ's state to ⊤
return Γ'
```

Relation *may-exist* is a conservative approximation of the $\parallel$ relation for contexts and can be computed for every thread θ during the construction of the CSDG, by collecting for each fork site context all reachable contexts via a call-string-based traversal of the TCFG.

Lemma 3.1. *Let c, d be two contexts, and let $\theta(c)$ and $\theta(d)$ be the threads of c and d. If $c \parallel d$, then may-exist$(c, \theta(d))$ and may-exist$(d, \theta(c))$.*

Proof. According to definition 3.13, we have the following cases:

1. $\theta(c) = \theta(d)$ and $\theta(c)$ is a multi-thread, or
2. $\theta(c) \neq \theta(d)$, neither node of one of the two contexts is dominated by a must-join of the thread of the other context, and
   - there exists a fork site context f in G that indirectly forks the thread of the one context via an outgoing fork edge and reaches the other context via a different outgoing edge.

In case 1, the context of the start node of $\theta(c)$ reaches c and d, thus may-exist$(c, \theta(d))$ and may-exist$(d, \theta(c))$. In case 2, it follows from the existence of fork site context f that may-exist$(c, \theta(d))$ and may-exist$(d, \theta(c))$. □

The *may-exist* relation is used during the slice to deactivate a thread in the state tuples as soon as the traversal visits a context c at which the thread may not exist. This means that the thread can also not exist at contexts which lie before c in a timing-sensitive path. Therefore, a deactivated thread cannot be reactivated during the remaining traversal. This updating mechanism is depicted in Algorithm 3.4. Called with a context c and a state tuple Γ, it sets the state of c's thread to c if that thread is not a multi-thread. Every thread for which *may-exist*(c, θ) does not hold is set to state $\top$.

It remains to extend the *reaches* relation as follows: Let C be the set of contexts of a program p, then $\forall c \in C \cup \{\bot, \top\} : c \rightsquigarrow^* \bot$ and $\top \rightsquigarrow^* c$. The effect of this extension is that a traversal towards a thread in state $\bot$ is always valid and a traversal towards a thread in state $\top$ is always

rejected. This mechanism cannot lead to a rejection of a timing-sensitive path, which is shown by the next lemma.

Lemma 3.2. *Let $c_0 \rightarrow_{ts}^{*} c_k$ be a timing-sensitive path in a CSDG G. Let θ be the thread of c_0. For every $0 \leq i \leq k$, may-exist(c_i, θ) holds.*

Proof. Let c_i be a context in the path. Since the path is timing-sensitive, it follows that either $c_0 \rightsquigarrow^{*} c_i$ or that $c_0 \parallel c_i$. In the first case, the context of the start node of θ reaches c_i, too, because 'reaches' is transitive, so *may-exist*(c_i, θ) holds. In the second case, it follows from lemma 3.1 that *may-exist*(c_i, θ) holds. □

Let us apply our mechanism to our examples. In Fig. 3.13, the state of thread 2 is set to state $\top$ as soon as the traversal arrives at node 13. The traversal from node 9 to node 20 is rejected because state $\top$ is not reachable. Like Nanda's approach, our mechanism is not able to detect the timing-insensitivity in Fig. 3.14. Its advantage is that it gets along with one entry per thread in the state tuples. Furthermore, in opposite to the MHP information used by Nanda's approach the may-exist information is context-sensitive. We therefore expect to yield smaller slices.

3.8. Limitations of Timing-Sensitive Slicing

The definition of timing-sensitive paths is based on the assumption that every program execution of the analyzed program executes the statements in accordance with the order specified by the TCFG. Unfortunately, during the execution of a program its code may be reordered (by a just-in-time compiler, the processor or the memory architecture [48]), which may turn previously timing-insensitive paths into valid ones. Consider the example in Fig. 3.15. It shows a program fragment of two threads, on the left side their ICFGs and on the right side a possible actual execution order at runtime. For Java programs, this reordering is feasible, because the accesses to the shared variables are not properly synchronized [48, chapt. 17]. The timing-sensitive slice for statement 2 computed by using the ICFGs on the left side consists of the statements 1, 2, 4 and 5. The interference dependence between statements 3 and 5 is not traversed because it would result in a timing-insensitive path. But as a result of the reordering on the right side statement 3 is in fact able to influence the slicing criterion.

If statement reordering may turn timing-sensitive slices incorrect, what does that mean for timing-sensitive slicing? Fortunately, the Java language specification [48] makes a strong guarantee for Java programs that are correctly synchronized, i.e. that contain no data races. If a Java program is correctly synchronized, then all executions of the program will appear to be *sequentially consistent* [48, §17.4.5]. If an execution is sequentially consistent, then all *inter-thread actions* in it occur in a total order consistent with *program order* [48, §17.4.3]. Inter-thread actions are actions that may interact with other threads, such as reads or writes to shared variables,

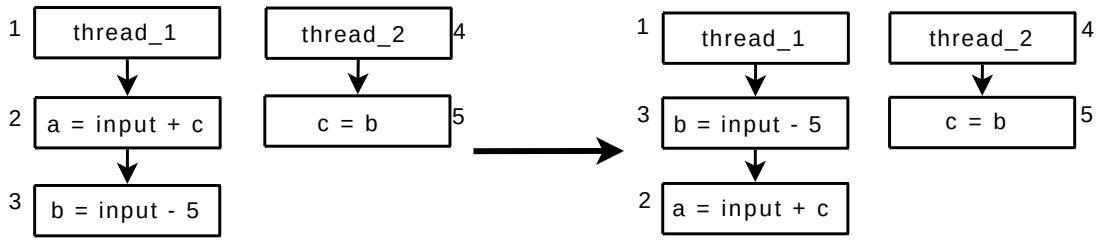


Figure 3.15.: Statement reordering at runtime may switch statements 2 and 3.

forks and joins, or synchronization actions; an action whose statement may cause a concurrency edge in our CSDGs is an inter-thread action. The program order of a thread t is a total order of t's inter-thread actions. It is defined as "... a total order that reflects the order in which these actions would be performed according to the intra-thread semantics of t " [48, §17.4.3].

Using that guarantee we can show for correctly synchronized Java programs that statements that occur only in timing-insensitive paths to a slicing criterion cannot influence the slicing criterion.

Theorem 3.2. *Let G be the CSDG of a correctly synchronized Java program and let s be a node in G. A node that occurs only in timing-insensitive paths to s cannot influence s.*

Proof. Let n be a node such that no context path from a context c_n of n to a context c_s of s is timing-sensitive: $\nexists c_n \rightarrow^*_{ts} c_s \in \{c_n \rightarrow^* c_s\}$. Let $\Phi = c_n \rightarrow^* c \rightarrow d \rightarrow^*_{ts} c_s$ be a timing-insensitive path such that the sub-path $c' \rightarrow^*_{ts} c_s$ is timing-sensitive and $c \rightarrow d$ denotes the interference dependence at which Φ becomes timing-insensitive (we can ignore the other kinds of dependences because of the prepending property; only at interference dependences the TCFG is used to verify the timing-sensitivity of the resulting path). According to the prepending property, path $c \rightarrow d \rightarrow^*_{ts} c_s$ is timing-insensitive because c fails to reach the first context f in the sub-path $d \rightarrow^*_{ts} c_s$ that belongs to the same thread. Note that at f the thread must have been left via a concurrency edge, so c, d and f generate inter-thread actions.

If a Java program is correctly synchronized, then every execution of that program is sequentially consistent and exhibits a total order of its inter-thread actions that is consistent with program order [48, §17.4.3 and §17.4.5]. Every execution E of p that realizes the dependences of sub-path $d \rightarrow^*_{ts} c_s$ has to execute actions generated by d and f. If E also executes an action generated by c, then E must execute that action behind the last action generated by f, because c cannot reach f in the TCFG.

In order to let c_n influence c_s via path Φ, the actions in an execution E have to be reordered such that an action generated by c appears before an action generated by d and before an action generated by f. But this order of inter-thread actions is not consistent with the program order of the thread of c' and f'. □

In summary, code reordering performed at runtime limits the application of timing-sensitive slicing to correctly synchronized Java programs. Similar results are not known for other languages. The problem of statement reordering has not been addressed by any previous approach to timing-sensitive slicing [31, 75, 106, 111, 112, 122]. The I2P slicer is not affected by statement reordering because it traverses each encountered interference edge and ignores execution orders.

3.9. Krinke's Timing-Sensitive Slicer

This section describes Krinke's original timing-sensitive slicer, provides several extensions and optimizations and presents pseudocode suitable for reproduction.

Krinke's original slicer is very close to Alg. 3.3. He assumes that all threads may happen in parallel and that a program has a statically known number of threads. His *threaded ICFG* consists of one ICFG per thread, and all ICFGs are disjoint. His *threaded IPDG* contains one IPDG per thread, which are connected via interference edges. Forking and joining of threads is not modeled, all threads are assumed to start directly after the program start and to run until program termination. These assumptions permit a 'lazy' updating of state tuples. It suffices to update state tuples only after the traversal of an interference dependence. As a consequence, his thread-local slicer does not need to work with state tuples at all. It is a slight modification of Alg. 2.2 that additionally returns the set of contexts at which the corresponding thread can be left. Since that algorithm turned out to be quite expensive, Krinke suggested the following optimization: The thread-local slice $\overline{S}$ for a context c is computed via the much faster context-restricted slicer (Alg. 2.3). Then, the nodes on all SDG paths between the node of c and those nodes in $\overline{S}$ where the thread can be left are collected in a set $Chop$. This can be done efficiently via *chopping* (cf. chapter 4). Finally, the contexts at which the thread can be left are determined by computing a slice for c with the modification of Alg. 2.2, but only along the paths formed by the nodes in $Chop$. Algorithm 3.5 presents pseudocode for Krinke's original algorithm.

Improvement Since Krinke did not implement his slicer, it offered several opportunities for improvement. We applied the following major optimizations:

- Development and integration of a fast, scalable analysis of reachability between contexts.
- Integration of more precise MHP information.
- Handling of multi-threads.
- Integration of the restrictive state tuple optimization.
- Integration of forking and joining of threads.

The remainder of this section describes these optimizations in detail.

Algorithm 3.5 Krinke's original timing-sensitive slicer.

Input: A CSDG $G = (N, E)$, a slicing criterion s.
Output: The slice S for s.

Let $C(n)$ be the set of all possible contexts of node n
Let $\theta(c)$ denote the thread of context c
Let $\Gamma(\theta)$ be the context stored in state tuple Γ for thread θ
Let $[c/\theta]\Gamma$ return a copy of Γ in which thread θ is mapped to context c
Let `IPDG`(c) be a modified IPDG slicer that returns the thread-local slice $\overline{S}$ for c and the set I of visited contexts at which the thread can be left

/ Initialization */*
$\Gamma_0 = (\bot, \ldots, \bot)$ *// initially, every thread is in a nonrestrictive state*
$W = \{(c, \Gamma_c) \mid c \in C(s) \wedge \Gamma = [c/\theta(s)]\Gamma_0\}$
$M = \{\,\} \cup W$ *// mark the visited configurations*

repeat
  $W = W \setminus \{(c_n, \Gamma)\}$ *// process the next configuration*
  Compute the thread-local slice $\overline{S}$ for c_n via the context slicer (Alg. 2.2)
  $S = S \cup \overline{S}$

  / Determine at which contexts the thread can be left */*
  $I_{nodes} = \{m \in \overline{S} \mid \exists m' \in N : \exists m' \rightarrow_{conc} m \in E\}$
  Compute a chop $Chop(I_{nodes}, n)$ (cf. chapter 4)
  Compute $(S', I) =$ `IPDG`(c_n), but only along the nodes in $Chop(I_{nodes}, n)$

  / Determine the valid interference dependences */*
  for all $i \in I$
    $\Gamma_i = [i/\theta(c_n)]\Gamma$ *// c_n's thread is left at context $i \Rightarrow$ update state tuple*
    for all $m \rightarrow_{id} m'$: m' is the node of context i
      // compute the valid contexts of m
      $C_m = \{c_m \in C(m) \mid \theta(c_m) \neq \theta(c_n) \wedge (\Gamma_i(\theta(c_m)) == \bot \vee c_m \text{ reaches } \Gamma_i(\theta(c_m)))\}$

      // Update worklist W
      for all $w \in \{(c_m, \Gamma_m) \mid c_m \in C_m \wedge \Gamma_m = [c_m/\theta(c_m)]\Gamma_i\}$
        if $w \notin M$
          $W = W \cup \{w\}$
          $M = M \cup \{w\}$

until $W = \emptyset$
return S

3.9.1. An Optimized Reachability Analysis

The slicing algorithm needs an analysis of reachability between contexts in order to determine valid traversals of interference dependences and to determine restrictive state tuples. Krinke suggests a context-driven traversal of the TCFG, similar to the IPDG slicer. We propose a more efficient algorithm.

Due to our proposed update mechanism for state tuples, the computation of valid traversals of interference dependences and of restrictive state tuples can be confined to a reachability analysis in ICFGs. In fact, it is not even necessary to leave and enter procedures at all. Consider the following example program:

```
void main():        void foo():        void bar():
  foo();              int i = 0;         int j = 1;
  skip;
  bar();
```

Let *start* be the context of `int i = 0` and *target* be the context of `int j = 1`. The call string of *start* consists of the call $foo()$, the call string of *target* consists of the call $bar()$. It is easy to see that *start* reaches *target*. The approach based on the IPDG slicer performs the following steps to reach *target*: First, it traverses from *start* to the intra-procedural successor of call `foo()`, statement `skip`. Then, it traverses from `skip` to `bar()`. And finally, it traverses from `bar()` to *target*. These three steps are performed by every successful reachability analysis: In the first step, the ICFG is traversed from *start* back to that procedure where the call strings of *start* and *target* start to diverge, which we call the *last shared procedure*. In the second step, the analysis traverses intra-procedurally from the return node of the call that leads to *start* to the call that leads to *target*. And in the third step, it traverses from the latter call to *target*. But the first and third step are redundant: The first step always succeeds, since a call string can always be decomposed and since there always exists a last shared procedure (all procedures are reachable from the main procedure). The third step always succeeds, too: The call string of *target* shows that there is a valid path from its call in the last shared procedure to *target*. So only the second step requires actual checking. This leads to the following reachability algorithm:

1. Determine the longest call string *pre* that prefixes both the call strings of *source* and *target*. The topmost element of *pre* is the call of the last shared procedure.

2. Determine the calls in the call strings of *source* and *target* that directly follow the prefix. These are the calls in the last shared procedure that lead to *source* and *target*. We call them *source_call* and *target_call*, respectively.

Algorithm 3.6 Algorithm `reach`: Determining reachability between contexts in ICFGs.

Input: An ICFG *G*, two contexts *source* and *target* in *G*.
Output: 'true' if *source* reaches *target* in *G*, 'false' otherwise.

```
Let pre be the longest common prefix of source and target

if source == target or source == X or target == ⊥   // special cases being always 'true'
   return true
else if source == ⊥ or target == X   // special cases being always 'false'
   return false
else if pre contains a synthetic call site
   return true
else
   // determine the call sites of source and target in the last shared procedure
   source_call = source.call_string[pre.length]
   target_call = target.call_string[pre.length]
   Let source_succ be the return node of source_call

   if source_succ reaches target_call intra-procedurally in G
      return true

return false
```

3. Perform an intra-procedural reachability analysis from the return node of *source_call* to *target_call* in the last shared procedure. If it is successful, then *source* reaches *target*, otherwise not.

Several modifications are required to deal with recursive calls. We exploit that recursive calls are represented by synthetic call site in the call strings. There are the following cases:

- The prefix *pre* contains a synthetic call site.
  In this case the reachability check succeeds: *start* and *target* belong to procedures called (transitively) from the same recursive cycle, which means that *start* and *target* reach each other.

- One of *source_call* or *target_call* is a synthetic call site.
  This case does not influence our algorithm, since *start* and *target* are not called from within the same recursive cycle. Context *source* reaches *target* if *source_call* reaches *target_call* in the last shared procedure.

- None of the previous cases applies.
  The algorithm is not influenced.

Hence, step 1 of our algorithm is modified as follows:

1. Determine the longest call string *pre* that prefixes both the call strings of *source* and *target*. The topmost element of *pre* is the call of the last shared procedure. If *pre* contains a synthetic call site, then *source* reaches *target*. Otherwise, proceed with step 2.

Figure 3.6 presents pseudocode for that algorithm.

3.9.2. Integration of MHP Information and Multi-Threads

We integrated our technique introduced in section 3.7 into Krinke's slicer so that it can process the information of our MHP analysis. Multi-threads are treated as described in section 3.6.5. A consequence of these extensions is that the thread-local slicer now has to propagate and update state tuples. Algorithm 3.7 is an appropriately enhanced version of Krinke's thread-local slicer. The algorithm also realizes the restrictive state tuple optimization. For that purpose, it receives a set M from the surrounding slicing algorithm that contains the already visited configurations.

3.9.3. Our Optimized Version of Krinke's Slicer

Algorithm 3.8 presents pseudocode for our optimized version of Krinke's slicer. A very important optimization, which is for space reasons not shown in the pseudocode, is that one iteration of the main loop should not process only the next configuration in W but all configurations in W belonging to the same thread at once. The slicing and chopping algorithms can be trivially extended to process a set of slicing or chopping criteria. This optimization strongly reduces the number of invocations of the context-restricted slicer and of the chopping algorithm.

Forking and joining of threads has been integrated as follows: Every context reachable via a join-out edge is identified as valid. At the traversal of a fork or fork-in edge the fork site context of the current thread is used to take only that context of the adjacent (fork or actual-in) node whose call string equals the call string of the fork site context.

Algorithm 3.7 Procedure `seq`: An IPDG slicer that propagates state tuples.

Input: A CSDG G, a configuration (s, σ_s, Γ_s), a set M storing the visited configurations.

Output: The thread-local slice $\overline{S}$ for (s, σ_s) and the set I of configurations at which the thread can be left via concurrency edges.

$W = \{(s, \sigma_s, \Gamma_s)\}$ *// initialize the worklist*
$\overline{S} = \{\}, I = \{\}$
repeat
  $W = W \setminus \{(n, \sigma, \Gamma)\}$ *// process next configuration*
  $S = S \cup \{n\}$
  for all $m \rightarrow_e n$ *// handle all incoming edges of n*
    if $e \in \{conc\}$
      $I = I \cup \{(n, \sigma, \Gamma)\}$ *// when encountering a concurrency edge, store* (n, σ, Γ) *in I*
    else if $e \in \{pi, call\}$ *// ascend to a calling procedure*
      Let c_m be the corresponding call site
      if $\sigma = \sigma'.c_m$ *// this test guarantees context-sensitivity*
        $\Gamma' = \text{update}(m, \sigma', \Gamma)$ *// update state tuple (Alg. 3.4)*
        if $\nexists (m, \sigma', \Gamma'') \in M : \Gamma'$ is restrictive to Γ'' *// detect redundant configurations*
          $W = W \cup \{(m, \sigma', \Gamma')\}$
          $M = M \cup \{(m, \sigma', \Gamma')\}$
        if c_m is recursive *// at recursive calls we also have to conserve call string* σ
          $\Gamma'' = \text{update}(m, \sigma, \Gamma)$
          if $\nexists (m, \sigma, \Gamma''') \in M : \Gamma''$ is restrictive to Γ'''
            $W = W \cup \{(m, \sigma, \Gamma'')\}$
            $M = M \cup \{(m, \sigma, \Gamma'')\}$
    else if $e \in \{po\}$ *// descend into a called procedure*
      Let c_n be the corresponding call site
      if c_n is recursive **and** $\sigma = \sigma'.c_n$ *// preserve the call string in recursive cycles*
        $\Gamma' = \text{update}(m, \sigma, \Gamma)$
        if $\nexists (m, \sigma, \Gamma'') \in M : \Gamma'$ is restrictive to Γ''
          $W = W \cup \{(m, \sigma, \Gamma')\}$
          $M = M \cup \{(m, \sigma, \Gamma')\}$
      else
        $\sigma'' = \sigma.c_n$ *// append* c_n *to* σ
        $\Gamma' = \text{update}(m, \sigma'', \Gamma)$
        if $\nexists (m, \sigma'', \Gamma'') \in M : \Gamma'$ is restrictive to Γ''
          $W = W \cup \{(m, \sigma'', \Gamma')\}$
          $M = M \cup \{(m, \sigma'', \Gamma')\}$
    else *// intra-procedural edge – preserve call string* σ
      $\Gamma' = \text{update}(m, \sigma, \Gamma)$
      if $\nexists (m, \sigma, \Gamma'') \in M : \Gamma'$ is restrictive to Γ''
        $W = W \cup \{(m, \sigma, \Gamma')\}$
        $M = M \cup \{(m, \sigma, \Gamma')\}$

until $W = \emptyset$
return (S, I)

Algorithm 3.8 Our optimized version of Krinke's timing-sensitive slicer.

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

Let $C(n)$ be the set of all possible contexts of node n
Let $\theta(c)$ denote the thread of context c
Let $\Gamma(\theta)$ be the context stored the state of thread θ in state tuple Γ
Let $[c/\theta]\Gamma$ return a copy of Γ in which thread θ is mapped to context c

/ Initialization. */*
$\Gamma_0 = (\bot, \ldots, \bot)$ *// initially, every thread is in the nonrestrictive state*
$W = \{(s, \sigma, \Gamma') \mid (s, \sigma) \in C(s) \wedge \Gamma' = \texttt{update}(s, \sigma, \Gamma_0)\}$ *// update via Alg. 3.4*
$M = \{\} \cup W$ *// stores the visited configurations*

repeat
  $W = W \setminus \{(n, \sigma_n, \Gamma_n)\}$ *// process the next configuration*
  Compute the thread-local slice $\overline{S}$ for (n, σ_n) with the context-restricted slicer (Alg. 2.3)
  $S = S \cup \overline{S}$

  / Determine at which contexts the thread of n can be left */*
  $I_{nodes} = \{m \in \overline{S} \mid \exists m' \in N : \exists m' \rightarrow_{conc} m \in E\}$
  Compute a chop $Chop(I_{nodes}, n)$
  Compute $(S', I) = \texttt{seq}(G, c_n, M)$, but only along the nodes in $Chop(I_{nodes}, n)$

  / Determine the valid concurrency edges */*
  for all $(i, \sigma_i, \Gamma_i) \in I$
    Let m' be the node of context i
    for all $m \rightarrow_e m', e \in \{conc\}$ *// all incoming concurrency edges*
      $C'_m = \emptyset$
      if $e = id$ *// interference edge*
        / Proceed with those contexts of m that lie in another thread*
          (unless we are in a multi-thread) and reach the state of their thread
          *in Γ_i. Use the reachability analysis in Alg. 3.6. */*
        $C'_m = \{c_m \in C(m) \mid (\theta(c_m) \neq \theta(c_n) \vee \theta(c_n)$ is a multi-thread$) \wedge c_m$ reaches $\Gamma_i(\theta(c_m))\}$
      else if $e = jo$ *// join-out edge*
        $C'_m = C(m)$
      else *// fork or fork-in edge; make sure to traverse to the right fork site*
        Let (f, σ_f) be the fork site context of $\theta(c_n)$.
        $C'_m = \{(m, \sigma_m) \in C(m) \mid \sigma_m = \sigma_f\}$

      / Update worklist W. */*
      for all $(m, \sigma_m) \in C'_m$
        $\Gamma_m = \text{update}(m, \sigma_m, \Gamma_i)$ *// update via Alg. 3.4*
        / Check whether Γ_m is a restrictive state tuple. */*
        if $\nexists (m, \sigma_m, \Gamma'_m) \in M : \Gamma_m$ is restrictive to Γ'_m
          $W = W \cup \{(m, \sigma_m, \Gamma_m)\}$
          $M = M \cup \{(m, \sigma_m, \Gamma_m)\}$

until $W = \emptyset$
return S

3.10. Nanda's Timing-Sensitive Slicer

This section describes Nanda's timing-sensitive slicer, who picked up Krinke's algorithm and extended it into several directions. She reported the first known implementation of a timing-sensitive slicer and thus was able to develop several crucial optimizations. Most importantly, she introduced the restrictive state tuple optimization (cf. section 3.6.4) and developed a compact representation of call strings by single integers. Furthermore, she was the first to investigate the relationship between timing-sensitive slicing and MHP information and integrated the results of her thread regions analysis into her slicer. Her *interprocedural threaded control flow graph* and her *threaded system dependence graph* are similar to our TCFG and CSDG. Fork sites and join sites are modeled the same, where we adopted her modeling of join sites. The major difference is that her ICFGs and SDGs are really disjoint, whereas in our graphs shared CFGs and PDGs are not duplicated. Furthermore, it is not entirely clear whether her graphs contain one ICFG or SDG per thread class or per thread. It seems that the latter is the case, although no thread invocation analysis which could be used to identify the existing threads is mentioned.

An important result of our investigation is that Nanda's original algorithm may compute incorrect slices, because it uses the the restrictive state tuple optimization too greedily. We describe that problem and present a remedy. Furthermore, we developed several new optimizations, which are also presented. A preliminary version of the following descriptions has been published in [43].

3.10.1. Context Representation and Reachability Analysis

Since timing-sensitive slicing uses contexts to memorize the execution states of threads, an efficient technique for working with contexts is mandatory. Nanda argues that call strings are impractical, because they grow with the size of the target program and thus lead to poor runtime performance and high memory consumption. Her slicer represents contexts by single integers instead. For that purpose, a preprocessing step collects and enumerates all possible contexts. This preprocessing step in turn employs call strings, but has to be executed only once. The result can be stored in form of a *context graph*, where each node represents one context.

The preprocessing step folds cycles in interprocedural control flow graphs and creates an *interprocedural strongly connected regions* (ISCR) graph, in which context-sensitive paths are free of cycles. This permits a topological enumeration of the remaining contexts in reverse postorder. Nanda realizes the preprocessing step with a rather intricate algorithm, wherefore we present a less complicated alternative. It consists roughly of four steps:

1. Create one ICFG for every thread distinguished by the thread invocation analysis.

2. Fold cycles in ICFGs context-sensitively.

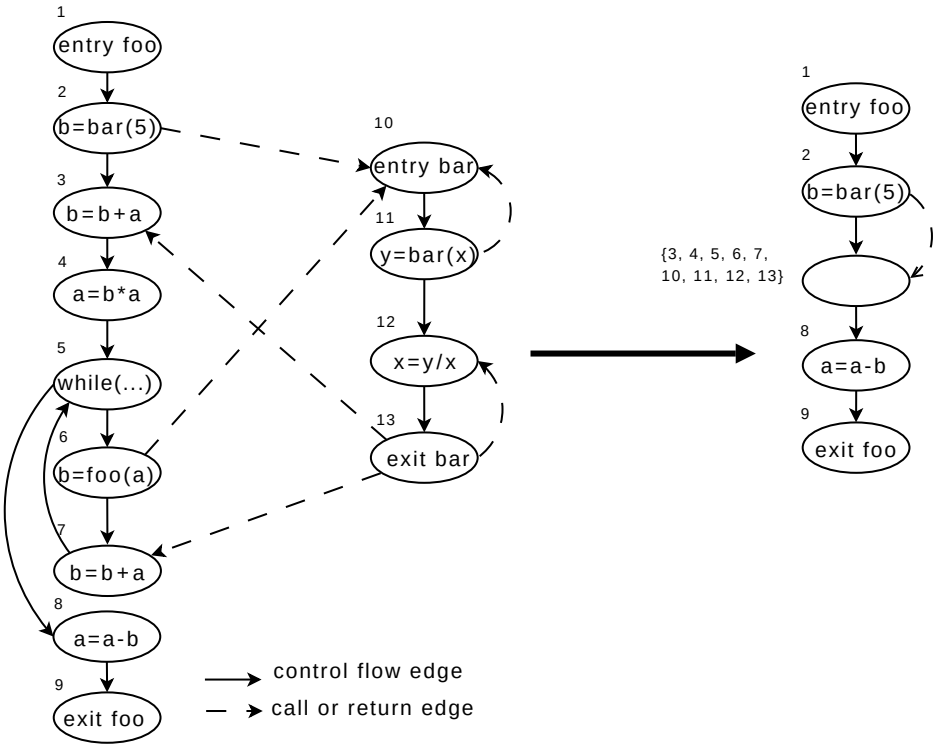


Figure 3.16.: Folding strongly connected components removes information about calling contexts.

3. Inline procedures.

4. Enumerate contexts in reverse postorder.

Create one ICFG for every thread distinguished by the thread invocation analysis The technique requires to have one ICFG per thread or multi-thread distinguished by the thread invocation analysis. To this end, the ICFGs of the thread classes have to be duplicated as often as is necessary.

Fold cycles in ICFGs context-sensitively The challenge in folding cycles in ICFGs is to do so in a context-sensitive manner so that no precision is lost. Simply folding strongly connected components is not sufficient, for which Fig. 3.16 presents an example. In the depicted ICFG node 6 reaches node 3, but only via context-insensitive paths. A context-sensitive analysis is able to detect and reject these paths. This is not possible anymore in the resulting folded graph. Instead of folding strongly connected components, only context-sensitive cycles are folded. A suitable two-phase algorithm has been presented by Krinke [75]:

- Phase 1 removes all return edges and folds the remaining strongly connected components. After that, the return edges are put back.

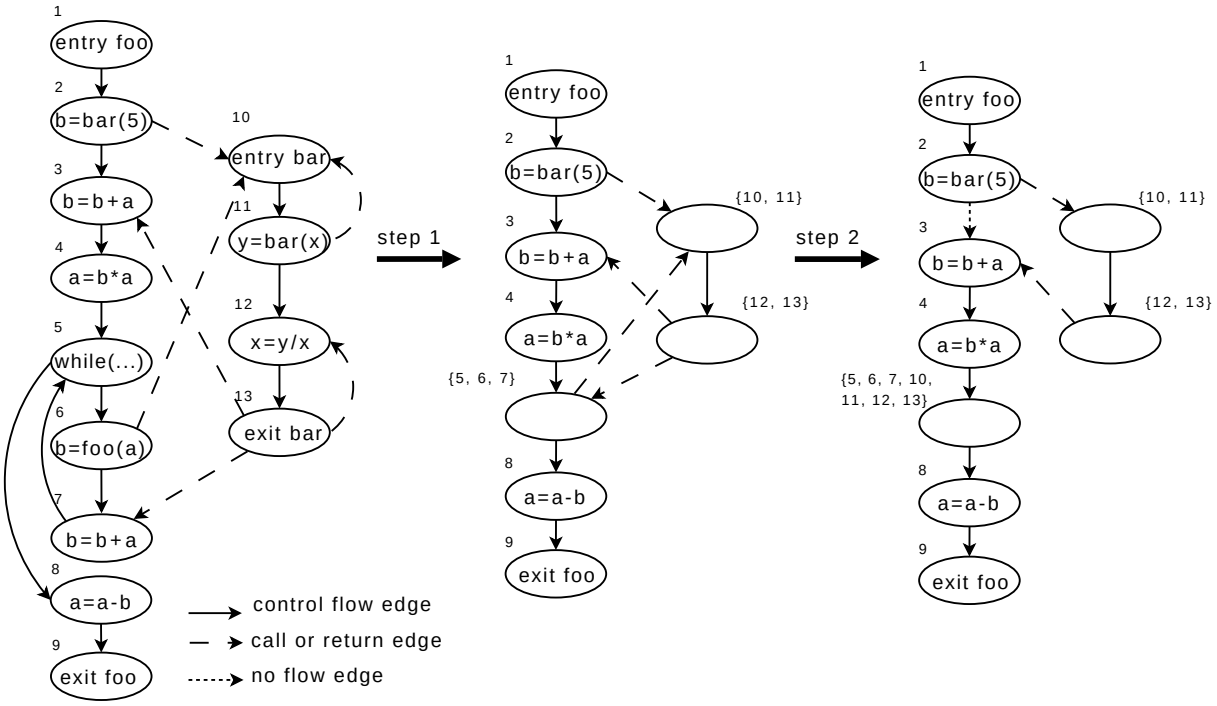


Figure 3.17.: ISCR graph with folded context-sensitive cycles (mid) and with inlined procedures (right).

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

The graph in the middle of Fig. 3.17 shows the result of this proceeding for our example. The loop is folded only intra-procedurally, the recursive call- and return-cycles are folded separately.

Inline procedures The next step inlines procedures called from within a loop or recursive procedure. For that purpose, all nodes of procedures called by a fold node are added to the fold node, and the call and return edges between the fold node and these procedures are deleted. The control flow edges between call nodes and their direct intra-procedural successors are replaced by *no flow edges*, which are needed later for the reachability analysis. The result for our example is shown on the right side of Fig. 3.17. Note that this step duplicates nodes; in our example, nodes 10, 11, 12 and 13 exist twice after the inlining.

Enumerate contexts in reverse postorder In order to enumerate the contexts, a *context graph* is generated from the ISCR graph. In a context graph, each node represents a single context, hence paths in that graph are always context-sensitive. The existing contexts are collected by a call-string driven traversal of the ISCR graph, edges are added accordingly. Since context-sensitive paths in the ISCR graph are free of cycles, no treatment of recursion is needed. Finally, the contexts in the context graph are enumerated in reverse postorder. The finished con-

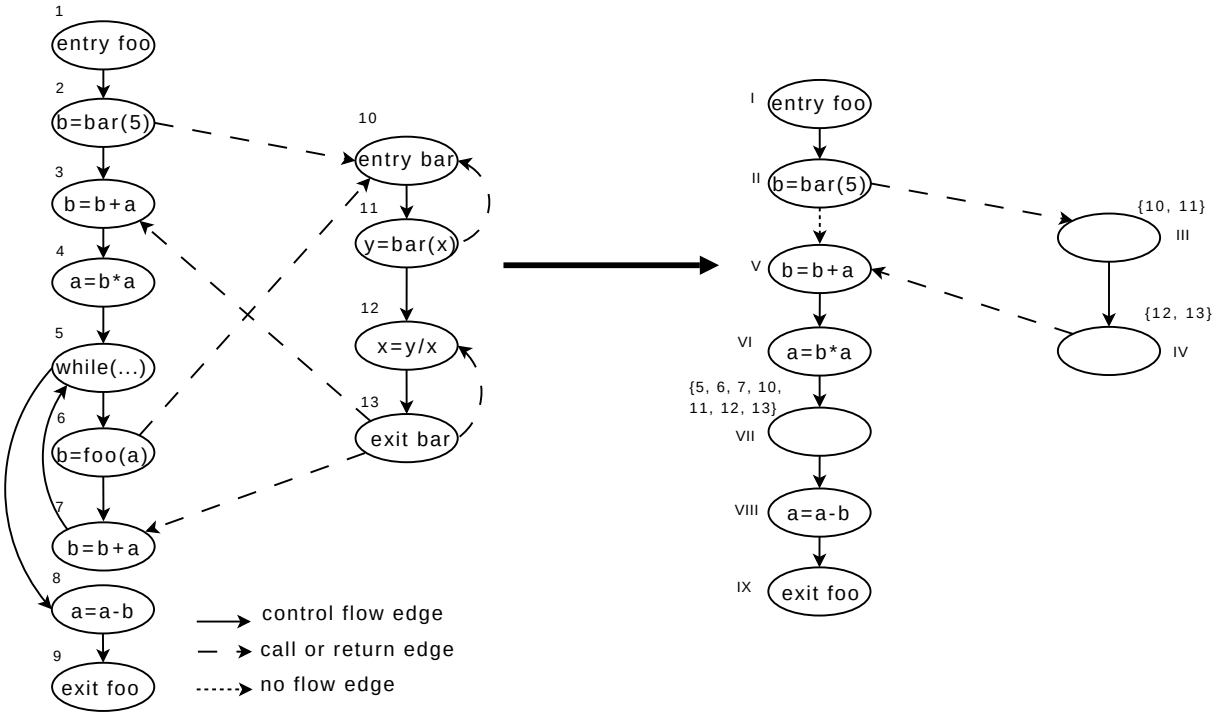


Figure 3.18.: An ICFG and its context graph.

text graph for our example is shown in Fig. 3.18, where the enumeration is represented by roman numbers.

Reachability analysis

Nanda's reachability analysis between two contexts consists of a backward traversal of the context graph. The reverse postorder enumeration of the nodes in the context graph guarantees that a node n has a smaller number than a node m if it reaches m. Nanda exploits that property to quickly check whether procedure calls can be jumped over during the reachability analysis. Assume that we want to know whether node VI can be reached by node I in the context graph in Fig. 3.18. By the time the traversal arrives at node V, it decides whether to traverse the incoming return edge by looking at the associated call node II. As II $\not<$ I, node I cannot lie in that procedure, and the traversal jumps directly to node II.

Context graphs and concurrency

Until now, the creation of context graphs did not mention any treatment of concurrency-related edges. This is so because fork and join edges may cause cycles, whose treatment is not clear. As a consequence of this, the ICFG of each thread is processed by the above technique separately, resulting in a set of disjoint context graphs. However, Nanda's update mechanism for state tuples may assign a thread region a context stemming from another thread, which may lead to

invocations of the reachability analysis for contexts from different threads. For that purpose, she precomputes for each thread region R the set of thread regions reachable in the TCFG by the start node of R. These precomputed results are used whenever a reachability analysis for contexts from different threads is needed. Since these results are computed on the level of nodes, the reachability analysis loses precision at this point.

We are able to solve that problem by using our thread invocation analysis. After the construction of the context graphs of the threads, we use the information of that analysis to connect via fork edges the nodes representing fork site contexts with the start nodes of the corresponding context graphs. Since the topological enumeration of the contexts of a context graph has been done independently from the other context graphs, the traversal of fork edges requires special treatment. Our reachability analysis is presented in section 3.10.4.

Practicability of context graphs

In general, the employment of context graphs should be considered carefully, because their size, compared with the size of the original TCFGs, does not scale well. However, we found that for programs within reach of timing-sensitive slicers context graphs are actually manageable, as long as effective cycle folding techniques are applied. An evaluation is presented in section 3.12.4.

3.10.2. Nanda's Original Slicing Algorithm

Nanda's original slicing algorithm iterates a modified two-phase slicer which queries the context graphs to retrieve the context of a node. It basically works as follows: After traversing an edge $m \to n$ towards m in the CSDG, where c_n is the current context of n, all contexts C_m of m are retrieved from the context graph. Then, a reachability analysis on the context graph discards every context $c_m \in C_m$ that cannot reach c_n. The slicer proceeds with the remaining contexts.

MHP information In order to yield more precise slices than Krinke's algorithm, Nanda lets her slicer exploit the MHP information from her thread region analysis (cf. section 3.3). To this end, state tuples contain one entry per thread region, as described in section 3.7. Because her MHP analysis lacks a thread invocation analysis and thus provides no information about multi-threads, her slicing algorithm uses the following conservative approximation to handle threads created within loops: Let l be a loop that forks a thread θ. During the computation of the ISCR graphs, all nodes of l and the nodes of t are folded into one single fold node. Since now every node of thread θ has the same context, every interference edge traversal towards an instance of θ is identified as valid. She does not address thread creation inside recursive procedures.

Algorithm 3.9 Nanda's original timing-sensitive slicer.

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

Let $\overline{C}(n)$ be a sorted set of all contexts of node n
Let $\theta(c)$ denote the thread of context c
Let $r(c)$ denote the thread region of context c

$W = \emptyset, W1 = \emptyset, W2 = \emptyset$ *// three worklists*
$M = \emptyset$ *// stores visited configurations*
$\Gamma_0 = [\bot, \ldots, \bot]$ *// an initial state tuple, having one entry per thread region*
for all $c_s \in \overline{C}(s)$
  `insert`(Γ_0, c_s, W, M)

repeat *// iterate a modified 2-phase-slicer until W is empty*
  $W = W \setminus \{(n, c_n, \Gamma_n)\}$
  $W1 = W1 \cup \{(n, c_n, \Gamma_n)\}$ *// initialize the next iteration*

  while $W_1 \neq \emptyset$ *// phase 1, only ascend to calling procedures*
    $W1 = W1 \setminus \{(n, c_n, \Gamma_n)\}$
    $S = S \cup \{n\}$ *// add node n to the slice*
    for all $m \rightarrow_e n$
      if $e \in \{conc\}$ *// edge leaves the thread*
        for all $c_m \in \overline{C}(m) : \theta(c_m) \neq \theta(c_n)$ *// make sure we really switch threads*
          if `reaches`$(c_m, \Gamma[r(c_m)])$ *// remain on timing-sensitive paths*
            `insert`(Γ_n, c_m, W, M)
      else if $e \in \{po\}$ *// postpone to phase 2*
        for all $c_m \in \overline{C}(m) : \theta(c_m) == \theta(c_n)$ *// remain in this thread*
          if `reaches`(c_m, c_n) *// proceed only with the reaching contexts*
            `insert`$(\Gamma_n, c_m, W2, M)$
      else *// intra-procedural edges, call- and parameter-in edges*
        for all $c_m \in \overline{C}(m) : \theta(c_m) == \theta(c_n)$ *// remain in this thread*
          if `reaches`(c_m, c_n) *// proceed only with the reaching contexts*
            `insert`$(\Gamma_n, c_m, W1, M)$

  while $W2 \neq \emptyset$ *// phase 2, only descend to called procedures*
    $W2 = W2 \setminus \{(n, c_n, \Gamma_n)\}$
    $S = S \cup \{n\}$ *// add node n to the slice*
    for all $m \rightarrow_e n$
      if $e \in \{conc\}$ *// edge leaves the thread*
        for all $c_m \in \overline{C}(m) : \theta(c_m) \neq \theta(c_n)$ *// make sure we really switch threads*
          if `reaches`$(c_m, \Gamma[r(c_m)])$ *// remain on timing-sensitive paths*
            `insert`(Γ_n, c_m, W, M)
      else if $e \notin \{pi, call\}$ *// intra-procedural edges and parameter-out edges*
        for all $c_m \in \overline{C}(m) : \theta(c_m) == \theta(c_n)$ *// remain in this thread*
          if `reaches`(c_m, c_n) *// proceed only with the reaching contexts*
            `insert`$(\Gamma_n, c_m, W2, M)$
until $W = \emptyset$
return S

Algorithm 3.10 `insert`: Manages the updating of the worklists.

Input: A state tuple Γ_{old}, a context c_m, a worklist W, a set M.

```
/* Create an updated state tuple. */
Γ_m = Γ_old  // create a copy of Γ_old
for all thread regions r that do not execute in parallel to c's thread region
    Γ_m = [c/r]Γ_m  // set r's state to c

/* Run the restrictive state tuple optimization. */
for all (m, c_m, Γ'_m) ∈ M :
    if Γ_m is restrictive to Γ'_m
        return  // the new configuration is redundant

W = W ∪ {(m, c_m, Γ_m)}  // insert the configuration into the worklist
M = M ∪ {(m, c_m, Γ_m)}  // mark it as visited
```

The slicer Algorithm 3.9 shows the original slicing algorithm. It consists of a main loop that iterates a two-phase slicer working with configurations. The two-phase slicer iterates over the worklists $W1$ and $W2$, the main loop over worklist W. A set M is used to mark already visited configurations. Worklist W is initialized with all configurations of slicing criterion s, by calling procedure `insert` (Alg. 3.10) with every context c_s of s and an initial state tuple with all entries set to the nonrestrictive state $\perp$. Procedure `insert` works as follows: It copies the given state tuple and sets the states of all thread regions which cannot happen in parallel to the thread region of the given context c to c. After that, it applies the restrictive state tuple optimization to filter redundant configurations.

When the embedded two-phase slicer traverses an edge $m \to n$, coming from a context c_n, it determines all valid contexts of the reached node m. If the traversed edge is a concurrency edge, these are all contexts of m whose visiting would not cause a timing-insensitive path. To this end, a context c_m of m has to reach the context stored as the state of c_m's thread region in the current state tuple. For all other kinds of edges, all contexts of m which reach context c_n are valid. Via procedure `insert` (Alg. 3.10) the valid contexts are annotated with updated state tuples and inserted into the suitable worklist.

A very effective optimization Nanda developed is to sort the contexts in $\overline{C}(n)$, the set of contexts of a node n, in descending order of their topological numbers. This sortation enables the restrictive state tuple optimization to detect more redundant configurations of node n.

3.10.3. Correctness

As pointed out in previous work [44, 46], Nanda's original algorithm may compute incorrect slices, which we demonstrate at an example. The basic algorithm is correct, but it applies the restrictive state tuple optimization after each edge traversal, which might prune valid interference edges when applied in phase 2. The restrictive state tuple optimization is based on the

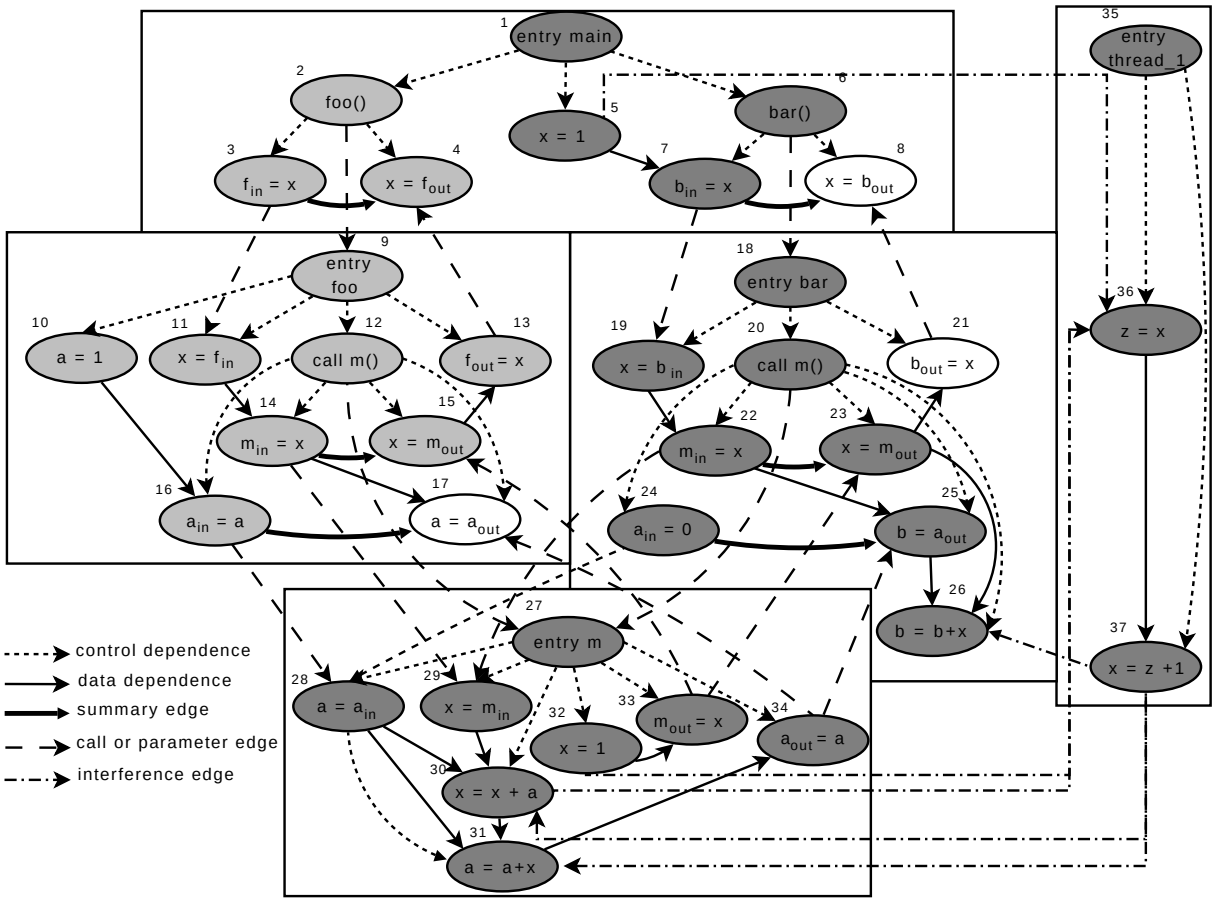


Figure 3.19.: Incorrect slice computed by Nanda's algorithm.

observation that a slice for a configuration (n, c_n, Γ) cannot contain more nodes than a slice for a configuration (n, c_n, Γ') if Γ is restrictive to Γ'. However, this does only hold in phase 1 of the presented slicer. If the slice for configuration (n, c_n, Γ') is confined to phase 2 and the slice for configuration (n, c_n, Γ) is not, then the latter slice may contain more nodes, since phase 2 does not ascend to calling procedures.

Figure 3.19 shows a CSDG for which the slicer computes incorrect slices. It consists of two threads, the main thread and thread_1. To keep matters simple, we assume that the threads may happen in parallel. Procedure `m` is called by procedures `foo` and `bar`. Thus, each node of procedure `m` has two different contexts, where the context resulting from procedure `foo` reaches the context resulting from procedure `bar`. All other nodes have one context. We denote every context of a node with the node itself, nodes of `m` receive a subscript *foo* or *bar* (e.g. 30_{bar} denotes node 30 in the calling context of procedure `bar`). The shaded nodes represent the timing-sensitive slice for node 26, the darker shaded nodes represent the slice computed by Nanda's slicer. It performs the following steps:

Initialization Worklist W is initialized with configuration $(26, 26, [26, \bot])$.

First iteration The first configuration in W, $(26,[26,\bot])$, is taken out and inserted into $W1$. In phase 1, the algorithm visits nodes $\{26,25,24,23,34,33,22,20,19,18,7,6,5,1\}$, traverses the interference edge $37 \rightarrow_{id} 26$ towards node 37 and inserts configuration $(37,[26,37])$ into worklist W. The configurations $(33_{bar},[33_{bar},\bot])$, $(33_{foo},[33_{foo},\bot])$, $(34_{bar},[34_{bar},\bot])$ and $(34_{foo},[34_{foo},\bot])$ are visited and inserted into worklist $W2$. In phase 2, the algorithm visits the nodes $\{34,33,32,31,30,29,28,27\}$, where for every node n the configurations $(n_{bar},[n_{bar},\bot])$ and $(n_{foo},[n_{foo},\bot])$ in $W2$ are inserted into worklist $W2$. It also traverses the interference edges $37 \rightarrow_{id} 31$ and $37 \rightarrow_{id} 30$ and inserts the configurations $(37,[31_{bar},37])$ and $(37,[30_{bar},37])$ into worklist W. The configurations $(37,[31_{foo},37])$ and $(37,[30_{foo},37])$ are discarded by the restrictive state tuples optimization.

Second iteration The next configuration in W, $(37,[26,37])$, is taken out and inserted into $W1$. In phase 1, the algorithm visits the nodes $\{37,36,35\}$. At node 36, with state tuple $[26,36]$, the thread can be left via interference edge $30 \rightarrow_{id} 36$ towards node 30. Contexts 30_{foo} and 30_{bar} are valid according to the reachability analysis, since both can reach the current state of main, context 26. But the state tuples of the resulting configurations, $(30_{foo},[30_{foo},36])$ and $(30_{bar},[30_{bar},36])$, are restrictive to the state tuples of the earlier inserted configurations $(30_{foo},[30_{foo},\bot])$ and $(30_{bar},[30_{bar},\bot])$. Thus these configurations are discarded. The same happens at interference edge $32 \rightarrow_{id} 36$ and during the following iterations for the remaining configurations in woklist W, $(37,[31_{bar},37])$ and $(37,[30_{bar},37])$: Procedure `m` cannot be entered again, wherefore the algorithm omits nodes that belong to the slice.

The remedy The application of the restrictive state tuples optimization has to distinguish between phase 1 and phase 2: A state tuple of a configuration visited in phase 1 must only be checked against the state tuples of other configurations visited in phase 1.

3.10.4. Improvement

Besides the problem concerning the restrictive state tuple optimization, we detected several opportunities for optimizations.

Integrating MHP information

We replaced Nanda's treatment of threads created inside loops with our treatment of multi-threads. That way, her algorithm becomes capable of handling threads created inside recursive cycles. We also replaced her update mechanism for state tuples with ours. Nanda's reachability analysis is adjusted accordingly and is shown in Alg. 3.11.

Algorithm 3.11 `reaches`: Our extension of Nanda's reachability analysis.

Input: Two contexts, s and t, the context graph G of the program.
Output: `true` if s reaches t, `false` otherwise.

```
if t == ⊥ or s == ⊤
  // every state reaches the nonrestrictive state, the exclusive state reaches every state
  return true
else if s == ⊥ or t == ⊤
  // the nonrestrictive state can only reach itself, the exclusive state can only be reached by itself
  return false
else // θ(s) == θ(t) is guaranteed
  /* The following code is Nanda's original reachability analysis (enriched with a treatment of fork
  edges). */
  W = {t} // a worklist
  M = {t} // marks the visited contexts

  repeat // traverse G backwards, starting from t
    W = W \ {c}
    if c = s // s reaches t
      return true
    if c < s // c cannot be reached by s ⇒ skip c
      continue

    for all d →_e c // traverse backwards all incoming edges
      if e == fork // ignore fork edges – s and t belong to the same thread
        continue
      else if e ∈ {ret} // a return edge – check if we can bypass the procedure
        Let d' be the context connected with c through a no flow edge (d' is the procedure call)
        if d' < s // s cannot be in the procedure called by d' ⇒ jump directly to d'
          if d' ∉ M
            W = W ∪ {d'}
            M = M ∪ {d'}
        else if d ∉ M // enter procedure
          W = W ∪ {d}
          M = M ∪ {d}
      else if e ∈ {cf, call} and d ∉ M // a call or control flow edge
        W = W ∪ {d}
        M = M ∪ {d}
  until W = ∅

return false // s does not reach t
```

Algorithm 3.12 Our repaired and optimized version of Nanda's slicer.

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

Let $\overline{C}(n)$ be a sorted set of all contexts of node n
Let $\theta(c)$ denote the thread of context c
Let $proc(c)$ denote the procedure ID of context c

$W = \emptyset, W_{inner} = \emptyset$ *// two worklists*
$M = \emptyset$ *// a set storing the visited contexts*
$\Gamma_0 = [\bot, \ldots, \bot]$ *// an initial state tuple, having one entry per thread*
for all $c_s \in \overline{C}(s)$
  `insert`(Γ_0, c_s, W, M)

repeat *// iterate a thread-local slicer until W is empty*
  $W = W \setminus \{(n, c_n, \Gamma_n)\}$
  $W_{inner} = W_{inner} \cup \{(n, c_n, \Gamma_n)\}$ *// initialize the next iteration*

  while $W_{inner} \neq \emptyset$
    $W_{inner} = W_{inner} \setminus \{(n, c_n, \Gamma_n)\}$
    $S = S \cup \{n\}$ *// add node n to the slice*

    for all $m \rightarrow_e n$
      if $e \in \{id\}$ *// interference edge*
        for all $c_m \in \overline{C}(m) : \theta(c_m) \neq \theta(c_n) \vee \theta(c_n)$ is a multi-thread *// switch threads*
          if `reaches`$(c_m, \Gamma[\theta(c_m)])$ *// remain on timing-sensitive paths*
            `insert`(Γ_n, c_m, W, M)
      else if $e \in \{fork, fi\}$ *// fork or fork-in edge*
        Let c_m be the context of m connected with c_n through an outgoing fork edge
        if $c_m \neq \varepsilon$ *// if c_m does not exist, the traversal is context-insensitive*
          `insert`(Γ_n, c_m, W, M)
      else if $e \in \{jo\}$ *// join-out edge*
        for all $c_m \in \overline{C}(m)$
          `insert`(Γ_n, c_m, W, M)
      else if $e \in \{pi, call\}$
        Let c_m be the context of m directly predecessing c_n or equal to c_n in c_n's context graph
        if $c_m \neq \varepsilon$ *// if c_m does not exist, the traversal is context-insensitive*
          `insert`$(\Gamma_n, c_m, W_{inner}, M)$
      else if $e \in \{po\}$
        Let c_m be the context of m directly predecessing c_n or equal to c_n in c_n's context graph
        `insert`$(\Gamma_n, c_m, W_{inner}, M)$
      else *// e is an intra-procedural edge*
        for all $c_m \in \overline{C}(m) : proc(c_m) == proc(c_n) \wedge$ `reaches`(c_m, c_n) *// remain in the procedure*
          `insert`$(\Gamma_n, c_m, W_{inner}, M)$

until $W == \emptyset$

return S

The reachability analysis as a bottleneck

After each traversal of a dependence edge $m \to n$ towards m, where c_n is the current context of n, Nanda's slicer has to identify the valid contexts of m. These are determined via a reachability analysis, which can be a bottleneck in programs where statements have many different contexts. In order to speed up the traversal of thread-local dependence edges we annotate contexts with procedure IDs, which are assigned during the construction of the context graphs. These procedure IDs are used as follows: If the slicer traverses an intra-procedural dependence edge, it has to process only those contexts C'_m of m with the same procedure ID as c_n. The slicer proceeds with all contexts $c_m \in C'_m$ that reach c_n. There may in fact be several contexts of m with the same procedure ID as c_n, which happens if a procedure in the context graph contains several fold nodes and all of them contain the procedures of m and n. In such a case the reachability analysis is used to remove the contexts of m that cannot execute before context c_n.

If the slicer traverses an interprocedural edge $m \to n$ coming from context c_n, it exploits that parameter-passing nodes are mapped to the associated call, entry, exit or return nodes. The context of node m we are looking for is either the direct predecessor of c_n in the context graph or it equals c_n, which is the case if both contexts are folded in the same fold node.

The fork edges used by us to connect the context graphs are used by our timing-sensitive slicer to traverse fork and fork-in edges precisely. Let $e = m \to n$ be a fork or fork-in edge encountered by the slicer, where c_n is the current context of node n. The slicer investigates the context c connected with c_n via an outgoing fork edge. If c is a context of m, the slicer proceeds with that context. Otherwise, the traversal is rejected.

Our resulting algorithm

Algorithm 3.12 illustrates, in combination with Algs. 3.4, 3.11 and 3.13, our repaired and optimized variant of Nanda's slicer. It no longer needs two-phase slicing for the thread-local slicer because it retrieves the context reached by an interprocedural traversal directly from the context graphs and remains context-sensitive that way. Strictly speaking, the outer loop could be omitted, too; however, its usage partitions the slice into a sequence of thread-local slices, which greatly alleviates debugging, so we recommend to implement it that way.

Since this algorithm turns out to be the most performant in terms of precision and runtime costs, its correctness is proven by the following theorem.

Theorem 3.3. *Let G be a CSDG and S be a slice of G for node s computed by Alg. 3.12. The following holds for every node n in G:*

$$\exists c_n \to^*_{ts} c_s \text{ in } G \Rightarrow n \in S.$$

Algorithm 3.13 `insert`: Manages the updating of the worklists.

Input: A state tuple Γ_{old}, a context c_m, a worklist W, a set M.

```
/* Create an updated state tuple. */
Γ_m = update(c_m, Γ_old)  // call Alg. 3.4

/* Run the restrictive state tuple optimization. */
for all (m, c_m, Γ'_m) ∈ M :
  if ∄ thread θ : ¬reaches(Γ_m[θ], Γ'_m[θ])
    return  // the new element is redundant

/* Insert the configuration into the worklist and mark it as visited. */
W = W ∪ {(m, c_m, Γ_m)}
M = M ∪ {(m, c_m, Γ_m)}
```

Proof. We have to show that the slicer visits context c_n, which we do by a backward iteration over path $\Phi = c_n \rightarrow_{ts}^{*} c_s$. Assume that the iteration arrives at a context edge $c_m \rightarrow c$, then we have to show three properties to conclude that the slicer traverses that edge and visits context c_m. First, worklist W_{inner} must at some point contain a configuration of c. Second, the slicer must be able to detect the context edge. And third, it must be able to add a configuration of c_m to W or W_{inner}.

Since the initialization of the algorithm is bound to add a configuration of c_s to worklist W, we have a starting point for our iteration. Now, take the next context edge $c_m \rightarrow c_o$ in Φ. We know that worklist W_{inner} contains at some point at least one configuration (o, c_o, Γ) of c_o. Since the number of possible configurations is finite and each configuration can be added to the worklists at most once, the configuration is eventually processed by the thread-local slicer. Trivially, the slicer finds the CSDG edge $e = m \rightarrow o$ associated with the context edge. It remains to show that it is able to retrieve context c_m, which is shown by a case distinction over the kind of edge e.

- e is an interference edge
  Let $\theta(c_m)$ be c_m's thread. First, lemma 3.2 shows that $\theta(c_m)$ has not been deactivated by our update mechanism during the hitherto traversal of the path. Thus, the state of c_m's thread in Γ is either $\perp$ or a context d on the subpath of Φ starting behind c_m or a context d' reachable by d (the latter case accounts for the restrictive state tuple optimization). If the state is $\perp$, then the slicer is trivially able to find c_m. Otherwise, the update mechanism guarantees that d stems from $\theta(c_m)$ and that $\theta(c_m)$ is not a multi-thread. Since Φ is timing-sensitive, c_m reaches d (and d', since 'reaches' is transitive). Therefore, the slicer is able to find c_m.

- e is a fork or fork-in edge
  The slicer retrieves the context c that is connected with c_n via an outgoing fork edge. If c is a context of m, the slicer proceeds with that context. Thus, the slicer is able to find c_m.

- e is a join-out edge
  The slicer proceeds with all contexts of m.

- e is an intra-procedural edge
  Context c_m must stem from the same procedure as c_o and must be able to reach c_o. Otherwise, the dependence denoted by e would not exist. Thus, the slicer is able to find c_m.

- e is an interprocedural edge
  Since the context of a parameter-passing node is mapped to the context of the associated call-, return-, start- or exit node, context c_m is either a direct predecessor of c_o in the context graph or, if the contexts are folded in the same fold node, it equals c_o. Thus, the slicer is able to find c_m.

Having found c_m, the slicer can proceed in two different ways: it embeds c_m in some configuration and either adds it to W or W_{inner} or rejects it due to the restrictive state tuple optimization, which means that a configuration of c_m has been added to W or W_{inner} sometime earlier. Either way, at least one configuration of c_m has been added to W or W_{inner}.

Since path Φ is finite, the iteration eventually reaches the first edge in Φ and the slicer adds a configuration of c_n to W or W_{inner} and thus, n to the slice S. □

The algorithm is not completely timing-sensitive for general MHP information, due to the problem with the prepending property described in section 3.7. Furthermore, join-out edges and intra-procedural dependences are treated conservatively. Concerning intra-procedural dependences, the slicer proceeds with all contexts of the source node of the traversed dependence edge which reach the current context c. As stated before, there may in fact be several contexts of the source node with the same procedure ID as c, which happens if a procedure in the context graph contains several fold nodes and all of them contain the two nodes connected by the dependence edge. Here the algorithm may consider contexts of the source node as valid that are not connected with c via an outgoing context edge.

3.11. Trading Precision for Speed: The Timing-Aware Slicer

The presented timing-sensitive slicers show that the handling of contexts is difficult and expensive. Therefore, we present and investigate another algorithm, whose purpose is to trade precision for speed and simplicity. It represents the execution states of threads only by nodes and thereby circumvents the handling of contexts. Therefore, it has a much smaller asymptotic runtime complexity of $\mathcal{O}(N^t)$. The algorithm assumes that all threads may happen in parallel to each other and thus does not need a MHP analysis. In order to compute correct slices, it only requires the results of a thread invocation analysis. A weaker form of the restrictive state

Algorithm 3.14 The timing-aware slicer.

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

$\Gamma_s = (\bot, \ldots, \bot)$ *// initial state tuple (one entry for each thread)*
if $\theta(s)$ is not a multi-thread
  $\Gamma_s = [s/\theta(s)]\Gamma_0$ *// set the state of* $\theta(s)$ *to* s

$W = \{(s, \Gamma_s)\}$ *// the main worklist*
$R = \{(s, \Gamma_s)\}$ *// store the visited worklist elements*
$M = \{s \mapsto true\}$ *// maps visited nodes to true (phase 1) or false (phase 2)*

repeat
  $W = W \setminus \{(n, \Gamma)\}$
  for all $m \rightarrow_e n$
    if $e \in \{fork, fi, jo\} \vee e == id \wedge (\theta(m) \neq \theta(n) \vee \theta(n)$ is a multi-thread$) \wedge m$ reaches $\Gamma(\theta(m))$
      /* Lazy updating of state tuples. */
      $\Gamma' = [n/\theta(n)]\Gamma$ *// remember where n's thread has been left*
      if $\theta(m)$ is not a multi-thread
        $\Gamma_m = [m/\theta(m)]\Gamma'$ *// update state tuple*

      /* Use a weaker form of the restrictive state tuple optimization. */
      if $\nexists (m, \Gamma'_m) \in R : \Gamma_m$ is restrictive to Γ'_m
        // the state tuple is not restrictive
        $W = W \cup \{(m, \Gamma_m)\}$
        $R = R \cup \{(m, \Gamma_m)\}$
        $M = M \cup \{m \mapsto true\}$

    else if $m \notin$ dom $M \vee M(n) \wedge \neg M(m)$
      // at thread-local edges we basically perform two-phase slicing
      if $M(n) \vee e \notin \{pi, c\}$
        // since all threads are deemed to happen in parallel,
        // state tuples have only to be updated after switching threads
        $W = W \cup \{(m, \Gamma\}$
        if $M(n) \wedge e = po$
          $M = M \cup \{m \mapsto false\}$
        else
          $M = M \cup \{m \mapsto M(n)\}$
until $W = \emptyset$

return dom M

tuple optimization based on reachability between nodes is used to identify and omit redundant configurations. Algorithm 3.14 presents pseudocode of this *timing-aware slicer*, which is a modification of the iterated two-phase slicer. We are not aware of any previous work describing a similar algorithm.

3.12. Evaluation

We have implemented the presented slicing algorithms in Java. The implementations work on CSDGs computed by ValSoft/Joana. Our evaluation investigates the impact and practicability of our MHP analysis (section 3.12.1), the precision and runtime behavior of the presented slicing algorithms (sections 3.12.2, 3.12.3 and 3.12.5) and the practicability of context graphs (section 3.12.4). The evaluation was done on a 2.2Ghz Dual-Core AMD workstation with 32 GB of memory running Ubuntu 8.04 (Linux version 2.6.24) and Java 1.6.0. Each slicer had a working memory of 8 GB at his disposal.

Our benchmark consists of 33 programs, whose statistics are shown in Table 3.1. The programs in the upper part stem from the Bandera [1] benchmark and solve a certain task in a concurrent manner. For example, LaplaceGrid solves the Laplace equation over a rectangular grid. The programs in the middle part stem from the Java Grande multi-threaded benchmark [3], from which we took only the most advanced versions of the programs (size 'B' or 'C'). The programs in the lower part are real JavaME [4] applications taken from the SourceForge repository[2], which use threads for realizing graphical user interfaces. Table 3.1 reports the number of nodes, edges and procedures in their CSDGs[3]. Column 'LOC' shows how many lines of code of the source code and of library code the CSDGs envelope. The numbers for the library code were retrieved by analyzing the source code information present in the Java bytecode. Column 'Thread Classes' shows how many different thread classes a program contains (subclasses of `java.lang.Thread`, plus the main thread). The entries in column 'Single Threads' denote for each thread class the number of unique threads that may exist at runtime, the entries in column 'Multi-threads' denote the number of identified multi-threads. The total number of distinguished threads is given in the last column. For example, KnockKnock contains three thread classes. The first one is the main thread, of which one single instance exists at runtime (the first entry in the tuple in column 'Single Threads'). The second thread class is instantiated inside a loop, wherefore the second entry in the tuple in column 'Multi-threads' indicates the existence of one multi-thread. There exists one single instance of the third thread class (the last entry in the tuple in column 'Single Threads'). In total, our analysis distinguishes three threads in KnockKnock. Note that the existence of both single threads and multi-threads of the same thread class is absolutely possible, albeit not happening in our benchmark programs.

Several of these programs have been used in our previous publications [46, 43], partly with very different statistics. The CSDGs in [46] were computed by Joana's old CSDG generator based on the Harpoon framework [65]. Since that framework is no longer maintained, Jürgen Graf rewrote our generator from scratch [49] on top of the WALA framework [5]. Besides from being built on different frameworks, the generators differ in the way how parameter objects

[2] `http://sourceforge.net/`

[3] The number of edges are higher than the ones given in Table 2.1, because there concurrency edges were ignored.

Table 3.1.: Statistics of our benchmark programs.

Name	LOC			Nodes	Edges	Procs.	Thread Classes	Single Threads	Multi-threads	Total
	src		lib							
Example	29	+	313	2509	36411	89	2	(1,1)	(0,0)	2
ProdCons	83	+	335	3331	39614	100	2	(1,0)	(0,2)	3
DiskScheduler	220	+	457	4389	43645	133	2	(1,0)	(0,1)	2
AlarmClock	187	+	366	4781	44952	124	3	(1,2,1)	(0,0,0)	4
DiningPhils	90	+	519	5115	125377	116	2	(1,0)	(0,1)	2
LaplaceGrid	175	+	531	6175	50876	161	2	(1,0)	(0,1)	2
SharedQueue	357	+	1138	11284	78797	199	2	(1,2)	(0,0)	3
EnvDriver	2677	+	472	19106	181428	180	2	(1,1)	(0,0)	2
KnockKnock	592	+	2471	35852	312678	506	3	(1,0,1)	(0,1,0)	3
DaisyTest	1114	+	2340	43138	437937	527	2	(1,0)	(0,1)	2
DayTime	371	+	4407	59708	632146	696	2	(1,1)	(0,0)	2
ForkJoin	326	+	1265	16972	59921	194	2	(1,0)	(0,1)	2
Sync	370	+	1261	17315	62033	194	2	(1,0)	(0,2)	3
Barrier	421	+	1265	17579	62471	199	2	(1,0)	(0,2)	3
Series	422	+	1277	17709	61014	211	2	(1,0)	(0,1)	2
LUFact	749	+	1271	18888	65449	214	2	(1,0)	(0,1)	2
SOR	431	+	1388	19151	66418	234	2	(1,0)	(0,1)	2
SparseMatmult	432	+	1390	19502	67646	236	2	(1,0)	(0,1)	2
Crypt	557	+	1386	19973	69547	236	2	(1,0)	(0,2)	3
MolDyn	777	+	1278	22373	90331	222	2	(1,0)	(0,1)	2
RayTracer	943	+	1316	23481	90346	268	2	(1,0)	(0,1)	2
MonteCarlo	1422	+	2129	35009	150313	460	2	(1,0)	(0,1)	2
Logger	279	+	1165	10333	52307	227	2	(1,1)	(0,0)	2
Maza	921	+	1125	11235	67677	250	2	(1,1)	(0,0)	2
Barcode	783	+	1242	12344	64674	271	2	(1,1)	(0,0)	2
Guitar	761	+	1256	13257	68684	296	2	(1,1)	(0,0)	2
J2MESafe	512	+	1621	17754	126409	309	2	(1,1)	(0,0)	2
HyperM	366	+	1332	17768	97575	277	3	(1,1,4)	(0,0,0)	6
Podcast	2012	+	1965	23576	159116	407	3	(1,1,1)	(0,0,0)	3
GoldenSMS Key	1139	+	1736	21860	177911	362	2	(1,1)	(0,0)	2
GoldenSMS Msg	900	+	1913	26333	215399	414	3	(1,2,1)	(0,0,0)	4
GoldenSMS Rec	695	+	1796	22088	149039	370	2	(1,1)	(0,0)	2
Cellsafe	3024	+	2137	41707	862654	534	2	(1,1)	(0,0)	2

are treated (object trees vs object graphs), in the treatment of library code (the old one uses stubs for all `java.lang` classes, see [52], chapt. 6) and in the available points-to analyses. In [43], we used the old CSDG generator for the programs also present in [46] to keep the results comparable. These were the programs Example, ProdCons, DiningPhils, LaplaceGrid, AlarmClock and SharedQueue. The other CSDGs, particularly those for the JavaME programs, were built by the new generator because the old one could not manage them. The CSDGs used

in this evaluation are all computed by the new generator. As it has been developed further since the time the evaluation in [43] was done, the current CSDGs of these programs also differ slightly from those used there.

There are some limitations to the CSDGs of the JavaME programs. Their computation did not include all used libraries because the graphs would have grown too big. This means that some parts of the programs are missing in the TCFGs and CSDGs because some callbacks from the library to the programs are missed.

3.12.1. The MHP Analysis

The first part of the evaluation investigates the costs and benefits of our MHP analysis.

Pruning spurious interference edges

Joana's CSDG generator uses a wide-spread trick to safely approximate the interference dependences in concurrent programs [52]: It assumes that a program has two threads of each thread class at runtime, except for the main thread, which is unique. Furthermore, all threads of the program are assumed to happen in parallel. These assumptions make sure that all possible interference dependences between threads of different thread classes and between threads of the same thread class are included. In a postprocessing step, we use our MHP analysis to identify and remove redundant interference edges. Table 3.2 shows the impact of that refinement on our benchmark. On average, about 75% of all interference edges could be removed. Program EnvDriver showed the most drastic impact – all of the previously 1661 interference edges have been removed. An inspection of its code reveals that the sole purpose of EnvDriver's main thread is to invoke the other thread, in which all the work is happening, so in fact there are no interference dependences between these two threads. Since the CSDG generator assumed that the program contains two concurrently executing instances of the second thread class, it computed 1661 interference edges between the statements of that class. An interesting pattern appears in the lower part of the Table: The JavaME programs Maza, Barcode, Guitar, J2MESafe, GoldenSMS Key, GoldenSMS Rec and Cellsafe have identical interference edges. They result from JavaME initialization code present in all of these programs. Similar to EnvDriver, these programs do their work mainly in the second thread, whereas the main thread primarily serves for initialization.

Table 3.2 furthermore shows that our MHP analysis is performant enough for our benchmark programs. The execution times, which also include the thread invocation analysis, ranged between 0.1 and 2.3 seconds. The main reason for that performance is that the number of thread regions and of the distinguished threads is small compared to the size of the programs.

There exist other evaluations of interference dependence computations that use the Java Grande benchmark. Ranganath and Hatcliff [116, 117] explain how escape analysis can be

Table 3.2.: Costs and effects of our MHP analysis.

Name	Number of interference edges before	after		Number of thread regs.	TCFG nodes	Runtime (in sec.)
Example	8	4	(50%)	9	966	.2
ProdCons	1383	175	(13%)	205	1088	.2
DiskScheduler	1583	338	(21%)	277	1484	.1
AlarmClock	1805	309	(17%)	204	1375	.1
DiningPhils	3164	225	(7%)	252	2262	.1
LaplaceGrid	2009	496	(25%)	391	1881	.2
SharedQueue	3060	754	(25%)	407	3242	.2
EnvDriver	1661	0	(0%)	7	8694	.2
KnockKnock	10320	1262	(12%)	2010	9023	.6
DaisyTest	11873	2619	(22%)	1935	7985	1.2
DayTime	52777	3807	(7%)	1619	11857	.9
ForkJoin	3766	220	(6%)	708	9501	.6
Sync	3819	266	(7%)	732	9620	.6
Barrier	3240	264	(8%)	715	9683	.7
Series	42	34	(81%)	732	9886	.5
LUFact	1334	806	(60%)	732	10787	.6
SOR	1005	181	(18%)	834	10542	.6
SparseMatmult	395	25	(6%)	846	10760	.6
Crypt	839	169	(20%)	862	11054	.6
MolDyn	12649	6161	(49%)	788	12870	.9
RayTracer	6754	5382	(80%)	1023	11648	1.0
MonteCarlo	4954	523	(11%)	1627	15000	2.3
Logger	280	32	(11%)	681	3118	.5
Maza	279	5	(2%)	662	3303	.3
Barcode	279	5	(2%)	846	3728	.3
Guitar	279	5	(2%)	868	4298	.3
J2MESafe	279	5	(2%)	977	4611	.3
HyperM	9208	8139	(88%)	623	3942	.4
Podcast	3140	128	(4%)	1316	5809	.5
GoldenSMS Key	279	5	(2%)	1164	5081	.4
GoldenSMS Msg	6114	4661	(76%)	1318	5972	.7
GoldenSMS Rec	287	5	(2%)	1172	5457	.7
Cellsafe	279	5	(2%)	1812	10424	1.0
Total	149143	37015	(25%)			

used to prune spurious interference dependences. They use a variation of Ruf's thread allocation analysis [123] to determine which threads may exist at runtime and to detect multi-threads. This information is combined with an escape analysis to determine which objects are shared between which threads. Only these objects are considered by their computation of interference dependences. However, their approach does not include a MHP analysis, hence all threads are assumed to happen in parallel to each other. Hammer [52] compared their interference com-

Table 3.3.: The number of interferences in the Java Grande benchmark computed by Ranganath and Hatcliff [116], Hammer [52] and us. The Table has been taken from page 167 of Hammer's PhD thesis [52] and extended with our results.

Name	Results from [116]	Results from [52]	Our results	Real numbers
ForkJoin	16	11	0	0
Sync	36	26	46	6
Barrier	29	17	26	7
Series	8	8	32	0
LUFact	24	276	750	139
SOR	23	115	179	85
SparseMatmult	14	11	23	1
Crypt	58	95/31	135	0
MolDyn	209	5288	5924	?
RayTracer	166	221	1133	?
MonteCarlo	11	126	157	?

putation with that in the old Joana CSDG generator, which differed from theirs by using our MHP analysis, by working on CSDGs with Hammer's object trees as highly precise representations of parameter objects and by using a context-sensitive points-to analysis. He used the same benchmark as Ranganath and Hatcliff for that purpose and concluded that both techniques yield a similar gain of precision on average. Table 3.3 summarizes the Table on page 167 in Hammer's PhD thesis [52] and extends it with our results. It shows per program the number of interferences computed by Ranganath and Hatcliff, by Joana's old CSDG generator as reported by Hammer and by the new generator. The rightmost column depicts the real number of interference dependences in the program as identified by Hammer. The numbers refer only to the interference dependences in the source code of the programs and exclude those in library code, wherefore our numbers differ from those in Table 3.2.

The comparison shows that the interference dependence computation in the new CSDG generator is less precise than that in the old one, the only exception being program ForkJoin. We identified Joana's points-to analysis as the causer of that imprecision. The points-to analysis we used is an Andersen-style analysis following a 0-1-CFA policy, which means that it distinguishes the allocation sites of objects, but is context-insensitive and does not distinguish different contexts of one allocation site. This may lead to the situation where local objects in different threads are not distinguished by the points-to analysis and are later interpreted as shared variables by the computation of interference dependence. This conclusion is supported by Hammer's evaluation, who noted that context-sensitivity has an important influence on the precise identification of shared objects. A context-sensitive points-to analysis is currently not

Table 3.4.: Precision of the thread invocation analysis.

	Real number of		Our results		
Name	classes	threads	classes	threads	mult.
Example	2	(1,1)	2	(1,1)	(0,0)
ProdCons	2	(1,6)	2	(1,0)	(0,2)
DiskScheduler	2	(1,3)	2	(1,0)	(0,1)
AlarmClock	3	(1,2,1)	3	(1,2,1)	(0,0,0)
DiningPhils	2	(1,5)	2	(1,0)	(0,1)
LaplaceGrid	2	(1,2)	2	(1,0)	(0,1)
SharedQueue	2	(1,2)	2	(1,2)	(0,0)
EnvDriver	2	(1,1)	2	(1,1)	(0,0)
KnockKnock	3	(1,15,1)	3	(1,0,1)	(0,1,0)
DaisyTest	2	(1,4)	2	(1,0)	(0,1)
DayTime	2	(1,1)	2	(1,1)	(0,0)
ForkJoin	2	(1,∞)	2	(1,0)	(0,1)
Sync	2	(1,∞)	2	(1,0)	(0,2)
Barrier	2	(1,∞)	2	(1,0)	(0,2)
Series	2	(1,∞)	2	(1,0)	(0,1)
LUFact	2	(1,∞)	2	(1,0)	(0,1)
SOR	2	(1,∞)	2	(1,0)	(0,1)
SparseMatmult	2	(1,∞)	2	(1,0)	(0,1)
Crypt	2	(1,∞)	2	(1,0)	(0,2)
MolDyn	2	(1,∞)	2	(1,0)	(0,1)
RayTracer	2	(1,∞)	2	(1,0)	(0,1)
MonteCarlo	2	(1,∞)	2	(1,0)	(0,1)
Logger	2	(1,1)	2	(1,1)	(0,0)
Maza	2	(1,1)	2	(1,1)	(0,0)
Barcode	3	(1,1,1)	2	(1,1)	(0,0)
Guitar	3	(1,1,1)	2	(1,1)	(0,0)
J2MESafe	2	(1,1)	2	(1,1)	(0,0)
HyperM	6	(1,1,4,1,1,1)	3	(1,1,4)	(0,0,0)
Podcast	3	(1,1,1)	3	(1,1,1)	(0,0,0)
GoldenSMS Key	3	(1,1,1)	2	(1,1)	(0,0)
GoldenSMS Msg	3	(1,1,1)	2	(1,1)	(0,0)
GoldenSMS Rec	2	(1,1)	3	(1,1,2)	(0,0)
Cellsafe	3	(1,1,2)	2	(1,1)	(0,0)

available in Joana. An alternative could be a thread-sensitive points-to analysis [99], which is tailored to precisely identify shared variables.

The approach of Ranganath and Hatcliff seems to be superior for the bigger programs. However, Hammer stated that their technique does not account for interferences resulting from array access, wherefore their results for programs LUFact and SOR are smaller than the real number of interference dependences. It is not clear if this explains their much smaller results for the last three programs, because these are too big for a manual examination.

Thread invocation analysis

Table 3.4 compares the results of our thread invocation analysis with the manually determined number of threads that may exist at runtime in the programs. Some interesting cases are described closer.

Program ProdCons has no specific thread classes for the producer and consumer threads but uses the `Runnable` interface to realize them and instantiates each one thread with the producer- and consumer-`Runnable`. This happens inside a loop, which is executed three times. The analysis approximates the loop by recognizing both invocations as multi-threads. DiskScheduler's main thread creates threads of the other thread class in a loop that is executed three times, which are approximated as a multi-thread by the analysis. DiningPhils starts five philosopher-threads in a loop, which are subsumed by one multi-thread. LaplaceGrid starts a worker thread in a loop that is executed twice, which is subsumed by one multi-thread. KnockKnock has three thread classes: the class of the main thread, a TCP server and a class for service threads maintaining the communication with a client. The main thread starts one TCP server that holds 5 – 15 service threads. Since the service threads are started inside a loop, the analysis subsumes them by a multi-thread. DaisyTest starts 4 threads in a loop, which are subsumed by a multi-thread.

Most programs from the Java Grande benchmark consist of a main thread and a second thread class and start in a loop a user-defined number of threads of that class. Amongst them are ForkJoin Barrier, Series, LUFact, SOR, SparseMatmult, MolDyn, RayTracer and MonteCarlo. Our analysis identifies these threads as multi-threads. Sync uses the `Runnable` interface to create two different kinds of threads in a loop. Our analysis reports two multi-threads, which distinguish these two kinds of threads. Crypt starts a user-defined number of threads in one loop to encrypt and later the same number of threads do decrypt. Again, our analysis reports two multi-threads.

The results for the JavaME programs show the current limitations of our analysis. At some point the TCFGs of these programs cut the called libraries off, so several threads resulting from callbacks are missing. All investigated JavaME programs contain at least a main thread and an event-handler thread. Several programs, Logger, Maza, Barcode, J2MESafe and GoldenSMS Rec, consist only of these threads. Podcast contains a third thread, whose task is to update clients. GoldenSMS Msg uses a third thread to create a text box for SMS messages. Barcode uses a media player from the Java ME library that internally uses threads, but is missing in the TCFG. Guitar contains a thread that is missing due to a missing callback from the event handler thread. In GoldenSMS Key, a file chooser implemented as a thread is missing. Cellsafe uses a thread to import and export stored data, which is missing. In HyperM, even three thread classes are missing, each of which is instantiated once.

Table 3.5.: Runtime costs in seconds of the MHP analyses of Li and Verbrugge [87], of Barik [18] and of us.

	Results from [87]			Results from [18]			Our results	
Name	Threads	PEG nodes	Time	Threads	ICFG nodes	Time	TCFG nodes	Time
ForkJoin	4	109	.61	–	–	–	9501	.6
Sync	5	255	9.28	–	–	–	9620	.6
Barrier	5	262	21.61	–	–	–	9683	.7
Series	3	109	.91	2	53+21	.01	9886	.5
LUFact	3	105	.80	2	57+57	.03	10787	.6
SOR	3	101	.72	2	48+69	.03	10542	.6
SparseMat.	3	85	.32	2	68+20	.01	10760	.6
Crypt	5	121	1.55	3	52+61+61	.08	11054	.6
MolDyn	3	144	3.22	2	280+758	13.12	12870	.9
RayTracer	3	211	7.10	2	387+221	2.03	11648	1.0
MonteCarlo	3	132	.84	2	520+316	3.19	15000	2.3

Comparison with Li and Verbrugge and with Barik

Li and Verbrugge [87] used the Java Grande benchmark to evaluate their MHP analysis. We summarized their analysis in section 3.3.1. The left part of Table 3.5 presents some of their results. Their technique requires a fixed number of threads, which explains the values in column 'Threads'. Column 'PEG nodes' denotes the number of nodes in their *parallel execution graphs*, on which the MHP analysis takes place. Column 'Time' shows the execution times of their analysis in seconds. It consists of the time needed to create the PEGs and to compute the MHP information. The execution times cannot be compared directly with ours because they used different (slower) hardware, and they did not include the libraries called within the programs. However, it seems that their analysis starts to get expensive if the PEG of a program exceeds 200 nodes. If the impact of synchronization on the MHP information can be neglected, our MHP analysis is a promising alternative.

Barik [18] also used parts of the Java Grande benchmark to evaluate his MHP analysis. The right part of Table 3.5 presents some of his results. It shows the number of threads distinguished by his analysis, which are the same as identified by our analysis. Again, the execution times cannot be compared directly with ours because he did not include the libraries called within the programs. However, the execution times seem not to scale well. There is a significant discrepancy between the times for the biggest three programs and for the other programs.

Nanda [106] does not provide an evaluation of her thread region analysis.

Table 3.6.: Table of features.

Name	MHP analysis	Runtime complexity	Capable of multi-threads	Granularity of state tuples
I2P	all threads parallel	$\mathcal{O}(\|E\|)$	yes	none
S	all threads parallel	$\mathcal{O}(\|N\|^t)$	yes	threads
K	all threads parallel	$\mathcal{O}(\|N\|^{pt})$	no	threads
GK	thread regions	$\mathcal{O}(\|N\|^{pt})$	yes	threads
GN	thread regions	$\mathcal{O}(\|N\|^{pt})$	yes	threads
GNR	thread regions	$\mathcal{O}(\|N\|^{pt})$	yes	threads
GNP	all threads parallel	$\mathcal{O}(\|N\|^{pt})$	yes	threads
GNT	thread regions	$\mathcal{O}(\|N\|^{pt})$	yes	thread regions
N	thread regions	$\mathcal{O}(\|N\|^{pt})$	loops only	thread regions

	Further remarks
I2P	Alg. 3.2; is timing-insensitive
S	Alg. 3.14; uses context-insensitive state tuples
K	Alg. 3.5
GK	Alg. 3.8
GN	Alg. 3.12
GNR	Alg. 3.12, but uses reachability analysis after each edge traversal
GNP	Alg. 3.12, but assumes all threads happen in parallel to each other
GNT	Alg. 3.12, but with Nanda's state tuple mechanism
N	Alg. 3.9, but without the faulty restrictive state tuple optimization

3.12.2. Comparison of Timing-Sensitive Slicers

This part of the evaluation investigates which one of the presented timing-sensitive slicing algorithms is the most practical. It compares the precision and runtime behavior of the following algorithms: The iterated two-phase slicer (I2P), whose results serve as a measure of quality, the timing-aware slicer of section 3.11 (S), Krinke's original algorithm (K) and our optimized version (GK), and several variants of Nanda's algorithm. The first variant, GN, is Alg. 3.12 and contains all our suggested optimizations. We inspect three other variants, each of which omits one of our major optimizations: Variant GNT uses Nanda's update mechanism for state tuples, variant GNP assumes that all threads happen in parallel to each other and variant GNR uses reachability analysis after each edge traversal to determine the reached contexts. Finally, we inspect a version of Nanda's original algorithm, N, in which the faulty application of the restrictive state tuple optimization has been repaired. Table 3.6 summarizes the algorithms.

We use a different benchmark for this comparison, because Krinke's original algorithm needs a statically bound number of threads and because several of the investigated algorithms do not scale well enough. It is described in Table 3.7 and consists of variations of the programs Example, ProdCons, DiskScheduler, LaplaceGrid and AlarmClock, in which the information about the existing threads has been manipulated. Our CSDGs and TCFGs store the results

Table 3.7.: The programs of the benchmark.

Name	Nodes	Edges	Procs.	Thread classes	Single threads	Total
Example-1	2509	36411	89	2	(1,1)	2
Example-2	2509	36411	89	2	(1,2)	3
Example-3	2509	36411	89	2	(1,3)	4
ProdCons-1	3331	39614	100	2	(1,2)	3
ProdCons-2	3331	39614	100	2	(1,4)	5
ProdCons-3	3331	39614	100	2	(1,6)	7
DiskScheduler-1	4389	43645	133	2	(1,1)	2
DiskScheduler-2	4389	43645	133	2	(1,2)	3
DiskScheduler-3	4389	43645	133	2	(1,3)	4
LaplaceGrid-1	6175	50876	161	2	(1,1)	2
LaplaceGrid-2	6175	50876	161	2	(1,2)	3
LaplaceGrid-3	6175	50876	161	2	(1,3)	4
AlarmClock-1	4781	44952	124	3	(1,2,1)	4
AlarmClock-2	4781	44952	124	3	(1,4,2)	7
AlarmClock-3	4781	44952	124	3	(1,6,3)	10

of the thread invocation analysis in form of annotations. The MHP analysis and the slicing algorithms process these annotations and trust their accurateness. By manually changing these annotations we transformed the multi-threads into single threads. In order to observe how the algorithms cope with combinatorial explosion of state tuples, we artificially raised the number of threads. In the programs with a '-1' suffix, the multi-threads have been converted to single threads. In the programs with a '-2' suffix, the numbers of all threads, except for the main thread, have been duplicated, and in the programs with a '-3' suffix, the numbers of all threads, except for the main thread, have been triplicated. Our implementation of CSDGs contains one SDG per thread class and not one for each existing thread, so it was not necessary to modify the original CSDGs. Thus, the three CSDGs of one program have the same number of nodes and edges. Different behavior of a slicing algorithm for different versions of one program must be caused by the different numbers of threads.

The results

For each program we randomly determined 100 slicing criteria, computed the slices with each algorithm and measured the size of the slices and the computation times. Table 3.8 shows the average size per slice in number of nodes, Table 3.9 shows the average computation time per slice in seconds. Missing values mean that the corresponding slicer was not able to compute the slices in reasonable time (denoted by '–') or ran out of memory (denoted by '*').

Table 3.8.: Average size per slice in number of nodes.

Name	I2P	S	K	GK	GN, GNR	N, GNT	GNP
Example-1	900.55	900.55	900.55	900.55	900.55	900.55	900.55
Example-2	900.55	900.55	900.55	900.55	900.55	900.55	900.55
Example-3	900.55	900.55	900.55	900.55	900.55	900.55	900.55
ProdCons-1	1532.74	1532.02	–	1358.60	1358.60	1452.74	1513.92
ProdCons-2	1532.74	1532.74	–	1358.60	1358.60	1453.46	1514.64
ProdCons-3	1532.74	1532.74	–	–	1358.60	1453.46	1514.64
DiskScheduler-1	2059.74	1895.37	–	1628.02	1628.02	1676.82	1878.93
DiskScheduler-2	2059.74	2036.54	–	1628.02	1628.02	1817.67	2020.10
DiskScheduler-3	2059.74	2036.62	–	–	1628.02	1817.75	2020.18
LaplaceGrid-1	3445.53	3420.37	–	2667.37	2667.37	3094.81	3259.80
LaplaceGrid-2	3445.53	3445.53	–	–	2775.17	–	3369.70
LaplaceGrid-3	3445.53	3445.53	–	–	2775.17	–	3369.85
AlarmClock-1	2807.46	2792.79	–	–	2652.17	–	2754.92
AlarmClock-2	2807.46	2792.80	–	–	–	*	–
AlarmClock-3	2807.46	2792.80	–	–	–	*	–

Precision Table 3.8 shows that the investigated algorithms are able to compute distinctly smaller slices than the iterated two-phase slicer, I2P. The slices computed by our most precise slicer, GN, were on average about 13% smaller than the ones computed by I2P. In the best case, for LaplaceGrid-1, the slices were even about 22% smaller. The results also show that the way how MHP information is integrated has a significant impact on the precision. The slices computed by algorithm GNP, which assumes that all threads happen in parallel to each other, were on average only 2% smaller than the I2P slices. Algorithm GN was also more precise than Nanda's original algorithm, N, its slices being on average 5% smaller. The reason for that difference is that her state tuple mechanism may require reachability analyses between contexts from different threads and her reachability analysis is context-insensitive in these cases. Our improved algorithm circumvents that imprecision, because our update mechanism for the state tuples ensures that the reachability analysis is always thread-local and our may-exist analysis is context-sensitive. Algorithms GK and GNR computed the same slices as GN, because they also use our state tuples mechanism. Algorithms GNT and N computed the same slices, because they both use Nanda's mechanism. The slices computed by algorithm S were only marginally smaller than those computed by I2P, except for program DiskScheduler-1, where they were 8% smaller. This indicates that the context of the execution states of threads is important for detecting timing-insensitivity.

It is remarkable that an increasing number of threads decreases the benefit of timing-sensitive slicing, whereas the computation times rise significantly: The more threads of a thread class exist, the more interference edge traversals find a thread that is in a suitable execution state.

Table 3.9.: Average execution time per slice in seconds.

Name	I2P	S	K	GK	GN	GNR	GNT	N	GNP
Ex-1	.007	.008	.452	.006	.006	.016	.007	.017	.007
Ex-2	.006	.008	.625	.008	.007	.019	.008	.021	.011
Ex-3	.007	.009	1.006	.010	.009	.021	.013	.017	.008
PC-1	.009	.203	–	41.093	.062	.734	8.852	10.497	.165
PC-2	.009	.646	–	655.975	.217	.872	49.235	53.622	.820
PC-3	.009	3.851	–	–	.745	1.480	211.617	213.983	3.289
DS-1	.010	.094	–	29.866	.119	39.511	47.825	174.426	.602
DS-2	.010	.335	–	252.562	.134	38.845	90.319	270.657	1.793
DS-3	.010	.670	–	–	.146	40.426	227.507	556.789	4.403
LG-1	.019	.369	–	384.564	2.992	81.259	896.453	1618.964	9.860
LG-2	.021	1.051	–	–	13.380	194.790	–	–	65.736
LG-3	.022	1.512	–	–	21.647	288.537	–	–	407.848
AC-1	.015	.562	–	–	54.434	–	–	–	78.750
AC-2	.017	2.517	–	–	–	–	*	*	–
AC-3	.016	15.913	–	–	–	–	*	*	–

Runtime behavior The average execution times presented in Table 3.9 foremost show that timing-sensitive slicing is expensive and cannot compete with the I2P slicer. They also show that among all timing-sensitive slicers our version of Nanda's algorithm, algorithm GN, is the most performant.

The results of the different variants of Nanda's algorithm in this comparison show the effectiveness of our new optimizations. Foremost, it pays off to have one entry per thread in the state tuples instead of having one entry per thread region. This can be seen by comparing algorithms GN and GNT: Algorithm GN was on average 359 times faster than GNT; in the best case, for DiskScheduler-3, it was even 1563 times faster. This is so because the state tuples in GNT are much bigger. For example, GNT's state tuples for DiskSchelduler-3 have 279 entries, whereas GN's state tuples have only 4 entries. Another consequence is that GNT consumes much more memory than GN. For AlarmClock-2 and -3, it ran out of memory and aborted.

A comparison of algorithms GN and GNR shows that the reachability analysis after each edge traversal originally suggested by Nanda can be a serious bottleneck. Our optimization based on procedure IDs used in GN led to an overall speedup of 81.5; in the best case, for DiskScheduler-1, GN was even 330 times faster. Both effects culminate in Nanda's original algorithm, N, which contains none of our suggested optimizations and was not able to slice several of our programs in reasonable time. For the programs which were sliced by both GN and N, GN was on average 855 times faster. A very interesting result is that algorithm GN was even faster than algorithm GNP, which assumes that all threads happen in parallel to each other

Table 3.10.: Total number of elements visited via interference edges.

Name	I2P	S	K	GK	N	GN
Ex-1	105	106	1321	105	104	104
Ex-2	105	112	1322	107	105	105
Ex-3	105	118	1323	109	106	106
PC-1	5006	22490	–	375922	36607	15528
PC-2	4983	105782	–	1356744	135609	74170
DS-1	4951	5496	–	256494	24742	21681
DS-2	4955	27628	–	799596	61495	21681
LG-1	6785	10923	–	1182994	62504	182790

and thus has a much simpler update mechanism for state tuples. The reason seems to be the increased precision of GN.

The runtime measurements reveal that Krinke's original algorithm scales poorly. The reason is in our opinion that Krinke did not provide an implementation and thus was not able to investigate optimizations. Our improved variant, algorithm GK, is able to compete with Nanda's original algorithm. However, it cannot compete with algorithm GN. Its representation of contexts via call strings declines performance in bigger programs, because the size of the call strings can grow arbitrarily. Furthermore, it has to distinguish more contexts than GN, because the virtual inlining of procedures during the construction of the context graphs strongly reduces the number of distinguished contexts.

Iterations of the embedded thread-local slicers Table 3.10 shows for several programs and algorithms the total number of elements visited via interference edges. This number is equivalent to the number of iterations of the embedded thread-local slicers. The Table shows that the number of iterations may grow very fast if the number of threads are raised. The usage of the restrictive state tuple optimization strongly relieves this combinatorial explosion, which can be seen by comparing the numbers for K and GK. The numbers for GK are often more than ten times bigger than those for GN or N, because GK has to distinguish more contexts. Algorithm GN commits less iterations than algorithm N, because our state tuple mechanism may subsume state tuples that are distinguished by Nanda's mechanism (since ours does not distinguish thread regions). Algorithm S is even less affected, because its state tuples use nodes instead of contexts.

Interestingly, the number of iterations committed by the I2P slicer also vary for different versions of the same program, even though they should not be affected at all, since I2P does not have any state tuples at all. A repeated execution of the test for I2P even resulted in completely different numbers. The reason for that behavior is that there may exist nodes belonging to the slice which the I2P slicer may visit during phase 2 of a thread-local slice or directly via an

interference edge. It then depends on the order in which the slicer traverses the edges whether it visits such a node first via the interference edge or during phase 2 of a thread-local slice. In the first case it has to visit the node once, in the latter case, twice. Since our implementation of that algorithm (Alg. 3.2) does not prioritize different kinds of edges, this order may vary in different executions.

Summary The comparison shows that Nanda's slicing technique is much more practical than Krinke's. This is due to the more adequate context representation as single integers and the usage of context graphs, which reduce the number of distinguished contexts. Our new optimizations raised its performance even more: Algorithm GN was on average 855 times faster than the original version, N, and its computed slices were on average 5% smaller. The most important of our optimizations is our new update mechanism for state tuples, which allows to exploit MHP information and still gets by with one entry per thread in the state tuples.

Another important result of this comparison is that an employment of MHP information does not only raise precision, but may also lead to a significant speedup. Algorithm GNP, which assumes that all threads happen in parallel to each other, had only slightly smaller slices than algorithm I2P and was even slower than our most precise algorithm, GN. Therefore, we recommend to base an employment of timing-sensitive slicing on a MHP analysis and its information.

3.12.3. Precision and Runtime Behavior of Timing-Sensitive Slicing

As a consequence of the previous comparison, we investigated the algorithms I2P, S and GN in greater detail, by slicing the programs presented in Table 3.1. For that purpose, we randomly determined 1000 slicing criteria for each program, computed the slices with each algorithm and compared the size of the slices and the execution times. Table 3.11 presents on the left side the average size per size in number of nodes, and on the right side the average execution time per slice in seconds. The entries for DayTime and algorithm GN are missing, because the slicer could not finish the computation in reasonable time. For the same reason, we computed only 100 slices of KnockKnock, DaisyTest, MolDyn MonteCarlo and RayTracer.

In Table 3.12 these results are refined. The left side of the table compares the precision of the algorithms. The first column shows the ratio of the overall number of nodes in the CSDGs to the average size of the I2P slices. The other columns show the ratio of the average slice sizes of the first algorithm to those of the second algorithm given in the column title. The columns on the right side of the Table show the slowdown of the more precise algorithms compared with the less precise algorithms. Due to the missing values for GN, program DayTime is omitted.

Precision The I2P slices contained on average about 44% of all nodes of the program, which is a significantly larger share than observed for context-sensitive slices of sequential programs

Table 3.11.: Average size per slice in number of nodes (left side), and average execution time per slice in seconds (right side).

	size per slice (nodes)			time per slice (sec.)		
Name	I2P	S	GN	I2P	S	GN
Example	867.02	866.99	866.99	.006	.007	.007
ProdCons	1497.33	1497.33	1395.71	.009	.070	.068
DiskScheduler	1974.53	1954.40	1544.10	.010	.030	.112
AlarmClock	2692.50	2680.26	2580.27	.011	.242	53.018
DiningPhils	2883.88	2882.15	1921.40	.025	.042	.169
LaplaceGrid	3353.35	3353.35	2783.88	.014	.068	10.195
SharedQueue	7441.98	7363.39	5573.03	.030	1.630	19.258
EnvDriver	8502.59	8502.59	8502.59	.047	8.503	65.021
KnockKnock	22799.23	22799.23	13884.86	.113	7.051	6323.566
DaisyTest	32625.10	32625.10	26956.22	.243	12.222	867.701
DayTime	47402.67	46619.00	–	.285	14.025	–
ForkJoin	8834.12	8834.12	3546.72	.029	.145	18.267
Sync	9107.79	9107.79	4848.33	.031	.459	23.286
Barrier	9275.34	9275.34	4723.45	.029	.377	6.202
Series	3440.31	3438.73	2755.25	.009	.012	.215
LUFact	3855.47	3853.02	2966.56	.010	.043	.271
SOR	3716.99	3716.92	2912.47	.010	.025	.252
SparseMatmult	3721.57	3721.48	2904.01	.011	.019	.273
Crypt	4251.60	4251.60	2414.78	.012	.056	1.141
MolDyn	13948.87	13948.87	6568.77	.047	6.729	409.502
RayTracer	14206.14	14206.14	6599.66	.032	1.095	333.824
MonteCarlo	21265.17	21262.29	9855.67	.094	2.783	278.504
Logger	3012.45	2945.36	2934.46	.010	.013	.013
Maza	4007.72	3981.26	3981.26	.011	.013	.021
Barcode	3356.62	2931.10	2931.10	.010	.009	.024
Guitar	3679.38	3619.26	3619.26	.011	.012	.016
J2MESafe	6409.32	6383.09	6383.09	.020	.022	.036
HyperM	9536.75	9468.15	8038.71	.037	75.574	76.285
Podcast	12170.88	11903.72	8645.34	.069	.187	.442
GoldenSMS Key	6852.48	6792.74	6792.74	.021	.029	.110
GoldenSMS Msg	12151.87	12151.29	8718.54	.050	7.829	30.390
GoldenSMS Rec	5789.61	5639.93	5639.93	.021	.023	.051
Cellsafe	23993.88	23938.91	23938.91	.218	.224	.272

(cf. sect. 2.7). Particularly, the slices for the programs from the Bandera benchmark contained on average about 54% of all nodes.

The slices computed by algorithm S were on average only about 0.8% smaller than the I2P slices. Only for the JavaME programs the algorithm was able to yield notably smaller slices, being on average about 2.2% smaller than the I2P slices. Similar to the results of section 3.12.2,

Table 3.12.: Average size ratio (in percent, left side), and slowdown (right side) per slice.

Name	size ratio (percent)				slowdown		
	I2P vs #nodes	S vs I2P	GN vs I2P	GN vs #nodes	S vs I2P	GN vs I2P	GN vs S
Example	34.6	100.0	100.0	34.6	1.2	1.2	1.0
ProdCons	44.9	100.0	93.2	41.9	7.8	7.6	1.0
DiskScheduler	45.0	99.0	78.2	35.2	3.0	11.2	3.7
AlarmClock	56.3	99.6	95.8	54.0	23.0	5027.3	218.7
DiningPhils	56.4	99.9	66.6	37.6	1.7	6.9	4.0
LaplaceGrid	54.3	100.0	83.0	45.1	4.8	721.5	151.0
SharedQueue	66.0	98.9	74.9	49.4	54.5	644.0	11.8
EnvDriver	44.5	100.0	100.0	44.5	181.1	1384.6	7.7
KnockKnock	63.6	100.0	60.9	38.7	62.5	56089.8	896.8
DaisyTest	75.6	100.0	82.6	62.5	50.2	3565.4	71.0
ForkJoin	52.1	100.0	40.2	20.9	4.9	619.9	126.1
Sync	52.6	100.0	53.2	28.0	15.0	759.6	50.7
Barrier	52.8	100.0	50.9	26.9	12.9	212.0	16.5
Series	19.4	99.9	80.1	15.6	1.4	23.8	17.4
LUFact	20.4	99.9	76.9	15.7	4.2	26.6	6.4
SOR	19.4	100.0	78.4	15.2	2.6	26.4	10.3
SparseMatmult	19.1	100.0	78.0	14.9	1.7	24.2	14.3
Crypt	21.3	100.0	56.8	12.1	4.6	92.6	20.3
MolDyn	62.4	100.0	47.1	29.4	142.9	8696.2	60.9
RayTracer	60.5	100.0	46.5	28.1	34.5	10517.5	304.9
MonteCarlo	60.7	100.0	46.4	28.2	29.5	2953.4	100.1
Logger	29.2	97.8	97.4	28.4	1.4	1.3	1.0
Maza	35.7	99.3	99.3	35.4	1.2	1.9	1.6
Barcode	27.2	87.3	87.3	23.8	0.9	2.4	2.7
Guitar	27.8	98.4	98.4	27.3	1.1	1.5	1.4
J2MESafe	36.1	99.6	99.6	36.0	1.1	1.8	1.6
HyperM	53.7	99.3	84.3	45.2	2049.4	2068.7	1.0
Podcast	51.6	97.8	71.0	36.7	2.7	6.4	2.4
GoldenSMS Key	31.4	99.1	99.1	31.1	1.4	5.3	3.8
GoldenSMS Msg	46.2	100.0	71.8	33.1	155.9	605.3	3.9
GoldenSMS Rec	26.2	97.4	97.4	25.5	1.1	2.4	2.2
Cellsafe	57.5	99.8	99.8	57.4	1.0	1.3	1.2
Bandera avg. values	**54.1**	**99.7**	**83.5**	**44.3**	**39.0**	**6745.9**	**136.7**
Java Grande avg. values	**40.1**	**100.0**	**59.5**	**21.4**	**23.1**	**2177.5**	**66.2**
JavaME avg. values	**38.4**	**97.8**	**91.4**	**34.5**	**201.6**	**245.3**	**2.1**
Total	**43.9**	**99.2**	**78.0**	**33.1**	**89.4**	**2940.9**	**66.2**

these numbers show that the context of the execution states of threads matters if one aims to avoid timing-insensitive paths.

The slices computed by algorithm GN were by far the most precise. On average, they were about 22% smaller than the I2P slices and about 21% smaller than the S slices. In the best case,

for ForkJoin, its slices were even almost 60% smaller. There is a significant discrepancy between the results for the three parts of our benchmark. Whereas the GN slices for the programs from the JavaME benchmark were on average only about 8.6% smaller than the I2P slices, the GN slices for the Java Grande programs were about 40.5% smaller. This could mean that the benefit of timing-sensitive slicing depends on the application area.

The timing-sensitive slices contained on average 33% of all nodes of the benchmark programs, which is comparable with the ratio of the size of context-sensitive slices to the overall number of nodes in sequential programs. It seems that timing-sensitivity has the same impact on the precision of slices of concurrent programs as context-sensitivity has for sequential programs.

Execution times As expected, the I2P slicer had no problems with slicing the benchmark, since its asymptotic running time is linear to the number of edges.

The execution times of S seem very high compared with its small gain of precision. They are on average 89 times higher than those of I2P. However, to be fair, we have not exhausted the possibilities for optimizing this algorithm. In particular, it could be improved to exploit our MHP information, which it currently does not.

The high precision of algorithm GN is accompanied by significantly increased runtime costs, which do not scale well. In the worst case, for KnockKnock, GN was more than 56,000 times slower than I2P. Notably, its runtime costs for the JavaME programs were in most cases acceptable, only the costs for GoldenSMS-Msg and HyperM were disproportionately higher than those of I2P. Interestingly, its runtime costs for the JavaME programs have the same dimensions as those of algorithm S. Since the main difference of GN and S is the propagation of contexts in GN, this similarity indicates that the investigated JavaME programs have comparatively few contexts per node.

Observations An important observation is that neither the gain of precision nor the runtime costs of timing-sensitive slicing seem to be predictable, even not for programs of roughly the same size. Compare, for example, the results for AlarmClock and DiningPhils or for the three GoldenSMS programs, where the gain of precision and the runtime costs vary strongly. Furthermore, strongly increased precision and strongly increased runtime do not necessarily accompany each other, for which again AlarmClock and DiningPhils are good examples. Thus, it is hard to estimate whether timing-sensitive slicing of a program pays off or is unsuccessful, as it is, for example, for AlarmClock. We cannot even use algorithm S as an aid in such an estimation, because its gain of precision is not related to the one of GN (again, AlarmClock and DiningPhils).

Another important observation is that the mere size of a program does not seem to dominate the runtime costs of timing-sensitive slicing. Even though Cellsafe has almost 9 times as

much nodes and more than 19 times as much edges as AlarmClock, GN computed its slices 195 times faster. The runtime costs seem also not to depend on the number of interference edges: DaisyTest has 8 times as much interference edges as DaisyTest, but GN sliced it 7 times faster. It is currently not clear which property of a CSDGs is mainly responsible for the disproportionately high runtime costs of timing-sensitive slicing for programs like AlarmClock, KnockKnock, RayTracer or HyperM.

3.12.4. On the Practicability of Context Graphs

In general, the employment of context graphs should be considered carefully, because their size, compared with the size of the original TCFGs, does not scale well. Since Nanda's slicing technique strongly depends on context graphs, we measured the size of the context graphs of our benchmark programs and the time needed for their computation. We found that for programs within reach of the timing-sensitive slicer context graphs are actually manageable if effective cycle folding is used. Table 3.13 compares the size of the TCFGs, of the CSDGs and of the context graphs of our benchmark programs[4]. The last column shows the time needed for the computation of the context graphs. Even though the context graphs are much bigger than the TCFGs, having about 15 times more nodes and edges in the worst case (for GoldenSMS-Key), their size is comparable with the size of the CSDGs. In the worst case, for GoldenSMS-Key, the context graph has only about 3.5 times more nodes and has even less edges than the corresponding CSDG. Remarkably, the context graphs have in all cases fewer edges than the CSDGs. This effect is due to the strong cycle folding committed by the computation of the ISCR graphs. However, the results reveal nevertheless that context graphs do not scale with the size of the programs: The bigger the programs, the bigger is the difference between the size of the TCFGs, the CSDGs and the context graphs. An aggravating factor is that the computation times increase strongly, from 0.1 seconds for DiningPhils to 51.2 seconds for DayTime.

3.12.5. Hot Spots

During our evaluation, we observed that the timing-sensitive slices for different slicing criteria of the same program did not require the same computation times. There were certain slicing criteria whose slices needed significantly more computation time than the others. We analyzed the impact of such *hot spots* to see whether there is only a handful of them in a program which consume the lion's share of the overall computation time. In that case it could be beneficial to analyze how such hot spots arise, i.e. if there are certain patterns of dependences in a CSDG which cause them. Then an analysis that detects hot spots could help to trade precision for

[4]The number of edges in the TCFGs seem to be quite high compared with the number of nodes. This results from exceptional flow being modeled by conditional branching: Most Java statements may throw exceptions and thus have more than one possible successor.

Table 3.13.: The size and computation time of the context graphs of our benchmark programs.

Name	TCFGs		CSDGs		context graphs		time
	nodes	edges	nodes	edges	nodes	edges	(sec.)
Example	966	1,696	2,509	36,411	812	1,396	0.4
ProdCons	1,088	1,927	3,331	39,614	770	1,331	0.2
DiskScheduler	1,484	2,624	4,389	43,645	7,949	13,203	1.1
AlarmClock	1,375	2,430	4,781	44,952	2,092	3,517	0.2
DiningPhils	2,262	4,203	5,115	125,377	2,853	5,083	0.1
LaplaceGrid	1,881	3,328	6,175	50,876	8,376	13,908	0.8
SharedQueue	3,242	5,639	11,284	78,797	7,266	11,956	0.5
EnvDriver	8,694	15,230	19,106	181,428	38,264	62,317	1.7
KnockKnock	9,023	14,937	35,852	312,678	62,854	95,888	6.2
DaisyTest	7,985	14,384	43,138	437,937	83,032	137,695	20.4
DayTime	11,857	21,298	59,708	632,146	176,098	292,220	51.2
ForkJoin	9,501	12,905	16,972	59,921	31,624	44,581	4.0
Sync	9,620	13,083	17,315	62,033	30,254	42,731	4.4
Barrier	9,683	13,149	17,579	62,471	29,850	42,162	4.5
Series	9,886	13,453	17,709	61,014	33,495	47,154	4.2
LUFact	10,787	14,641	18,888	65,449	33,902	47,677	3.8
SOR	10,542	14,359	19,151	66,418	43,818	62,348	7.5
SparseMatmult	10,760	14,659	19,502	67,646	43,898	62,452	8.4
Crypt	11,054	15,045	19,973	69,547	43,736	62,246	7.3
MolDyn	12,870	17,360	22,373	90,331	42,960	59,938	5.7
RayTracer	11,648	16,036	23,481	90,346	59,440	82,856	8.8
MonteCarlo	15,000	21,030	35,009	150,313	65,816	92,932	25.3
Logger	3,118	5,441	10,333	52,307	4,719	7,895	0.2
Maza	3,303	5,729	11,235	67,677	26,218	43,367	4.1
Barcode	3,728	6,513	12,344	64,674	18,286	29,966	0.9
Guitar	4,298	7,540	13,257	68,684	10,034	16,854	0.5
J2MESafe	4,611	8,043	17,754	126,409	26,262	43,184	1.6
HyperM	3,942	6,969	17,768	97,575	18,682	30,731	1.2
Podcast	5,809	10,198	23,576	159,116	41,944	67,078	3.6
GoldenSMS Key	5,081	8,871	21,860	177,911	77,519	128,905	10.0
GoldenSMS Msg	5,972	10,416	26,333	215,399	60,309	99,943	6.5
GoldenSMS Rec	5,457	9,485	22,088	149,039	51,262	84,848	4.7
Cellsafe	10,424	18,448	41,707	862,654	98,869	157,512	11.1

speed: For a hot spot, one could apply the fast iterated two-phase slicer instead of expensive timing-sensitive slicing. To this end, we computed 100 slices of our benchmark programs with the algorithms I2P and GN and measured the execution times for every single slice. Figures 3.20 and 3.21 show the results for SharedQueue, HyperM, Barcode and Cellsafe, which are representative. The upper chart of each program shows the gain of precision of the timing-

sensitive slices compared with the I2P slices. A gain of precision of $x\%$ means that the timing-sensitive slice is $x\%$ smaller than the I2P slice. The lower chart of each program shows for each slice its portion of the overall runtime costs in percent. The slices are sorted in the order of their execution, the same x-coordinate in the charts of the same program corresponds to the same slice.

The results for SharedQueue and HyperM in Fig. 3.20 are promising, because they suggest that there are very few hot spots in a program and that they do not necessarily correlate with those slices in which the most precision is gained. A conservative treatment of these hot spots could improve the runtime behavior without sacrificing too much precision. However, Fig. 3.21 indicates the opposite: The hot spots in Barcode and Cellsafe are exactly those slices that are responsible for the overall gain of precision. Treating them conservatively would reduce the gain of precision of timing-sensitive slicing to zero.

The results also show that the hot spots may have a different impact on the runtime costs. In Cellsafe they are responsible for almost 100% of the overall runtime costs. However, in the other programs their impact is not that drastic. If we classify a hot spot as a slice that consumed more than 5 times the average runtime, then their portion of the overall runtime is 46% for SharedQueue, 63% for HyperM and 22% for Barcode. Even though these portions are large, they seem to be not large enough to tackle the explosion of runtime costs by a conservative treatment of hot spots.

What can we conclude from this analysis? Timing-sensitive slicing is expensive even if it does not raise precision due to absence of timing-insensitive paths. Instead of detecting hot spots it could be useful to search for patterns of interference dependences in CSDGs that guarantee absence of timing-insensitive paths and to deactivate the detection of timing-insensitive paths for these interference dependences.

3.12.6. Threats to Validity

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

Our case study lacks big programs. The size of a program does not necessarily influence the number of thread-shared data, so the algorithms might work well for bigger programs with sparse interference dependences. This assumption is nourished by the results for programs like Cellsafe and Podcast.

In most parts of our evaluation, we have only computed 100 - 1000 slices per program, because timing-sensitive slicing was not fast enough for several of our bigger programs. Computing the slices for every node in these programs would not have been possible in reasonable time. However, we argue that our sample slices are sufficient for a qualitative comparison of

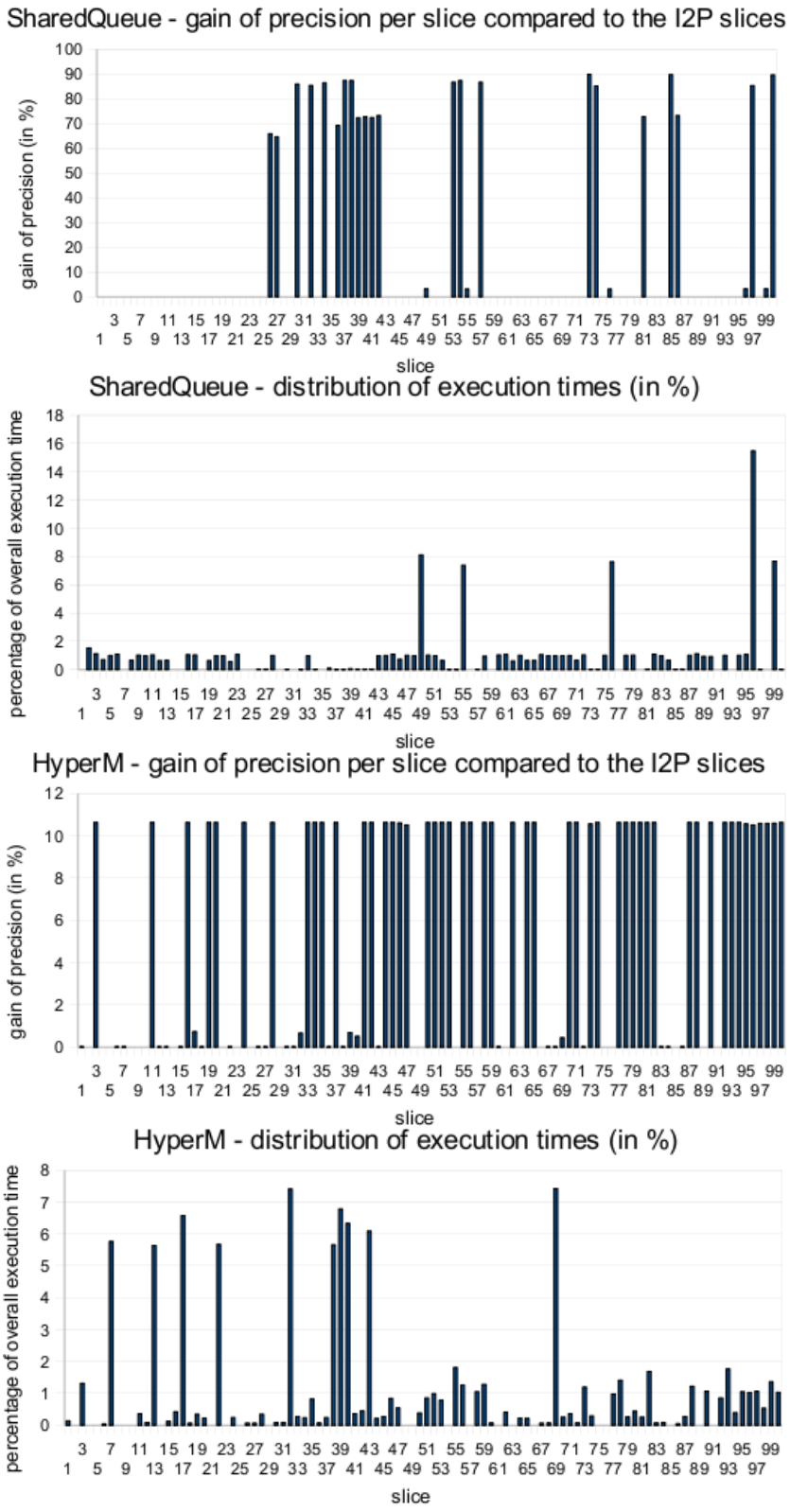


Figure 3.20.: Hot spots and gain of precision of 100 slices in ‘SharedQueue’ and ‘HyperM’. The diagrams show that there are very few hot spots in these programs and that they do not correlate with the slices which strongly increase the overall precision.

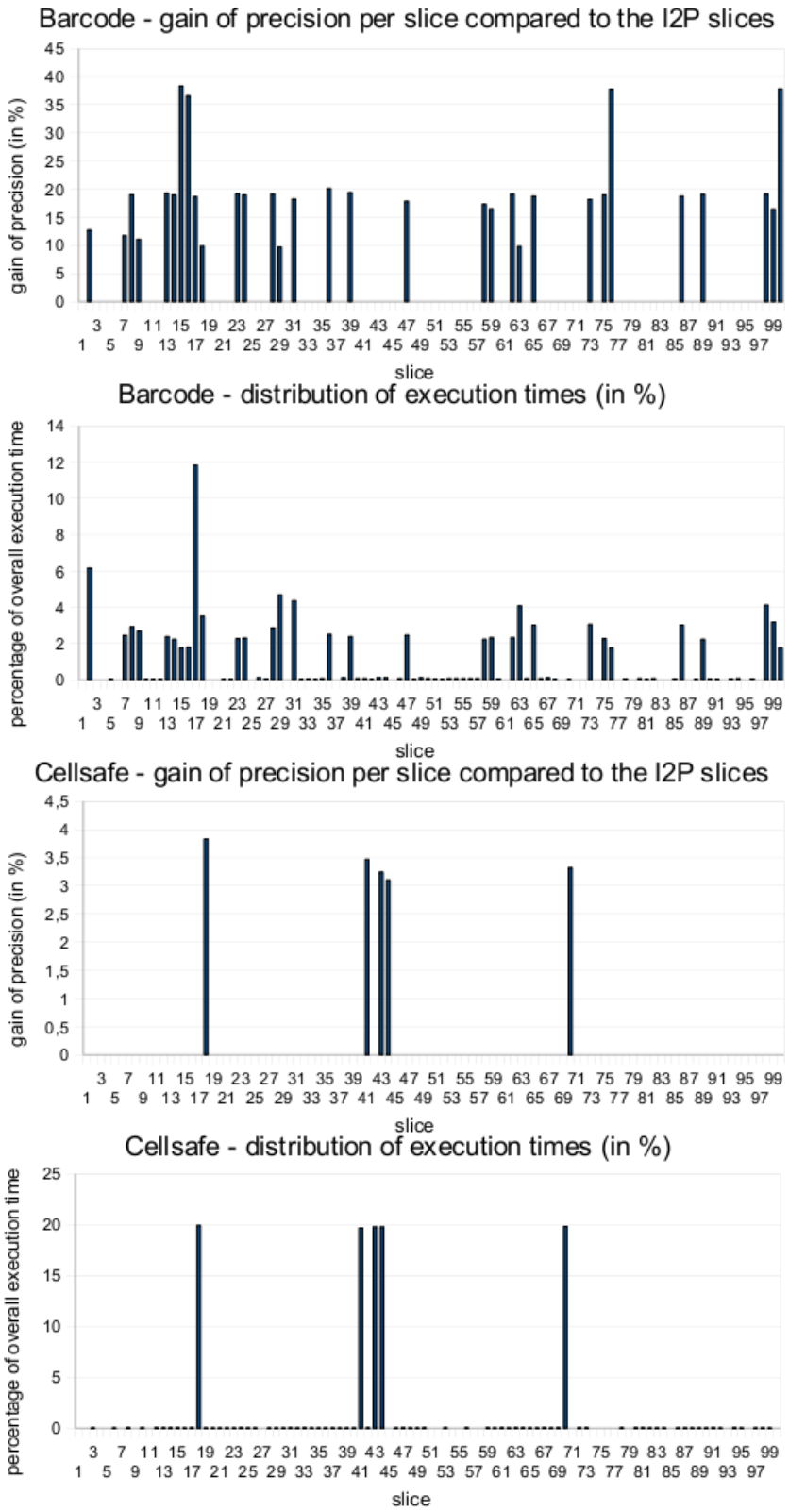


Figure 3.21.: Hot spots and gain of precision of 100 slices in ‘Barcode’ and ‘Cellsafe’. Here the hot spots are almost identical with the slices which strongly increase the overall precision.

the algorithms in terms of precision and runtime behavior, even though the concrete numbers might differ for a different set of slicing criteria.

Since our slicing criteria were created randomly without any filtering technique, our results should be verified for concrete applications of slicing, whose settings may a priori exclude some kinds of slicing criteria. In particular, we did not remove slicing criteria whose slices were completely thread-local. Excluding these slicing criteria from the evaluation would increase the average gain of precision significantly, because thread-local slices do not contain timing-insensitive paths.

Further threats to validity are possible bugs in our implementations, because these algorithms are extremely complicated.

3.12.7. Study Summary

Timing-sensitive slicing is able to decrease the size of the slices significantly – about 22% on average in our tests – but at a high price: The execution times rise dramatically and are dependent on the numbers of threads in the analyzed program. The application area of these algorithms is bound to concurrent programs with a low number of threads, since increasing numbers of threads decrease the precision benefits, while at the same time raising execution times. The iterated two-phase slicer is by far the most efficient algorithm. Additionally, it is easy to implement, so we recommend its use for slicing bigger concurrent programs, for programs with high numbers of threads and in application areas where its imprecision is of minor interest.

In our opinion, a vital requirement for the application of timing-sensitive slicing is to use Nanda's restrictive state tuple optimization and their context representation. Exploiting MHP information in the timing-sensitive slicers does not only raises precision, but can even decrease computation times, so we also recommend it. Our optimizations have shown to be very effective and should be included into implementations of these algorithms. Our optimized version of Nanda's algorithm, GN, proved to be the currently most performant and precise timing-sensitive slicer.

Our hot spot analysis shows that avoiding hot spots would not ease the exponential growth of computation costs, because the dimension of the cost reduction would be too small. Moreover, avoiding hot spots might lower precision, although hot spots and gain of precision do not necessarily correspond. An alternative could be to identify patterns of dependences in CSDGs that guarantee absence of timing-insensitive paths, in which case the detection of such paths could be deactivated. Such an optimization would not lower precision.

We presented an earlier version of this evaluation in [46]. However, that evaluation focused on the comparison of timing-sensitive slicers, similar to section 3.12.2, and did not contain a part comparable with section 3.12.3. An evaluation of the MHP analysis and of context graphs was

lacking, too. Furthermore, we developed several of our optimizations only after that publication, most importantly our new update mechanism for state tuples. The MHP analysis used in [46] was a preliminary version of the one presented in this chapter, lacking, for example, information about thread joining. Eventually, the benchmark used here contains more and more complex programs as the one in [46], where the biggest program had approximately 18.000 nodes and 140.000 edges.

Nanda provides an evaluation of her algorithm [106]; however, it is difficult to compare its results with ours because the original algorithm may compute incorrect slices by pruning valid interference edges. We fixed the algorithm according to section 3.10.3, which avoids such pruning, but raises execution times.

Krinke did not implement his algorithm. To the best of our knowledge, our implementation is the first, so we could not compare it with another one.

3.13. Discussion

We want to discuss several issues of slicing of concurrent programs in this section.

Context- vs timing-sensitivity

We seem to be the first to explicitly distinguish context-sensitive and timing-sensitive slicing of concurrent programs, our motivation being that timing-insensitivity may appear in programs without procedures. Nanda [106] equates context- with timing-sensitivity and calls her timing-sensitive slicer ‘context-sensitive’, her I2P slicer ‘imprecise’. Krinke [75] seems to distinguish context- and timing-sensitivity, but he does not investigate timing-insensitive context-sensitive slicing of concurrent programs.

Java’s happens-before relation

An interesting question is whether Java’s *happens-before relation* [48, 93] can be used to improve our MHP analysis. The happens-before relation is defined in the context of the Java memory model [93]. As summarized at the beginning of this chapter, the Java memory model may give rise to situations where a thread reading a shared variable `x` may not be aware of another thread having redefined `x` in the meantime. This may occur if the two actions are not properly synchronized, i.e. if they form a data race. The happens-before relation relates two actions in a program execution if the later executed action is guaranteed to observe the effects of the earlier executed action. If in a program execution an action a writes a variable `x` and an action b reads it and a happens before b, then b is guaranteed to read the value written to `x` by a.

```
void thread_1():
  lock l;
  x = 1;
  unlock l;
```

```
void thread_2():
  lock l;
  y = x;
  unlock l;
```

Figure 3.22.: An example illustrating the happens-before relation. Dependent on which lock action a program execution executes first either `x = 1` happens-before `y = x` or `y = x` happens-before `x = 1`.

The *happens-before order* [48, §17.4.5] of a program execution is defined as the transitive closure of the program order (cf. section 3.8) and the *synchronizes-with relation* [48, §17.4.4]. The synchronizes-with relation relates inter-thread actions that synchronize with each other. For example, an unlock action on a monitor `l` synchronizes-with all subsequent lock actions on `l`, a write to a volatile variable `y` synchronizes-with all subsequent reads of `y`. The important thing about the happens-before order is that it is dynamic; different program executions may exhibit different happens-before orders. Particularly, happens-before does not impose an execution order on the related actions. As an example, consider the program fragment in Fig. 3.22, in which two threads access a shared variable `x`. We assume that the lock and unlock statements work on the same lock `l`. The happens-before order of a program execution involving these threads depends on which lock action is performed first. If the one in thread 1 wins the race, then `x = 1` happens-before `y = x`, otherwise, `y = x` happens-before `x = 1`. This example illustrates that happens-before generally does not contain enough information to improve the results of our MHP analysis. The underlying problem is that Java's synchronization mechanism does not impose a fixed execution order between two synchronized blocks. However, in special cases synchronization may lead to a fixed execution order, and an analysis for detecting these special cases could improve the MHP analysis. Developing such an analysis remains future work.

Developing new optimizations

Our evaluation shows that additional optimizations are needed in order to make timing-sensitive slicing practical for bigger programs. In particular, it is necessary to identify which properties of a CSDG lead to disproportionately high execution times, while other SDGs of similar size are processed with acceptable costs. We suspect specific patterns of interference dependences to be responsible. Having identified these patterns, special optimizations could be developed to treat them.

3.14. Related Work

In this section we summarize the most related work.

Slicing concurrent programs

Probably the first author who addressed slicing of concurrent programs was Cheng [32, 33]. He defines the *process dependence net* (PDN) to represent dependences in parallel or distributed programs without procedures, where the concurrent tasks communicate via channels. Chen's target language is Occam 2, a concurrent language close to the CSP calculus, but he states that his system is also able to convert programs written in C, Ada or Pascal into PDNs. PDNs contain, besides data-, control- and weak control dependences, selection dependences, which are a form of control dependence for nondeterministic choice operators (such as the ALT command in Occam 2), synchronization dependences, which exist between statements reading and writing a channel, and communication dependences, which are Chen's counterpart to interference dependence. PDNs are sliced via simple graph reachability.

Zhao et al. [164] extended Cheng's PDN to the *system dependence net* (SDN) to represent concurrent object-oriented programs. Their language of choice is CC++ (Compositional C++), an extension of C++ with cobegin-coend concurrency, shared memory and rendezvous synchronization. The SDN integrates the dependences used in the PDN into Larsen and Harold's ClDGs [85] (cf. sect. 2.7). In a subsequent publication, Zhao [163] developed the *multi-threaded dependence graph* (MDG) for Java, which is similar to our CSDG and additionally contains synchronization dependences. A statement s is synchronization dependent on a statement t if either s is a `wait()` and t the corresponding `notify()`, or if s is a join node and t is the exit of the joined thread. SDNs and MDGs are sliced via the two-phase slicer, extended to additionally traverse the concurrency-related dependences in both phases. Nanda [106] has shown that this treatment of concurrency-related dependences may yield incorrect slices.

Hatcliff et al. [61] use slicing in their Bandera project, a tool set for compiling Java programs into inputs of several existing model-checkers, to identify and omit program parts unrelated to a given specification. Their dependence graph is similar to the CSDG, but lacks summary edges and additionally contains synchronization and ready dependences (cf. sect. 3.4) and weak control dependences. Their slicer works similar to the IPDG slicer, but k-limits the size of the call strings and additionally traverses interference edges. Thus, it is timing-insensitive and not completely context-sensitive.

Ramalingam [114] proved that synchronization-sensitive context-sensitive slicing of concurrent programs is undecidable. The proof consists of reducing Post's Correspondence Problem to the synchronization-sensitive context-sensitive reachability problem. Most approaches to slicing of concurrent programs sacrifice synchronization-sensitivity in favor of context-sensitivity, and it would be interesting to see the results of an opposite approach, favoring synchronization-sensitivity. However, Taylor [144] came to the daunting conclusion that a synchronization-sensitive analysis, for rendezvous-style synchronization, is even for intra-procedural programs without branches and loops NP-hard.

Müller-Olm and Seidl [102] have shown that optimal slicing of concurrent interprocedural programs is undecidable: If a node n is interference dependent on a node m due to a variable v, then it is not decidable whether another statement that redefines v may execute between n and m. This is proven by reducing the termination problem of two-counter machines to that problem. The problem becomes solvable when procedure calls are forbidden, but is still PSPACE-hard. In a subsequent publication, Müller-Olm [101] observed that the common assumption of statements executing atomically intended to simplify program analyses actually increases their complexity. If that assumption is abandoned, the problem of precise interprocedural analysis of variable dependences in parallel programs investigated in [102] becomes decidable. The paper shows how the set of possible program executions needed for that purpose, called *bridging runs*, can be characterized by a constraint system, so that the dependences can be analyzed via abstract interpretation. The runtime complexity of the resulting algorithm is exponential in the number of the shared variables and polynomial in the program size.

Mohapatra et al. [100] investigated dynamic slicing of concurrent object-oriented programs. Their algorithm statically builds a kind of CSDG and computes on it the dynamic slice during the program execution. After each step in the program execution, the algorithm marks all dependences in the CSDG which turned active and unmarks all which ceased to exist. The authors report a space complexity of $\mathcal{O}(N^2)$ and a runtime complexity of $\mathcal{O}(N * m)$ for the dynamic phase of their algorithm, where N is the number of nodes and m the maximal in-degree in the CSDG. They have implemented their algorithm in the system *DSCOP*, where it can be used to dynamically slice programs written in a subset of Java.

Timing-sensitive slicing Chen and Xu [31] present a different approach to deal with timing-insensitivity in Java programs. They incorporate synchronization in their *concurrent control flow graphs* (CCFGs) such that thread-local control flow edges between calls of `wait()` and their direct successors are removed and redirected over the notifying thread via wait and synchronization edges. Having constructed the CCFG of a program, all edges and nodes which cannot be reached from its entry node are removed. Then, one SDG per thread is created and connected via interference dependences, the result being the *concurrent program dependence graph* (CPDG). During the slicing process, their proposed slicing algorithm identifies and omits nodes that cannot be part of the slice because they cannot be executed before the nodes in the intermediate slice. This is determined on the level of nodes, ignoring calling contexts, hence the detection of timing-insensitivity is context-insensitive. Furthermore, the slicer itself traverses the CPDG context-insensitively. However, embedding that algorithm into the iterated two-phase slicer seems to be straightforward. We suspect that the precision of the suchlike improved algorithm is in the range of our timing-aware slicer, which also treats timing-insensitivity in a context-insensitive manner. However, the authors do not report an implementation and do not provide empirical data. A shortcoming of their approach is that the construction of a CCFG

requires to inline procedures that call synchronized procedures or contain synchronized blocks, hence it cannot completely handle recursion.

Rousseau [122] presents a slicer for a subset of Ada that is able to compute timing-sensitive slices. The slicer is embedded in QUASAR, a program analysis tool based on model checking, where it serves as a preprocessing step to reduce the size of the models. The slicer does not work on a CSDG but on the abstract syntax tree of the program and determines the program dependences on the fly. It computes the dependences by using stacks, on which the statements reside for which the dependences still have to be computed. For each thread, the slicer maintains one stack, effectively simulating the state tuples used in the timing-sensitive slicers of Krinke and Nanda. The author argues that this on-the-fly-computation is more performant than to compute a whole CSDG. However, if several slices of the same program are needed, this may lead to computing the same dependences repeatedly, hence that claim is disputable. The author does not provide empirical data supporting his claim, hence the runtime performance of his algorithm remains unclear.

Qi and Xu [111] present the *task communication reachability graph* (TCRG) to compute timing-sensitive slices of Ada programs. The TCRG is basically a control flow graph that unrolls the symbolic execution committed by Krinke's and Nanda's algorithms and describes all possible execution orders of a program. For that purpose, its nodes represent configurations (def. 3.18) and the edges model control flow between them. By using an optimization similar to restrictive state tuples it is possible to subsume many configurations by one representative, so it is not necessary to include all possible configurations in the graph. A data flow analysis on the TCRG creates as special dependence graph, in which all paths are context- and timing-sensitive. Context- and timing-sensitive slices can be computed via simple graph reachability. The authors do not present an evaluation of their technique.

In a subsequent publication [112], Qi et al. present the *threaded interaction reachability graph* (TIRG), a sort of control flow graph in which the possible ways of interleaving are unrolled – each node in a TIRG corresponds to a possible interleaving situation. This permits not only to build a dependence graph free of context- and timing-insensitive paths, as with the TCRG, but also to identify situations where an interference dependence is intercepted by a killing definition. Hence, their approach promises extremely precise slices. However, several issues are still to be solved. Most importantly, TIRGs not only unroll the possible ways of interleaving, but also have to inline procedures, presumably leading to monstrous sizes (to date, no empirical data has been reported). Thus, compression techniques have to be developed in order to get their sizes into grip. A possible starting point could be the work on dynamic dependence graphs for dynamic slicing, which at the beginning experienced similar problems.

Concurrency analysis for Java

Naumovich et al. [107] present a may-happen-in-parallel (MHP) analysis for Java programs that computes for every pair (s, s') of statements whether s and s' may execute concurrently. Their approach considers fork and join points as well as synchronization and is more precise than the technique we used. The analysis works on a *parallel execution graph* (PEG) which is derived from the control flow graph of the input program. PEG creation imposes several restrictions on the input program, most notably absence of recursion, because procedures have to be inlined. The MHP analysis on the PEG has a runtime complexity cubic to the number of PEG nodes and seems to be practical only for PEGs with < 2000 nodes [87].

Li and Verbrugge [87] implemented Naumovich et al's MHP analysis based on the Soot framework and developed several PEG simplification techniques, which may strongly reduce PEG sizes. These techniques basically identify areas in the PEG that can be merged to a single node. Their evaluation results suggest that MHP analysis for Java can be made practical for programs of reasonable size. However, restrictions like absence of recursion still persist.

Ruf [123] investigated synchronization removal techniques for Java programs and developed a *thread allocation analysis* for determining the number of instances a thread class may have at runtime. Basically, the analysis collects the allocations of thread objects in the program and determines conservatively how often such an allocation may be executed. Since a Java thread can only be started once, counting the allocations of thread objects conservatively approximates the existing threads. Our thread invocation analysis in section 3.1 was inspired by that work.

Lammich and Müller-Olm [81] present a conflict analysis for concurrent interprocedural programs with monitor-style synchronization. Their setting is similar to Java and permits dynamic thread creation and reentering of already acquired monitors. The analysis decides whether a conflict situation is reachable by checking whether the synchronization permits to reach the involved statements simultaneously, which is shown to be NP-hard. It determines via which paths the statements can be reached sequentially and then uses the acquisition histories of the monitors to check which of these paths can be interleaved. The crucial idea is to break down the paths into *macrostep* paths, which are same-level paths prepended by a single procedure call. That way the analysis achieves context-sensitivity without keeping track of calling contexts. The authors present an algorithm using abstract interpretation with an asymptotic running time exponential in the number of monitors. This technique could be adapted to prune spurious interference dependences: If there exists no valid interleaving in which the usage of a variable uses a certain definition of that variable, then the corresponding interference dependence can be removed.

4. Chopping

This chapter investigates chopping of concurrent programs. Chopping answers the question of which statements are involved in conveying effects from a statement s to another statement t. Chops are usually computed on dependence graphs, where a chop $chop(s,t)$ from s to t consists of all nodes on paths from s to t. Prior to our work, no chopping algorithm for concurrent programs has been reported at all. We developed the first published context-sensitive and timing-sensitive chopping algorithms for concurrent programs and present them in this chapter.

A simple way to compute $chop(s,t)$ for s and t is to intersect the forward slice for s with the backward slice for t. However, this may result in context-insensitive chops, even if context-sensitive slicing is used, because the slices (= sets of nodes) do not contain information about calling contexts. Consider the program in Fig. 4.1: Statement 3 is not influenced by statement 2, so $chop(2,3)$ should be empty. But the forward slice for statement 2 consists of the statements $\{2,4,5\}$, the backward slice for statement 3 of the statements $\{3,4,5\}$, so the intersection results in chop $\{4,5\}$. Reps and Rosay [120] developed the first context-sensitive chopping algorithm for sequential interprocedural programs. Their algorithm is the state of the art for chopping sequential programs. We abbreviate it with *RRC* throughout the thesis.

The RRC is, however, rather complicated to implement, and its asymptotic running time is not linear to the size of the analyzed program. We present a new chopping technique for sequential programs that is not entirely context-sensitive, but is easy to implement and very fast in practice, offering a genuine alternative for quick deployment. We evaluated the precision and runtime costs of this new technique together with several variants of the RRC on a set of 22 Java programs.

Unfortunately, the Reps-Rosay chopper cannot be applied directly to concurrent programs because it relies on all interprocedural effects being summarized by summary edges. We show how to extend it to compute context-sensitive chops in concurrent programs. After that, we

```
1 void main():          4 int foo():
2   int m = foo();      5   return 1;
3   int n = foo();
```

Figure 4.1.: A small example illustrating context-insensitive chopping.

transfer the techniques for timing-sensitive slicing to chopping and present a timing-sensitive chopping algorithm for concurrent programs.

Since detection of timing-insensitive paths is expensive and difficult to implement, we developed six chopping algorithms for concurrent programs. These algorithms offer different degrees of precision, from imprecise (but fast) over context-sensitive to timing-sensitive. We implemented these algorithms and evaluated their precision and runtime costs on a set of concurrent Java programs. Context-sensitive chopping reduced the chop sizes up to 25%, while moderately increasing execution times. Timing-sensitive chopping strongly reduced the chop sizes – up to 90% in the best case –, but at the expense of considerably increased execution times.

This chapter is organized as follows: Section 4.1 introduces chopping algorithms for sequential programs, on which those for concurrent programs are based. Section 4.2 presents our new, almost context-sensitive chopping algorithm for sequential programs. Section 4.3 concludes the part about chopping of sequential programs with one of the few published evaluations of these algorithms. Section 4.4 is concerned with context-sensitive chopping of concurrent programs and presents a suitable extension of the Reps-Rosay chopper. Section 4.5 explains timing-sensitive chopping and presents our resulting algorithms. Our chopping algorithms are evaluated in section 4.6. Section 4.7 discusses several issues and future work and section 4.8 summarizes related work.

Previous publications by the author contain preliminary versions of this chapter [42, 43].

4.1. Chopping Sequential Programs

Concerning sequential programs, there exist two different kinds of chopping, *same-level* and *unbound* chopping. Same-level chopping requires from the chopping criterion (s,t) that s and t stem from the same procedure p and considers only p and procedures called by p. Unbound chopping permits arbitrary chopping criteria and takes all procedures of the program into account. Figure 4.2 illustrates that difference: The unbound chop from `return x` to `x=x*x` consists of the gray shaded nodes because the return value of the first call of `foo` is fed to the second. The same-level chop from `return x` to `x=x*x` is empty because inside one invocation of `foo` `return x` does not influence `x=x*x`.

4.1.1. Same-Level Chopping

Since same-level chopping is used to compute unbound chops, it is briefly explained in this section. The basis of all presented chopping algorithms is the *intra-procedural* chop. An intra-procedural chop from a node s to a node t in procedure p consists of all nodes on paths from s

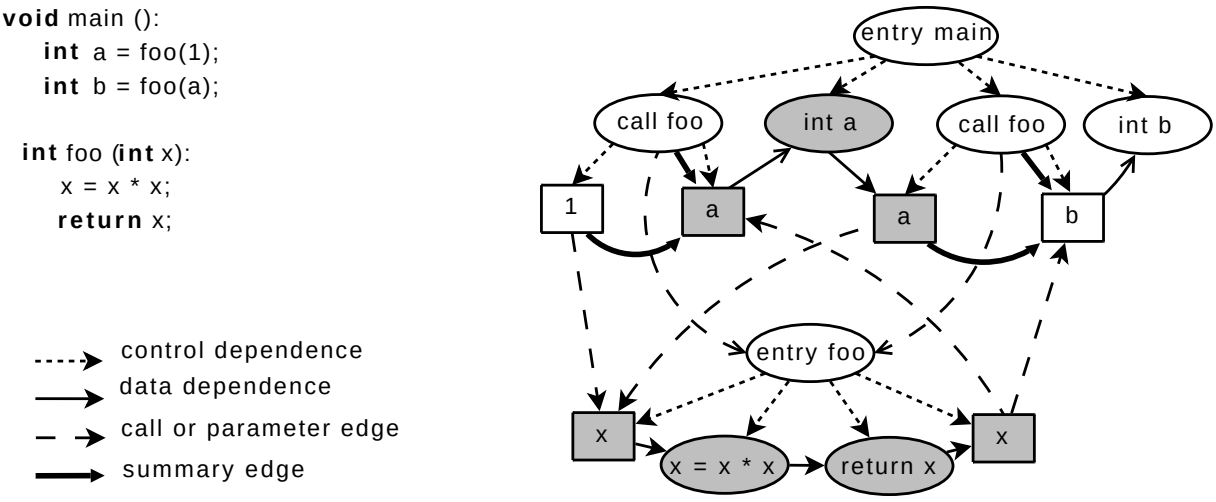


Figure 4.2.: Same-level vs unbound chopping: The unbound chop for (`return x`, `x=x*x`) consists of the gray shaded nodes, the same-level chop is empty. Note that the SDG contains special summary edges from call nodes to actual-out nodes, which are needed for chopping.

to t in p's PDG. It is commonly computed by intersecting the intra-procedural backward slice for t with the intra-procedural forward slice for s.

Definition 4.1 (Intra-procedural chop)**.** *Let G be the PDG of a procedure p. Let s and t be two nodes in G. The intra-procedural chop of G for chopping criterion (s,t) consists of the set of nodes*

$$\{n \mid \exists \text{ a path from } s \text{ to } t \text{ in } G \text{ that contains } n\}.$$

Jackson and Rollins [67] extended intra-procedural chopping to same-level chopping, where the nodes on paths through procedures called underway are added to the chop. A (context-sensitive) same-level chop from a node s to a node t of the same procedure consists of all nodes on matched paths from s to t.

Definition 4.2 (Same-level chop)**.** *The same-level chop of a SDG G for chopping criterion (s,t) consists of the set of nodes*

$$\{n \mid \exists \text{ a matched path from } s \text{ to } t \text{ in } G \text{ that contains } n\}.$$

Jackson and Rollins suggested computing same-level chops through an iterated approach that exploits summary edges. To this end, SDGs have to be extended with summary edges from call nodes to the associated actual-out nodes. Their approach first computes an intra-procedural chop from s to t. Then, for every traversed summary edge $ai \rightarrow_{su} ao$, it computes another intra-procedural chop for criterion (fi,fo), where fi is the formal-in- or entry node connected with the actual-in- or call node ai, and fo is the formal-out node connected with the actual-out node ao.

Algorithm 4.1 SMC: Krinke's Summary-Merged Chopper [74].

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

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

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

repeat
$W = W \setminus L$ *// process the next element*

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

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

until $W = \emptyset$
return C

This step is repeated until no new summary edge is visited. The same-level chop consists of all nodes visited in that process.

Jackson and Rollins' approach computes a new chop for every pair of {formal-in, entry} and formal-out nodes that have a summary edge between the corresponding {actual-in, call} and actual-out nodes included in the chop. Therefore, it may traverse the same edges multiple times – since two of such chops may overlap – and has an asymptotic running time bounded by $O(|E| * MaxFormalIns^2)$, where *MaxFormalIns* is the maximum number of formal-in nodes in any procedure's PDG. Krinke [74] developed an improved algorithm, which relieves that redundancy and is significantly faster in practice: If two summary edges of one call site are included in the chop, one does not need to compute two separate chops. Instead, a single chop between the *set* of corresponding {formal-in, entry} nodes and the *set* of corresponding formal-out nodes exhibits the same precision and traverses a smaller number of edges. Krinke's improved algo-

Algorithm 4.2 Reps and Rosay's same-level chopper [120].

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

$M = \emptyset, B = \emptyset, F = \emptyset, ActOut = \emptyset$
Let C be the intra-procedural chop for (s,t)

// initialize a worklist with all summary edges that lie in the chop
$W = \{(m,n) \mid m,n \in C, \exists$ a summary edge $m \rightarrow_{su} n\}$

repeat
  $W = W \setminus (m,n)$ *// process next element*
  Let *fi* be the formal-in or entry node connected with m
  Let *fo* be the formal-out node connected with n

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

    if $F(fi) = \emptyset$ *// store the intra-procedural forward slice for fi in F*
      $F(fi) = forw\_slice(fi)$

    if $B(fo) = \emptyset$ *// store the intra-procedural backward slice for fo in B*
      $B(fo) = back\_slice(fo)$
      $ActOut(fo) = \{n \in back\_slice(fo) \mid n$ *is an actual-out node*$\}$

    for all $x \in B(fo)$
      if $x \in F(fi)$ *// x belongs to the chop*
        $C = C \cup x$
        $B(fo) = B(fo) \setminus \{x\}$
        if x *is an actual-out or call node*
          for all summary edges $x \rightarrow_{su} y$ with $y \in ActOut(fo)$
            $W = W \cup \{(x,y)\}$

until $W = \emptyset$
return C

rithm, called *Summary Merged Chopper* and depicted in Alg. 4.1, exploits that observation as follows: After computing the initial intra-procedural chop for s and t, all traversed summary edges of visited call sites are collected. Then, for every visited call site, a new chop is computed between the set of corresponding {formal-in, entry} nodes and the set of corresponding formal-out nodes. This procedure is repeated with the newly visited summary edges until no new summary edges is visited. The resulting chop consists of all nodes visited in the process.

Though significantly faster in practice, the Summary Merged Chopper has still the same runtime complexity. An asymptotically faster technique with $O(|E| * MaxFormalIns)$ has been proposed by Reps and Rosay [120]. We explain that technique on its pseudocode in Figure 4.2. Starting from the initial intra-procedural chop for (s,t), it computes for every summary

edge $ai \rightarrow_{su} ao$ being part of the chop the corresponding {formal-in, entry}/formal-out pair (fi,fo). Then, it stores the intra-procedural forward slice for fi in map F (= forward) and the intra-procedural backward slice for fo in map B (= backward). It further stores for fo the set of actual-out nodes lying in the intra-procedural backward slice for fo. Then, every node x that lies in both the intra-procedural backward slice for fo and the intra-procedural forward slice for fi stored in the maps B and F is added to the chop and is removed from the stored backward slice for fo. The removal guarantees that each node is touched at most once. If x is a formal-in or entry node and there is a summary edge $x \rightarrow_{su} y$ to an actual-out node y lying in the intra-procedural backward slice for fo, then this edge is added to the worklist. That way, the algorithm iteratively processes the procedures called within the chop.

4.1.2. Unbound Chopping

An unbound chop[1] $chop(s,t)$ for criterion (s,t) takes all realizable paths from s to t into account. In the remainder of this thesis, the sole term 'chop' usually refers to unbound chopping. The concretion 'unbound' is only used where we have to distinguish between same-level and unbound chops.

Definition 4.3 (Unbound context-sensitive chop)**.** *The unbound context-sensitive chop of a SDG G for chopping criterion (s,t) consists of the set of nodes*

$$\{n \mid \exists s \rightarrow_{cs}^{*} t \text{ in } G \text{ that contains } n\}.$$

Reps and Rosay [120] developed a sophisticated algorithm that chops programs context-sensitively. It exploits that in a SDG all interprocedural effects are propagated via call sites and summarized by summary edges. Figure 4.3 gives a schematic overview: The ovals symbolize procedures, upgoing edges are procedure calls and downgoing edges are returns. The two graphs show how the chopper proceeds in computing $chop(s,t)$. First, it determines the common callers of s and t, i.e. the procedures that (indirectly) call both the procedures of s and t. This is achieved by computing a forward slice for s and a backward slice for t, which only visit the calling procedures. Intersecting them reveals the common callers and the set of nodes A in these procedures that belong to the chop. This is shown in the upper graph. In the next step, the RRC collects the nodes in the procedures leading from A to s or t that belong to the chop. For the procedures leading to s, this is done by intersecting the forward slice of s and the backward slice of A, where the forward slice visits only the calling procedures and the backward slice visits only the called procedures. For the procedures leading to t, this works analogously. The result is shown in the lower graph as light gray highlighted areas. This step ignores the procedures called underway by the visited nodes – in our example procedures 5 and 7. In a

[1] Also called *non-same-level* chop in several publications.

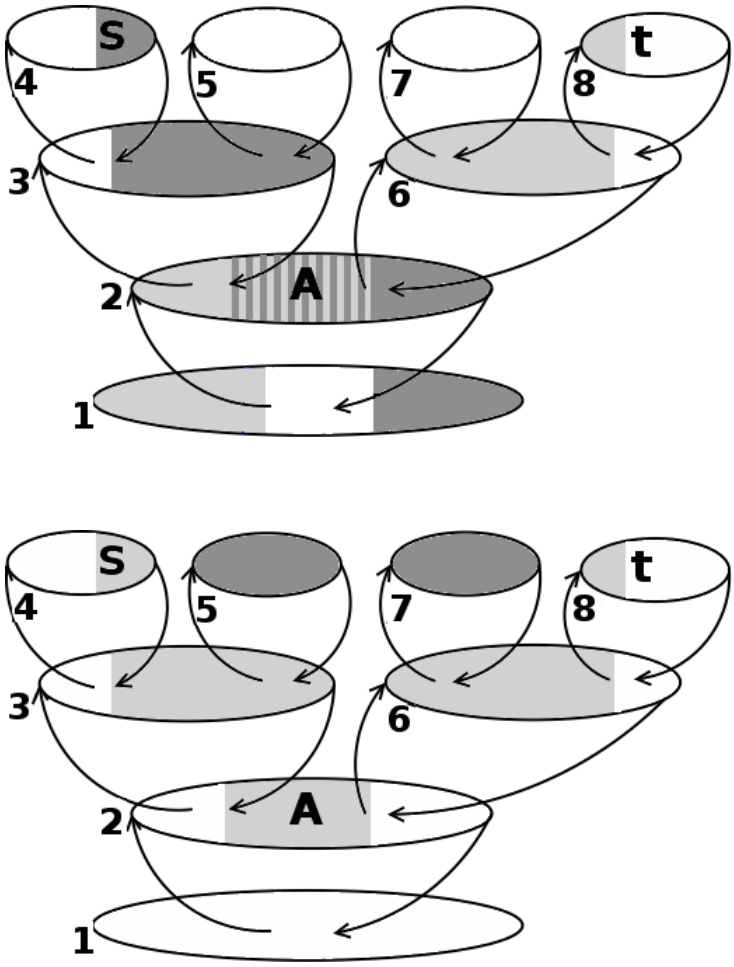


Figure 4.3.: Schematic overview of the Reps-Rosay chopper for chopping criterion (s,t). The upper part shows step 1, the lower part shows steps 2 and 3.

third step, these omitted procedures are analyzed via same-level chopping, starting from the summary edges between the nodes collected in step 2. The resulting chop consists of the nodes collected in steps 2 and 3. By exploiting summary edges in the second step and using same-level chopping in the third step, the algorithm maintains context-sensitivity. According to Reps and Rosay, RRC's asymptotic running time is in $\mathcal{O}(|E| * \mathit{MaxFormalIns})$ if their same-level chopper shown in Alg. 4.2 is used for step 3. Algorithm 4.3 shows its pseudocode; function *slc*, which computes the same-level chops, is realized by extending Alg. 4.2 to sets of chopping criteria.

4.1.3. The Reps-Rosay Chopper for Sets of Nodes

Though not explicitly stated by Reps and Rosay [120], the RRC is also able to compute context-sensitive chops for chopping criteria consisting of sets S, T of nodes, the result being the union of the chops for every pair $(s,t) \in S \times T$. For that purpose, the underlying slicers in the RRC are extended to compute slices for sets of nodes. This extension is needed for the computation

Algorithm 4.3 RRC: The Reps-Rosay chopper [120].

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

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

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

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

/ Step 3: collect the nodes in procedures called underway */*
// collect all traversed summary edges
// distinguishing the branches C_1 and C_2 ensures that the edge has actually been traversed,
// because otherwise ai could have been visited in C_1 and ao in C_2
$S = \{(ai,ao) \mid \exists\, ai \rightarrow_{su} ao \wedge (ai,ao \in C_1 \vee ai,ao \in C_2)\}$

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

return *Chop*

of context-sensitive chops of concurrent programs, and thus we prove its correctness in this subsection.

Following grammar H in definition 2.3, let $m \rightarrow^*_{unbr} n$ denote an *unbalanced-right* path in a SDG. Similarly, let $m \rightarrow^*_{unbl} n$ denote an *unbalanced-left* path. Reps and Rosay define the following operations to compute context-sensitive chops in SDGs [120]:

- $f_{unbr}(S) = \{n \mid \exists s \in S : s \rightarrow^*_{unbr} n\}$ (conforms to f_1 in Alg. 4.3)
- $f_{unbl}(S) = \{n \mid \exists s \in S : s \rightarrow^*_{unbl} n\}$ (conforms to f_2 in Alg. 4.3)
- $b_{unbl}(T) = \{n \mid \exists t \in T : n \rightarrow^*_{unbl} t\}$ (conforms to b_1 in Alg. 4.3)
- $b_{unbr}(T) = \{n \mid \exists t \in T : n \rightarrow^*_{unbr} t\}$ (conforms to b_2 in Alg. 4.3)

In other words, f_{unbr} is the set of nodes lying on unbalanced-right paths starting at a node $s \in S$, f_{unbl} is the set of nodes lying on unbalanced-left paths starting at a node $s \in S$, b_{unbr} is the set of nodes lying on unbalanced-right paths leading to a node $t \in T$ and b_{unbl} is the set of nodes lying on unbalanced-left paths leading to a node $t \in T$. The operations f_{unbr} and b_{unbl} can

be implemented by forward and backward two-phase slicers committing only phase 1, i.e. only ascending to calling procedures, f_{unbl} and b_{unbr} can be implemented by forward and backward two-phase slicers committing only phase 2, i.e. only descending to called procedures [120].

The RRC employs a function $SLC(e)$ that takes a summary edge $e = ai \rightarrow_{su} ao$ and computes a same-level chop for the corresponding {formal-in, entry}/formal-out pair. However, its concrete functionality is irrelevant for this proof. As explained in greater detail further above, the RRC performs the following 3 steps to compute the chop $RRC(s,t)$ [120]:

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

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

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

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

Lemma 4.1. $RRC(S,T) = \bigcup_{\substack{s \in S \\ t \in T}} RRC(s,t)$

Proof.

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

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

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

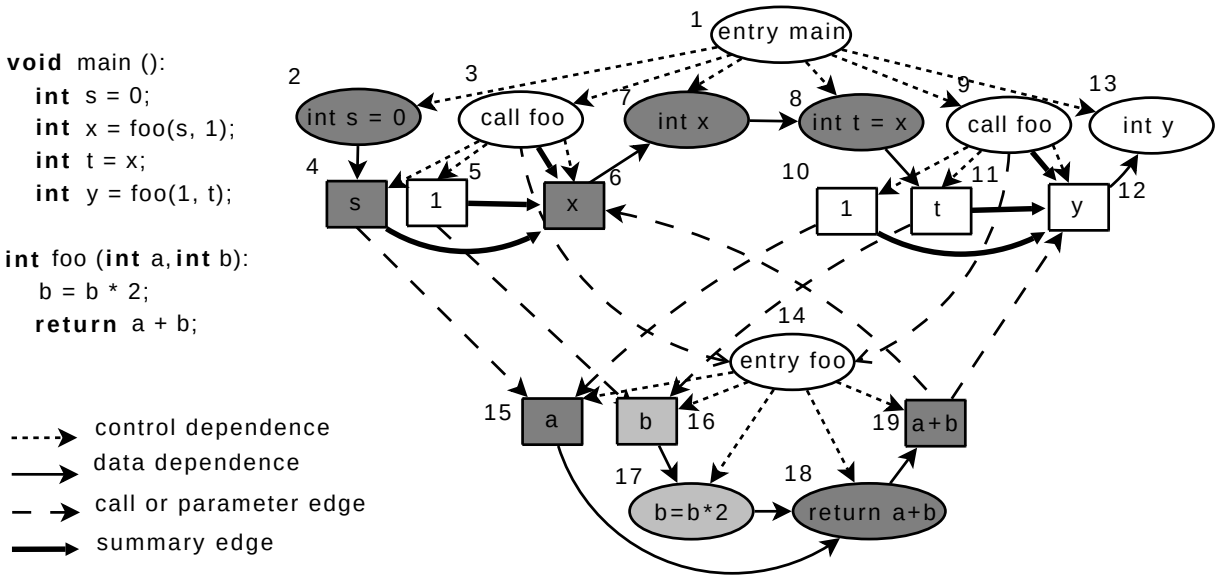


Figure 4.4.: Chops for chopping criterion $(2,8)$. The highlighted nodes denote the chop determined by computing the backward slice for node 8 on the forward slice for node 2. The dark gray nodes denote the context-sensitive chop.

either $s \rightarrow^*_{unbr} e_s \rightarrow_{su} e_t \rightarrow^*_{unbr} w \rightarrow^*_{unbl} t$ or $s \rightarrow^*_{unbr} w \rightarrow^*_{unbl} e_s \rightarrow_{su} e_t \rightarrow^*_{unbl} t$ holds. Therefore, e is also visited by the chop $RRC(s,t)$ in step 2, which means that $SLC(e)$ is added to that chop. Hence $n \in RRC(s,t)$.

□

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

4.2. The Fixed-Point Chopper

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

Although the RRC is known for 15 years, intersection-based chopping is often considered a convenient alternative for a quick deployment. A well-known optimization computes a forward slice for s and then a backward slice for t restricted to the subgraph traversed by the forward slicer. The resulting backward slice is already the chop, eliminating the intersection. Its runtime complexity is in $\mathcal{O}(|E|)$, like that of the underlying two-phase slicer. Moreover, it already removes some spare nodes from the chop. For example, it detects that $chop(2,3)$ in the program of Fig. 4.1 is empty, as statement 3 is not in the forward slice for statement 2.

During our work, we made the following observation: Computing another forward slice on the result of the above algorithm may result in an even more precise chop, for which Fig. 4.4

Algorithm 4.4 Fixed-point chopping: Computing almost context-sensitive chops.

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

```
Let f(c,M) be a two-phase forward slicer that only visits nodes in set M.
Let b(c,M) be a two-phase backward slicer that only visits nodes in set M.

/* Compute the initial chop. */
F = f(s,N)  // N be the set of all nodes in the SDG
Chop = b(t,F)
changed = true
/* Iterate until reaching a fixed-point. */
repeat
   Tmp = f(s,Chop)
   Tmp = b(t,Tmp)

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

   Chop = Tmp

until ¬changed
return Chop
```

provides an example. The context-sensitive chop for node pair (2, 8) consists of the dark gray nodes: Variable `s` is passed as a parameter to `foo` and flows via the return statement into variable `t`. If we employ the algorithm described above, the resulting chop also contains nodes 16 and 17: The forward slice for node 2 consists of the nodes {2,4,15,18,19,6,7,8,10,11,16,17,12,13}, the backward slice for node 8, restricted to these nodes, is the set {8,7,6,19,18,17,16,15,4,2}. Nodes 17 and 16 were visited by the forward slicer in the context of the second invocation of `foo`, which does not influence node 8. But that information is not available anymore in the set representing the forward slice, thus the backward slicer traverses from node 18 to the nodes 17 and 16. If we compute a second forward slice for node 2, restricted to that set of nodes, we are able to remove these spurious nodes. The forward slice visits the nodes {2,4,15,18,6,7,8}, which is in this example the context-sensitive result. This will not always be the case, but repeating that optimization may gradually remove more spurious nodes, resulting in a fixed-point style algorithm shown in Alg. 4.4. However, fixed-point chopping is not generally context-sensitive, for which Fig. 4.5 provides an example. The context-sensitive chop for (14, 12) consists of the dark gray nodes, but fixed-point chopping additionally includes node 13. Although this new algorithm has an asymptotic runtime complexity of $\mathcal{O}(|E| * |N|)$, our evaluation indicates that the fixed-point is reached very fast, usually after the second iteration of the loop.

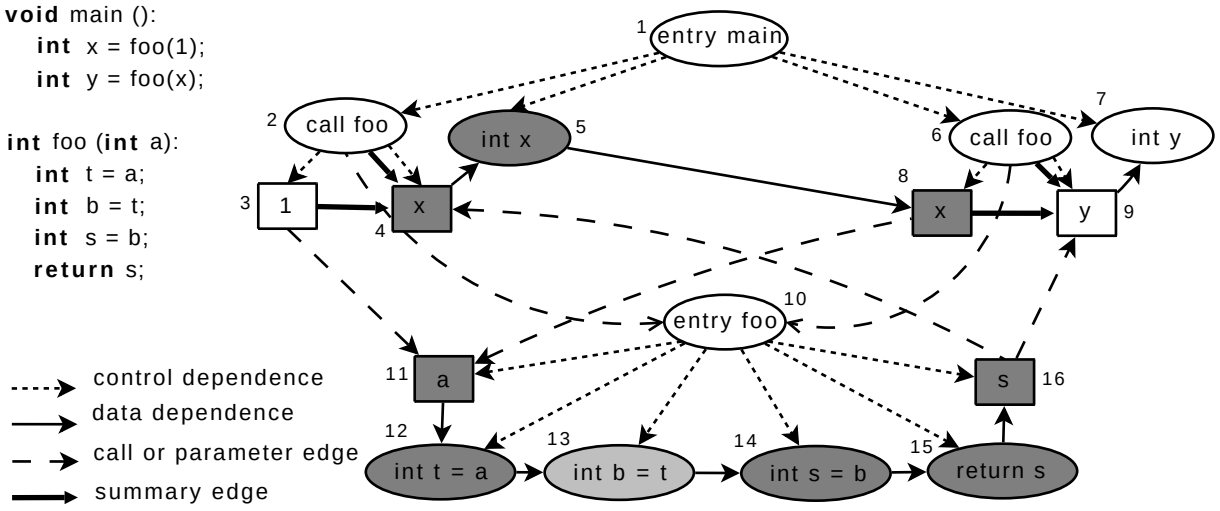


Figure 4.5.: Fixed-point chopping is not context-sensitive in general.

4.3. Evaluation

We have integrated the presented algorithms for unbound chopping into the Joana framework and evaluated them on the benchmark used in section 2.6, using the same hardware and settings. For each benchmark program, we randomly determined 10,000 chopping criteria consisting of one source and one target node. We measured the average chop sizes and execution times of our fixed-point chopper *FC*, the simple intersection-based chopper *SIC* and the RRC. For a more in-depth investigation, we further employed the algorithms *IC* (intersection chopper), which determines the chop for (s,t) by computing the forward slice for s on the backward slice for t, and *Opt-1*, which iterates the while-loop of FC only once. Additionally, we combined the RRC with the three same-level chopping algorithms introduced in section 4.1.1 and examined the runtime differences. In summary, we evaluated the following algorithms:

- SIC, intersects the forward slice for s with the backward slice for t.
- IC, computes the forward slice for s on the backward slice for t.
- FC, the fixed-point chopper of Alg. 4.4.
- Opt-1, executes the while-loop of FC only once.
- RRC, the Reps-Rosay Chopper.
- RRC-Unopt, uses Jackson and Rollins' iterative approach to compute the same-level chops in the last step of RRC.
- RRC-SMC, uses Krinke's Summary-Merged Chopper to compute the same-level chops.

Table 4.1.: Average size per chop (number of nodes). Column 'RRC' subsumes our three RRC variants, which always computed the same chops.

Name	SIC	IC	Opt-1	FC	RRC
Example	329.03	323.58	316.39	316.39	316.39
ProdCons	556.39	535.61	535.61	535.61	535.61
DiskScheduler	455.18	379.49	376.52	376.52	375.77
AlarmClock	770.06	628.87	608.76	608.76	608.56
DiningPhils	584.31	458.49	447.77	447.77	447.57
LaplaceGrid	895.43	739.92	731.66	731.66	731.29
SharedQueue	2335.69	2254.26	2252.76	2252.76	2251.94
EnvDriver	3326.14	3263.47	3248.71	3248.71	3248.35
KnockKnock	1798.35	1291.63	1258.56	1258.56	1235.91
Daisy	11843.49	10762.85	10750.78	10750.78	10750.78
Daytime	21454.00	20654.90	20492.54	20492.53	20480.94
Logger	889.01	873.53	873.15	873.15	873.14
Maza	1440.62	1390.85	1332.75	1332.75	1315.49
Barcode	528.93	414.26	403.73	403.73	401.93
Guitar	922.66	878.58	868.41	868.41	867.44
J2MESafe	2667.09	2475.75	2438.80	2438.80	2428.10
HyperM	267.65	202.19	190.00	190.00	189.32
Podcast	3197.20	2547.91	2461.84	2461.84	2456.19
GoldenSMS_Key	2163.53	1989.03	1952.65	1952.59	1947.89
GoldenSMS_Msg	1513.91	1243.03	1218.25	1218.25	1205.29
GoldenSMS_Rec	1402.34	1228.43	1193.48	1193.48	1187.30
Cellsafe	16202.76	15604.19	15398.24	15398.24	15130.68

4.3.1. Precision

Table 4.1 shows the average chop size for each chopping algorithm and benchmark program. Since all evaluated RRC variants compute the same chops, they are subsumed by column 'RRC'. The measured values demonstrate that context-sensitive chopping is able to reduce chop sizes significantly: The RRC chops are on average 13.3% smaller than the ones based on naïve intersection (SIC), in the best case even about 31.3% (for KnockKnock).

The simple, well-known optimization applied in IC turns out to be very effective: The RRC chops are on average only 2.2% smaller than the IC chops. Fixed-point chopping reduces imprecision even more: The FC chops are on average only 0.4% bigger than the RRC chops. For 10 out of our 22 programs, the difference is even below 0.1%. Notably, several of our larger programs are amongst these 10 programs (e.g. Daisy and DayTime), so FC's effectiveness is not restricted to small and simple programs.

The differences between Opt-1 and FC are marginally small – in Table 4.1 they are only visible for DayTime and GoldenSMS_Key. In fact, only for 47 out of our 220,000 chopping

Table 4.2.: Average execution time per chop (in milliseconds).

Name	SIC	IC	Opt-1	FC	RRC		
					Unopt.	SMC	Orig.
Example	9	7	9	9	48	17	17
ProdCons	12	10	13	13	74	33	26
DiskScheduler	12	9	11	12	45	21	18
AlarmClock	13	9	12	13	68	25	23
DiningPhils	24	18	23	25	62	41	38
LaplaceGrid	18	13	18	21	77	35	29
SharedQueue	33	23	37	37	1,274	74	139
EnvDriver	76	59	94	124	18,325	151	1,097
KnockKnock	73	40	49	58	556	83	83
Daisy	206	149	254	326	103,880	1,468	2,784
DayTime	347	279	501	721	296,799	1,295	8,491
Logger	16	9	14	14	230	22	40
Maza	25	16	26	36	896	38	100
Barcode	15	8	10	11	52	8	14
Guitar	25	16	22	24	245	21	36
J2MESafe	47	31	50	67	4,239	106	218
HyperM	22	13	14	15	36	7	12
Podcast	65	41	61	77	2,761	63	180
GoldenSMS_Key	52	31	48	64	4,183	112	284
GoldenSMS_Msg	48	23	33	42	2,120	60	140
GoldenSMS_Rec	39	23	32	41	1,598	56	114
Cellsafe	400	353	658	951	185,975	1,553	5,213

criteria – 26 in AlarmClock, 18 in DayTime and 1 in each Maza, J2MESafe and GoldenSMS-Msg – Opt-1 and FC computed different results. For these 47 chops, FC needed three loop iterations, two iterations removing spurious nodes and a last one to detect the fixed-point. For the other chops, it iterated the loop twice, thus basically performing Opt-1 plus an additional loop iteration detecting the fixed-point.

4.3.2. Runtime Behavior

Table 4.2 shows the average execution time per chop in milliseconds the different algorithms needed. The execution times measured for our three RRC variants reveal that the 3rd step of RRC – the computation of the same-level chops – is the critical part concerning runtime performance. The naïve iterative approach taken in RRC-Unopt did not scale well for our larger programs. For Cellsafe, GoldenSMS-Rec and EnvDriver it was more than 100 times slower than RRC-SMC. Surprisingly, RRC-SMC was the most performant variant, even though the original

RRC is asymptotically faster. It was faster than RRC for 15 out of 22 programs, particularly for the larger programs. On average, it was 2.2 times faster than RRC.

Given its imprecision, algorithm SIC performed rather poorly. For 4 programs it was even slower than RRC-SMC. IC was by far the fastest algorithm, followed by Opt-1 and FC. Algorithm RRC-SMC emerged to be competitive with the intersection based algorithms, even being fastest for HyperM. On average, RRC-SMC needed only twice as much time as the fastest algorithm, IC.

4.3.3. Study Summary

Concerning context-sensitive chopping, our evaluation shows that the unoptimized version of RRC is not practical. An application should always employ one of the optimized versions. Surprisingly, RRC-SMC was on average twice as fast as the original RRC and seems to be the most practical variant for middle-sized programs. However, as the original RRC is asymptotically faster than RRC-SMC, this might be different for larger programs.

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

4.4. Context-Sensitive Chopping of Concurrent Programs

A context-sensitive chop of a CSDG is defined as follows:

Definition 4.4 (Context-sensitive chops)**.** *The (unbound) context-sensitive chop of a CSDG G for chopping criterion (s,t) consists of the set of nodes*

$$\{n \mid \exists s \rightarrow_{cs}^{*} t \text{ in } G \text{ that contains } n\}.$$

Intersection-based chopping in combination with the iterated two-phase slicer enables fast and simple chopping of concurrent programs. Our first algorithm, abbreviated with CIC (*concurrent intersection chopper*), intersects the backward slice for t and the forward slice for s computed with the I2P slicer. This algorithm is the easiest chopping algorithm for concurrent programs. Our second algorithm, the *iterated two-phase chopper* (I2PC) computes a backward slice for t and then a forward slice for s, which only visits the nodes already visited during the backward slice. Its runtime complexity is in $\mathcal{O}(|E|)$, like that of the underlying I2P slicer. Our

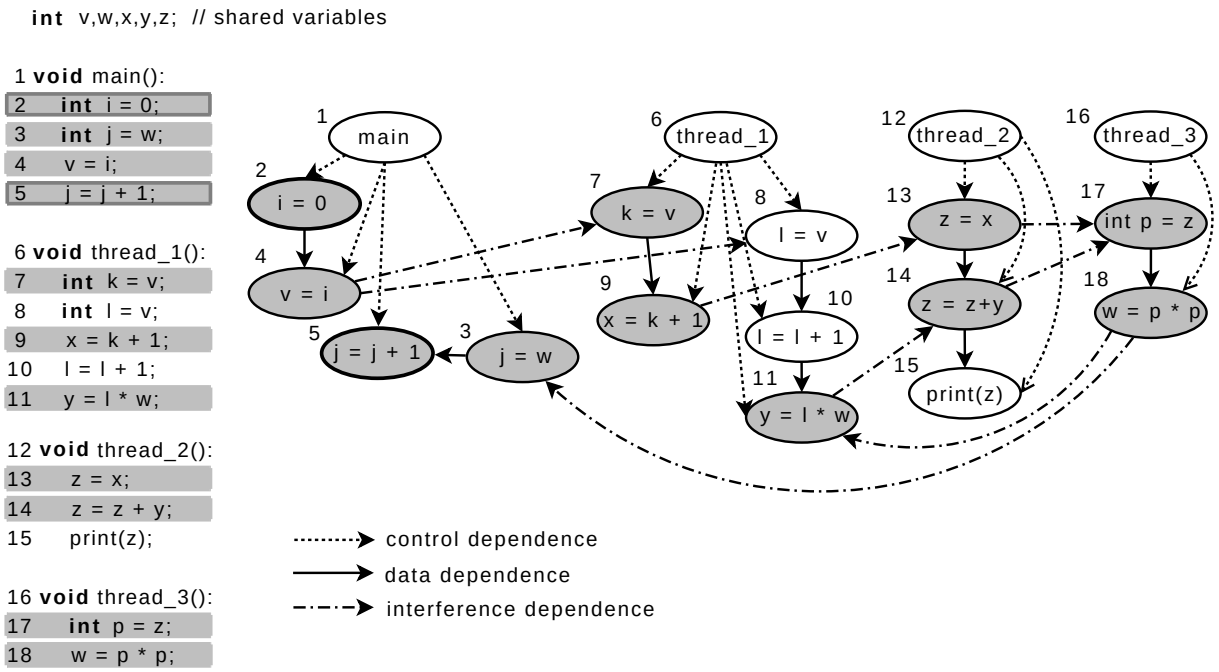


Figure 4.6.: The context-sensitive chop for chopping criterion $(2,5)$.

third algorithm, CFC, extends the fixed-point chopper from section 4.2 by substituting the two-phase slicers with iterated two-phase slicers. It has the same runtime complexity as the original fixed-point chopper.

As in the case of sequential programs, intersection-based chopping of concurrent programs is not context-sensitive. Unfortunately, the RRC cannot be applied to concurrent programs, due to interference dependence. Interference edges cannot be treated as the other kinds of edges because they cross procedure borders arbitrarily, breaking the well-formedness of SDGs for sequential programs. Our context-sensitive algorithm, the *context-sensitive chopper* (CSC), is an extension of the RRC able to handle interference dependence and has the same runtime complexity. The CSC is based on the following observation: A chop in a concurrent program can be divided into a set of sequential chops. Figure 4.6 presents an example: It shows four threads communicating via shared variables (for simplicity of presentation, all threads are assumed to happen in parallel to each other). The chop from statement 2 to statement 5 in `main` is highlighted gray. It can be partitioned into the thread-local sets $\{2,3,4,5\}$, $\{7,9,11\}$, $\{13,14\}$ and $\{17,18\}$. As one looks closer, these sets correspond to the sequential chops $\mathit{RRC}(\{2,3\},\{4,5\}) = \{2,3,4,5\}$, $\mathit{RRC}(\{7,11\},\{9,11\}) = \{7,9,11\}$, $\mathit{RRC}(\{13,14\},\{13\}) = \{13,14\}$, and $\mathit{RRC}(\{17\},\{18\}) = \{17,18\}$. These chopping criteria have the following property: The source criterion contains every node at which the whole chop enters the according thread via a concurrency edge and the original source criterion if it lies in that thread, e.g. $\{2,3\}$ in `main`. The target criterion contains every node at which the whole chop leaves the thread via

Algorithm 4.5 CSC: Context-sensitive chopping of concurrent programs.

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

```
// collect the set I of concurrency edges lying in the I2PC chop
// this should be done during the I2PC chop, to increase performance
I = {m →conc n | m,n ∈ I2PC(s,t)
S = {s}  // a set for the source criterion
T = {t}  // a set for the target criterion

// build the chopping criterion
for all m →id n ∈ I
   S = S ∪ {n}  // add sink node n to the source criterion
   T = T ∪ {m}  // add source node m to the target criterion

// compute the chop with the Reps-Rosay chopper
C = RRC(S,T)
return C
```

a concurrency edge and the original target criterion if it lies in that thread, e.g. $\{4,5\}$ in `main`. So if we have the concurrency edges that belong to the whole chop, we are able to compute the single sequential chops context-sensitively using the RRC. The CSC employs the I2PC to determine these edges, using a modified I2PC that collects the concurrency edges I that lie in its chop. Then, it picks for every thread T the concurrency edges $E \subseteq I$ that enter T and the concurrency edges $L \subseteq I$ that leave T. Let N_E be the sink nodes of the edges E, i.e. the nodes where T is entered, and let N_L be the source nodes of the edges L, i.e. the nodes where T is left. The chop $RRC(N_E,N_L)$ is the context-sensitive sequential chop from N_E to N_L. The chop for the whole program consists of the union of these chops for all threads. This algorithm has the same asymptotic runtime behavior as the original RRC: The worst-case runtime complexity of the I2PC is in $\mathcal{O}(|E|)$. As the subgraphs of the single threads in a CSDG are disjoint, the computation of the sequential chops using the RRC is in $\mathcal{O}(|E| * MaxFormalIns)$. Thus CSC's worst-case runtime complexity is in $\mathcal{O}(|E| * MaxFormalIns)$.

Figure 4.5 shows pseudocode for the CSC. The second step can be computed by a single call of RRC because the subgraphs of the threads in a CSDG are disjoint and RRC ignores concurrency edges: The source criterion is formed by the sink nodes of all concurrency edges in I plus the original source criterion, and the target criterion is formed by the source nodes of all concurrency edges in I plus the original target criterion. In our example, the concurrency edges are $I = \{4 \rightarrow_{id} 7, 9 \rightarrow_{id} 13, 13 \rightarrow_{id} 17, 18 \rightarrow_{id} 3\}$. The source criterion is $S = \{2,3,7,13,17\}$, the target criterion is $T = \{4,5,9,13,18\}$, and the chop $CSC(2,5)$ is computed by $RRC(S,T)$.

At first glance, it is not clear that CSC is context-sensitive, because set I is computed by a context-insensitive technique. However, we can show that each concurrency edge in I belongs to the context-sensitive chop. If we traverse a concurrency edge towards node n in thread θ,

then we do not know in which calling context we reach n, since interleaving cannot be forecast in general. We have to assume conservatively that we reach n in every possible context of n. Hence, if a concurrency edge $m \to n$ is in I, then every possible instance of n is in the context-sensitive forward slice for s and there must exist at least one instance of n in the context-sensitive backward slice for t. Thus, according to definition 3.10, there exists a context-sensitive path from s to t via edge $m \to n$.

Theorem 4.1. *Let G be a CSDG, and let $CSC(s,t)$ be the chop from s to t in G computed by the algorithm in Fig. 4.5. For every node n in G, the following holds:*

$$n \in CSC(s,t) \Leftrightarrow \exists \text{ a context-sensitive path } s \to^* n \to^* t.$$

Proof.

'$\Rightarrow$' For every node $v \in CSC(s,t)$, there exist nodes $s' \in S$ and $t' \in T$ such that $v \in RRC(s',t')$, thus there exists a context-sensitive sequential path $p : s' \to^* v \to^* t'$ that can be generated from nonterminal *realizable* by grammar H_{conc} of definition 3.10. We are left to show that we can extend p to a context-sensitive path $q : s \to^* s' \to^* v \to^* t' \to^* t$. We distinguish four cases:

1. $s = s' \wedge t = t'$
   In this case, $q = p$ and is therefore context-sensitive.
2. $s = s' \wedge t \neq t'$
   According to the creation of set T of the chopping criterion in Fig. 4.5, there exists a context-sensitive path $t' \to t'' \to^* t$ such that $t' \to t''$ is a concurrency edge (because t' has to be visited by the backward slicer after traversing a concurrency edge). Thus $t' \to t'' \to^* t$ has the form *conc* $(realizable\ conc)^*$ *realizable*. The concatenation $s' \to^* v \to^* t' \to t'' \to^* t$ can be generated from nonterminal *conc_realizable*.
3. $s \neq s' \wedge t = t'$
   According to the creation of set S of the chopping criterion in Fig. 4.5, there exists a context-sensitive path $s \to^* s'' \to s'$ such that $s'' \to s'$ is a concurrency edge (because s' has to be visited by the forward slicer after traversing a concurrency edge). Thus it has the form $(realizable\ conc)^*$ *conc*. The concatenation $s \to^* s'' \to s' \to^* v \to^* t'$ can be generated from nonterminal *conc_realizable*.
4. $s \neq s' \wedge t \neq t'$
   This is simply the combination of the two previous cases.

'$\Leftarrow$' We can rewrite the path as $s \to^* s' \to^* v \to^* t' \to^* t$ such that $s' \to^* v \to^* t'$ is a context-sensitive sequential path, s' is either s or is preceded by a concurrency edge and t' is

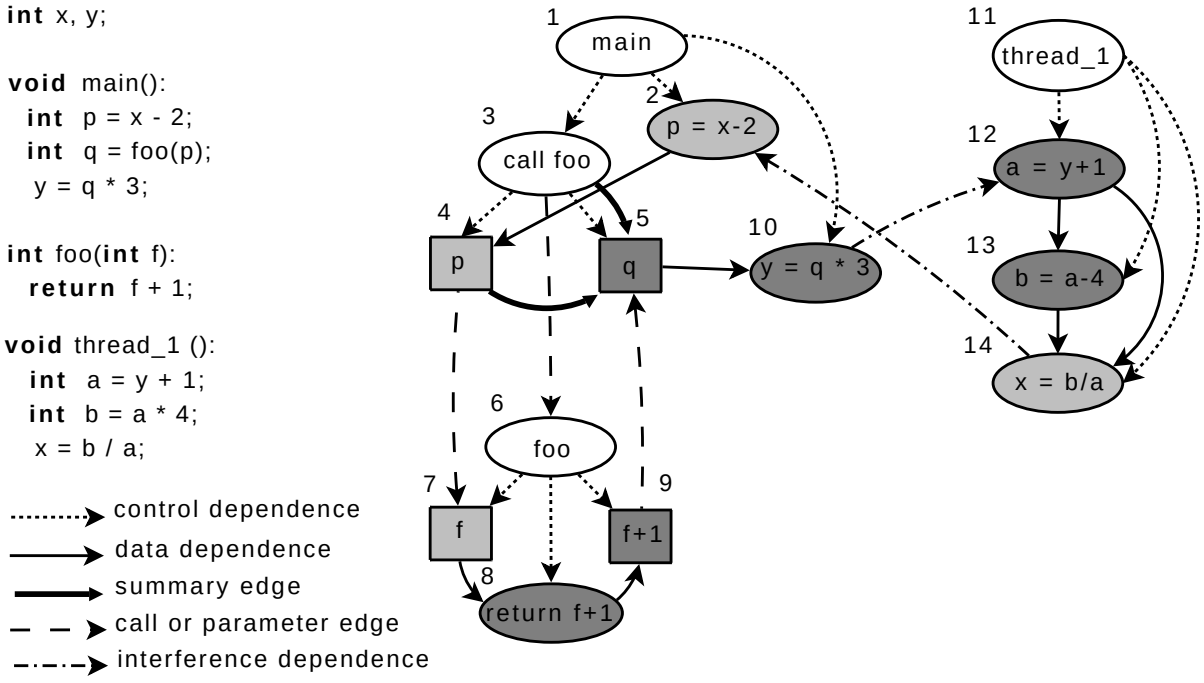


Figure 4.7.: Chops for chopping criterion $(8,13)$. The gray shaded nodes mark the context-sensitive chop, the dark gray shaded nodes mark the timing-sensitive chop.

either t or succeeded by a concurrency edge. We have to show that $s' \in S$ and $t' \in T$. In that case the algorithm is guaranteed to compute the chop $RRC(s',t')$, from which $v \in CRC(s,t,)$ follows. We have that $s,t,s',t' \in I2PC(s,t)$ because the path is context-sensitive, wherefore the backward slicer and later the forward slicer of the I2PC chopper visit these nodes. If $s' = s$ or $t' = t$, $s' \in S$ or $t' \in T$ follows trivially. For the other cases, the concurrency edge incoming to s' or outgoing from t' lies in the I2PC chop and is added to set I. It follows $s' \in S$ and $t' \in T$.

□

4.5. Timing-Sensitive Chopping

A chop $chop(s,t)$ is timing-sensitive if it contains exactly the nodes on all timing-sensitive paths between s and t in the CSDG.

Definition 4.5 (Timing-sensitive chop)**.** *A timing-sensitive chop of a CSDG G for a chopping criterion (s,t) consists of the set of nodes*

$$\{n \mid \exists c_s \rightarrow_{ts}^{*} c_n \rightarrow_{ts}^{*} c_t \text{ in } G\}.$$

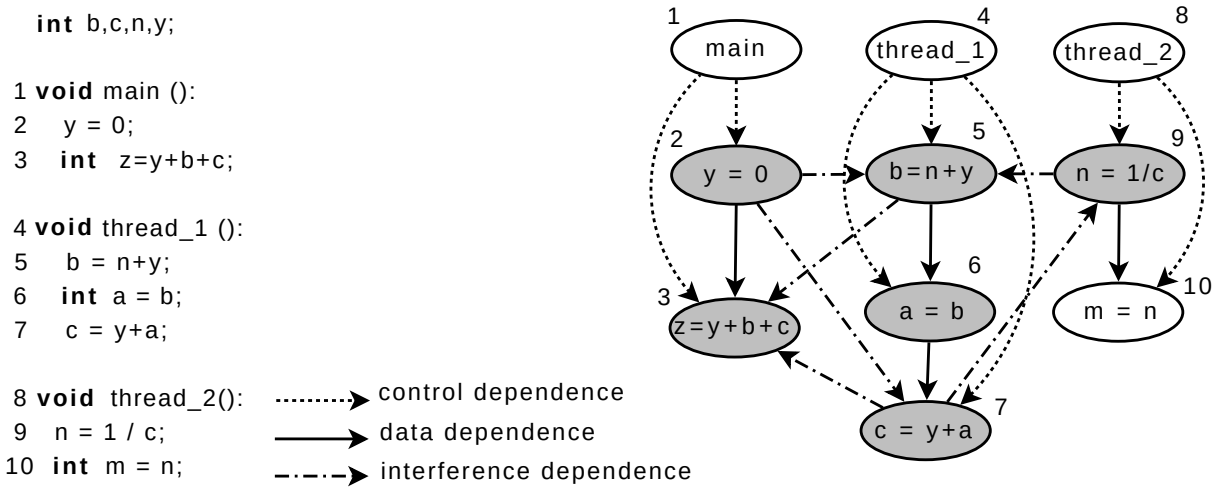


Figure 4.8.: Intersecting timing-sensitive slices does not yield timing-sensitive chops. The suchlike computed chop for chopping criterion $(2,3)$ contains time travels.

Since context-sensitive chopping treats interference dependence as being transitive, its computed chops may be timing-insensitive. Consider the example in Fig. 4.7. The gray shaded nodes are the context-sensitive chop $CSC(8,13)$. However, node 14 cannot influence node 13, because it cannot be executed before node 13. Similarly, node 8 cannot influence nodes 2, 4 and 7, so all these nodes should be removed from the chop. The first intuitive idea is to employ timing-sensitive slicing algorithms for that task. One first computes the context-sensitive chop $CSC(s,t)$ and then removes every node that is not in both the timing-sensitive backward slice for t and the timing-sensitive forward slice for s. This technique would determine the dark gray shaded nodes as the chop for $(8,13)$, which is timing-sensitive. However, it does not always compute timing-sensitive chops, for which Fig. 4.8 provides an example. The depicted graph shows the suchlike computed chop for criterion $(2,3)$. The timing-sensitive backward slice for node 3 consists of the nodes {1, 2, 3, 4, 5, 6, 7, 8, 9}. The timing-sensitive forward slice for node 2, computed on these nodes[2], visits the nodes {2, 3, 5, 6, 7, 9}, which also form the resulting chop. Unfortunately, node 2 cannot influence node 3 via node 9: All paths from node 2 to node 3 via node 9 contain the path $7 \to 9 \to 5$, which is timing-insensitive, as it leaves `thread_1` at node 7 and reenters it later at node 5. We therefore call this algorithm the *almost timing-sensitive chopper* (ATSC). A straightforward solution is to inspect every possible path in the chop for timing-insensitivity. Fortunately, there is an easier and more efficient solution.

Let us examine why ATSC is not timing-sensitive. To keep matters simple, we assume that all threads of a program may happen entirely in parallel. The prepending property in definition 3.1 ensures that a path traversed by a context-sensitive slicer can only become timing-insensitive due to leaving and reentering a thread via concurrency edges. As explained in section 3.6,

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

Table 4.3.: The state tuples for chop(2,3) in Fig. 4.8.

Node	State tuples in the backward slice	State tuples in the forward slice
2	$[2,\bot,\bot],[2,7,\bot],[2,5,\bot]$	$[2,\top,\top]$
3	$[3,\bot,\bot]$	$[3,\top,\top],[3,5,\top],[3,7,\top]$
5	$[3,5,\bot]$	$[2,5,\top]$
6	$[3,6,\bot]$	$[2,6,\top]$
7	$[3,7,\bot]$	$[2,7,\top]$
9	$[3,5,9]$	$[2,7,9]$

propagating the execution states of threads in form of state tuples during the slice enables to remain on timing-sensitive paths. Let us take a look at the state tuples computed by the chop $ATSC(2,3)$ in Fig. 4.8. Every node has only one context, so we represent it simply by the node itself. The initial state tuple for the backward slice for node 5 is $[\bot,\bot,\bot]$, where the first entry denotes `main`'s state, the second `thread_1`'s state and the third `thread_2`'s state. The slicer visits the gray highlighted nodes[3] with state tuples $\{(3,[3,\bot,\bot])$, $(2,[2,\bot,\bot])$, $(7,[3,7,\bot])$, $(6,[3,6,\bot])$, $(5,[3,5,\bot])$, $(2,[2,7,\bot])$, $(2,[2,5,\bot])$, $(9,[3,5,9])\}$. The traversal from $(9,[3,5,9])$ to node 7 is rejected because node 5 is not reachable from node 7. The initial state tuples for the forward slice for node 2 is $[\top,\top,\top]$ (dual to '$\bot$', '$\top$' represents a state that reaches every context). The slicer visits the gray shaded nodes with state tuples $\{(2,[2,\top,\top])$, $(3,[3,\top,\top])$, $(5,[2,5,\top])$, $(6,[2,6,\top])$, $(7,[2,7,\top])$, $(3,[3,5,\top])$, $(3,[3,7,\top])$, $(9,[2,7,9])\}$. The traversal from $(9,[2,7,9])$ to node 5 is rejected because node 5 is not reachable from node 7. Table 4.3 summarizes these state tuples.

We observe the following property of state tuples: A state tuple for a context c computed by the timing-sensitive backward slicer for a slicing criterion t represents a sequence of interference dependences over which c may influence t. For example, state tuple $[3,5,9]$ of node 9 describes the sequence $9 \rightarrow_{id} 5 \rightarrow_{id} 3$, through which node 9 may influence node 3. State tuple $[2,7,\bot]$ of node 2 describes the sequence $2 \rightarrow_{id} 7$, through which node 2 may influence node 3, provided that node 3 is executed after node 7. A program execution may only trigger such a sequence if its threads have not executed further than the associated state tuple. Assume that a program execution reaches node 9 in state $[3,6,9]$, then it is impossible for node 9 to influence node 3 via $9 \rightarrow_{id} 5 \rightarrow_{id} 3$ in that program run because node 5, which is needed to transfer the effects of node 9 to node 3, has already been executed. The program execution must not exceed the states in state tuple $[3,5,9]$ before reaching node 9. Formally, the state tuple describing its thread execution states has to be restrictive to $[3,5,9]$. This observation can be generalized as follows: Let $\Gamma_{back}(c,t)$ be the set of state tuples in which the timing-sensitive backward slice for a node t visited a context c. If a program execution reaches context c with state tuple Γ,

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

Algorithm 4.6 `insertChop`: Manages the updating of the worklists.

Input: A state tuple Γ_{old}, a context c_m, a worklist W, a set M, a set *STATES*.

```
/* Create an updated state tuple. */
Γ_m = updateChop(c_m, Γ_old)

/* Run the restrictive state tuple optimization. */
for all (m, c_m, Γ'_m) ∈ M :
  if ∄ thread θ : ¬reaches(Γ'_m[θ], Γ_m[θ])
    return // the new element is redundant

/* Check wheter the traversal is timing-sensitive wrt. the backward slice. */
for all (m, c_m, Γ'_m) ∈: STATES
  if ∃ thread θ : ¬reaches(Γ_m[θ], Γ'_m[θ])
    return // the traversal is not timing-sensitive

/* Insert the configuration into the worklist and mark it as visited. */
W = W ∪ {(m, c_m, Γ_m)}
M = M ∪ {(m, c_m, Γ_m)}
```

Algorithm 4.7 Procedure `updateChop`: Updating state tuples.

Input: A context c and a state tuple Γ.
Output: A state tuple Γ'.

```
Γ' = Γ  // create a copy of Γ
Let θ(c) be the thread of c
if θ(c) is not a multi-thread  // leave multi-threads alone
  Γ' = [c/θ(c)]Γ'  // set θ(c)'s state to c
return Γ'
```

then $\Gamma_{back}(c,t)$ must contain at least one state tuple Γ_{back} to which Γ is restrictive. Otherwise, c cannot influence t in this program execution.

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

We transfer that observation to chopping. Assume that the chopping algorithm ATSC visits context c with state tuples $\Gamma_{back}(c,t)$ during the backward slice and state tuples $\Gamma_{forw}(c,s)$ during the forward slice. There must exist state tuples $\Gamma_{forw} \in \Gamma_{forw}(c,t)$, $\Gamma_{back} \in \Gamma_{back}(c,s)$ such that Γ_{forw} is restrictive to Γ_{back}, otherwise c cannot be in the timing-sensitive chop for s and t, because no program execution is able to satisfy both conditions. In our example, $\Gamma_{forw}(9,2)$ for

Algorithm 4.8 TSC: A timing-sensitive chopper.

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

```
Compute the timing-sensitive slice for t via Alg. 3.12
Let STATES be the set of state tuples created during the slice for t
W = ∅, W_inner  // two worklists
M = ∅  // a set storing the visited contexts
Γ_0 = [⊤, ..., ⊤]  // an initial state tuple, having one entry per thread
for all c_s ∈ C̄(s)
  insertChop(Γ_0, c_s, W, M, STATES)

repeat  // iterate a thread-local forward slicer until W is empty
  W = W \ {(n, c_n, Γ_n)}
  W_inner = W_inner ∪ {(n, c_n, Γ_n)}  // initialize the next iteration

  while W_inner ≠ ∅
    W_inner = W_inner \ {(n, c_n, Γ_n)}
    S = S ∪ {n}  // add node n to the slice
    for all n →_e m
      if e ∈ {id}  // interference edge
        for all c_m ∈ C̄(m) : θ(c_m) ≠ θ(c_n) ∨ θ(c_n) is a multi-thread  // switch threads
          if reaches(Γ[θ(c_m)], c_m)  // remain on timing-sensitive paths
            insertChop(Γ_n, c_m, W, M, STATES)
      else if e ∈ {fork, fi}  // fork or fork-in edge
        Let c_m be the context of m connected with c_n through an incoming fork edge
        if c_m ≠ ε  // if c_m does not exist, the traversal is context-insensitive
          insertChop(Γ_n, c_m, W, M, STATES)
      else if e ∈ {jo}  // join-out edge
        for all c_m ∈ C̄(m)
          insertChop(Γ_n, c_m, W, M, STATES)
      else if e ∈ {po}
        Let c_m be the context of m directly successing c_n or equal to c_n in c_n's context graph
        if c_m ≠ ε  // if c_m does not exist, the traversal is context-insensitive
          insertChop(Γ_n, c_m, W_inner, M, STATES)
      else if e ∈ {pi, call}
        Let c_m be the context of m directly successing c_n or equal to c_n in c_n's context graph
        insertChop(Γ_n, c_m, W_inner, M, STATES)
      else  // e is an intra-procedural edge
        for all c_m ∈ C̄(m) : proc(c_m) == proc(c_n) ∧ reaches(c_n, c_m)  // remain in the procedure
          insertChop(Γ_n, c_m, W_inner, M, STATES)

until W = ∅
return S
```

node 9 is $\{[2,7,9]\}$ and $\Gamma_{back}(9,3)$ is $\{[3,5,9]\}$. There exists no possible program execution in which node 2 influences node 3 via node 9, because a state tuple Γ to which $[2,7,9]$ is restrictive cannot be restrictive to $[3,5,9]$. Our last algorithm, the *timing-sensitive chopper* (TSC), exploits that property to compute timing-sensitive chops. Its pseudocode is shown in Fig. 4.8. Called for

a chopping criterion (s,t) it first calls our timing-sensitive backward slicer, Alg. 3.12, for t and retrieves the set *STATES* of all configurations that were visited during the backward slice. Then, it performs a timing-sensitive forward slice for s, which is basically dual to Alg. 3.12. The major difference is that the update mechanism for the state tuples shown in Alg. 4.7 does not deactivate threads anymore because the traversal is in forward direction. Procedure `insertChop` in Alg. 4.6 is an extension of Alg. 3.13, which checks additionally whether the determined state tuple of context c_m is restrictive to any state tuple of a configuration of c_m stored in *STATES*. Note that the restrictive state tuple optimization in `insertChop` is inverted: Concerning forward slicing, a state tuple Γ is redundant if the same context has been annotated with a state tuple Γ' restrictive to Γ.

4.5.1. Optimizations

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

4.5.2. Correctness of TSC

The following theorems prove that TSC computes correct chops. Theorem 4.2 proves that the forward slicer computes correct forward slices, given that set *STATES* contains all possible configurations of the program.

Theorem 4.2. *Let $G = (N,E)$ be a CSDG. Let S be the set of nodes visited by the forward slicer in Alg. 4.8 for a slicing criterion $s \in N$. If STATES contains all possible configurations of the nodes in G, the following holds for every node n in G:*

$$\exists c_s \rightarrow_{ts}^{*} c_n \text{ in } G \Rightarrow n \in S.$$

Proof. We have to show that the slicer visits context c_n, which we show by a forward iteration over path $\Phi = c_s \rightarrow_{ts}^{*} c_n$. Assume that the iteration arrives at a context edge $c \rightarrow c_m$, then we have to show three properties to conclude that the slicer traverses that edge and visits context c_m. First, worklist W_{inner} must at some point contain a configuration of c. Second, the slicer must be able to detect the context edge. And third, it must be able to add a configuration of c_m to W or W_{inner}.

Since the initialization of the algorithm is bound to add a configuration of c_s to worklist W, we have a starting point for our iteration. Now, take the next context edge $c_m \rightarrow c_o$ in Φ. We know that worklist W_{inner} contains at some point at least one configuration (o, c_o, Γ) of c_o. Since the number of possible configurations is finite and each configuration can be added to the worklists at most once, the configuration is eventually processed by the thread-local slicer. Trivially, the

slicer finds the CSDG edge $e = o \to m$ associated with the context edge. It remains to show that it is able to retrieve context c_m, which is shown by a case distinction over the kind of edge e.

- e is an interference edge
  Since Φ is timing-sensitive, the state σ of c_m's thread in Γ is either $\top$ or a context d on the subpath of Φ ending at c_m or a context d' that reaches d on the subpath of Φ ending at c_m (the latter case accounts for the restrictive state tuple optimization). If the state is $\top$, then the slicer is trivially able to find c_m. Otherwise, the update mechanism guarantees that d stems from c_m's thread, which is not a multi-thread. Since Φ is timing-sensitive, d reaches c_m (and d' reaches c_m, since 'reaches' is transitive). Therefore, the slicer is able to find c_m.

- e is an interference edge
  Let $\theta(c_m)$ be c_m's thread. First, lemma 3.2 shows that $\theta(c_m)$ has not been deactivated by our update mechanism during the hitherto traversal of the path. Thus, the state of c_m's thread in Γ is either $\bot$ or a context d on the subpath of Φ starting behind c_m or a context d' reachable by d (the latter case accounts for the restrictive state tuple optimization).

- e is a fork or fork-in edge
  The slicer retrieves the context c that is connected with c_n via an incoming fork edge. If c is a context of m, the slicer proceeds with that context. Thus, the slicer is able to find c_m.

- e is a join-out edge
  The slicer proceeds with all contexts of m.

- e is an intra-procedural edge
  Context c_m must stem from the same procedure as c_o and must be reachable by c_o. Otherwise, the dependence denoted by e would not exist. Thus, the slicer is able to find c_m.

- e is an interprocedural edge
  Since the context of a parameter-passing node is mapped to the context of the associated call-, return-, start- or exit node, context c_m is either a direct successor of c_o in the context graph or, if the contexts are folded in the same fold node, it equals c_o. Thus, the slicer is able to find c_m.

Having found c_m, the slicer can proceed in two different ways: it embeds c_m in some configuration and either adds it to W or W_{inner} or rejects it due to the restrictive state tuple optimization, which means that a configuration of c_m has been added to W or W_{inner} sometime earlier. Either way, at least one configuration of c_m has been added to W or W_{inner}.

Since path Φ is finite, the iteration eventually reaches the last edge in Φ and the slicer adds a configuration of c_n to W or W_{inner} and thus, n to the slice S. □

Theorem 4.2 proves that TSC computes correct chops.

Theorem 4.3. *Let G be a CSDG and $TSC(s,t)$ be a chop from s to t in G. The following holds for every node n in G:*

$$\exists c_s \rightarrow_{ts}^* c_n \rightarrow_{ts}^* c_t \text{ in } G \Rightarrow n \in TSC(s,t).$$

Proof. We have to show that both slicers visit context c_n. This is clear for the backward slicer, because the path is timing-sensitive. The forward slicer only visits c_n if it can visit every context c in the subpath $c_s \rightarrow_{ts}^* c_n$ with a state tuple Γ_c^{forw} restrictive to that state tuple Γ_c^{back} with which the backward slicer has visited c. We show by an iteration over the subpath $c_s \rightarrow_{ts}^* c_n$ that this is the case. The iteration terminates because the subpath is finite.

The iteration starts at c_s, which is initially annotated with a state tuple $\Gamma_{c_s}^{forw}$ in which the state of c_s's thread is c_s and the states of the other threads are set to $\top$. We know that the state of c_s's thread in $\Gamma_{c_s}^{back}$ must also be c_s, so it follows that $\Gamma_{c_s}^{forw}$ is restrictive to $\Gamma_{c_s}^{back}$.

We now traverse from the current context c_i in the path to the successor c_{i+1}. Let $\Gamma_{c_{i+1}}^{forw}$ be the state tuple with which the forward slicer visits c_{i+1} and let $\Gamma_{c_{i+1}}^{back}$ be the state tuple with which the backward slicer has visited c_{i+1}. For every thread θ, we have the following three cases: θ's state in $\Gamma_{c_{i+1}}^{forw}$ is either $\top$, a context c_j lying on the subpath from c_s to c_{i+1} or a context d_j that reaches a context c_j lying on the subpath from c_s to c_{i+1}. The latter case accounts for the restrictive state tuple optimization.

Since $\top$ reaches every context, it remains to handle the second and third case. We know that θ's state in $\Gamma_{c_{i+1}}^{back}$ cannot be $\top$ because otherwise the backward slicer would not be able to visit context c_j later on. It must either be $\bot$, a context c_k lying on the subpath from c_{i+1} to c_t or a context d_k reachable by a context c_k lying on the subpath from c_{i+1} to c_{i+1}. Due to our update mechanism, c_j and c_k stem from thread θ. Since the whole path is timing-sensitive and the *reaches* relation is transitive, it follows that θ's state in $\Gamma_{c_{i+1}}^{forw}$ reaches θ's state in $\Gamma_{c_{i+1}}^{back}$.

It follows that $\Gamma_{c_{i+1}}^{forw}$ is restrictive to $\Gamma_{c_{i+1}}^{back}$. □

Since the TSC employs Alg. 3.12, its chops are not completely timing-sensitive.

4.6. Evaluation

We have integrated the presented chopping algorithms for concurrent programs into Joana and evaluated them on the benchmark used in section 3.12, using the same hardware and settings. For each benchmark program, we randomly determined 1,000 chopping criteria consisting of

one source and one target node, with the exception of KnockKnock, DaisyTest, DayTime, MolDyn, RayTracer and MonteCarlo, where we determined only 100 chopping criteria, because the algorithms ATSC and TSC were not performant enough. We measured the average chop sizes and execution times of our chopping algorithms. Furthermore, we investigated whether the more precise algorithms are able to detect more empty chops than the imprecise ones. This is valuable information for analyses that employ chopping as a preprocessing step, because if a chop $chop(s,t)$ is empty, it is guaranteed that s cannot influence t in any possible program run. In that case, the applications may omit the actual main analysis. In summary, we evaluated the following algorithms:

- CIC, intersects the forward slice for s with the backward slice for t, computed by the I2P slicer.
- I2PC, computes a backward slice for t on the forward slice for s, using the I2P slicer.
- CFC, basically the fixed-point chopper shown in Alg. 4.4, but employing the I2P slicer.
- CSC, our context-sensitive chopper shown in Alg. 4.5.
- ATSC, the almost timing-sensitive chopper.
- TSC, our timing-sensitive chopper shown in Alg. 4.8.

4.6.1. Precision

Table 4.4 shows the average chop size for each chopping algorithm and program. ATSC and TSC have no entries for DayTime, because they could not compute the chops in reasonable time. Table 4.5 summarizes these results for the three parts of our benchmark and presents, for a chosen number of pairs of chopping algorithms, the ratio of the sizes of the chops.

Context-sensitive chopping The context-sensitive chops computed by CSC were on average 6% smaller than the imprecise ones computed by CIC, and even about 25% smaller for several programs (for Barcode, Series, LUFact, SOR and SparseMatmult). Similar to their pendants for chopping of sequential programs, the intersection-based algorithms I2PC and CFC were almost as precise as the context-sensitive CSC. On average, the CSC chops were only 0.9% smaller than the I2PC chops and 0.1% smaller than the CFC chops.

It is interesting that context-sensitivity has not the same impact on chopping concurrent programs as it has on chopping sequential programs. A subset of these programs was used in the evaluation in section 4.3, where the context-sensitive chops were 13% smaller than the naïve intersection-based ones. The cause of that effect seems to be that the context-sensitive traversal of a CSDG drops the current context when it switches threads via concurrency edges.

Table 4.4.: Average size per chop (number of nodes).

Program	CIC	I2PC	CFC	CSC	ATSC	TSC
Example	303.87	295.97	289.60	289.60	289.60	289.60
ProdCons	669.53	669.51	669.50	669.50	579.43	577.00
DiskScheduler	890.64	890.58	889.75	889.75	511.8	508.24
AlarmClock	1522.56	1522.49	1522.49	1522.49	1385.10	1041.08
DiningPhils	1436.51	1436.32	1436.30	1436.30	688.93	677.53
LaplaceGrid	1798.02	1797.48	1796.60	1796.60	1152.95	993.88
SharedQueue	4212.27	4212.05	4212.05	4212.05	2560.50	2361.00
EnvDriver	3292.04	3192.71	3179.76	3179.42	3179.76	3179.42
KnockKnock	16416.76	16416.52	16390.63	16390.63	6906.12	4016.11
DaisyTest	32697.40	32697.40	32697.40	32697.40	22843.76	21910.71
DayTime	37943.78	37943.52	37906.23	37906.22	–	–
ForkJoin	4457.70	4457.61	4457.61	4457.61	469.00	414.08
Sync	4601.54	4601.44	4601.43	4601.43	671.39	619.23
Barrier	4421.42	4421.34	4421.34	4421.33	680.43	631.81
Series	660.55	499.19	490.76	490.41	142.24	133.19
LUFact	790.57	615.77	607.89	607.67	150.06	130.26
SOR	637.77	485.12	479.16	479.04	104.65	95.06
SparseMatmult	616.53	469.99	462.39	462.26	100.67	97.53
Crypt	885.66	762.85	754.58	754.46	123.93	108.25
MolDyn	6615.36	6614.94	6614.94	6614.94	1084.02	867.56
RayTracer	6227.53	6227.53	6226.61	6226.61	1049.43	709.74
MonteCarlo	12942.14	12941.90	12941.90	12941.30	2904.15	2605.22
Logger	1104.71	1091.19	1091.12	1091.12	1067.17	1065.24
Maza	1982.63	1939.92	1885.45	1853.05	1864.51	1722.78
Barcode	814.74	662.49	628.68	628.30	418.35	403.36
Guitar	758.01	708.88	702.90	701.35	695.86	690.64
J2MESafe	2330.52	2173.46	2139.00	2127.47	2122.38	2108.45
HyperM	6014.95	6012.18	6012.18	6012.18	4972.65	2785.19
Podcast	6686.62	6670.88	6670.82	6670.81	2695.41	2637.46
GoldenSMS Key	2270.08	2080.67	2046.56	2043.10	2003.32	1989.76
GoldenSMS Msg	5589.13	5571.21	5548.79	5548.31	2605.53	2191.40
GoldenSMS Rec	1544.05	1368.98	1345.15	1341.50	1213.81	1209.51
Cellsafe	14288.65	13817.44	13624.31	13513.02	13528.37	13361.42

Timing-sensitive chopping The results show that timing-sensitive chopping drastically reduces the chop sizes. The TSC chops had on average only half the size of the CIC or CSC chops. In individual cases, the chop sizes were reduced by 90% (for ForkJoin and RayTracer). There is also a significant difference between the precision of ATSC and TSC. The TSC chops were on average 9.6% smaller than the ATSC chops, in the best case, for KnockKnock, even more than 40%.

Table 4.5.: Average ratio of the chop sizes per part of the benchmark for chosen pairs of chopping algorithms. Each column shows the ratio of the average slice sizes of the first algorithm to those of the second algorithm given in the column title.

	I2PC vs CIC	CSC vs I2PC	CSC vs FPC	CSC vs CIC	TSC vs CSC	TSC vs ATSC
Bandera	99.5%	99.7%	100.0%	99.2%	66.2%	90.5%
Java Grande	90.2%	99.4%	100.0%	89.7%	16.9%	87.8%
JavaME	94.7%	98.3%	99.7%	93.1%	78.6%	93.0%
Total	94.8%	99.1%	99.9%	94.0%	53.5%	90.4%

Table 4.5 indicates that the benefits of timing-sensitive chopping may depend on the application area. The gain of precision for the programs in the Java Grande benchmark was much higher than for the other parts of the benchmark. For the Java Grande benchmark, the TSC chops contained on average only 16.9% of the nodes contained in the context-sensitive chops, whereas for the JavaME benchmark they contained on average about 78.6% of that nodes.

Program EnvDriver shows an interesting result: The ATSC chops were bigger than the CSC chops. Since EnvDriver's CSDG does not contain interference edges and thus no timing-insensitivity, this means that ATSC is due to its intersection-based nature not context-sensitive.

4.6.2. Runtime Behavior

Table 4.6 shows the average execution time in seconds the algorithms needed for one chop.

Context-sensitive chopping Since context-sensitive chopping of concurrent programs gains less precision compared to intersection-based chopping than in the sequential case, the CSC is also somewhat slower than intersection-based chopping. Algorithms CIC, I2PC and even CFC were distinctly faster than CSC. Particularly for bigger programs, KnockKnock, DayTime, DaisyTest and MonteCarlo, CIC and I2PC were more than 10 times faster than CSC. By far the most performant chopper in our evaluation was I2PC.

Timing-sensitive chopping The runtime evaluation shows that the high precision of timing-sensitive chopping is at the expense of their execution times. Only for the smaller programs ATSC and TSC could keep up with the other algorithms. For the other programs, performance declined as expected, due to their exponential asymptotic running time. In the worst case, for KnockKnock, a TSC chop needed almost 200 minutes on average to compute a single chop, ATSC even needed almost 220 minutes. Noteworthy, TSC has due to its increased precision a similar runtime performance as ATSC and even outperforms it for several programs.

Table 4.6.: Average execution time per chop (in seconds).

Program	CIC	I2PC	CFC	CSC	ATSC	TSC
Example	.007	.005	.007	.019	.015	.029
ProdCons	.013	.011	.015	.063	.071	.128
DiskScheduler	.016	.012	.023	.077	.105	.174
AlarmClock	.022	.018	.029	.135	101.793	114.086
DiningPhils	.050	.045	.075	.195	.212	.373
LaplaceGrid	.026	.021	.045	.154	6.722	7.131
SharedQueue	.058	.047	.084	.402	13.350	13.500
EnvDriver	.080	.058	.125	.197	47.987	49.548
KnockKnock	.249	.217	.592	2.910	13107.446	11831.954
DaisyTest	.510	.498	1.020	9.546	1362.810	1350.079
DayTime	.619	.580	1.674	8.177	–	–
ForkJoin	.075	.054	.093	.364	17.960	18.311
Sync	.073	.053	.094	.368	19.374	18.862
Barrier	.070	.050	.086	.336	5.821	5.867
Series	.022	.011	.016	.030	.174	.184
LUFact	.024	.013	.019	.035	.214	.225
SOR	.022	.011	.016	.028	.166	.184
SparseMatmult	.024	.012	.018	.030	.193	.204
Crypt	.027	.014	.025	.046	.503	.518
MolDyn	.109	.080	.146	.532	369.483	353.073
RayTracer	.111	.085	.210	.539	2039.421	1768.135
MonteCarlo	.219	.173	.325	2.164	169.804	178.996
Logger	.020	.012	.019	.043	.055	.090
Maza	.032	.022	.050	.081	.437	.449
Barcode	.023	.012	.020	.027	.034	.049
Guitar	.024	.012	.018	.028	.066	.085
J2MESafe	.050	.031	.068	.129	.529	.633
HyperM	.093	.074	.137	.483	268.581	244.084
Podcast	.103	.078	.140	.409	.498	.826
GoldenSMS Key	.062	.037	.076	.170	2.339	2.286
GoldenSMS Msg	.111	.081	.201	.584	30.071	23.872
GoldenSMS Rec	.044	.025	.046	.089	.650	.671
Cellsafe	.424	.353	.946	1.988	235.551	210.810

4.6.3. Detection of Empty Chops

Table 4.7 shows how many empty chops our algorithms determined for our chopping criteria. Even though context-sensitive chopping increased precision only about 4% on average, it was very effective in finding additional empty chops. On average, it determined 55.7% of the chops to be empty, compared to 42.1% empty chops found by algorithm CIC. Interestingly, I2PC, CFC and CSC found exactly the same number of empty chops. Algorithm TSC found considerably

Table 4.7.: Percentage rate of chops within our chopping criteria that the chopping algorithms detected to be empty.

Program	CIC	I2PC	CFC	CSC	ATSC	TSC
Example	57.1	63.8	63.8	63.8	63.8	63.8
ProdCons	52.9	53.1	53.1	53.1	56.3	56.3
DiskScheduler	50.4	52.5	52.5	52.5	62.1	62.1
AlarmClock	42.3	43.6	43.6	43.6	46.6	55.7
DiningPhils	44.7	48.6	48.6	48.6	59.9	59.9
LaplaceGrid	40.5	45.6	45.6	45.6	56.4	59.3
SharedQueue	41.4	43.6	43.6	43.6	59.9	62.0
EnvDriver	46.9	59.5	59.5	59.5	59.5	59.5
KnockKnock	20.0	28.0	28.0	28.0	44.0	59.0
DaisyTest	0.0	0.0	0.0	0.0	23.0	24.0
ForkJoin	43.9	45.8	45.8	45.8	79.3	79.4
Sync	42.8	45.2	45.2	45.2	71.6	71.7
Barrier	45.8	47.5	47.5	47.5	71.4	71.5
Series	46.0	79.9	79.9	79.9	89.8	90.0
LUFact	43.8	78.0	78.0	78.0	89.6	89.6
SOR	50.0	82.4	82.4	82.4	92.1	92.1
SparseMatmult	49.1	80.7	80.7	80.7	90.6	90.6
Crypt	49.0	80.5	80.5	80.5	92.0	92.0
MolDyn	41.0	45.0	45.0	45.0	71.0	71.0
RayTracer	54.0	54.0	54.0	54.0	81.0	81.0
MonteCarlo	34.0	40.0	40.0	40.0	72.0	72.0
Logger	52.6	65.9	65.9	65.9	66.6	66.9
Maza	52.1	56.7	56.7	56.7	57.1	57.2
Barcode	43.2	72.8	72.8	72.8	78.3	79.2
Guitar	52.3	76.0	76.0	76.0	76.2	76.2
J2MESafe	42.7	64.8	64.8	64.8	64.9	64.9
HyperM	34.9	42.0	42.0	42.0	42.5	64.4
Podcast	40.0	47.3	47.3	47.3	61.7	61.7
GoldenSMS Key	32.0	66.3	66.3	66.3	67.1	67.1
GoldenSMS Msg	38.4	55.3	55.3	55.3	66.7	68.1
GoldenSMS Rec	37.3	72.0	72.0	72.0	73.4	73.4
Cellsafe	26.1	44.6	44.6	44.6	44.8	44.8
Total	42.1	55.7	55.7	55.7	66.6	68.3

more empty chops than the other algorithms. On average, 68.3% of its computed chops were empty.

4.6.4. Study Summary

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

Context-sensitive chopping Compared to naïve intersection-based chopping, context-sensitive chopping reduced chop sizes about 6% on average and about 25% in the best cases. Since this gain of precision is smaller than in the case of sequential programs, CSC's runtime performance is not as competitive as that of its pendant for sequential programs. Nevertheless, CSC seems to be practical for practical programs. As in the case of chopping for sequential programs, there are no arguments in favor of algorithm CIC. Algorithms I2PC and CFC have a similar implementation effort, are equally performant or even faster and are more precise. Since both are also almost as precise as CSC, with respect to both chop sizes and finding empty chops, they are genuine alternatives to CSC.

Timing-sensitive chopping Timing-sensitive chopping strongly increased precision. The TSC chops were on average about 46.5% smaller than the context-sensitive chops. The TSC also detects a significant number of empty chops that are considered not empty by the intersection-based and context-sensitive choppers. However, one has to pay a price for that precision. The algorithms TSC and ATSC do not scale well, because the underlying technique has a worst-case exponential runtime behavior. Overall, TSC computes smaller chops than ATSC and is able to outperform ATSC due to its increased precision, thus we consider TSC superior to ATSC.

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

Our case study lacks big programs and consists only of 33 programs. Table 4.6 shows that the execution times of timing-sensitive chopping may vary greatly between programs of similar size (e.g. AlarmClock and DiningPhils). In order to make a robust statement about the practicability of timing-sensitive chopping, an extended runtime evaluation on a much bigger and more differentiated benchmark is needed.

In most parts of our evaluation, we have only computed 100 - 1000 chops per program. Computing all possible chops would not have been possible in reasonable time. However, we argue that our sample chops are sufficient for a qualitative comparison of the algorithms in terms of precision and runtime behavior, even though the concrete numbers might differ for a different set of chopping criteria.

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

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

4.7. Discussion

TSC basically computes one forward and one backward slice, therefore it bears the same worst-case runtime complexity as timing-sensitive slicing, being $\mathcal{O}(|N|^{pt})$. The present evaluation and our recent evaluation of timing-sensitive slicing [46] indicate that timing-sensitive slicing and chopping, using the optimizations developed so far by Krinke [75], Nanda [106] and us, can handle programs with 5,000 -10,000 lines of code in reasonable time. New optimizations to further relieve the combinatorial explosion remain an important issue for future work.

An interesting capacity of timing-sensitive chopping is its ability to detect empty chops that are deemed non-empty by more imprecise techniques. Future work could explore if it is possible to detect these empty chops without computing the complete timing-sensitive forward and backward slices. This could result in a practical timing-sensitive scanner for empty chops.

We have published earlier versions of this chapter [42, 43], which also contain evaluations of the chopping algorithms executed on a subset of the benchmark used here. Several differences shall be pointed out: Since Joana's SDG generator has been developed further since then [49], the CSDGs differ both in size and structure. Furthermore, we have developed several of our optimizations only later, hence the ATSC and TSC algorithms used in the earlier versions do not correspond to those presented here. This affects particularly the runtime behavior of the algorithms. However, the qualitative results in [42, 43] support those observed in this chapter.

4.8. Related Work

Chopping originates from Jackson and Rollins' work on modularizing SDGs for reverse engineering [67]. They define chops to be confined to a single procedure. The source and the target of a chop must be within the same procedure, and only that procedure's code is analyzed. They suggest an iterative approach to extend such an intra-procedural chop to procedures called within that chop: If the chop contains a call to another procedure, another intra-procedural chop is computed, where the parameter variables of the called procedure form the source criterion, and the return variables of the called procedure form the target criterion. This kind of chopping is called *same level chopping*, because it does not take callers of the initial procedure into account.

Reps and Rosay [120] extend Jackson and Rollins' same-level chopping to unbound chopping. Their chopping algorithm, the RRC described in section 4.1.2, is context-sensitive and the state-of-the-art algorithm for sequential programs. The authors integrated their algorithm in the Wisconsin Program-Slicing Tool for C, which was the foundation of CodeSurfer [15], a commercial program analysis tool for C.

Krinke [74] developed a new same-level chopper called *summary-merged chopper*, which is context-sensitive, as the iterative approach of Jackson and Rollins, and much faster in practice.

In a subsequent work [76], he introduces the concept of *barrier chopping* and *slicing*, where a user can specify a barrier consisting of nodes or edges which must not be crossed by the chopping algorithm. This permits to exclude program parts from the analysis one is not interested in, e.g. library calls. Krinke implemented all these algorithms in the VALSOFT system [75].

5. Information Flow Control For Concurrent Programs

Information flow control is concerned with the security of sensitive information processed by software. Whereas security of information is predominantly understood as an *accessing* problem, tackled by techniques such as access control or encryption, information flow control sees it as a *processing* problem: Software that rightfully accesses sensitive information might intentionally or unintentionally leak it to unauthorized sinks, violating its *confidentiality*, or taint it with data from unauthorized sources, violating its *integrity*. It is complementary to access control, encryption and likewise techniques and can be combined with them in order to achieve *end-to-end* security, protecting sensitive information during its whole lifetime.

Sensitive information flow in a program can refer to *confidentiality* or *integrity*. Information flow preserves confidentiality if information from confidential sources does not flow to unauthorized recipients. It preserves integrity if information from unauthorized sources does not flow to confidential recipients. Biba [22] observed that integrity is dual to confidentiality, so an IFC technique verifying the one can be adopted to verify the other one. This thesis focuses on confidentiality, the presented techniques can be adopted to verify integrity as well.

Types of information leaks

Programs contain various kinds of information flow, which may intentionally or unintentionally unveil confidential data. The most intuitive kinds of information flow are *explicit* and *implicit flow*, the former resulting from assignments and the latter from conditional branching. The program on the left side of Fig. 5.1 leaks at statement `print(0)` information about the input PIN being smaller than 1234, which is an illicit implicit flow. The program also directly prints the PIN stored in variable `y`, which is an illicit explicit flow. Concurrent programs may additionally contain *possibilistic* and *probabilistic channels*. Possibilistic channels are information leaks that depend on the program's interleaving – a program run may leak information or not. This happens in the program in the mid of Fig. 5.1, which only leaks the PIN via `print(x)` if the assignment `x = y` happens between the two statements of thread 1. Probabilistic channels also depend on interleaving, but leak information through the probability distribution of interleaving orders. The program on the right side of Fig. 5.1 serves as an example: There is no possibilistic channel leaking information about the PIN in this program, because the printed value of `x` is always 0 or 1. But the PIN's value may alter the *probabilities* of these possible outputs, because the running time of the loop may influence the interleaving order of the two assignments to

```
void main():
  x = inputPIN();
  if (x < 1234)
     print(0);
  y = x;
  print(y);
```

```
void thread_1():
  x = input();
  print(x);

void thread_2():
  y = inputPIN();
  x = y;
```

```
void thread_1():
  x = 0;
  print(x);

void thread_2():
  y = inputPIN();
  while (y != 0)
     y--;
  x = 1;
  print(2);
```

Figure 5.1.: Examples for information leaks in sequential programs (left), for a possibilistic channel (mid) and for probabilistic channels (right). The threads are meant to execute concurrently.

`x`. Assume that the scheduler picks each thread with the same probability and schedules after every executed statement. If the PIN is 0, then 0 is printed with probability $\frac{44}{64}$, if PIN is 1, this probability raises to $\frac{57}{64}$. If an attacker knows these probability distributions and can repeatedly observe program runs with the same PIN, he is able to deduce information about it. Note that the program contains another probabilistic channel: The PIN may also influence the probability in which order the two `print`-statements are executed.

Lampson [83] declares all kinds of information flow other than explicit flow as *channels*, arguing that they propagate information only as a secondary effect. Following Sabelfeld and Myers [126], information leaks can be categorized as follows:

- Illicit explicit flow leaks information through assignments.
- Illicit implicit flow leaks information through conditional branching.
- Possibilistic channels leak information by changing the set of possibly occurring events.
- Probabilistic channels leak information by changing the probability with which an event occurs.
- Termination channels leak information about data which caused (or not caused) nontermination. The program on the right side of Fig. 5.1 contains a termination channel: If the input PIN is negative, the loop diverges and 2 is never printed to the screen.
- Timing channels leak information by influencing the time which elapses between two events. Timing channels can be a serious threat to security, an infamous case being the timing attack on RSA encryption [70].
- Resource channels leak information through the degree of usage of resources, such as memory, disk space or CPU.

- Physical channels leak information by changing the degree of physically measurable entities, such as heat or noise development or power consumption.

Since channels manifest themselves in very different ways, the usage of different, specific countermeasures seems to be the most promising approach. Physical channels can best be avoided by physical isolation of the hardware, resource channels can be encountered by denying unauthorized users information about resources. Timing channels can be closed by delaying the returning from a critical computation until a fixed time interval has passed. The other kinds of channels as well as explicit and implicit flow are the subject of *language-based* information flow control. Language-based IFC inspects the code of a program in order to reconstruct and validate the information flow between the program statements. This thesis investigates the usage of language-based IFC for the detection of information leaks in concurrent programs. It thereby excludes timing-, resource- and physical channels. In the remainder, the terms 'information flow control' and 'IFC' refer to language-based information flow control.

5.1. Background

The degree of confidentiality granted by an IFC technique depends on its *security policy*, its *attacker model*, its *security property*, the *classification* of the program in question and the *security constraint* used to enforce the security property.

The *security policy* defines the admissible flows of information, which are usually specified by means of a set of security levels and a flow relation $\rightsquigarrow$ on pairs of security levels. The information flows in a program must be in accordance with this flow relation. The security policy may additionally define exceptions of the flow rules, so-called *declassifications* [129], which permit inevitable leakage of information in a controlled manner.

The *attacker model* defines the abilities of an attacker. A common assumption is that the attacker knows the source code and is able to see certain parts of the program behavior, the so-called *observable behavior*. Unfortunately, there is no real agreement on which parts of the program behavior should be taken into account. Typical points of contention are whether an attacker is able to observe the sheer termination of the program or not, whether he is able to observe parts of the memory at any time or is restricted to I/O events, or whether he is able to exploit timing-, physical- or resource channels. Defining a reasonable attacker model is a major prerequisite for the development of an IFC technique.

The *classification* of a program states which of its parts constitute its observable behavior and which parts provide information of a certain security level. Typical methods of classification are the assignment of security levels to variables, to events, or to I/O streams.

The *security property* defines restrictions on the observable behavior a program has to adhere in order to be secure with respect to the given security policy, attacker model and classification.

The *security constraint* defines necessary constraints on the source code of a program, whose abidance guarantees that the program satisfies the security property. They usually, but not necessarily, constrain the syntax of the program.

5.1.1. Noninterference

A widely accepted security policy, which is also used in this work, is the *noninterference policy* [47]. Given a set of security levels and a flow relation $\rightsquigarrow$ on pairs of security levels, it requires that information of a level l' may only interfere with information of a level l if $l' \rightsquigarrow l$ holds. In the context of language-based IFC noninterference is usually understood as follows: There exists an attacker who is authorized with a certain security level l, which permits him to observe program behavior classified with l or a level $l' \rightsquigarrow l$. A program is said to be noninterferent if data of a level $h \not\rightsquigarrow l$ does not affect that behavior.

Note that level l partitions the set of security levels into two equivalence classes: A class *low* containing l and all levels $l' \rightsquigarrow l$ and a class *high* containing the remaining levels. We will often use these two equivalence classes to simplify our explanations. The program behavior observable by the attacker is the *low-observable behavior*. Keep in mind that this partition into high and low is always relative to the level of a certain attacker. Different attackers may have different levels, and a program has to be noninterferent with respect to all possible attackers.

Security properties based on noninterference typically require that changes in the high input data of a program do not change the low-observable behavior. As a consequence, the low-observable behavior of the program contains no exploitable information about the high input data, so that an attacker cannot draw conclusions about it by observing program runs.

The input a program receives during its execution can be seen as a list, whose content is predetermined at the program start and whose elements are consumed subsequently by input reading operations. In order to signal which input value has which security level, we assume one input list per security level. Two inputs are said to be *low-equivalent* if for each security level $l \in \mathit{low}$ the corresponding lists are equal. Two program runs are called *low-equivalent* if their low-observable behavior is indistinguishable for the attacker. Based on these terms, security properties based on noninterference typically have the following form:

Definition 5.1 (Noninterference)**.** *A program p is noninterferent if it satisfies the following condition: Whenever p is run on two low-equivalent inputs, then the resulting program runs are low-equivalent.*

This is an *abstract security property*, which has to be instantiated with a suitable definition of low-observable behavior. We will do so in section 5.4.

5.1.2. Denning-Style Information Flow Control

Some of the earliest work on IFC has been published by Denning and Denning [37, 38]. Even though the notion of noninterference was defined only several years later by Goguen and Meseguer [47], their technique enforces an intuitive understanding of noninterference. The ideas presented in their publications serve as the very foundation of today's noninterference-based IFC techniques and are therefore summarized here.

Denning [37] suggested arranging security levels in a complete bound lattice such that piece of information of a certain level is only allowed to influence other pieces of the same or a higher level. Such a lattice is called a *security lattice*. It permits to use the meet and join operators to compute the actual security level of information flowing through a program statement. We use the security lattice $\mathfrak{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, taken from [52], in the remainder. L is the set of security levels, $\sqsubseteq$ defines their partial order, $\sqcap$ and $\sqcup$ are the meet and join operations, $\bot$ is the bottom element and $\top$ is the top element[1].

Denning formulates her IFC technique with the help of an *abstract program S*, which is recursively defined by:

1. (Statement) S is an elementary statement, e.g. an assignment or I/O.
2. (Composition) There exist S_1 and S_2 such that $S = S_1; S_2$.
3. (Choice) There exist $S_1, \ldots, S_m$ and an m-valued variable c such that $S = c : S_1, \ldots, S_m$.

Point 3 declares conditional structures, where ':' is a choice operator that uses variable c to choose among the alternatives $S_1, \ldots, S_m$.

The abstract program works on a *storage* $M = \{a, b, \ldots\}$, which is a set of locations. Each location is assigned a security level from the security lattice in charge. This assignment can be static, where it is constant throughout the program execution, or dynamic, where it depends on the content of the location. The security level of a location a is addressed by $lvl(a)$.

Program S is declared secure if all program statements are secure. An elementary statement, $b = a_1, \ldots, a_n$, is secure if any explicit flow caused by it is secure. This is the case if

$$(lvl(a_1) \sqcup \cdots \sqcup lvl(a_n)) \sqsubseteq lvl(b)$$

holds after the execution of the statement, i.e. the security level of location b permits the storage of the result of the computation. The choice operator $S = c : S_1, \ldots, S_m$ is secure if each S_k is secure and the implicit flows caused by it are secure: Let $b_1, \ldots, b_n$ be the possible locations in which $S_1, \ldots, S_m$ may store their computations, then the implicit flows are secure if

$$lvl(c) \sqsubseteq (lvl(b_1) \sqcap \cdots \sqcap lvl(b_n))$$

[1] It is always possible to extend a given lattice with $\top$ and $\bot$.

holds after the execution of S, i.e. the information about the branching is allowed to flow into the locations written to in the branches. Finally, the composition $S_1; S_2$ is secure if S_1 and S_2 are secure.

The subsequent work of Denning and Denning [38] extends the above mechanism to a certification procedure committed by a compiler. For that purpose, they convert the above abstract flow rules into a concrete *certification semantics*, which is processed by the compiler during the semantic analysis.

5.1.3. Declassification

The information flow rules defined by a security lattice are generally too restrictive for practical programs, because some leakage of information is often inevitable. For example, a password check leaks information about the given password being correct or not. Suchlike information leaks are often unavoidable and it should be possible to permit them. The idea of *declassification* is to mark leaks as acceptable, so that the IFC analysis does not reject the program.

Sabelfeld and Sands [129, 130] identify four different dimensions of declassification, namely *what* information is declassified, *where* is it declassified, *when* is it declassified and *who* declassified it. They further argue that declassification mechanisms should satisfy four basic principles:

1. Semantic consistency
   Semantics-preserving transformations of declassification-free subprograms should not change the (in)security of a program.

2. Conservativity
   A security policy which allows declassification should be weaker than noninterference. A secure program without declassification should be noninterferent.

3. Monotonicity of release
   Adding a declassification should not turn a secure program insecure.

4. Non-occlusion
   A declassification should not mask other covert information leaks.

To date, there does not exist a dominating approach to declassification, because most existing approaches only address one or two dimensions of declassification, and even within one dimension no approach was yet able to become widely accepted. Thus, there does not exist a generally accepted extension of noninterference that accounts for declassification. A good overview, categorization and comparison of existing declassification mechanisms is presented in Sabelfeld and Sands' recent publication [130].

5.1.4. Slicing-Based Information Flow Control

The currently predominant approach to enforce noninterference is the employment of non-standard type systems, which encode the elements of the security lattice via a type hierarchy such that the subtype relation corresponds to the partial order relation of the lattice [126]. Denning's flow rules are translated into typing rules such that a well-typed program is noninterferent. These *security-type systems* have several advantages and disadvantages, which have been closely investigated by Hammer [52, 53]: IFC based on security-type systems is fast and scales well, its soundness can be rigorously proven with proof techniques like automated theorem provers [139] and it (often) allows a modular verification of programs. Modularity is a strongly desired property, because it enables to certify libraries as secure. On the contrary, modularity is achieved through conservativity – security-type systems are usually context-, flow- and object-insensitive, which, according to Hammer, may result in many false alarms. Furthermore, the integration of a security-type system into a programming language results in a new, separate language. It is not possible to analyze existing programs with a security-type system without (manually) converting the programs into the corresponding language. And finally, the implementation of a security-type system for mature languages is difficult and has to be redone for every desired language, wherefore only a few such systems are known to exist; for example, FlowCaml [132], a language similar to OCaml, or Jif [103], a Java-like language.

Several authors suggest using dependence graphs and slicing to address these difficulties [52, 58, 138, 159]. It is known for some time that slicing allows to enforce noninterference in sequential programs: If the slice for the statements that constitute the low-observable behavior is free of statements reading high input, the program is noninterferent [7, 21, 138]. However, the technical challenges concerning the creation of SDGs for full-fledged languages and real-world applications have prevented implementations of slicing-based IFC until only recently. The first reported implementations stem from Yokomori et al. [159], for Pascal, and from Hammer et al. [52, 58], for sequential Java bytecode. A machine-checked proof that slicing of sequential interprocedural programs can be used to enforce noninterference has recently been published by Wasserrab and Lohner [153].

Slicing-based IFC promises higher precision, due to its context-, flow- and object-sensitivity, and it can be applied to existing programs and languages. Moreover, it permits to decouple the IFC algorithm from the concrete programming language: The same algorithm can be used for every language for which a SDG generator is available. Of course, slicing-based IFC also has its weaknesses. The construction of a SDG is a whole-program analysis that analyzes procedures only in the context of their usage in the analyzed program. To date, there exists no method to create a SDG for a library which accounts for every possible usage of the library. Thus, a modular verification of information flow via SDGs and slicing is currently not possible.

Furthermore, slicing-based IFC has higher runtime costs than type systems and therefore a limited range, and proving its soundness is difficult.

The approach of Hammer et al. [52, 58] is currently the most sophisticated, as it allows the usage of arbitrary security lattices and provides a declassification mechanism. The user annotates statements identified as sources of information with a *provided security level*, P, and statements which output information with a *required security level*, R. The technique verifies whether information flowing to a statement with a required security level is allowed to do so. For that purpose, it basically realizes Denning's flow rules by a monotone data flow analysis framework (cf. sect. 2.4.1) working on SDGs. The lattice of data flow facts is the security lattice $\mathfrak{L}$ in charge, and the transfer-function of a node n, $f_n(l) = (l \setminus kill(n)) \sqcup gen(n)$, is composed of the *gen*-function

$$gen(n) = \begin{cases} P(n) & n \text{ is annotated with a provided security level} \\ \bot & otherwise \end{cases}$$

and the *kill*-function

$$kill(n) = \top.$$

The *gen*-function returns the provided level, if existing, and otherwise $\bot$, the identity element for $\sqcup$. Statements do not kill information flowing through, wherefore the *kill*-function always returns $\top$, the identity element for $\sqcap$.

The *actual security level* $S(n)$ of information flowing to a node n is determined by the framework functions $out(n) = f_n(in(n))$ and $in(n) = \bigsqcup_{m \in pred(n)} out(m)$:

$$S(n) = out(n) = gen(n) \sqcup \bigsqcup_{m \in pred(n)} S(m).$$

Hammer shows that the transfer functions are monotone and distributive, which means that $S(n)$ always has a minimal fixed point, which is due to the distributivity of the transfer functions equivalent to the precise meet-over-all-paths solution. The final solution of $S(n)$ describes the maximal level of information that may actually flow through node n.

Having the minimal fixed point solutions $S(n)$ for all nodes n in the SDG, it remains to check whether the information flow violates the flow rules of lattice $\mathfrak{L}$. This is the case if the required security level of any statement s disallows information of the actual security level $S(s)$. Hence, the following equation must hold for every node n in the SDG :

$$S(n) \sqsubseteq \begin{cases} R(n) & n \text{ is annotated with a required security level} \\ \top & otherwise. \end{cases} \quad [5.1]$$

Declassification

Hammer et al.'s approach includes a declassification mechanism, which can be categorized as a *where*-declassification [130]. A declassification is a node d with both a provided and a required security level. Incoming information is converted to level $P(d)$, and the computation of the actual security level is adjusted as follows:

$$S(n) = \begin{cases} P(n) & n \text{ is a declassification} \\ gen(n) \sqcup \bigsqcup_{m \in pred(n)} S(m) & \text{otherwise} \end{cases}$$

In order to prevent declassifications from declassifying arbitrary information, the security levels of incoming information have to be smaller than d's required level, $R(d)$. This leads to the following equation that must hold for every declassification d in the SDG in order to avoid illicit information flow. For all other nodes in the SDG, equation 5.1 must hold.

$$R(d) \sqsupseteq \bigsqcup_{m \in pred(d)} S(m)$$

The authors proved that the following restrictions to the declassification mechanism guarantee conservativity and monotonicity of release[2]. Let d be a declassification:

- $R(d) \sqsupseteq \bigsqcup_{m \in pred(d)} S(m)$, and
- $R(d) \sqsupseteq P(d)$, i.e. declassification works only 'down the lattice'.

5.2. Information Leaks in Concurrent Programs

Programs with threads and shared-memory communication may contain possibilistic and probabilistic channels. Interleaving can lead to a *set* of possible observable behaviors per input and possibilistic and probabilistic channels may manifest in these sets. IFC for concurrent programs has to deal with these channels.

If a program contains a possibilistic channel, then changes in the high input data can cause *different sets* of possible low-observable behaviors. Probabilistic channels are even more subtle; here changes in the high input data may alter the *probability* of a low-observable behavior. Note that possibilistic channels imply probabilistic channels: If a change in the high input data leads to different sets of possible low-observable behaviors due to a possibilistic channel, several of these behaviors have probability 0 in one set and > 0 in the other, thus there is also a probabilistic channel. On the other hand, a program containing probabilistic channels may indeed produce the same sets of possible low-observable behaviors, only with different probabilities. We focus on detecting probabilistic channels, covering possibilistic channels implicitly.

[2] It is not clear whether these are necessary conditions.

Probabilistic channels

Looking closer at probabilistic channels reveals that there exist different kinds of them. Consider again the program on the right side of Fig. 5.1 and assume that the output of the `print`-statements constitutes its low-observable behavior. We described two probabilistic channels in that program in the introduction, which in fact represent two different kinds of probabilistic channels: The first kind are *probabilistic data channels*, which leak information via the quality of an observable event. In the example, `print(x)` leaks information about the PIN, because the probability distribution of the interleaving order of statements `x = 0` and `x = 1` is influenced by the running time of the loop and thus transfers information about the PIN to the printed value of `x`. The second kind are *probabilistic order channels*, which leak information via the relative ordering of observable events. In the example, the probability distribution of the relative order of `print(x)` and `print(2)` contains information about the PIN, again because the running time of the loop influences the outcome of that interleaving.

We distinguish these two kinds of probabilistic channels because they require different treatment. Probabilistic data channels can be tracked down by inspecting implicit and explicit flow, whereas the detection of probabilistic order channels requires an inspection of the possible interleaving of low-observable events.

Preconditions of probabilistic channels

According to Zdancevic and Myers [160], probabilistic channels have two necessary preconditions.

First, a program must contain *conflicts*, concurrently executing statements whose interleaving order is not fixed. A conflict is a *data conflict* if both statements access the same shared variable v and at least one of them defines v. The example on the right side of Fig. 5.1 contains two data conflicts, between statement `x = 1` in thread 2 and statements `x = 0` and `print(x)` in thread 1. Data conflicts may cause probabilistic data channels. A conflict is an *order conflict* if both statements produce observable behavior. The example contains one order conflict, between `print(x)` and `print(2)`. Order conflicts may cause probabilistic order channels.

Second, there must exist a program part whose execution time depends on high data and influences the outcome of a conflict. Without such a program part a conflict is benign. In our example, all conflicts are influenced by the loop, whose running time depends on the PIN. Hence, the data and order conflicts transfer information about the PIN to the observable behavior, resulting in the probabilistic data and order channels.

Note that it is not sufficient to consider only loops or conditional structures as candidates for such program parts. A single statement may execute with different speed when fed with different data, due to techniques like caching or pipelining. These different execution times can

be sufficient to influence the outcome of a conflict. An in-depth investigation of the effects of caching on program security has been committed by Agat [8].

5.3. Probabilistic Noninterference

A suitable abstract security property for concurrent programs is *probabilistic noninterference* [150]. An input of a concurrent program p may cause a set of possible program runs, each of them with a certain probability, which add up to 1. Probabilistic noninterference can be defined as follows:

Definition 5.2 (Probabilistic noninterference)**.** *A program p is probabilistic noninterferent if for all pairs* (t,u) *of low-equivalent inputs the following holds:*

Let Θ *be the set of possible program runs resulting from t and u. For each* $T \in \Theta$*, the following must hold: Let* $\mathfrak{T}$ *be the set of program runs possibly caused by t that are low-equivalent to T. Let* $\mathfrak{U}$ *be the analogical set for u. Then the sum of the probabilities of the program runs in* $\mathfrak{T}$ *must equal the sum of those in* $\mathfrak{U}$*.*

There emerged two general approaches to enforce probabilistic noninterference: The first approach aims to ensure that high input cannot influence the probability of the outcome of a conflict (e.g. [28, 94, 128, 135, 150]). This is usually done by enforcing security constraints based on *probabilistic bisimulation* [84], a form of bisimulation in which transitions are labeled with probabilities, which is used to guarantee that two low-equivalent inputs cause the same possible low-observable behaviors with the same probabilities.

The second approach aims to ensure that conflicts cannot influence the low-observable behavior (e.g. [64, 98, 121, 145, 160]). This is achieved by using security constraints that guarantee *low-observational determinism* [98, 121]. In a low-observational deterministic program a certain input may still cause a set of possible program executions, but all of them are low-equivalent. From the point of view of an attacker the program behaves deterministically. If additionally all program runs resulting from low-equivalent inputs are low-equivalent, the program is *low-security observational deterministic* [98, 121], which is a specialization of probabilistic noninterference:

Definition 5.3 (Low-security observational determinism)**.** *A program p is low-security observational deterministic if it satisfies the following condition: Whenever p is run on two low-equivalent inputs, then all possible program runs are low-equivalent.*

The differences between these two approaches are subtle, but have important consequences. The first approach is generally less restrictive, because it is closer to the definition of probabilistic noninterference. For example, security properties based on the second approach often reject programs that do not work with high data at all, simply because their low-observable

behavior is not deterministic [64, 145, 160]. The second approach, however, makes the concrete scheduling strategy insignificant, because it cannot manifest in the probabilities of the low-observable behaviors. It allows to define *scheduler-independent* security properties, which hold for every possible scheduler. In turn, independence from the scheduler is the major problem of the first approach. There exists no reasonable scheduler-independent security property based thereon, and much effort is spent in the development of security properties that comprise as many scheduling strategies as possible. For example, *strong security* [128], a well-known security property of this category, comprises the class of schedulers whose scheduling strategies cannot be influenced by high data.

We chose to base our IFC technique on low-observational determinism, because we consider dependence from schedulers dangerous. If the confidentiality of a program is only guaranteed for a certain set of schedulers, it is fragile: It can be intentionally or unintentionally flawed by choosing an inappropriate scheduler. Since we have a flow-sensitive program analysis at hand, we were able to reduce the restrictions coming with low-observational determinism to a minimum.

Zdancevic and Myers [160] showed that programs complying with the following security constraint are low-security observational deterministic: (1) The program parts contributing to the low-observable behavior are free of conflicts, which means that the program is low-observational deterministic, and (2) the implicit and explicit flow in the program does not transfer high data to the low-observable behavior. This observation was the main inspiration for our IFC technique.

5.4. A Trace-Based Definition of Low-security Observational Determinism

In this section we develop our security property, a trace-based definition of low-security observational determinism. To this end, we define the low-observable behavior of programs via traces.

We are interested in three aspects of a program statement: (1) The sets of variables it defines or uses, (2) whether it branches the control flow, and (3) to which other statements it may happen in parallel. Since other aspects are of no interest, we forgo to define a concrete programming language. Instead, we assume several wellformedness properties of the structure of the program and of the semantics of the statements, similar to those made by Wasserrab et al. [154]:

- The control flow of the program can be modeled by a TCFG as defined in definition 3.1.
- The execution of a statement leaves all variables that are not in its *def*-set unchanged.
- The semantic effect of a statement *s* is deterministic and depends solely on the values of the variables in its *use*-set. Assume that two program states agree on all values of the

variables in the *use*-set. Then the states resulting from executing s agree on all variables in the *def*-set of s and both traces proceed with the same branch in the control flow.

- The first usage of a variable in a trace has to be preceded by at least one definition (no undefined variables).

5.4.1. Traces

Let p be a program that operates on a set *Var* of variables. $Ops(p)$ is the (possibly infinite) set of all possible *operations* executed by p. An operation is an instance of a statement and $stmt(o)$ maps operation o to that statement. $use(o) = use(stmt(o))$ is the set of variables used by o, and $def(o) = def(stmt(o))$ is the set of variables defined by o. Operations are unique, i.e. an operation is executed at most once in a trace and is uniquely identifiable among different traces, by the sequence of procedure calls and predicates leading to the operation. How this is done exactly is described in definition 5.5.

A *memory* m is a map from variables to values, where $m(v)$ denotes the value of v in m. The undefined value is denoted by $\perp$. The projection of a memory m to a subset $V \subseteq Var$ of variables is denoted by $m\,|_V$. Two memories m and m' are equal with respect to a set $V \subseteq Var$, written $m\,|_V = m'\,|_V$, iff $\forall v \in V : m(v) = m'(v)$.

A *trace* T of a program p is a (possibly infinite) list of ordered *configurations*

$$(\overline{m_0}, o_0, m_0) \cdots (\overline{m_j}, o_j, m_j) \cdots$$

where a configuration $(\overline{m_i}, o_i, m_i)$ in T consists of an operation $o_i \in Ops(p)$, the memory $\overline{m_i}$ right before o_i's execution and the resulting memory m_i. For every two adjacent configurations, $(\overline{m_i}, o_i, m_i)$ and $(\overline{m_{i+1}}, o_{i+1}, m_{i+1})$, $m_i = \overline{m_{i+1}}$ holds. All traces of the same program start with the same operation *start*, representing the program start. Traces are thought to be *maximal*, which means that they are either infinite or describe a finished program execution. We write $o \in T$ to express that T executes operation o.

Input of a program is realized by one input list per security level. The input is predefined at the program start, but is *not* part of the initial memory of the program. It has to be explicitly made accessible by *import statements*, which import an input list by writing its content into the memory. This simulates the opening of an input stream and prevents that high data is in the system before it is explicitly needed. If an operation reads input, it removes the first element of the list, hence the input list is in its *use*-set and in its *def*-set.

5.4.2. Dynamic Program Dependences and Trace-Slices

In order to model information flow in traces, we employ dynamic versions of data and control dependence. These dependences are sufficient, because in a trace procedure calls and interleaving are inlined.

Dynamic control dependence

Xin and Zhang [157] introduced a context-sensitive definition of *dynamic control dependence*, which suits best for our purpose. It is based on *branching points*, operations with at least two possible direct successors in the control flow graph. Intuitively, an operation o is dynamically control dependent on the branching point of the innermost enclosing conditional, or, if o is a procedure entry, on the operation that invoked that procedure. We have to extend their definition to programs with threads.

Xin and Zhang define dynamic control dependence via *regions*, which in turn are based on postdominance (cf. sect. 2.1.2). For every possible branch, a branching point b *directs* one region. A region of one branch of b contains all operations that are executed in this branch between b and its *immediate dynamic postdominator* d and belong to the same thread as d. The immediate dynamic postdominator d of b is the first operation executed after b, whose statement $stmt(d)$ postdominates $stmt(b)$. Xin and Zhang have shown that the regions of one thread in an execution are either disjoint or nested, which allows to define dynamic control dependence via regions.

Definition 5.4 (Dynamic control dependence (from [157], extended to threads))**.** *Let T be a trace of a program p. An operation $o \in T$ is dynamically control dependent on operation $b \in T$, written $b \overset{dcd}{\dashrightarrow} o$, iff*

- *o is a thread entry and b is the corresponding fork operation, or*
- *o is a procedure entry and b is the operation that invoked that procedure, or*
- *b is the director of the innermost enclosing region of o.*

Xin and Zhang have also shown that every operation in a trace is dynamically control dependent on exactly one other operation, except for operation *start*, which is independent. Hence, dynamic control dependence can be used to uniquely identify operations among different traces:

Definition 5.5 (Uniqueness of operations)**.** *Let o_i and o_j be two operations that are executed in different traces. Then $o_i = o_j$ iff $stmt(o_i) = stmt(o_j)$ and*

1. *$o_i = o_j = start$, or*

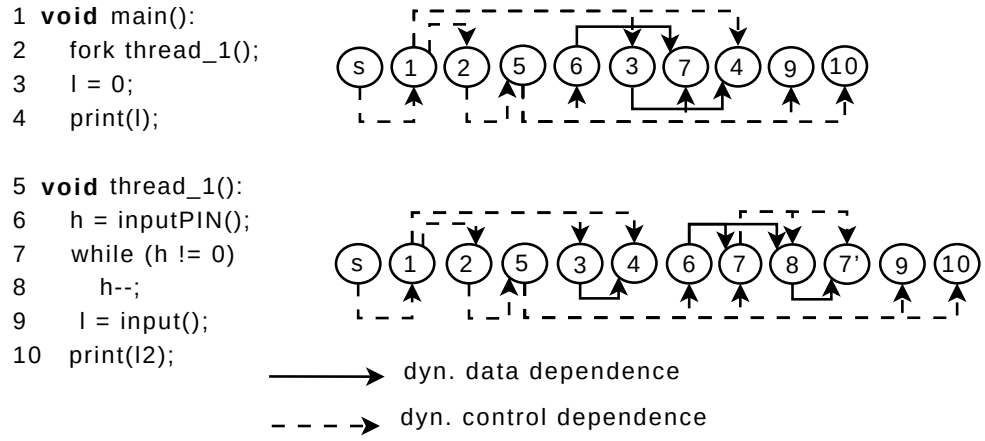


Figure 5.2.: A program and two possible traces. The first trace results from input (inputPIN() = 0, input() = 0), the second from (inputPIN() = 1, input() = 0).

2. *$b_i = b_j$, where b_i is the operation on which o_i is dynamically control dependent and b_j is the operation on which o_j is dynamically control dependent.*

This recursive definition terminates, because the trace-prefixes ending at o_i and o_j are finite.

The uniqueness of operations makes dynamic control dependence a static property. Every operation which is executed in at least one trace is dynamically control dependent on exactly one other operation, the only exception being operation *start*.

An important concept for us is the transitive closure of dynamic control dependences of an operation o, written $DCD(o)$. $DCD(o)$ is a list $\langle b_1, \ldots, b_n \rangle$ of operations, ordered by dynamic control dependence, such that $b_1 = start$, $b_n = o$ and each b_{i+1} is dynamically control dependent on b_i. $DCD(o)$ is a static property, which according to definition 5.5 characterizes operation o.

Dynamic data dependence

Definition 5.6 (Dynamic data dependence)**.** *Let T be a trace of a program p. An operation o is dynamically data dependent on operation a in T, written $a \overset{v}{\dashrightarrow} o$, iff there exists a variable $v \in use(o) \cap def(a)$, o executes after a in T and there is no operation o' with $v \in def(o')$ executing between a and o in T.*

Figure 5.2 shows two possible traces of an example program.

5.4.3. Trace-Slices

The operations and dependences in a trace form a sort of dependence graph, which can be sliced with respect to an operation o.

Definition 5.7 (Trace-slice)**.** *Let T be a trace of a program p. The trace-slice $S(o,T)$ of T for an operation $o \in Ops(p)$ is an acyclic graph consisting of all operations and dependences in T which form a path to o. In case $o \notin T$, $S(o,T)$ is empty.*

The *data slice* for o restricts the trace slice to paths consisting only of dynamic data dependences.

Definition 5.8 (Data slice)**.** *The data slice $D(o,T)$ of a trace T for an operation o is an acyclic graph consisting of all operations and dynamic data dependences in T which form a path to o. In case $o \notin T$, $D(o,T)$ is empty.*

The *potential influence* of o consists of all operations that are able to influence o in any possible execution.

Definition 5.9 (Potential influence)**.** *Let Θ be the set of all possible traces of a program p. The potential influence of an operation o, $Pot(o)$, is the set closed under the following rules:*

$$o \in Pot(o),$$

$$\frac{T \in \Theta : q \overset{v}{\dashrightarrow} q' \in T \vee q \overset{dcd}{\dashrightarrow} q' \in T \qquad q' \in Pot(o)}{q \in Pot(o)}$$

Trace-slices and data slices allow us to reason about information flow in single traces and to show whether single traces are free of information leaks. The potential influence allows us to reason about information flow in all possible traces and to show whether all possible traces are free of information leaks.

5.4.4. Dynamic Control Dependence Determines Execution Orders

Dynamic control dependence determines the relative execution order of operations that cannot happen in parallel. Xin and Zhang [157] observed that a branching point b imposes a total execution order on all operations being dynamically control dependent on b and lying in the same branch of b. Based on that observation, it can be shown that two operations of the same thread have a definite relative execution order:

Observation 5.1. *Let q and r be two different operations of the same thread, and let T and U be two traces which both execute q and r. Further, let T execute q before r. Then U also executes q before r.*

Proof. Let branching point b be the last operation in the common prefix *pre* of $DCD(q)$ and $DCD(r)$ (*pre* contains at least *start*). Let b_q be the direct successor of b in $DCD(q)$, and let b_r be the direct successor of b in $DCD(r)$. These are the operations where $DCD(q)$ and $DCD(r)$

start to diverge. Note that it is possible that $b_q = q$ or $b_r = r$. We present the proof for the case $b_q \neq q$ and $b_r \neq r$. The proofs for the other cases can be easily derived from it.

Since T executes q before r, it executes b_q before b_r. This is so because the regions directed by b_q and b_r have to be disjoint (otherwise, they would be nested and then one of both operations would be part of prefix *pre*). According to Xin and Zhang's observation, U also executes b_q before b_r, because both are dynamically control dependent on b and lie in the same branch of b. Since the regions directed by b_q and b_r are disjoint, U executes the operations in b_q's region before the operations in b_r's region. Thus, it executes q before r. □

We extend that observation to operations which cannot happen in parallel:

Observation 5.2. *Let q and r be two different operations which cannot happen in parallel, and let T and U be two traces which both execute q and r. Further, let T execute q before r. Then U also executes q before r.*

Proof. There are two cases: If q and r belong to the same thread, the claim follows from observation 5.1. Otherwise, our MHP information guarantees that either q executes before r's thread is forked, or that r executes after q's thread has been joined. Both cases impose a definite execution order between q and r, thus U executes q before r. □

The execution of a branching point b in a trace triggers the execution of all operations in the chosen branch which are dynamically control dependent on b. The only way to prevent their execution is to delay them infinitely.

Definition 5.10 (Infinite delay of operations)**.** *Let T be a trace of a program and let T execute b. Let o be an operation dynamically control dependent on b that belongs to the branch b chooses to execute. If $o \notin T$, then T infinitely delays o.*

The motivation behind that definition is to express that trace T is bound to execute o, unless it is hindered by infinitely repeating a loop or recursive cycle. The following observation follows directly from definition 5.10:

Observation 5.3. *Let T and U be two traces of a program, and let T execute operation o. Let b be the operation on which o is dynamically control dependent. If U executes b and b chooses the same branch as in T, then U executes o or infinitely delays o.*

5.4.5. Low-Observable Behavior and Low-Equivalence of Traces

The following attacker model is the foundation of our security property.

Valid information flow is specified by a security lattice $\mathfrak{L}$. Operations can be classified with a level $l \in \mathfrak{L}$, which means that their effects are visible to an attacker with a level l or higher. To this end, we adopt the classification mechanism of Hammer et al. [58]: The user annotates

statements with a *source* or a *sink level* (called *provided* and *required* levels in [58]). A source level x means that the annotated statement provides data of level x to the program, e.g. by reading input from an input stream. A sink level x expresses that the annotated statement exposes its data to recipients authorized with level x, e.g. by printing it to the screen. The classification of a program is mapped to the operations being instances of annotated statements. In the remainder, we often use the terms 'source' and 'sink' as abbreviations for statements and operations annotated with a source or sink level.

The attacker in our model has the following capabilities: He knows the program's source code and is authorized with a certain security level $l \in \mathfrak{L}$, which permits him to observe the execution of operations classified as a source or sink of level $x \in low$, where low is the set containing l and all levels smaller than l. Such an operation is called a *low-observable operation*. A configuration in a trace associated with a low-observable operation is called a *low-observable event*. The following function $event_{low}$ defines which properties of a low-observable event, $(\overline{m}, o, m)$, are observable to the attacker, where λ is the empty word:

$$event_{low}((\overline{m}, o, m)) = \begin{cases} (\overline{m}\,|_{use(o)}, o) & o \text{ is a source of level } x \in low \\ (m\,|_{def(o)}, o) & o \text{ is a sink of level } x \in low \\ \lambda & \text{otherwise.} \end{cases}$$

In effect, the attacker sees which low-observable operation is responsible for the event, since he knows the source code, and he sees the values of the involved variables. If the event involves a source o, the attacker sees o and the values of the variables used by o in memory $\overline{m}$. If the event involves a sink o, the attacker sees o and the values of the variables defined by o in memory m. Since the attacker is also able to see in which order a trace executes low-observable events, the *low-observable behavior* of a trace is a sequence of low-observable events:

Definition 5.11 (Low-observable behavior)**.** *Let $T = (\overline{m_0}, o_0, m_0) \cdots (\overline{m_i}, o_i, m_i) \cdots$ be a trace. The low-observable behavior of T is a list of low-observable events, which results from the application of function $event_{low}$ to every configuration in T:*

$$obs_{low}(T) = event_{low}((\overline{m_0}, o_0, m_0)) \cdots event_{low}((\overline{m_i}, o_i, m_i)) \cdots .$$

The attacker is also aware of the probability with which an input causes a certain low-observable behavior, which allows him to exploit probabilistic channels. Besides timing channels and physical and resource channels we also exclude termination channels from our attacker model, because their treatment would impose severe restrictions on the programs.

```
1 void main():
2   x = inputPIN();
3   while (x > 0)
4     print("x");
5     x--;
6   while (true)
7     skip;
```

```
1 void main():
2   x = inputPIN();
3   while (x != 0)
4     x--;
5   print(1);
```

```
1 void main():
2   x = inputPIN();
3   while (x == 0)
4     skip;
5   print("x");
6   while (x == 1)
7     skip;
8   print("x");
9   ...
10  while (x == 42)
11    skip;
12  print("x");
13  ...
```

Figure 5.3.: Three tough nuts for termination-insensitive definitions of low-equivalent traces. The program on the left must be rejected because it gradually leaks the PIN, the one in the mid could be accepted because its leak is a termination channel. The program on the right exploits termination channels to leak the input PIN.

Low-equivalent traces

Developing a termination-insensitive definition of low-equivalent traces turns out to be a major sticking point. Trivially, two events (m,o) and (n,q) are equal if $m = n$ and $o = q$ holds, and two low-observable behaviors are equal if they consist of the same number k of events and for every $0 \leq i \leq k$ the according events are equal. A manifest way to define low-equivalent traces is to require equal low-observable behaviors, which permits one trace to terminate and the other to not terminate. But this requirement effectively forbids low-observable events behind loops whose guards contain high data, which we believe to be too restrictive in practice. Consider the program in the middle of Fig. 5.3, whose input in line 2 is high data and whose `print`-statement is low-observable. If a run of the program does not terminate, the `print`-statement is delayed infinitely, which leads to the conclusion that the input was < 0. We want to permit this and similar programs, because these leaks are a sort of termination channel.

Several existing definitions [146, 160] that aim to permit these termination channels declare traces low-equivalent if their low-observable behavior is equal up to the length of the shorter sequence of low-observable events. But as pointed out by Huisman et al. [64], this may lead to unintended information leaks. Consider the program on the left side of Fig. 5.3, whose traces always diverge, and assume that the input PIN is high data and that the executions of statement 4 comprise its low-observable behavior. The program exposes the input PIN by printing an equal number of x's to the screen. If low-equivalence of traces is confined to the length of the shorter sequence of low-observable events, this behavior is perfectly legal, because all traces with low-equivalent inputs are equal up to the length of the shorter sequence.

In order to solve that problem, we suggest the following approach: If we have two finite traces, then they are low-equivalent if their low-observable behaviors are equal. If both traces are infinite, then the low-observable behaviors must be equal up to the length of the shorter sequence, *and* the low-observable events missing in the other trace must be missing due to infinite delay. This additional constraint makes sure that the missing events leak information only via termination channels. Similarly, if one of both traces is finite and the other is infinite, then the finite trace must have *at least as much* low-observable events as the infinite one, the low-observable behaviors must be equal up to the length of the shorter sequence and the low-observable events missing in the infinite trace must be missing due to infinite delay. Or more formally:

Definition 5.12 (Low-equivalence of traces, $\sim_{low}$)**.** *Let p be a program and let T and U be two traces of p. Let $obs_{low}(T) = (m_0, o_0)\cdots$ and $obs_{low}(U) = (n_0, q_0)\cdots$ be their low-observable behaviors. Let k_T be the number of events in $obs_{low}(T)$ and k_U be the number of events in $obs_{low}(U)$. T and U are low-equivalent, written $T \sim_{low} U$, if one of the following cases holds:*

1. *T and U are finite, $k_T = k_U$, and $\forall 0 \leq i \leq k_T : m_i = n_i \wedge o_i = q_i$.*
2. *T is finite and U is infinite, and*
   - *$k_T \geq k_U$,*
   - *$\forall 0 \leq i \leq k_U : m_i = n_i \wedge o_i = q_i$, and*
   - *$\forall k_U < j \leq k_T$, U infinitely delays an operation $b \in DCD(o_j)$.*
3. *T is infinite and U is finite, and*
   - *$k_U \geq k_T$,*
   - *$\forall 0 \leq i \leq k_T : m_i = n_i \wedge o_i = q_i$, and*
   - *$\forall k_T < j \leq k_U$, T infinitely delays an operation $b \in DCD(o_j)$.*
4. *T and U are infinite, and*
   - *if $k_T = k_U$, then $\forall 0 \leq i \leq k : m_i = n_i \wedge o_i = q_i$.*
   - *if $k_T > k_U$, then $\forall 0 \leq i \leq k_U : m_i = n_i \wedge o_i = q_i$, and $\forall k_U < j \leq k_T$, U infinitely delays an operation $b \in DCD(o_j)$.*
   - *if $k_T < k_U$, then $\forall 0 \leq i \leq k_T : m_i = n_i \wedge o_i = q_i$, and $\forall k_T < j \leq k_U$, T infinitely delays an operation $b \in DCD(o_j)$.*

Note that $\sim_{low}$ is reflexive and symmetric, but not transitive.

Thanks to our additional constraints, we are able to reject the program on the left side of Fig. 5.3, because the missing low-observable events are not missing due to infinite delay. The

program in the mid, however, is accepted. Our definition of low-equivalent traces exploits that we have a program analysis, slicing, at hand, which enables us to analyze whether low-observable events are missing due to infinite delay. We compare our definition with several others in section 5.7.3.

It should be mentioned that definition 5.12 leaves room for pathological programs like that on the right side of Fig. 5.3. It encodes the possible positive values of the high input into a sequence of x's before it enters a nonterminating loop, but all of its traces are considered low-equivalent. This is a problem experienced by many existing termination-insensitive approaches to IFC; for example, Hammer et al.'s IFC technique [58] and the Jif security-type system [103] accept this or similar programs. To date, we are not aware of a satisfactorily solution to this problem. Some ideas are discussed in section 5.8.

Low-Security Observational Determinism

It remains to insert our definition of low-equivalent traces into our chosen security property.

Security Property 1 (Low-security observational determinism)**.** *Program p is low-security observational deterministic if the following holds for every pair* (t,u) *of low-equivalent inputs: Let* $\mathfrak{T}$ *and* $\mathfrak{U}$ *be the sets of possible traces resulting from t and u.* $\forall T,U \in \mathfrak{T} \cup \mathfrak{U} : T \sim_{low} U$.

We proceed by investigating how this security property can be enforced via slicing.

5.5. A Slicing-Based Security Constraint for Low-Security Observational Determinism

In order to detect probabilistic channels in traces, we have to map data and order conflicts to operations. To this end, we define that two operations may happen in parallel if their statements may happen in parallel, which is a sound, albeit imprecise, approximation.

Definition 5.13 (Data and order conflicts)**.** *Let a and b be two operations that may happen in parallel.*

- *There is a data conflict from a to b, written* $a \overset{dconf}{\rightsquigarrow} b$*, iff a defines a variable v that is used or defined by b.*
- *There is an order conflict between a and b, written* $a \overset{oconf}{\leftrightsquigarrow} b$*, iff both operations are low-observable.*

Note that conflicts are ignored by trace-slices, data slices and potential influences. We say that an operation o is *potentially influenced by a data conflict* if there exists an operation $b \in Pot(o)$ and an operation $a \in Ops(p)$ such that $a \overset{dconf}{\rightsquigarrow} b$ holds. We make and prove the following claim:

Security Constraint 1 (LSOD)**.** *A program p is low-security observational deterministic if*

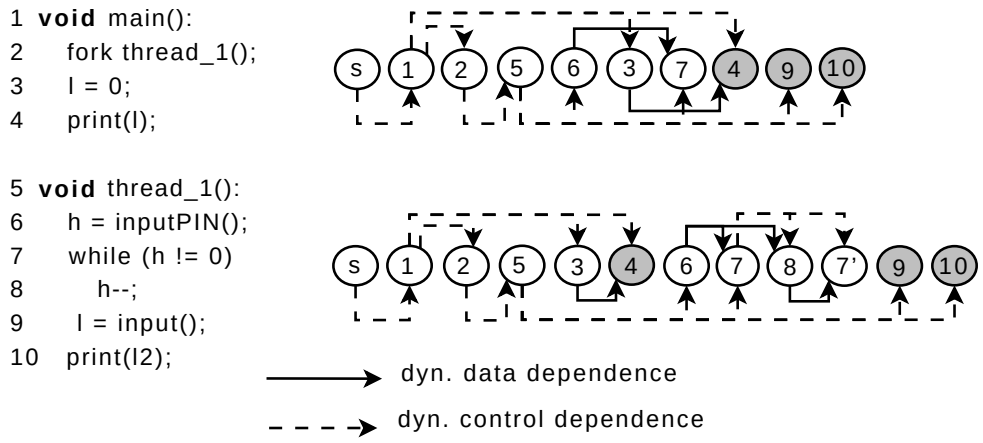


Figure 5.4.: A program and two possible traces. The first trace results from input (inputPIN() = 0, input() = 0), the second from (inputPIN() = 1, input() = 0). The shaded nodes represent the low-observable behavior.

1. *no low-observable operation o is potentially influenced by an operation reading or importing high input,*
2. *no low-observable operation o is potentially influenced by a data conflict, and*
3. *there is no order conflict between any two low-observable operations.*

The first rule ensures that the implicit and explicit flow to o does not transfer high data. The second rule ensures that high data cannot influence the data flowing to o via interleaving. The third rule ensures that high data cannot influence the execution order of low-observable operations via interleaving.

5.5.1. Security Constraint LSOD Enforces Low-Security Observational Determinism

The proof of our claim is based on the following observation: Let T and U be two traces of program p with the same input. If the trace-slices of T and U for an operation o are equal, then o uses the same values and writes the same results to the same variables in both traces. This observation can be specialized with respect to low-equivalent inputs: If T and U have low-equivalent inputs, the trace-slices for o are equal and no operation in these trace-slices reads or imports high input, then o uses the same values and writes the same results to the same variables in T and U. It follows that T and U are low-equivalent if this holds for every low-observable operation in T and U and if the traces execute the low-observable operation in the same relative order. A program is low-security observational deterministic if this holds for all traces with low-equivalent inputs.

Consider Fig. 5.4 as an example. Assume that the operations 2, 6 and 7 are low-observable. The two inputs and the two traces are low-equivalent with respect to this classification. The

traces execute the low-observable operations in the same order, the trace-slices of these operations are free of operations reading or importing high input and each of these operations has the same trace-slices in both traces. Unfortunately, this does not hold for all possible traces resulting from these inputs. In the trace on the right side, operations 2 and 6 could be swapped, the resulting trace is not low-equivalent to those shown in the Figure.

We start by proving that if an operation o has equal data slices in T and U and no operation in the data slices reads or imports high input, then o uses the same values in both traces and computes the same results. We introduce the following abbreviations: Let $(\overline{m}, o, m)$ be a configuration in a trace T. $\overline{T_o} = \overline{m}|_{use(o)}$ denotes the part of memory $\overline{m}$ that contains the variables used by o, and $T_o = m|_{def(o)}$ denotes the part of memory m that contains the variables defined by o.

Lemma 5.1. *Let T and U be two traces of a program p with low-equivalent inputs. Let $o \in Ops(p)$ be an operation. If $D(o,T) = D(o,U)$ and no operation in these data slices reads or imports high input, then $\overline{T_o} = \overline{U_o}$ and $T_o = U_o$.*

Proof. If $D(o,T)$ and $D(o,U)$ are empty, the lemma trivially holds – T and U do not execute o. Otherwise, we exploit that $D(o,T)$ is an acyclic directed graph in which all paths are finite. We show that the lemma in fact holds for every operation q in $D(o,T)$, by an induction over the length of the longest path in the data slice that ends at q. The induction terminates because the number of operations and the length of the longest path in $D(o,T)$ is finite.

The base case handles all operations in $D(o,T)$ without incoming dynamic data dependences. Let q be such an operation. Trivially, $\overline{T_q} = \overline{U_q}$ because q does not use any variables. Hence, q writes constant values to the variables in $def(q)$ or imports low input. It follows $T_q = U_q$.

In the induction step we have an operation q which is dynamically data dependent on a set Q of operations and whose longest incoming path of dynamic data dependences has length $n+1$. We know for each $q' \in Q$ that $D(q',T) = D(q',U)$, $\overline{T_{q'}} = \overline{U_{q'}}$ and $T_{q'} = U_{\prime q}$, as its longest incoming path of dynamic data dependences has length $\leq n$ and thus q' was already covered by the induction.

- $\overline{T_q} = \overline{U_q}$
  Let $q' \overset{v}{\dashrightarrow} q$ be a dynamic data dependence due to a variable $v \in use(q)$. We know that $T_{q'}(v) = U_{q'}(v)$. The dynamic data dependence signals that there are no redefinitions of v executed between q' and q in T and U. It follows that $\overline{T_q}(v) = T_{q'}(v) = \overline{U_q}(v)$.

- $T_q = U_q$
  Operation q uses the same data for its computations in T and U and possibly imports low input. Thus, it writes the same values to the variables in $def(q)$.

□

Since a data slice $D(o,T)$ is a subgraph of the trace-slice $S(o,T)$, lemma 5.1 also holds in case the trace-slices $S(o,T)$ and $S(o,U)$ are equal.

Corollary 5.1. *Let T,U be two traces of a program p with low-equivalent inputs. Let $o \in Ops(p)$ be an operation. If $S(o,T) = S(o,U)$ and no operation in these trace-slices reads or imports high input, then $\overline{T_o} = \overline{U_o}$ and $T_o = U_o$.*

If we have two traces with low-equivalent inputs, all low-observable operations satisfy the conditions of corollary 5.1 and the traces execute these operations in the same relative order, then the traces are low-equivalent.

Corollary 5.2. *Let p be a program. Let T and U be two finite traces of p with low-equivalent inputs. T and U are low-equivalent if for every low-observable operation o, $S(o,T) = S(o,U)$ holds and no operation in the trace-slices reads or imports high input and T and U execute the same low-observable operations in the same relative order.*

Proof. We have to show $obs_{low}(T) = obs_{low}(U)$. The condition of the corollary ensures that T and U execute the same low-observable operations in the same relative order.

Let o be the i-th low-observable operation. If o is classified as a source, the corresponding events are $(\overline{m}|_{use(o)}, o)$ in T and $(\overline{n}|_{use(o)}, o)$ in U. If o is classified as a sink, the corresponding events are $(m|_{def(o)}, o)$ in T and $(n|_{def(o)}, o)$ in U. According to corollary 5.1, $\overline{m}|_{use(o)} = \overline{n}|_{use(o)}$ and $m|_{def(o)} = n|_{def(o)}$ hold, therefore the events are equal. Since this holds for every low-observable event in T and U, $obs_{low}(T) = obs_{low}(U)$ follows. □

The following two corollaries cover the cases where one or both of the two traces are infinite.

Corollary 5.3. *Let p be a program. Let T and U be two infinite traces of p with low-equivalent inputs such that $obs_{low}(T)$ is of equal length or longer than $obs_{low}(U)$ (switch the names if necessary). T and U are low-equivalent if*

- *they execute the shared low-observable operations in the same relative order,*
- *for every low-observable operation $o \in U$ $S(o,T) = S(o,U)$ holds and no operation in the trace-slices reads or imports high input*
- *and for every low-observable operation $o \in T$ and $o \notin U$ U infinitely delays an operation $b \in DCD(o)$.*

Proof. Similar to the proof of corollary 5.2, we can use corollary 5.1 to show that T and U have the same low-observable behavior up to the last low-observable event in U. Thus, T and U satisfy case 2 in definition 5.12. □

Corollary 5.4. *Let p be a program. Let T and U be two traces of p with low-equivalent inputs, such that T is finite and U is infinite (switch the names if necessary). T and U are low-equivalent if*

- *$obs_{low}(T)$ is of equal length or longer than $obs_{low}(U)$,*
- *T and U execute the shared low-observable operations in the same relative order,*
- *for every low-observable operation $o \in U$ $S(o,T) = S(o,U)$ holds and no operation in the trace-slices reads or imports high input*
- *and for every low-observable operation $o \in T$ and $o \notin U$ U infinitely delays an operation $b \in DCD(o)$.*

Proof. Similar to the proof of corollary 5.2, we can use corollary 5.1 to show that T and U have the same low-observable behavior up to the last low-observable event in U. Thus, T and U satisfy case 4 in definition 5.12. □

If all pairs of traces with low-equivalent inputs satisfy one of these three corollaries, the program is low-security observational deterministic.

Corollary 5.5. *A program p is low-security observational deterministic if for every pair (t,u) of low-equivalent inputs the following condition holds: Let $\mathfrak{T}$ and $\mathfrak{U}$ be the sets of possible traces resulting from t and u. Then every pair of traces $T,U \in \mathfrak{T} \cup \mathfrak{U}$ is low-equivalent according to corollary 5.2, 5.3 or 5.4.*

Proof. This follows directly from definition 1. □

It remains to show that a program is low-security observational deterministic if it satisfies our security constraint LSOD. The following lemma shows that an operation o has a unique trace-slice if it is not potentially influenced by a data conflict.

Lemma 5.2. *Let p be a program and o be an operation of p. If o is not potentially influenced by a data conflict, then $S(o,T) = S(o,U)$ holds for all traces T and U which execute o.*

Proof. The proof is done by an induction over the operations in T. The induction shows that $S(q,T) = S(q,U)$ holds for each $q \in T$ which potentially influences o.

Base case: The induction starts at *start*, which potentially influences o (via a chain of dynamic control dependences). Operation *start* is executed by both T and U, and both trace-slices for *start* contain only *start*.

Induction step: We take the next operation q executed by T. If q does not potentially influence o, we have nothing to show. Otherwise, we know for all operations r on which q depends in T that $S(r,T) = S(r,U)$ because they were already covered by our induction ($r \in Pot(o)$). Note that this implicitly shows that r is executed in U.

First, we show that all dependences of the form $r \overset{v}{\dashrightarrow} q$ or $r \overset{dcd}{\dashrightarrow} q$ in $S(q,T)$ are also in $S(q,U)$. This is clear for $r \overset{dcd}{\dashrightarrow} q \in S(q,T)$ because q is dynamically control dependent on exactly one operation. A dependence $r \overset{v}{\dashrightarrow} q$ in $S(q,T)$ is in $S(q,U)$ if q executes after r in U and there exists no operation c that redefines v and executes between r and q in U.

1. q executes after r in U:
   Since q and r must not form a data conflict, they cannot happen in parallel. According to corollary 5.1, q executes after r in U.

2. There exists no c with $v \in def(c)$ which executes between r and q in U:
   Assume there exist redefinitions of v between r and q in U, and let c be the last of them before q executes. It follows that c is potentially influencing o because q is dynamically data dependent on c in U. There are two cases: Either c, r and q cannot happen in parallel to each other or at least two of them may happen in parallel.

   a) In the latter case, they form at least one data conflict which potentially influences o. This contradicts the assumption of the lemma.

   b) Otherwise, corollary 5.1 requires T to also execute c between r and q. But this means that the dependence $r \overset{v}{\dashrightarrow} q$ does not exist in T because c redefines v, and that is a contradiction.

Thus, every dependence $r \overset{v}{\dashrightarrow} q$ and $r \overset{dcd}{\dashrightarrow} q$ in $S(q,T)$ is also in $S(q,U)$. It remains to show that q does not depend on additional operations in U.

- There exists no additional dynamic control dependence $r' \overset{dcd}{\dashrightarrow} q$ in U because q is dynamically control dependent on exactly one operation.

- There exists no dynamic data dependence $c \overset{v}{\dashrightarrow} q$ in U which does not exist in T:
  Assume that q is dynamically data dependent on an operation c in U due to a variable v but not in T. This means that q is dynamically data dependent on a different operation r in T due to v. Hence, c executes between r and q in U but not in T. In can be shown in analogy to point 2 that such an operation c cannot exist.

It follows that q has the same incoming dependences in T and in U. According to the induction, $S(r,T) = S(r,U)$ holds for every r on which q is dependent in T and U. Therefore, $S(q,T) = S(q,U)$ holds. □

Next, we show that if a trace T executes an operation o that is not potentially influenced by a data conflict or an operation reading or importing high input, then all traces U whose inputs are low-equivalent to the one of T either execute o or delay it infinitely.

Corollary 5.6. *Let p be a program and let o be an operation of p that is not potentially influenced by a data conflict or an operation reading or importing high input. Let T be a trace of p and Θ be the set of possible traces whose inputs are low-equivalent to the one of T. If $o \in T$, then every $U \in \Theta$ either executes o or infinitely delays an operation in $DCD(o)$.*

Proof. We assume that $U \in \Theta$ does not execute o and show that in this case U infinitely delays an operation in $DCD(o)$. There exist two adjacent operations in $DCD(o)$, b and b', such that $b \in U$ and $b' \notin U$. This holds because the first element in $DCD(o)$ is *start*, which is always executed, and the last element in $DCD(o)$ is o, which is not executed. We apply lemma 5.2 to show $S(b,T) = S(b,U)$ and then lemma 5.1 to show that b evaluates in U with the same results as in T and thus chooses the same branch in T and U. According to observation 5.3, it follows that U infinitely delays b'. □

If there is no order conflict between the low-observable operations of program p, then all traces with low-equivalent inputs have the same relative execution order of low-observable operations up to infinite delay:

Corollary 5.7. *Let p be a program and let T and U be two traces with low-equivalent inputs. If there are no order conflicts between any two low-observable operations, then all low-observable operations executed by both T and U are executed in the same relative order.*

Proof. Let o and o' be two low-observable operations. We assume that T and U execute them, but not in the same relative order, which means that their execution order is switched. According to corollary 5.1, this can only be the case if o and o' may happen in parallel. But then they form an order conflict, which contradicts the assumption. □

Now we have all necessary pieces together:

Theorem 5.1 (Security constraint LSOD enforces low-security observational determinism)**.** *A program p is low-security observational deterministic if it satisfies security constraint 1.*

Proof. Let t and u be two low-equivalent inputs, and let $\mathfrak{T}$ and $\mathfrak{U}$ be the resulting sets of possible traces. We have to show that each pair of traces T, U in $\mathfrak{T} \cup \mathfrak{U}$ is low-equivalent.

Corollary 5.7 guarantees that T and U execute the shared low-observable operations in the same relative order. We can apply lemma 5.2 to all low-observable operations o executed by both T and U, so we also have $S(o,T) = S(o,U)$. Since the potential influence of a low-observable operation o does not contain operations reading or importing high input, this also holds for the operations in $S(o,T)$ and $S(o,U)$. It remains to distinguish finite and infinite traces:

1. T and U are finite
   Corollary 5.6 guarantees that T and U execute the same low-observable operations because infinite delay would require infinite traces. The conditions of corollary 5.2 are satisfied and T and U are low-equivalent.

2. T and U are infinite
   Let $obs_{low}(T)$ be of equal length or longer than $obs_{low}(U)$ (switch the names if necessary). Corollary 5.6 guarantees that T executes the same low-observable operations as U and that U infinitely delays an operation $b \in DCD(o)$ of all low-observable operations o executed in T but not in U. Thus, the conditions of corollary 5.3 are satisfied and T and U are low-equivalent.

3. T is finite and U is infinite
   Corollary 5.6 guarantees that T executes the same low-observable operations as U does and that U infinitely delays an operation $b \in DCD(o)$ of all low-observable operations o executed in T but not in U. Note that this means that $obs_{low}(T)$ has at least as many events as $obs_{low}(U)$. Thus, the conditions of corollary 5.4 are satisfied and T and U are low-equivalent.

4. T is infinite and U is finite
   Switch the names and apply the previous case.

□

5.5.2. A Sound Approximation of LSOD via Slicing of CSDGs

Conveniently, the potential influence of an operation o can be conservatively approximated by the slice for $stmt(o)$ of the CSDG. We are able to show this claim on the basis of one assumption. An operation q has a fixed calling context, which is encoded in $DCD(q)$, because procedure calls are branching points. Our assumption is that if there exists a dynamic dependence $q \dashrightarrow q'$ in a trace T, then there exists a context path from the context of q to the context of q' in the CSDG. This assumption is eligible, since it is the purpose of a CSDG to account for every possible dependence which may appear in a program run. However, proving that property is beyond the scope of this work, because it would require to prove a whole CSDG generator correct.

Theorem 5.2. *Let p be a program and G its CSDG. Let o be an operation in $Ops(p)$. The following holds for every operation $q \in Pot(o)$: There is a context-sensitive path from $stmt(q)$ to $stmt(o)$ in G.*

Proof. Since the case $q = o$ is trivial, we assume $q \neq o$. Let $c(r)$ denote the context of an operation r. There exists a sequence of dynamic dependences $q_1 \dashrightarrow \cdots \dashrightarrow q_k$ such that $q =$

q_1 and $q_k = o$. Our assumption states that for each dynamic dependence $q_i \dashrightarrow q_{i+1}$ in that sequence there exist a context path from $c(q_i)$ to $c(q_{i+1})$. According to definition 3.15, these paths can be successively concatenated to a a context path from $c(q)$ to $c(o)$. Thus, there exists a context-sensitive path from $stmt(q)$ to $stmt(o)$ in G. □

Since the context-sensitive slice of G for $stmt(o)$ contains all nodes on context-sensitive paths to $stmt(o)$, it covers the potential influence of operation o. This means that our security constraint LSOD can be verified by slicing CSDGs. A suitable algorithm is presented in section 5.6.

5.5.3. Limitations

We made the following limitations in order to keep this part manageable:

Pointers In order to avoid modeling pointer indirection, our memories do not include pointers. For pointer indirection an intermediate layer of locations has to be added, so that a variable points to a location and the location to a value. Dynamic data dependences have to account for changes of the location a variable points to. Lemma 5.1 can then be extended to show that each variable of operation o undergoes equivalent changes of locations in both traces.

Sequential consistency In order to model traces as sequences of configurations, we assumed that programs are *sequentially consistent*. Sequential consistency [82] is a relation between a program and its effects on the memory, defining which effects correspond to valid program executions. Sequential consistency requires that the effects of any program execution must be in accordance with a sequential execution of the operations in which the operations of each thread appear in the order specified by the program. This limits the validity of our proof to concurrent programs that behave sequentially consistent, for example correctly synchronized Java programs [48, §17.4.5].

5.6. Slicing-Based Verification of LSOD

In the following, we present an algorithm based on slicing of CSDGs that verifies whether a program satisfies our security constraint LSOD. In order to detect data and order conflicts, CSDGs are enriched with *conflict edges*.

Definition 5.14 (Data and order conflict edges)**.** *Let G be a CSDG and let m and n be two nodes in G that may happen in parallel. There is a data conflict edge $m \rightarrow_{dconf} n$ to G if m defines a variable v that is used or defined by n. There is an order conflict edge $m \leftrightarrow_{oconf} n$ to G if both nodes are classified as sources or sinks.*

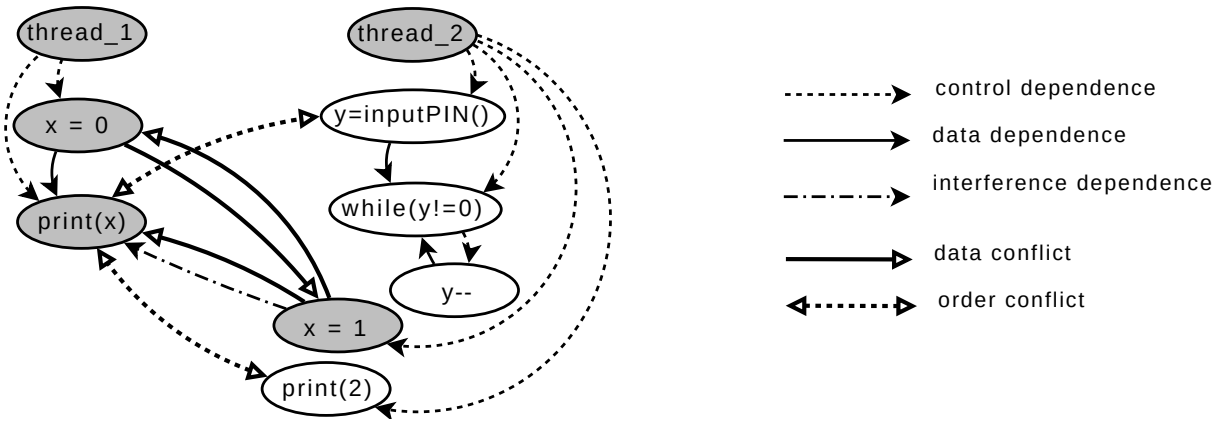


Figure 5.5.: The CSDG of the program on the right side of Fig. 5.1, enriched with conflict edges. The gray nodes denote the slice for node `print(x)`. Note that the slice ignores conflict edges.

Figure 5.5 shows the CSDG of the program on the right side of Fig. 5.1, enriched with conflict edges. The example assumes that the security lattice consists of two elements, $l \sqsubseteq h$, and that `y = inputPIN()` is classified as a source of h data and `print(x)` and `print(2)` are classified as sinks of l data. The CSDG contains two order conflict edges, one between `print(x)` and `print(2)` and one between `print(x)` and `y = inputPIN()`, and three data conflict edges, from `x = 0` to `x = 1`, from `x = 1` to `x = 0` and from `x = 1` to `print(x)`.

Note that data conflicts can be computed independently from the classification of sources and sinks because they only depend on conflicting variable accesses. Conveniently, interference dependences already represent the data conflicts between a read and a write. Joana's CSDG generator uses *interference-write dependences* to represent conflicts between two writes. These special dependences are ignored by the hitherto presented slicing and chopping algorithms.

Definition 5.15 (Interference-write dependence)**.** *A statement n is interference-write-dependent on statement m, abbreviated by $m \rightarrow_{iw} n$, if both statements may write to the same memory location and may happen in parallel up to synchronization.*

Order conflicts have to be determined after a user has classified the program and have to be adjusted whenever the classification changes. Since order conflict edges are inserted between every pair of sources or sinks that may happen in parallel, the IFC algorithm has to check later if a certain order conflict is actually visible to the attacker in question: If the attacker has a level l, then both nodes involved in the order conflict have to be classified with a level equal to or lower than l. Otherwise, the conflict can be ignored because the attacker cannot observe the execution of both nodes and thus not their relative ordering. We call an order conflict *low-observable* if it is visible to the attacker.

The following algorithm verifies whether a program p is LSOD:

1. Retrieve a security lattice $\mathfrak{L}$ and a classification of p (from a user).

2. Enrich the CSDG with order conflict edges.

3. Compute a slice for every source or sink s. Let $l \in \mathfrak{L}$ be the level of s.

   - If the traversal encounters a source of level $h \not\sqsubseteq l$, then the program may leak data of level h via explicit or implicit flow and is rejected.
   - If the traversal encounters an incoming data conflict edge, the program may contain a probabilistic data channel and is rejected.
   - If the traversal encounters an order conflict edge, check if the order conflict is low-observable. If so, the program may contain a probabilistic order channel and is rejected.

As an example, consider Fig. 5.5. The slice for `print(2)` encounters the order conflict edge between the nodes `print(2)` and `print(x)`, so the program may contain a probabilistic order channel. The slice for `print(x)`, highlighted gray in Fig. 5.5, encounters all data conflict edges, so the program may contain a probabilistic data channel as well, whereas its implicit and explicit flow is secure.

5.6.1. Adding a Declassification Mechanism

The presented algorithm detects probabilistic channels and malicious explicit and implicit flow, as desired. However, it is too restrictive in practice because it does not permit declassification. The principle of *non-occlusion* [129, 130] (cf. section 5.1.3) aids us in integrating existing declassification mechanisms into our algorithm. In order to satisfy non-occlusion, a declassification mechanism for implicit and explicit flow should not declassify probabilistic channels. This requirement allows us to divide the security check into two independent phases, one that inspects the implicit and explicit flow and one that scans the program for probabilistic channels. The inspection of implicit and explicit flow can be done with already existing techniques, for example, with the slicing-based IFC check of Hammer et al. [58] or even with a security-type system, as long as our classification system can be mapped over. That way, it is possible to reuse existing declassification mechanisms for implicit and explicit flow. The chosen technique has to be extended to account for explicit flow resulting from shared-memory access. For example, Hammer et al.'s IFC technique has to include concurrency edges in the predecessor relation $pred(n)$ used in the data flow equations (cf. sect. 5.1.4) and has to ignore conflict edges.

5.6.2. Optimizations

Low-security observational determinism has one hitch, programs can fail it even if they do not work with any high data at all. We therefore use the following optimization: A conflict is

considered harmless if there exists no source of high data that can execute before the nodes forming the conflict. Our algorithm is extended as follows:

1. Retrieve a security lattice $\mathfrak{L}$ and a classification of p (from a user).
2. Enrich the CSDG with order conflict edges.
3. Compute a slice for every source or sink s. Let $l \in \mathfrak{L}$ be the level of s.
   - If the traversal encounters a source of level $h \not\sqsubseteq l$, then the program may leak data of level h via explicit or implicit flow and is rejected.
   - If the traversal encounters an incoming data conflict edge, check in the TCFG if any node that can be executed before both conflicting nodes is a source of level $h \not\sqsubseteq l$. If so, the program may contain a probabilistic data channel and is rejected.
   - If the traversal encounters an order conflict edge, check if the order conflict is low-observable. If so, check in the TCFG if any node that can be executed before both conflicting nodes is a source of level $h \not\sqsubseteq l$. In that case the program may contain a probabilistic order channel and is rejected.

Note that this optimization is not covered by our proofs in section 5.5 and should be used with care.

5.6.3. Pseudocode

Algorithms 5.1, 5.3 and 5.2 present pseudocode for our IFC algorithm. It receives a CSDG in which sources and sinks are already classified and which already contains the order conflict edges, the corresponding TCFG and the security lattice in charge. It then runs an external check of the implicit and explicit flow. If the program passes that check, it is scanned for probabilistic channels. This is done by Alg. 5.3.

Algorithm 5.3 receives the CSDG, the TCFG, the security lattice and a source or sink s of a certain security level l. The algorithm first checks whether s is involved in a low-observable order conflict that can be preceded by a source of high data. This task is delegated to the auxiliary procedure `benign` in Alg. 5.2. After that, it executes an extended I2P slicer (cf. section 3.2) which additionally checks if s is potentially influenced by a data conflict whose nodes can be preceded by a source of high data. This check is again delegated to procedure `benign`.

Procedure `benign` first checks whether the given conflict is an order conflict and whether it is low-observable, which it is if both nodes involved in the conflict are visible to the attacker. After that, it checks for both order and data conflicts whether the involved nodes can be preceded by

Algorithm 5.1 Information flow control for concurrent programs.

Input: A classified CSDG $G = (N, E)$, its TCFG C, a security lattice $\mathfrak{L}$.
Output: 'true' if the program is LSOD (up to declassification and harmless conflicts), 'false' otherwise.

```
Let src(n) be the source level of node n (= ⊥ if n is not a source).
Let sink(n) be the the sink level of node n (= ⊤ if n is not a sink).

/* Inspect the implicit and explicit flow of the program. */
Let flow(G,C,𝔏) be a function that returns false if G contains illicit implicit or explicit flow.
if flow(G,C,𝔏) == false
   return false

/* Scan the program for probabilistic channels. */
for all n ∈ N : src(n) ≠ ⊥   // check the sources of information
   if prob(G,C,n,src(n),𝔏) == false
      return false
for all n ∈ N : sink(n) ≠ ⊤   // check the sinks of information
   if prob(G,C,n,sink(n),𝔏) == false
      return false

return true
```

Algorithm 5.2 Procedure `benign` identifies benign conflicts.

Input: A TCFG $C = (N, E)$, two conflicting nodes a and b, the kind e of the conflict, a security lattice $\mathfrak{L}$, a security level $l \in \mathfrak{L}$.
Output: 'true' if the conflict is harmless, 'false' otherwise.

```
Let reaches(m,n,C) return 'true' if there exists a realizable path from node m to node n in C.

/* Check visibility of order conflicts. */
if e == oconf
   x = (src(a) ≠ ⊥ ∧ src(a) ⊑ l) ∨ (sink(a) ≠ ⊤ ∧ sink(a) ⊑ l)  // is 'a' visible?
   y = (src(b) ≠ ⊥ ∧ src(b) ⊑ l) ∨ (sink(b) ≠ ⊤ ∧ sink(b) ⊑ l)  // is 'b' visible?
   if ¬x ∨ ¬y   // one of them is not visible
      return true // the order conflict is not visible

/* Check if a source of high data may execute before the conflicting nodes. */
for all n ∈ N
   if src(n) ⋢ l
      if (reaches(n,a) ∨ n ∥ a) ∧ (reaches(n,b) ∨ n ∥ b)
         return false // the conflict is harmful

return true // the outcome of the conflict cannot be influenced by high data
```

a source n of high data. This is the case if n reaches them on realizable paths in the TCFG or if it may happen in parallel to them.

The depicted pseudocode forgoes efficiency for clarity and therefore has a comparatively high asymptotic running time: If we exclude the check of the implicit and explicit flow done by the external procedure `flow`, the runtime complexity is dominated by the reachability check in in

Algorithm 5.3 Procedure `prob` detects probabilistic channels.

Input: A CSDG $G = (V, E)$, its TCFG C, a node s, its security level l, the security lattice $\mathfrak{L}$.
Output: 'false' if s leaks information through a probabilistic channel, 'true' otherwise.

```
/* Check G for probabilistic order channels. */
for all m ↔_oconf s   // inspect order conflicts
  if benign(C,m,n,e,𝔏,x) == false   // is the conflict harmless?
    return false

/* Check G for probabilistic data channels. */
// initialize the modified I2P-slicer
W = {s}  // a worklist
M = {s ↦ true}  // maps visited nodes to true (phase 1) or false (phase 2)

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

  // look for data conflicts
  if ∃m →_dconf n
    return false
  for all m →_dconf n
    if benign(C,m,n,e,𝔏,x) == false   // is the conflict harmless?
      return false // no, it is harmful

  // proceed with standard I2P slicing
  for all m →_e n, e ∉ {oconf, dconf}   // handle all incoming edges, exclude conflict edges
    // if m hasn't been visited yet or we are in phase 1 and m has been visited in phase 2
    if m ∉ dom M ∨ (¬M(m) ∧ (M(n) ∨ e == conc))
      // if we are in phase 1 or if e is not a call or param-in edge, add m to W
      if M(n) ∨ e ∉ {pi, call}
        W = W ∪ {m}

        /* determine how to mark m: */
        if M(n) ∧ e == po
          // we are in phase 1 and e is a param-out edge: mark m with phase 2
          M = M ∪ {m ↦ false}
        else if ¬M(n) ∧ e == conc
          // we are in phase 2 and e is a concurrency edge: mark m with phase 1
          M = M ∪ {m ↦ true}
        else
          // mark m with the same phase as n
          M = M ∪ {m ↦ M(n)}

until W = ∅

return true // we have found no probabilistic channels
```

procedure `benign`. For each conflict edge, $\mathcal{O}(|N_{CSDG}|)$ complete traversals of the TCFG may be necessary, and the number of conflict edges in the CSDG is bound by $\mathcal{O}(|N_{CSDG}|^2)$. This means a worst case complexity of $\mathcal{O}(|E_{CSDG}| + |N_{CSDG}|^3 * |E_{TCFG}|)$ for one call of procedure `prob`, the first summand being the upper bound for the costs of the extended I2P slicer, the second summand, for procedure `benign`. Since `prob` can be called $\mathcal{O}(|N_{CSDG}|)$ times in the worst case, the complete IFC check has a worst case complexity of $\mathcal{O}(|N_{CSDG}| * |E_{CSDG}| + |N_{CSDG}|^4 * |E_{TCFG}|)$.

Several optimizations trading memory for speed are possible. The harmlessness of a conflict edge needs to be checked at most once for each security level in the lattice $\mathfrak{L}$ and can be cached and reused. It also can be precomputed which sources of information can reach which conflicting nodes, which can be efficiently done in $\mathcal{O}(|N_{CSDG}| * |E_{TCFG}|)$ by computing a context-sensitive forward slice of the TCFG for each source. Both optimizations together reduce the runtime complexity to $\mathcal{O}(|N_{CSDG}|)$ slices of the CSDG in `prob`, $\mathcal{O}(|\mathfrak{L}| * |N_{CSDG}|^2)$ calls of `benign` and $\mathcal{O}(|N_{CSDG}|)$ slices of the TCFG, summing up to a total complexity of $\mathcal{O}(|N_{CSDG}| * |E_{CSDG}| + |\mathfrak{L}| * |N_{CSDG}|^2 + |N_{CSDG}| * |E_{TCFG}|)$.

5.7. Evaluation

We have implemented our IFC technique in Java and integrated it into the Valsoft/Joana project, where it can be used to analyze the information flow in concurrent Java programs. Implicit and explicit flow are investigated by an extension of Hammer et al.'s algorithm [52, 58] to concurrency edges. This section presents an evaluation of our technique. First, we compare it with Smith and Volpano's *weak probabilistic noninterference* [135, 150], with Sabelfeld and Sand's *strong security* [128] and with different realizations of *low-security observational determinism* [64, 98, 121, 160]. It follows a case study committed on a set of 10 programs, which investigates the practicability of our technique. A stress test committed on a set of concurrent Java programs, investigating the runtime behavior of our technique, completes the evaluation.

5.7.1. Weak Probabilistic Noninterference

Smith and Volpano's security property *weak probabilistic noninterference* [135, 150] enforces probabilistic noninterference via a variant of probabilistic bisimulation called *weak probabilistic bisimulation* [135, 150]. A program is weakly probabilistic noninterferent if for each pair of low-equivalent inputs each sequence of low-observable events caused by the one input can be caused by the other input with the same probability. It is called 'weak' because the number of steps between two events in the one run may differ from the number of steps between them in the other run.

Weak probabilistic noninterference addresses illicit explicit and implicit flow and probabilistic channels. Explicitly excluded are timing channels, which permits the probabilistic bisimulation to be weak, and termination channels. This renders their interpretation of low-observable behavior very similar to ours: It consists of a sequence of low-observable events, but lacks information about the time at which such an event occurs. The major difference between their definition of equivalent low-observable behavior and ours is that their definition disallows low-observable events to be delayed infinitely in the one low-observable behavior and being executed in the other. Thus, their definition is stricter with respect to termination channels and only permits the sheer termination of the program to differ. A general advantage of weak probabilistic noninterference over our security property determinism is that it accepts programs that do not work on high data at all.

Whereas their idea of low-observable behavior is very similar to ours, its concrete definition differs. The underlying classification mechanism partitions the variables of the program in question into high and low variables, and the attacker is able to see all low variables at any one time. Hence, the values of the low variables and their changes over time constitute the low-observable behavior. This makes their attacker generally more powerful than ours, because we cannot simulate that capability with our classification mechanism. Our mechanism aims to classify I/O operations as high or low and to treat unclassified operations as invisible, which is in turn not possible with Smith and Volpano's mechanism.

Security constraint

Smith and Volpano present a security-type system that guarantees that well-typed programs are weakly probabilistic noninterferent. Expressions are classified as high or low, dependent on the variables appearing in the expression. An expression is low if it does not contain any high variable, it is high otherwise. A program is weakly probabilistic noninterferent if

- only low expressions can be assigned to low variables,
- a conditional structure with a high guard cannot assign to a low variable and
- a conditional structure whose running time depends on high variables cannot be followed sequentially by assignments to low variables.

The running time of a conditional structure is said to depend on high variables if it is a loop with a high guard or an `if`-structure with a high guard whose branches have different numbers of statements. A special language construct `protect` can be applied to subprograms which do not contain loops and causes an atomic execution of that subprogram. The running time of a `protected if`-structure with a high guard is by definition independent from the high guard.

```
void thread_1():
  h = inputPIN();
  if (h < 0)
    h = h * (-1);
  l = 0;

void thread_2():
  x = 1;
```

```
void thread_1():
  h = inputPIN();
  if (h == 0)
    h = h + 2;
  else
    h = h - 2;
  l = 0;

void thread_2():
  l = 1;
```

Figure 5.6.: Two examples comparing the restrictions of LSOD and weak probabilistic noninterference. We assume that Smith and Volpano's technique classifies variables `h` and `x` as high and `l` as low, and that our technique classifies `h = inputPIN()` as a high source and `l = 0` and `l = 1` as low sinks. The left program is accepted by our condition and rejected by theirs, the right program is rejected by ours and accepted by theirs.

A weakness of Smith and Volpano's security-type system is that it lacks a detection of conflicts. Probabilistic channels are prevented by forbidding assignments to low variables sequentially behind conditional structures whose running time depends on high variables, which is very restrictive. The program on the left side of Fig. 5.6 is rejected by their security constraint, because the running time of the `if`-structure depends on high data and is followed sequentially by `l = 0`. However, it does not contain a probabilistic channel because `l = 0` is not involved in an order conflict or influenced by a data conflict. It therefore satisfies our security constraint LSOD. The `protected` construct produces relief, but it is non-standard and would have to be integrated into the desired language.

The type system assumes that a single statement has a fixed running time, which is not necessarily the case in presence of caching and pipelining. Based on this assumption, programs like that on the right side of Fig. 5.6 are accepted by their type system, because the branches of the `if`-structure have equal length and thus different values of `h` do not alter the probabilities of the possible ways of interleaving of `l = 0` and `l = 1`. We explicitly aim to reject such programs, arguing that different running times of `h = inputPIN()` could already cause a probabilistic channel, and our security constraint rejects the program because of the data conflict between `l = 0` and `l = 1`.

Smith and Volpano's security-type system is restricted to probabilistic schedulers and breaks, for example, in the presence of a round-robin scheduler, for which Smith provides an example [135]. This a disadvantage compared with LSOD, which holds for every scheduler. Similar to LSOD, the type system does not support a modular verification of programs and libraries.

5.7.2. Strong Security

Sabelfeld and Sands' security property *strong security* [128] addresses implicit and explicit flow, probabilistic channels and termination channels. It enforces probabilistic noninterference for a large class of schedulers, namely all schedulers whose decisions are not influenced by high data. It makes the following requirements to a program p and all possible pairs (t,u) of low-equivalent inputs: Let $\mathfrak{T}$ and $\mathfrak{U}$ be the set of possible program runs resulting from t and u. For every $T \in \mathfrak{T}$, there must exist a low-equivalent program run $U \in \mathfrak{U}$.

Even though it looks like a possibilistic property, strong security is capable of preventing probabilistic channels, the trick being the definition of low-equivalent program runs: Two program runs are low-equivalent if they have the same number of threads and they produce the same low-observable events and create or kill the same number of threads at each step under any scheduler whose decisions are not influenced by high data. This 'lockstep execution' requirement allows to ignore the concrete scheduling strategy.

Programs are classified by partitioning variables into high and low variables. The attacker sees all low variables at any one time and is aware of program termination, so the values of the low variables, their changes over time and the termination behavior constitute the low-observable behavior. Strong security assumes that the attacker is not able to see which statement is responsible for a low-observable event and is designed to identify whether two syntactically different subprograms have equivalent semantic effects on the low-observable behavior, which makes it possible to identify programs like that on the left side of Fig. 5.7 as secure. Even though the assignments to the low variable `l` are influenced by high data via implicit flow, strong security states that the low-observable behavior is not, because both branches lead to 0 being assigned to `l`. Our security property is not able to recognize this program as secure and rejects it, the same holds for weak probabilistic noninterference.

Comparing their definition of low-observable behavior with ours and with the one of Smith and Volpano shows typical disagreement on the capabilities of the attacker. Sabelfeld/Sands and Smith/Volpano assume that the attacker is able to see low variables at any time, which we do not, Smith/Volpano and we assume that the attacker cannot exploit termination channels and is able to identify statements responsible for low-observable events, which is contrary seen by Sabelfeld/Sands.

The requirement of lock-step execution renders strong security compositional with respect to sequential and parallel composition, which means that strongly secure programs can be combined sequentially or in parallel to a new strongly secure program. Sabelfeld [127] has proven that strong security is the least restrictive security property that provides this degree of compositionality and scheduler-independence. Its compositionality is its outstanding property and an advantage over our property, which is not compositional. On the other hand, lockstep execution

```
void main():
  h = inputPIN();
  if (h < 0)
    l = 0;
  else
    l = 0;
```

```
void thread_1():
  h = inputPIN();
  if (h < 0)
    h = h * (-1);
  else
    skip;
  l = 0;

void thread_2():
  x = 1;
```

Figure 5.7.: Two examples demonstrating the capabilities of strong security. We assume that `h` and `x` are classified as high and `l` as low. The left program is strongly secure, because both branches assign the same value to `l`. The right program is a transformation of the program on the left of Fig. 5.6, where the additional `skip` statement removes the probabilistic data channel.

imposes serious restrictions to programs, and an investigation of its practicability remains an important open issue.

The restriction to schedulers not working on high data means that any information possibly used by the scheduler, for example thread priorities or the mere number of existing threads, must be classified as low. This in turn means that the classification of a program becomes scheduler-dependent, so the scheduler-independence of strong security is bought by making the classification scheduler-dependent. This makes it possible to break strong security by running the program under a scheduler for which the attacker knows the classification of the program to be inappropriate. This is a disadvantage compared with our technique, whose security property, security constraint and classification mechanism are scheduler-independent.

Security constraint

Sabelfeld and Sands present a security-type system that ensures that a well-typed program is strongly secure. The type system checks similarly to that of Smith and Volpano whether implicit and explicit flow is secure and disallows loops with high guards completely in order to prevent termination channels.

Similar to Smith and Volpano, the authors assume that a single statement has a fixed execution time. Under that assumption, probabilistic channels may only appear if assignments to low variables are sequentially preceded by `if`-structures with high guards (since they forbid loops with high guards completely). The specific feature of their type system is that it transforms `if`-structures with high guards such that they cannot cause probabilistic channels. For that purpose, it pads the branches of such an `if`-structure with `skip`-statements until the branches have the same number of statements. For example, the program on the left side of Fig. 5.6 would be transformed to the program on the right side of Fig. 5.7 and would then be accepted.

The removal of probabilistic channels by padding `if`-structures with `skip`-statements is clearly a proof of concept, as it assumes that the execution time of `skip` is equal to the one of any other single statement. However, the idea of transforming out probabilistic channels is very appealing and is hopefully investigated further. At the time being, a practical employment of strong security would rather follow Smith and Volpano's idea of identifying whether the branches of `if`-structures have equal length.

5.7.3. Low-Security Observational Determinism

Next, we compare our IFC technique with two recent approaches to enforce low-security observational determinism, provided by Zdancevic and Myers [160] and by Huisman et al. [64].

Low-security observational determinism by Zdancevic and Myers

Zdancevic and Myers [160] pointed out that conflicts are a necessary condition of probabilistic channels. They suggested combining a security-type system for implicit and explicit flow with a conflict analysis, arguing that programs without conflicts have no probabilistic channels.

The authors address implicit and explicit flow as well as probabilistic data channels, called *internal timing channels* in [160]. They exclude termination channels and probabilistic order channels and justify that by confining the attacker to be a program itself (e.g. a thread). Such an attacker is not able to observe the relative order of low-observable events, because such an observation requires a probabilistic data channel in which the differing relative orders manifest.

A program is classified by partitioning variables into high and low, hence the low-observable behavior of a program run consists of the changes of low variables over time. Since the relative order of low-observable events does not matter, each low variable can be inspected in isolation. Let T be a program run and $T(v)$ be the sequence of values a variable v has during the program run, called the *location trace* of v. Two program runs T and U are low-equivalent if for every low variable l the location traces $T(l)$ and $U(l)$ are equal up to the length of the shorter run and up to *stuttering*, which means that up to the shorter sequence l undergoes the same changes in both program runs, but not necessarily at the same time. For example, if $T(l) = \langle 0,0,0,1,1,2,3\rangle$ and $U(l) = \langle 0,1,2\rangle$, then T and U are equivalent with respect to l.

The authors demonstrate how to enforce their security property for λ_{SEC}^{PAR}, a concurrent language with message-passing communication. Choosing message-passing makes the detection of data conflicts more specific: Data conflicts can only appear due to conflicting accesses to the same communication channel. The language provides *linear channels*, communication channels that are used for transmitting exactly one message and thus guarantee conflict-free communication. The authors present a security-type system for λ_{SEC}^{PAR} that verifies the confidentiality of implicit and explicit flow and that linear channels are used exactly once. The type system

guarantees that well-typed programs are low-security observational deterministic if they are additionally free of data conflicts. However, a suitable analysis of data conflicts is not presented.

Comparison Zdancevic and Myers' attacker is weaker than ours, as probabilistic order channels are excluded. It is explicitly designed to tackle malicious threads spying out confidential information in the host system. It is possible to modify our analysis to comply with their attacker, by simply skipping the detection of probabilistic order channels.

Their security constraint requires that programs are completely free of data conflicts, which is much stricter than ours. This requirement virtually prevents an application to languages with shared memory, because any program containing a data conflict would be rejected, even if the conflict does not influence the low-observable behavior at all.

Huisman et al. [64] pointed out that Zdancevic and Myers' security property contains a leak because its definition of low-equivalent program runs is restricted to the length of the shorter run. We describe that problem in the next part.

Low-security observational determinism by Huisman et al.

Huisman et al. [64] took up and improved the work of Zdancevic and Myers. Foremost, they pointed out that Zdancevic and Myers' security property contains a leak, because its definition of low-equivalent program runs is restricted to the length of the shorter run. Consider the program on the left of Fig. 5.8, taken from [64], which copies a secret PIN to variable `l` and then prints it. It is sequential and therefore free of conflicts. Because of the loop, no two program runs with low-equivalent inputs and different input values for `h` have the same length. The additional assignments to `l` in the longer run, after which `h` has been copied to `l`, always fall out of the comparison. But up to the length of the shorter program run `l` has the same values in both program runs after every step, hence the program is accepted by Zdancevic and Myers' property. Huisman et al. [64] close that leak by strengthening the definition of low-equivalent program runs: Two program runs T and U are low-equivalent if for every low variable l the location traces $T(l)$ and $U(l)$ are equal up to stuttering. This means that assignments to low variables sequentially behind loops iterating over high data are forbidden.

The authors present an additional definition of low-equivalent program runs that closes termination channels. It additionally requires that either both program runs terminate or none of them. They also describe the necessary measurements to encounter probabilistic order channels. This can be achieved by extending location traces to the set L of low variables: In that case two program runs T and U are low-equivalent if the set of low variables in T and U undergoes the same sequence of changes in both program runs up to stuttering.

The authors enforce their security property via model checking. They have formalized their different security properties via two temporal logics, CTL* and the polyadic modal μ-calculus,

```
void main():
  h = inputPIN();
  l = 0;
  while (h > 0)
    l++;
    h--;
```

```
void main():
  h = inputPIN();
  x = y;
  fork thread_1();
  fork thread_2();
```

```
void thread_1():
  x = 0;
  print(x);

void thread_2():
  y = 1;
```

Figure 5.8.: Two examples demonstrating the strengths and weaknesses of Huisman et al.'s security property. Both programs are rejected, the first because it copies the PIN to the low variable `l`, the second because the order of the assignments `x = 0` and `y = 1` depend on interleaving.

for which the model-checking problem is decidable if the program in question can be expressed by a finite-state-machine. This permits a very precise detection of relevant data conflicts, such that total freedom of data conflicts is not required. Hence, their approach can be applied to languages with shared-memory communication.

Comparison Huisman et al.'s security property is very flexible, as it permits to include and exclude termination channels and probabilistic order channels. However, it is also more restrictive than ours in several aspects. It is stricter towards termination channels, because it forbids low-observable events sequentially behind loops iterating over high data. Furthermore, the optional treatment of probabilistic order channels imposes severe restrictions on the analyzed programs. Since a program is classified by partitioning variables into high and low, each assignment to a low variable is regarded as a low-observable event. The security property addressing probabilistic order channels requires that two low-equivalent program runs must make the same sequence of changes to low variables. This means in effect that if two threads work on different low variables, then the assignments to these variables must have a fixed interleaving order, even if the variables are completely unrelated (apart from being 'low').

As an example, consider the program on the right side of Fig. 5.8. Its main thread reads a PIN, assigns `y` to `x` and then forks both threads. Thread 1 sets `x` to 0 and then prints it, thread 2 sets `y` to 1. Assume that the PIN is high data and the output is low-observable. Using Huisman et al.'s technique, `h` is classified as a high variable and `x` as a low variable. Now it is compulsory that `y` is also classified as low because otherwise the assignment `x = y` would be illegal. This means that the assignments `x = 0` and `y = 1` are low-observable. Since the order of these assignments is not fixed, the program is rejected. Using our IFC technique, `h = inputPIN()` is classified as a high source and `print(x)` as a low sink, and the program is accepted by our security property. We therefore argue that our approach of classifying operations instead of variables is better suited for low-security observational determinism, because it permits a much less restrictive treatment of probabilistic order channels.

5.7.4. Case Study

We have applied our implementation to 10 concurrent programs in order to investigate its usefulness and practicability. We used the simple two-point lattice $low \sqsubseteq high$ to classify sources and sinks in these programs and analyzed them with our implementation.

Two of the programs serve as example programs in the literature and contain different kinds of probabilistic leaks and handicaps. The first program is taken from Smith and Volpano [136, p. 3] and contains a probabilistic data channel. The second program is an example provided by Mantel et al. [95, p. 10] and is probabilistic noninterferent, which is difficult to discover. The third program is a self-written password check routine in which a malicious thread creates a probabilistic order channel. We ported these programs to Java, the code can be found in Appendix A. For these programs, we investigated whether our implementation is able to find the known leaks.

The remaining 7 programs stem from our benchmark (Table 3.1), namely KnockKnock, LaplaceGrid, SharedQueue, RayTracer, MonteCarlo, J2MESafe and Podcast. In these programs, we classified statements that intuitively looked like fitting sources and sinks and investigated the reported leaks.

Program 'SmithVolpano'

The first program, shown in section A.1, reads a `PIN` and employs three threads to compute a value `result`, which is finally printed. There is no explicit or implicit flow from `PIN` to `result`, so an IFC analysis considering only these kinds of information flow classifies the program secure. But the assignments to `result` in threads `Alpha` and `Beta` are conflicting, and the outcome of the conflict is influenced by the values of `trigger0` and `trigger1`, which in turn are changed dependent on `PIN`'s value in thread `Gamma`. Thus, this program contains a probabilistic data channel which leaks information about `PIN` to `result`. And actually, according to Smith and Volpano [136], if the input `PIN` is less twice the value of variable `mask`, then `PIN`'s value is eventually copied into `result` and printed to the screen (provided that scheduling is fair).

Conduction and Results We classified statement `PIN = Integer.parseInt(args[0])` as a high source and statement `System.out.println(result)` as a low sink. Our algorithm detected a probabilistic data channel from the source to the sink, as required.

Program 'Mantel'

The second program, shown in section A.2, describes an application managing a stock portfolio of Euro Stoxx 50 entries. It consists of four threads, coordinated by an additional `main`

thread. The program first runs the `Portfolio` and `EuroStoxx50` threads concurrently, where `Portfolio` reads the user's stock portfolio from storage and `EuroStoxx50` retrieves the current stock rates. When these threads have finished, threads `Statistics` and `Output` are run concurrently, where `Statistics` calculates the current profits and `Output` incrementally prepares a statistics output. After these threads have finished, the statistics are displayed, together with a pay-per-click commercial. An ID of that commercial is sent back to the commercials provider to avoid receiving the same commercial twice. The portfolio data, `pfNames` and `pfNums`, is secret, hence the Euro Stoxx request by `EuroStoxx50` and the message sent to the commercials provider should not contain any information about the portfolio. As `Portfolio` and `EuroStoxx50` do not interfere, the Euro Stoxx request does not leak information about the portfolio. The message sent to the commercials provider is not influenced by the values of the portfolio, too, because there is no explicit or implicit flow from the secret portfolio values to the sent message. Furthermore, the two outputs have a fixed relative ordering, as `EuroStoxx50` is joined before `Output` is started. Hence, the program should be considered secure.

Conduction and Results We classified the two statements reading the portfolio from storage, `pfNames = getPFNames()` and `pfNums = getPFNums()`, as high sources and the flushes of output stream `nwOutBuf` in `EuroStoxx50` and at the end of the `main` procedure as low sinks. The challenge of this program is to detect that `EuroStoxx50` is joined before `nwOutBuf` is flushed in the `main` procedure, because otherwise it cannot be determined that the two flushes of `nwOutBuf` have a fixed execution order. An then the program would have to be rejected because the resulting order conflict is influenced by both sources.

Our MHP analysis was able to detect that the joins of the threads are must-joins, which enabled our IFC algorithm to identify that there is no order conflict between the two flushes of `nwOutBuf`, therefore no probabilistic channel was reported. But surprisingly, it detected illicit implicit flow from both sources to the flush of `nwOutBuf` in `main`. An inspection of the CSDG and TCFG of the program revealed that these leaks result from possible exceptions, introduced by converting the program to Java: The execution of the sink `nwOutBuf.flush()` in `main` depends on the `join`-procedures being executed successfully – if they throw an exception, the program is aborted. Thus, there is implicit flow from the call of `join` for thread `Portfolio` to the sink. And since reading the portfolio from storage might cause an exception, there is also an implicit flow from these statements to the call of `join` for the `Portfolio` thread, resulting in the illicit implicit flow reported by our analysis.

Our CSDG generator permits to ignore exceptions during the construction of the CSDG. If done so, the algorithm does not detect any information leaks and accepts the program. If the treatment of joins in our MHP analysis is turned off, the analysis detects two probabilistic order channels between the two flushes of `nwOutBuf`, one leaking information about `pfNames = getPFNames()`, the other leaking information about `pfNums = getPFNums()`.

Program ‘PasswordCheck’

The third investigated program is a password check routine that contains a probabilistic order channel due to a malicious thread. Its code is shown in section A.3. The program is called with a user name and a password, which are checked by procedure `check` in the main thread against a password data base. The `passwords` array is considered secret data, whereas the output is publicly visible, so the password check already leaks information, which we aim to permit by declassifying the data at `check`'s return statement. The program contains a thread that executes concurrently to `check`. This thread simulates `check`, except for accessing the passwords array, where instead a string consisting of 8 chars is compared with itself. Finally, the thread performs an output signaling it has finished the loop. The intuition behind that thread is that its output conflicts with the output in the main thread and that the outcome of this conflict is influenced by the length of the password. Assume that the scheduler schedules after each statement and picks both threads with the same probability: If the password consists of 8 chars, then both `check` and the thread need the same execution time on average, so both possible relative orderings are balanced. If the password is shorter, the output in `main` more likely appears first; if it is longer, it more likely appears last. This is valuable information for the preparation of a password attack.

Conduction and Results The program demonstrates that our analysis is able to find probabilistic order channels and to declassify implicit and explicit flow without masking probabilistic channels. We classified `passwords = {"x", "y"}` as a high source and the two outputs as low sinks. Our analysis identified an illicit implicit flow from the source to the output in `main` and a probabilistic order channel leaking information about the source via the order conflict between the two outputs. In a second step, we defined statement `return match` to declassify high information to low, legalizing the implicit flow from the source to the output in `main`. This time, our analysis reported only the probabilistic order channel.

The other programs

In the remaining seven programs, we annotated statements printing to the System.out stream as low sinks if they were located in exceptional code or served for logging or debugging purposes. As high sources we chose the initializations of variables which contained the main data processed by the programs or information such as passwords or hostnames.

Table 5.1 summarizes the results of our analysis. It reports the number of high sources, low sinks, probabilistic data (PD) and order channels (PO). We additionally used timing-sensitive slicing to see if it can be used to lower the number of reported probabilistic channels. Even though it lowered the number of encountered data conflicts, it did not improve precision as much as we expected. Only for J2MESafe it was able to reduce the number of reported channels. For

Table 5.1.: The number of probabilistic data (PD) and order channels (PO) detected by our IFC algorithm. The rightmost two columns show the effect of timing-sensitive slicing on the precision of the results.

Name	Sources	Sinks	Timing-insens.		Timing-sens.	
			PD	PO	PD	PO
LaplaceGrid	2	1	2	2	2	2
SharedQueue	1	3	3	6	3	6
KnockKnock	2	2	4	2	4	2
RayTracer	1	2	2	2	–	–
MonteCarlo	2	2	4	4	4	4
J2MESafe	2	6	20	0	12	0
Podcast	1	1	1	1	1	1

program RayTracer, the timing-sensitive slicer did not finish computation within reasonable time.

Our implementation is able to collect the data and order conflicts responsible for a probabilistic channel. We used that feature to manually inspect the reported channels. The reported probabilistic order channels could be inspected and confirmed manually; the statements involved in the underlying order conflicts can be executed concurrently. A manual inspection of the probabilistic data channels was only possible for KnockKnock and for J2MESafe, because the channels in the other programs involved too many data conflicts. The probabilistic data channels reported for KnockKnock were induced by the same two data conflicts, which could be identified as false positives. They connect the fields of two string arrays, which are not shared between the involved threads. Program J2ME shows a similar result. Its probabilistic data channels are based on the same 13 - 14 data conflicts, which connect local variables of different threads, so these channels could also be identified as false positives.

5.7.5. Runtime Behavior

To complete our evaluation, we investigated how well our implementation scales with increasing program sizes, lattice sizes and numbers of sources and sinks. We applied our algorithm to the seven programs used in the previous part, KnockKnock, LaplaceGrid, SharedQueue, RayTracer, MonteCarlo, J2MESafe and Podcast, and to our biggest program, Cellsafe. We used three different security lattices: Lattice A is a simple chain of three elements, *public* $\sqsubseteq$ *confidential* $\sqsubseteq$ *secret*. Lattice B consists of 22 elements, arranged in a lattice of height 9, resulting in many incomparable pairs of elements. Lattice C is a huge lattice of height 7 with 254 elements. For each program and lattice, we randomly chose 10 sources and 10 sinks, 33 sources and 33 sinks and finally 100 sources and 100 sinks of random security levels and analyzed the

Table 5.2.: Average execution times of our IFC algorithm for different programs, lattices and numbers of sources and sinks (in seconds).

Name + Lattice	sources x sinks 10 x 10	33 x 33	100x 100	Name + Lattice	sources x sinks 10 x 10	33 x 33	100x 100
LG + A	1.6	4.8	21.2	MC + A	17.1	53.3	224.2
LG + B	1.6	5.8	29.7	MC + B	18.7	53.3	173.6
LG + C	2.0	9.5	170.7	MC + C	17.5	54.8	205.0
SQ + A	5.9	17.2	54.0	JS + A	2.2	5.2	18.8
SQ + B	5.5	17.3	68.0	JS + B	2.4	5.6	20.3
SQ + C	5.8	21.1	162.5	JS + C	2.4	5.8	40.7
KK + A	25.7	58.5	170.0	PO + A	6.4	18.1	54.6
KK + B	22.1	57.5	187.8	PO + B	7.4	19.1	66.8
KK + C	25.2	64.9	256.2	PO + C	7.0	20.4	89.8
RT + A	8.9	25.3	99.3	CS + A	19.4	52.5	153.3
RT + B	7.3	23.9	116.1	CS + B	21.5	52.1	160.2
RT + C	8.4	27.2	175.1	CS + C	21.0	53.6	177.3

classified programs with our algorithm. We thereby measured the execution times of the whole algorithm, of the scan for probabilistic channels and of the scan for illicit and explicit flow. Since the execution times are presumably heavily dependent on the randomly chosen sources and sinks, the test was run ten times and the presented results are the average values.

Table 5.2 shows the average execution times of our IFC algorithm. It contains one row for each combination of program and lattice, i.e. row 'LG + A' contains the results for LaplaceGrid and lattice A. The numbers reveal that the most important factor influencing the runtime behavior is, besides the sheer size of the program, the number of sources and sinks. If only 10 sources and 10 sinks were selected, the size of the lattice in charge did not really matter. The runtime for lattice C was in several cases faster than that for lattice B or A. The cause of that behavior is that with 10 sources and 10 sinks the size of lattice C is not exhausted – the classification can introduce at most 20 different security levels to the analysis, the same maximal number as with lattice B. With 33 sources and 33 sinks the huge size of lattice C slowly became noticeable. Here the analysis for a program with lattice C was in all cases the most expensive. With 100 sources and 100 sinks the size of lattice C eventually became the dominating cost factor.

Table 5.3 shows the percentage share of the probabilistic channel detection among the overall execution times. The remaining time was consumed by the algorithm of Hammer et al. [58], which we employed for verifying the explicit and implicit flow. The results show that the two checks were similarly fast. However, it should be noted that the performance of the detection of probabilistic channels seems to decline stronger for huge lattices than Hammer et al.'s algorithm. For lattice C and 100x100 sources and sinks, the detection of probabilistic channels was

Table 5.3.: The percentage share of the probabilistic channel detection among the overall execution times.

Name + Lattice	sources x sinks 10 x 10	33 x 33	100x 100	Name + Lattice	sources x sinks 10 x 10	33 x 33	100x 100
LG + A	53	48	62	MC + A	45	38	38
LG + B	56	54	73	MC + B	43	38	39
LG + C	58	73	94	MC + C	44	40	47
SQ + A	44	39	44	JS + A	52	46	41
SQ + B	43	40	55	JS + B	46	46	46
SQ + C	43	50	80	JS + C	53	54	71
KK + A	57	45	43	PO + A	46	38	35
KK + B	58	46	45	PO + B	45	37	36
KK + C	58	48	58	PO + C	43	39	46
RT + A	44	40	44	CS + A	38	29	28
RT + B	49	41	50	CS + B	34	30	28
RT + C	47	46	67	CS + C	32	31	34

in most cases more time-consuming. According to Hammer [52, 53], slicing-based IFC gets along with comparatively few annotations, which is an encouraging diagnosis.

5.7.6. Study summary

Our evaluation confirms that our IFC technique is a convenient way to enforce probabilistic noninterference for programs written in a contemporary language with threads. The power of our security property and our security constraint is competitive to well-known security properties and constraints in the literature, its strong points being independence from the scheduler and comparatively low restrictions.

Our algorithm based on slicing of CSDGs is capable of full Java bytecode, our case study shows that it is able to analyze concurrent Java programs. The examined programs contain typical Java constructs such as objects, dynamic dispatch, forking and joining of multiple threads and exception handling. Our technique needs only a few annotations per program – the user has to identify sources and sinks of information and declassification points.

The case of the ‘Mantel’ program shows that detecting the causes of unexpected information leaks remains complicated. It currently requires manual inspection of the dependence graph, even though slicing and chopping can be used to narrow down the responsible program parts. It remains future work to develop techniques that aid the user in identifying the program points responsible for information leaks.

The investigation of the seven programs from our benchmark reveals that the precision of our IFC technique suffers from many false positive data conflicts in the CSDGs, a problem that has to be solved to make the technique practical. In section 3.12.1, we concluded that the context-

insensitive points-to analysis used in the Joana framework causes many spurious interference and interference-write dependences. Future work should investigate methods that reduce that imprecision, for example, thread-sensitive points-to analyses. The problem of timing-sensitivity seems to come second at the moment.

Our stress test indicates that our algorithm is fast enough to analyze real-world programs. However, its scalability depends strongly on the number of sources and sinks in the analyzed program.

The weak point of our evaluation is that the used programs are no real security-sensitive programs. The evaluation should be extended to a set of real security-sensitive programs for which the security lattices and classifications can be derived from a specification. Important questions besides practicability, precision and runtime performance are:

- Which sizes of security lattices and which numbers of sources and sinks are typical?
- What is the effort of transferring the security specifications of typical security-sensitive programs to classifications? Is our classification mechanism practical?
- Can our technique be employed to guide the development of security-sensitive software? Is such an employment of any use?

5.8. Discussion and Future Work

We want to propose several directions for future work.

Termination channels

Our security property excludes termination channels. This is common practice in IFC techniques for sequential programs, because termination channels are assumed to be sufficiently small, an assumption that does only hold as long as programs are seen as black boxes (see [16] for a discussion). As soon as programs interact with a user, termination channels can be used to leak an arbitrarily amount of information. We presented an example of such a program in section 5.4.5, on the right side of Fig. 5.3. These channels can be prevented by forbidding low-observable behavior behind loops with guards that may receive high data, but this is a too severe restriction. A *termination analysis* [149] for loops could solve that problem: Low-observable behavior behind such a loop can be permitted if its termination is underwritten by a static analysis.

Declassification of probabilistic channels

We currently do not provide a declassification mechanism for probabilistic channels. Instead of declassifying probabilistic channels, we consider Zdancevic and Myers' idea of using *linear*

channels for a deterministic communication between threads [160] more promising. Linear channels can be integrated in form of a library into languages with shared memory. We have recently a added such a library as a proof-of-concept implementation to Valsoft/Joana, but our experiences with it are very preliminary and are not reported here.

Conditioned slicing

We suggest an investigation of how *conditioned slicing* [29] can be used for information flow control. A conditioned slice of a program p is computed with respect to a first order formula F on a subset of the input variables of p and has only to consider the program executions feasible with respect to F. Conditioned slicing is able to yield much smaller slices than traditional slicing, which are still correct with respect to F. An IFC technique built on conditioned slicing could yield extremely precise, correct results. However, to the best of our knowledge, conditioned slicing has not been extended to concurrent programs yet.

5.9. Related Work

Due to the vast amount of literature about language-based information flow control, we focus on related work on probabilistic noninterference and slicing-based IFC. For information about other topics of language-based information flow control we recommend the excellent survey published by Sabelfeld and Myers [126]. A good summarization of declassification mechanisms is presented in Sabelfeld and Sands' recent publication [130].

Probabilistic noninterference

Probabilistic noninterference has its origins in McLean's *Flow Model* [97], Gray's *P-Restrictiveness* [50] and Gray's *probabilistic nondeducibility on strategy* [51]. These security properties are rather abstract and leave a gap between their system models and concrete source code, which took some time to bridge. Smith and Volpano [136] were the first to bridge that gap and presented a security-type system that guarantees that type-safe programs are probabilistic noninterferent. Their work origins the usage of probabilistic bisimulation for security properties based on probabilistic noninterference.

Important subsequent publications on that field are Smith and Volpano's weak probabilistic noninterference [133, 134, 135, 150], Sabelfeld and Sands' strong security [128, 127], which we have both described in section 5.7, and Boudol and Castellani's work [28], who independently from Smith and Volpano developed a security-type system that analyzes whether the execution time of `if`-conditionals depends on high guards.

Mantel and Sudbrock [94] address and relieve strong securities' requirement of lock-step-execution by restricting the class of valid schedulers to *robust schedulers*, arguing that this class

comprises the most important schedulers, such as round-robin and priority-based scheduling. Their approach assigns each thread a security level which is set to high as soon as the thread executes a conditional structure with a high guard and cannot be reset to low. A scheduler is *robust* if the probability of choosing a low thread among the set of low threads is not influenced by the high threads. Their security property, FSI, is basically a relaxation of strong security which requires lock-step execution only from low threads. However, a FSI-secure program is not FSI-secure for the class of robust schedulers, but S-secure, which is another security property defined in that work. S-security restricts probabilistic noninterference to finite program runs, which means that S-secure programs containing nonterminating loops may leak arbitrary information. FSI-secure programs are disallowed to assign to low variables inside or sequentially behind conditional structures with high guards, so this potential leak seems to be closed. However, the gap between FSI-security and S-security needs a close investigation of possible covert channels. Furthermore, it should be explored whether the class of robust schedulers allows commonly used techniques for ensuring fairness and liveness.

Incorporating the scheduler Since scheduler-independence imposes many restrictions on programs, several authors suggest incorporating the scheduler.

Russo and Sabelfeld [125] describe how cooperative scheduling can be used to realize Smith and Volpano's `protect`-construct. Cooperative scheduling is a kind of non-preemptive scheduling, where the executing process has to yield the processor to allow the scheduler to choose another process. The authors forbid these yield-commands inside conditional structures with high guards and present a security-type system which transforms the `protect`-constructs in a program typed with Smith and Volpano's type system in [150] into appropriate yield-commands.

In a subsequent publication, Russo and Sabelfeld [124] show that if the employed scheduler itself satisfies a noninterference property it is possible to enforce probabilistic noninterference via a compositional yet permissive security property. The idea is that threads themselves receive a security level, dependent on the security level of conditional structures. During the execution of a conditional structure with a high guard a thread is considered high (and can be reset to low afterwards). The scheduler has to distinguish between low and high threads and has to guarantee that scheduling of low threads is independent from high threads. Under this assumption it is possible to achieve probabilistic noninterference by a security-type system which only has to ensure that the implicit and explicit flow is secure and that threads receive the security level of the guard of the currently executed conditional structure. Barthe et al. [19] transfer Russo and Sabelfeld's approach to a simple assembly language with dynamic thread creation and argue that a similar instantiation can be done for Java Bytecode.

Combining different security properties Mantel et al. [95] present the *combining calculus*, a framework that allows to compose concurrent programs that are secure with respect to different

security properties to a new program which satisfies a form of possibilistic noninterference. They define conditions secure programs additionally have to fulfill in order to be composable. For example, two programs can be composed sequentially if the first program is *main-surviving*, i.e. the last existing thread is always the main thread. They can be composed in parallel if the sets of variables occurring in them are disjoint. The combining calculus permits to analyze subprograms with different IFC techniques, so the degree of security and the imposed restrictions may vary.

Low-security observational determinism

McLean [98] and Roscoe [121] were the first to propose observational determinism as a means to avoid dealing with schedulers. McLean's approach [98] ensures that implementations of concurrent systems are low-security observational deterministic with respect to a trace-based specification. Roscoe [121] uses the standard representation of determinism in the CSP calculus and specifies an algorithm that checks whether a non-deterministic process p is low-security observational deterministic.

Zdancevic and Myers [160] and later Huisman et al. [64] showed how low-security observational determinism can be enforced on the level of source code. Their techniques have been described in section 5.7. Terauchi [145] continues their work and presents an improved security-type system for a concurrent language with message-passing communication, which permits data conflicts in program parts that do not contribute to the low-observable behavior. For that purpose, the author integrates a *fractional determinism checker* for concurrent programs [146] in his type system. The author's definition of low-observable behavior differs from the ones of Zdancevic and Myers and of Huisman et al. It is not based on trace-locations, but treats all low variables at once, just as Huisman et al.'s proposed treatment of probabilistic order channels. This strengthening enables their security property to detect probabilistic order channels, but makes it much more restrictive, as we have pointed out in section 5.7.3. Furthermore, it does not permit stuttering and requires in summary that two traces resulting from low-equivalent inputs must make the same updates to the low variables at every step. In order to permit termination channels, low-equivalence of traces is only required up to the length of the shorter trace. This limitation reintroduces the leak of Zdancevic and Myers's condition, although Terauchi argues that disallowing stuttering and treating all low variables as a whole closes that leak. However, the program on the left side of Fig. 5.8 is accepted by his condition, even though it is not secure.

IFC based on data flow analysis

Probably the first who connected data-flow analysis and IFC were Bergeretti and Carré [21]. They specify a data-flow analysis for a Pascal-like language without procedures that computes

which initial definitions of variables (i.e. input) may influence which final definitions of variables (i.e. output) and which statements are involved in transferring that information.

Abadi et al. [7] found that IFC and slicing share a common notion of dependency and present the Dependency Core Calculus, an extension of the computational lambda calculus. The authors show that slicing as well as a security-type system for possibilistic noninterference from Smith and Volpano [136] can be translated into the DCC. These translations enable a direct comparison between slicing and security-type systems. However, the authors model slicing through a type system, hence the results are not representative for SDG-based slicing.

Snelting [137] proposed to use slicing to verify the information flow in a program and to use *path conditions* as witnesses for illicit information flow. A path condition between two statements, s and t, is a necessary condition on the program state that a program run has to satisfy in order to reach t, when coming from s. The path condition is composed of all predicates influenced by s and influencing t. A path condition for an information leak enables to reproduce situations in which that leak occurs and thus possesses evidentiary value. Path conditions also permit to render slicing-based IFC more precisely, because a path condition may reveal unrealizable paths traversed by the slicing algorithm. In [138], Snelting et al. explain in detail how path conditions can be realized for C-like languages. Hammer and Schaade [56] extend path conditions to object-oriented languages and report a preliminary implementation for Java. However, to date an integration of path conditions into slicing-based IFC has not been reported, so its practicability remains to be demonstrated.

Wasserrab et al. [154] provide a machine-checked proof formalized in Isabelle\HOL that intra-procedural slicing can be used to enforce noninterference. The proof is based on CFGs and a set of wellformedness properties (which inspired ours in section 5.4), so that it holds for every language and program whose CFGs fulfill these properties. Wasserrab and Lohner [153] extend that proof to the two-phase slicer for interprocedural programs.

Yokomori et al. [159] show how a modified SDG generator can be used to compute the information flow in a procedural language and report an implementation for Pascal. A user has to mark the high input statements, then the system computes the security levels of the output statements during the data flow analysis. The SDG generator is modified such that it does not compute the program dependences but propagates the security levels through the control flow graph. The analysis is not context-sensitive, as it takes the least upper bound of the results of different invocations of one procedure. Furthermore, it is restricted to a two-point security lattice, even though an extension to arbitrarily security lattice seems to be straightforward.

Kuninobu et al. [80] use abstract interpretation to compute the security levels of return values of functions based on the security levels of the parameters of the functions. Their technique permits arbitrary security lattices, but does not include a declassification mechanism. Their

approach has been implemented for a subset of C and is able to deal with recursion and global variables.

Hammer et al.'s slicing-based IFC technique [52, 58] is currently the most sophisticated, because it is able to handle full sequential Java bytecode and permits declassification. His technique has been described in section 5.1.4. Preliminary versions of his technique have been published in [54, 55].

Quantitative information flow control Data flow analysis can also be used for *quantitative* information flow control, which measures the capacity of information leaks. Clark et al. [34] define a sound set of rules computing the capacity of information leaks in a sequential while-language. These rules are based on control and data dependence. Chen and Malacaria's approach [30] is able to compute the capacity of probabilistic channels: They transform a multi-threaded program into a single-threaded one, where an outer while-loop together with probabilistic operators simulates probabilistic scheduling, and apply an earlier developed formula [92] able to measure the leakage of loops.

McCamant and Ernst [96] developed an analysis that dynamically measures the maximal numbers of bits leaked by an information leak. The analysis creates a dependence graph for a program execution and augments each edge with the number of high bits that are maximally transferred by that edge. These numbers are determined by a bit capacity analysis, which marks all bits of high data and tracks them during the program execution. Having built the augmented dependence graph, the analysis proceeds by computing the maximum flow between the inputs and the outputs of the program, which with a suitable classification of the program can be translated to the size of possible information leaks. The analysis is dynamic and thus not conservative, but it allows to combine an arbitrary number of program runs. This analysis is an important step towards a practical employment of IFC in the software engineering process, as knowing the size of a leak permits to decide whether the leak is acceptable. It could also be used to determine possible points for declassifications.

6. The Joana Framework

It follows a brief overview of the Joana framework. Several aspects have already been described in Hammer's PhD thesis [52] and in a conjoint publication [45], hence we focus on our share of the framework.

Figure 6.1 provides a coarse overview of Joana's architecture. The rectangle labeled as 'Eclipse-Plugin' contains those parts that are accessible via our plugin [45] for Eclipse, a popular software development environment[1]. All white colored parts can be used without Eclipse, only the three graphical user interfaces depend on it.

The core of the system is the CSDG data structure, which is based on the public graph library JGraphT[2]. The CSDG decouples our slicing, chopping and IFC algorithms from the algorithms generating the dependence graphs. Joana currently supports CSDGs for Java and C programs. The 'JSDG + WALA' module creates CSDGs for Java programs and is mainly maintained by Jürgen Graf [49]. It builds on the IBM T.J. Watson Libraries for Analysis (WALA) [5] and contains our MHP analysis. The 'CodeSurfer + Plugin' module consists of a plugin for CodeSurfer [35], a commercial SDG generator for C, which converts CodeSurfer-SDGs in our CSDG format. The plugin was developed by Bernd Nürnberger in his master thesis [108]. The 'Slicing' module contains a wide range of slicing and chopping algorithms, including all algorithms presented in chapters 2, 3 and 4. The 'Graphviewer' is a stand-alone tool visualizing our CSDGs, developed and maintained by several student assistants. The 'IFC' module contains our IFC analyses. At the time being, it consists of Hammer et al's IFC algorithm [58], our algorithm and an improved IFC algorithm for sequential programs developed by Jens Krinke, which has not been published yet.

Figure 6.2 shows the graphical user interface of our IFC analysis. The code in part (A) is an excerpt of the PasswordCheck program investigated in section 5.7.4. Part (B) shows the classification of the program, which is the one used in section 5.7.4 (the choice of the security level is done in a pop-up menu and not shown here). The undermost annotation, which declassifies the result of the password check, is deactivated. Therefore, the analysis reports in part (C) the probabilistic order channel and the illicit flow from the passwords array to the output of the result of the password check. In the source code, the chop from the source to the sink of that leak is highlighted, depicting the paths over which the information flows to the sink. Part (D) shows several statistics about the leak, which aid the user in estimating its severeness.

[1] http://www.eclipse.org/
[2] http://www.jgrapht.org/

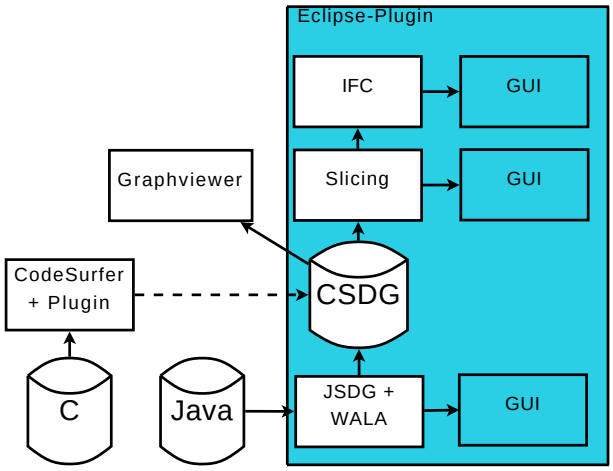


Figure 6.1.: A coarse overview of the architecture of ValSoft/Joana.

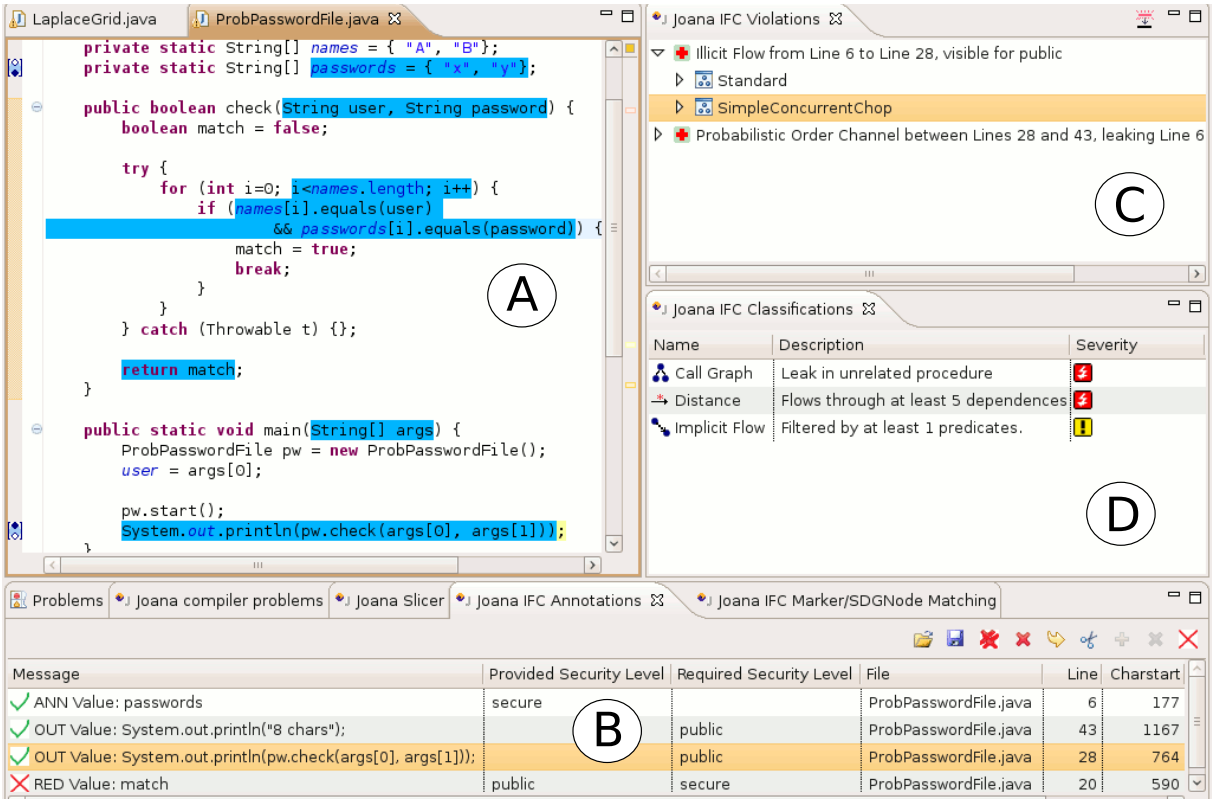


Figure 6.2.: An overview of the graphical user interface of our IFC analysis.

These statistics report how the leaks are positioned in the call graph, the number of dependences forming the shortest path from the source to the sink and the smallest number of predicates that the information has to flow through to reach the sink.

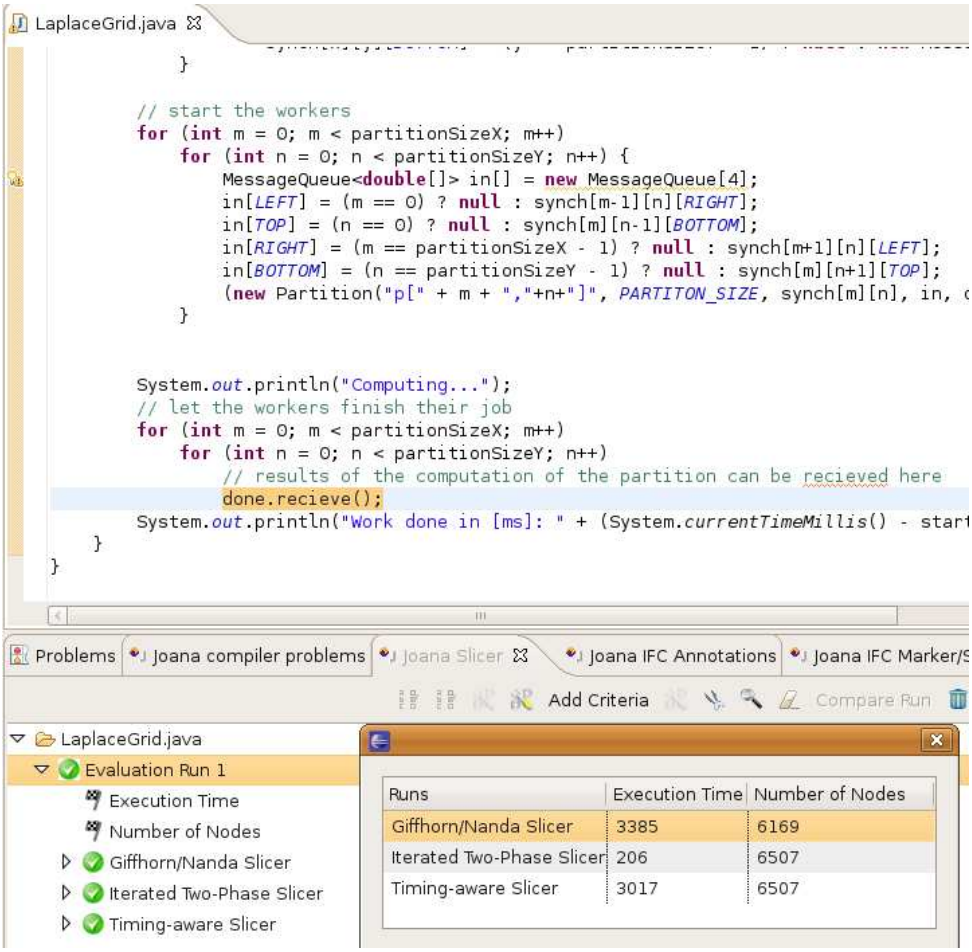


Figure 6.3.: An overview of the graphical user interface of our slicing and chopping algorithms.

Figure 6.3 shows the graphical user interface by which our slicing and chopping algorithms can be accessed. The code shown is an excerpt of the LaplaceGrid program used in our evaluations. The picture shows the comparison of the slices for statement `done.receive()` computed by three algorithms, our version of Nanda's slicer, the iterated two-phase slicer and the timing-aware slicer. Since slicing and chopping algorithms are not an end in themselves, this user interface is intended for demonstration purposes and has no real application.

6.1. Related Work

There exist only a few complete realizations of slicing-based program analysis – implementations that provide everything from dependence graph computation over slicing algorithms to applications, preferably accessible via a graphical user interface.

The CodeSurfer [14, 15, 35], a program analysis framework for C/C++ programs, is regarded as the most advanced and mature realization. It is developed and maintained by Gramma-

Tech[3] and the only known commercialized SDG generator. The tool is able to compute SDGs for programs with 300.000 LOC if summary edges are turned off and an imprecise points-to analysis is used. With summary edges enabled and a good points-to analysis programs it can still analyze programs with 100.000 – 150.000 LOC[4]. The tool is shipped with the standard slicing and chopping algorithms for sequential programs and provides an interface over which queries on the SDG can be freely formulated via scheme scripts, for which reason CodeSurfer is extremely extensible. It is therefore used as a basis by many research groups, an overview of corresponding publications can be found on GrammaTech's homepage.

The SAnToS Laboratory at the Kansas State University developed Indus/Kaveri [66, 118], a general-purpose slicing framework for Java programs which particularly focuses on a configurable CSDG generator. The user is able to choose among the standard intra- and interprocedural dependences as well as among interference-, synchronization- and ready dependence. However, summary edges are not supported. Context-sensitive slicing is done via a k-limited call-string-based traversal similar to the IPDG slicer, and due to the k-limiting some context-sensitivity is lost. Timing-sensitive slicing is not supported, but the tool also provides a context-restricted slicer. Indus/Kaveri is realized as a plugin for the Eclipse framework and offers a nice and user-friendly GUI.

[3] `http://www.grammatech.com/`

[4] This was stated by Paul Anderson, the Vice President of Engineering at GrammaTech, in his keynote at the SCAM conference in 2008.

7. Conclusion

This thesis investigated slicing of concurrent programs with shared memory and threads and its application to information flow control. The developed algorithms have been integrated into the program analysis framework ValSoft/Joana, where they can be used to analyze concurrent Java programs. They are capable of all features of Java bytecode, particularly of dynamic thread creation inside loops and recursive procedures, and can be adapted to analyze likewise languages.

We conducted the first realistic evaluation and comparison of timing-sensitive slicing algorithms. The timing-sensitive algorithms are significant achievements in slicing technology, being the only algorithms for pruning timing-insensitive paths in programs written in contemporary languages. Nanda developed the restrictive state tuple optimization, which is essential for applying these algorithms in practice. Unfortunately, their algorithm applies that optimization at one point where it might prune valid dependences. We detected and explained that problem and presented a correction. We have further applied several optimizations to the algorithms, which provide a significant speedup and improved precision, and have extended the algorithms to handle dynamic thread generation inside loops and recursion, which makes them capable of full Java bytecode. Among the investigated timing-sensitive algorithms, our optimized version of Nanda's algorithm performed best.

We were the first to investigate chopping of concurrent programs and transferred the ideas underlying timing-sensitive slicing to chopping. The resulting timing-sensitive chopping algorithm is a powerful technique able to drastically reduce the chop sizes. Due to its high runtime costs, we developed six different chopping algorithms for concurrent programs, ranging from imprecise to context-sensitive to timing-sensitive. This enables a user to choose the chopping algorithm whose ratio between precision and runtime costs suits best.

Since the chosen approach to timing-sensitive program analysis has a worst-case runtime complexity exponential to the number of threads of the analyzed program, algorithms based thereon may run into scalability problems. It seems that the high costs require a selective employment of timing-sensitive slicing and chopping. A pragmatic approach would employ a less precise algorithm first and apply the timing-sensitive algorithm to refine the results of chosen cases. That way, one can greatly reduce analysis overhead and still benefit from the precision of timing-sensitive program analysis.

Using our experiences with analyzing concurrent programs, we developed a technique for detecting probabilistic channels in concurrent programs. It can be combined with Hammer et

al.'s algorithm for information flow control for sequential programs, resulting in an analysis which is able to verify absence of illicit explicit and implicit flow and of probabilistic channels in concurrent programs and permits declassification. The resulting information flow control technique is based on observational determinism and a new, scheduler-independent security property, which is capable of competing with existing ones in terms of permissiveness and security guarantees. The whole technique has been integrated into ValSoft/Joana and is, to the best of our knowledge, the first existing realization of information flow control for concurrent programs written in a real-world language.

A. The Java Programs Of Our Case Study

A.1. Program 'SmithVolpano'

The following program is an example from Smith and Volpano [136], converted to Java.

```
class Alpha extends Thread {
  public void run() {
    while (mask != 0) {
      while (trigger0 == 0) ; /* busy wait */
      result = result | mask;
      trigger0 = 0;
      maintrigger++;
      if (maintrigger == 1) trigger1 = 1;
    }
  }
}
class Beta extends Thread {
  public void run() {
    while (mask != 0) {
      while (trigger1 == 0) ; /* busy wait */
      result = result & ~mask;
      trigger1 = 0;
      maintrigger++;
      if (maintrigger == 1) trigger0 = 1;
    }
  }
}
class Gamma extends Thread {
  public void run() {
    while (mask != 0) {
      maintrigger = 0;
      if ((PIN & mask) == 0) trigger0 = 1;
      else trigger1 = 1;
      while (maintrigger < 2) ; /* busy wait */
      mask = mask / 2;
    }
  }
}
```

```
class SmithVolpano {
  static volatile int maintrigger, trigger0, trigger1 = 0, PIN, result = 0;
  static volatile int mask = 2048; // a power of 2

  public static void main(String[] args) throws Exception {
    PIN = Integer.parseInt(args[0]);
    Thread a=new Alpha(); Thread b=new Beta(); Thread g=new Gamma();
    g.start(); a.start(); b.start();  // start all threads
    g.join(); a.join(); b.join();     // join all threads
    System.out.println(result);
  }
}
```

A.2. Program ‘Mantel’

The following program is an example from Mantel et al [95], converted to Java. For brevity, some methods are not shown.

```
class Mantel {
  // to allow mutual access, threads are global variables
  static Portfolio p = new Portfolio();
  static EuroStoxx50 e = new EuroStoxx50();
  static Statistics s = new Statistics();
  static Output o = new Output();

  static BufferedWriter nwOutBuf =
      new BufferedWriter(new OutputStreamWriter(System.out));
  static BufferedReader nwInBuf =
      new BufferedReader(new InputStreamReader(System.in));
  static String[] output = new String[50];

  public static void main(String[] args) throws Exception {
    // get portfolio and eurostoxx50
    p.start(); e.start();
    p.join();  e.join();
    // compute statistics and generate output
    s.start(); o.start();
    s.join();  o.join();
    // display output
    stTabPrint("No.\t␣|␣Name\t␣|␣Price\t␣|␣Profit");
    for (int n = 0; n < 50; n++)
      stTabPrint(output[n]);
    // show commercials
    stTabPrint(e.coShort+"Press␣#␣to␣get␣more␣information");
    char key = (char)System.in.read();
    if (key == '#') {
      System.out.println(e.coFull);
      nwOutBuf.append("shownComm:"+e.coOld);
      nwOutBuf.flush();                    // public output
    }
  }
}
```

```
class Portfolio extends Thread {
  int[] esOldPrices, pfNums;
  String[] pfNames; String pfTabPrint;

  public void run() {
    pfNames = getPFNames();          // secret input
    pfNums = getPFNums();            // secret input
    for (int i = 0; i < pfNames.length; i++)
      pfTabPrint += pfNames[i] + "|" + pfNums[i];
  }

  int locPF(String name) {
    for (int i = 0; i < pfNames.length; i++)
      if (pfNames[i].equals(name)) return i;
    return -1;
  }
}

class EuroStoxx50 extends Thread {
  String[] esName = new String[50];
  int[] esPrice = new int[50];
  String coShort;
  String coFull;
  String coOld;

  public void run() {
    try {
      nwOutBuf.append("getES50");
      nwOutBuf.flush();              // public output
      String nwIn = nwInBuf.readLine();
      String[] strArr = nwIn.split(":");
      for (int j = 0; j < 50; j++) {
        esName[j] = strArr[2*j];
        esPrice[j] = Integer.parseInt(strArr[2*j+1]);
      }
      // commercials
      coShort = strArr[100];
      coFull = strArr[101];
      coOld = strArr[102];
    } catch (IOException ex) {}
  }
}
```

```
class Statistics extends Thread {
  int[] st = new int[50];
  volatile int k = 0;

  public void run() {
    k = 0;
    while (k < 50) {
      int ipf = p.locPF(e.esName[k]);
      if (ipf > 0)
        set(k, (p.esOldPrices[k] - e.esPrice[k]) * p.pfNums[ipf]);
      else
        set(k, 0);
      k++;
    }
  }
  public synchronized void set(int k, int value) {
    st[k] = value;
  }
  public synchronized int get(int k) {
    return st[k];
  }
}

class Output extends Thread {
  public void run() {
    for (int m = 0; m < 50; m++) {
      while (s.k <= m) ;  /* busy wait */
      output[m] = m+"|"+e.esName[m]+"|" + e.esPrice[m]+"|"+s.get(m);
    }
  }
}
```

A.3. Program 'PasswordCheck'

A malicious password checker.

```
public class PasswordCheck extends Thread {
  private String user;
  private String[] names = { "A", "B"};
  private String[] passwords = { "x", "y"};

  public boolean check(String usr, String pwd) {
    boolean match = false;
    try {
      for (int i=0; i<names.length; i++) {
        if (names[i].equals(usr) && passwords[i].equals(pwd)) {
          match = true;
          break;
        }
      }
    }
    catch (Exception e) {};
    return match;
  }

  public static void main(String[] args) {
    ProbPasswordFile pw = new ProbPasswordFile();
    pw.user = args[0];
    pw.start();       // the malicious thread is started
    System.out.println(pw.check(args[0], args[1]));
  }

  public void run() {
    boolean match = false;
    try {
      for (int i=0; i<names.length; i++) {
        if (names[i].equals(user) && "ABCDEFGH".equals("ABCDEFGH")) {
          match = true;
          break;
        }
      }
    }
    catch (Exception e) {};
    System.out.println("8␣chars");
  }
}
```

B. Curriculum Vitae

In german

Name:	Dennis Giffhorn
Geboren am:	29.08.1978 in Wolfsburg
Staatsangehörigkeit:	deutsch

Bildungsweg

Juni 1997:	Abitur am Ratsgymnasium Wolfsburg
März 2005:	Diplom in Informatik mit Nebenfach Psychologie an der Carl-von-Ossietzky-Universität Oldenburg

Beruflicher Werdegang

01.06.2005 - 31.03.2008:	Wissenschaftl. Mitarbeiter am Lehrstuhl Softwaresysteme von Prof. Snelting an der Universität Passau
01.04.2008 - 31.05.2011:	Wissenschaftl. Mitarbeiter am Lehrstuhl Programmierparadigmen von Prof. Snelting am Karlsruher Institut für Technologie

In english

Name:	Dennis Giffhorn
Born:	29.08.1978 in Wolfsburg
Citizenship :	german

Education

June 1997:	Abitur (academic high school diploma) at the Ratsgymnasium Wolfsburg
March 2005:	Diplom (MSc) of Computer Science at the Carl-von-Ossietzky-Universität Oldenburg

Work Experience

01.06.2005 - 31.03.2008:	PhD student at the Software Systems group of Prof. Snelting at the University of Passau
01.04.2008 - 31.05.2011:	PhD student at the Programming Paradigms group of Prof. Snelting at the Karlsruhe Institute of Technology

C. List of Figures

D. List of Tables

E. Bibliography

[1] The Bandera tool set. http://bandera.projects.cis.ksu.edu/.

[2] Java Card Technology.
http://www.oracle.com/technetwork/java/javacard/overview/index.html.

[3] The Java Grande forum multi-threaded benchmarks.
http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html.

[4] The Java Mobile Edition for mobile devices. http://java.sun.com/javame/index.jsp.

[5] The T.J. Watson Libraries for Analysis (WALA).
http://wala.sourceforge.net/wiki/index.php/Main_Page.

[6] ValSoft/Joana: Information flow control in program dependence graphs.
http://pp.info.uni-karlsruhe.de/project.php?id=30.

[7] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM.

[8] J. Agat. Transforming out timing leaks. In *27th. ACM SIGPLAN Symposium on Princ. of Prog. Lang.*, pages 40–53, 2000.

[9] G. Agrawal and L. Guo. Evaluating explicitly context-sensitive program slicing. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 6–12, New York, NY, USA, 2001. ACM.

[10] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[11] H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Notices*, 25(6):246–256, June 1990.

[12] M. Allen and S. Horwitz. Slicing Java programs that throw and catch exceptions. *SIGPLAN Notices*, 38:44–54, June 2003.

[13] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[14] P. Anderson. 90% perspiration: Engineering static analysis techniques for industrial applications. In *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 3–12, 2008.

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

[16] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer Berlin / Heidelberg, 2008.

[17] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41, New York, NY, USA, 1979. ACM.

[18] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer Berlin / Heidelberg, 2006.

[19] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *ESORICS*, pages 2–18, 2007.

[20] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, 1993.

[21] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[22] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 04 1977.

[23] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23:498–516, August 1997.

[24] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program.*, 62(3):228–252, 2006.

[25] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 44, Washington, DC, USA, 2003. IEEE Computer Society.

[26] D. Binkley and M. Harman. A survey of empirical results on program slicing. In *Advances in Computers, 62:105-178*, pages 105–178, 2004.

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

[28] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[29] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595 – 607, 1998.

[30] H. Chen and P. Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 31–40, New York, NY, USA, 2007. ACM.

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

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

[33] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. *Advances in Parallel and Distributed Computing Conference*, 0:370, 1997.

[34] D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. *Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages (QAPL 2004)*, 112:149 – 166, January 2005.

[35] The CodeSurfer code browser for C/C++. http://www.grammatech.com/.

[36] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[37] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[38] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

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

[40] I. Forgács and T. Gyimóthy. An efficient interprocedural slicing method for large programs. In *SEKE'97: Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, pages 279–287. Springer-Verlag, 1997.

[41] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. CONSIT: A fully automated conditioned program slicer. *Softw. Pract. Exper.*, 34:15–46, January 2004.

[42] D. Giffhorn. Chopping concurrent programs. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 13–22, September 2009.

[43] D. Giffhorn. Advanced chopping of sequential and concurrent programs. *Software Quality Journal*, 19(2):239–294, 2011.

[44] D. Giffhorn and C. Hammer. An evaluation of slicing algorithms for concurrent programs. In *7th IEEE Int. Work. Conf. on Source Code Analysis and Manipulation*, pages 17–26, 2007.

[45] D. Giffhorn and C. Hammer. Precise analysis of Java programs using Joana (tool demonstration). In *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.

[46] D. Giffhorn and C. Hammer. Precise slicing of concurrent programs - An evaluation of static slicing algorithms for concurrent programs. *Journal of Automated Software Engineering*, 16(2):197–234, June 2009.

[47] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[48] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* AddisonWesley Prof., 3 edition, 2005.

[49] J. Graf. Speeding up context-, object- and field-sensitive SDG generation. In *10th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 105–114, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[50] J. W. Gray. Probabilistic interference. *Security and Privacy, IEEE Symposium on*, 0:170, 1990.

[51] J. W. Gray. Toward a mathematical foundation for information flow security. *Security and Privacy, IEEE Symposium on*, 0:21, 1991.

[52] C. Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), July 2009. ISBN 978-3-86644-398-3.

[53] C. Hammer. Experiences with PDG-based IFC. In *International Symposium on Engineering Secure Software and Systems (ESSoS'10)*, volume 5965 of *LNCS*, pages 44–60. Springer-Verlag, February 2010.

[54] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, pages 136–145, 2006.

[55] C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, pages 87–96, 2006.

[56] C. Hammer, R. Schaade, and G. Snelting. Static path conditions for Java. In *Proceedings of the 3rd Workshop on Programming Languages and Analysis for Security*, pages 55–66. ACM, June 2008.

[57] C. Hammer and G. Snelting. An improved slicer for Java. In *Proceedings of the 5th ACM SIGPLAN- SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22. ACM Press, 2004.

[58] 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*, 2009. Supersedes their publications at ISSSE 2006 and ISoLA 2006.

[59] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, pages 70–, Washington, DC, USA, 1997. IEEE Computer Society.

[60] M. Harman, L. Hu, M. Munro, X. Zhang, D. Binkley, S. Danicic, M. Daoudi, and L. Ouarbya. Syntax-directed amorphous slicing. *Automated Software Engg.*, 11:27–61, January 2004.

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

[62] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3:11–16, April 1992.

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

[64] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterization of observational determinism. In *19th IEEE Computer Security Foundations Workshop*, July 2006.

[65] The FLEX compiler infrastructure. http://flex.cscott.net/Harpoon/.

[66] The Indus slicer for Java. http://indus.projects.cis.ksu.edu/.

[67] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, 1994.

[68] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 38–47, New York, NY, USA, 2005. ACM.

[69] M. Kamkar, N. Shahmehri, and P. Fritzson. Bug localization by algorithmic debugging and program slicing. In *PLILP '90: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, pages 60–74, London, UK, 1990. Springer-Verlag.

[70] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[71] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[72] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29:155–163, October 1988.

[73] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, Montreal, Canada, June 1998.

[74] J. Krinke. Evaluating context-sensitive slicing and chopping. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 22, Washington, DC, USA, 2002. IEEE Computer Society.

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

[76] J. Krinke. Barrier slicing and chopping. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, 2003.

[77] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European software engineering conference / 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 178–187. ACM Press, 2003.

[78] J. Krinke. Context-sensitivity matters, but context does not. In *4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 29–35, Washington, DC, USA, 2004. IEEE Computer Society.

[79] J. Krinke. Effects of context on program slicing. *J. Syst. Softw.*, 79(9):1249–1260, 2006.

[80] S. Kuninobu, Y. Takata, H. Seki, and K. Inoue. An information flow analysis of recursive programs based on a lattice model of security classes. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 87(9):48–61, 2004.

[81] P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *Proceedings of the 15th international symposium on Static Analysis*, SAS '08, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag.

[82] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[83] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16:613–615, October 1973.

[84] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 94:1–28, September 1991.

[85] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[86] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1:121–141, January 1979.

[87] L. Li and C. Verbrugge. A practical MHP information analysis for concurrent Java programs. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, LNCS. Springer Verlag, Sept. 2004.

[88] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 358, Washington, DC, USA, 1998. IEEE Computer Society.

[89] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[90] J. R. Lyle, D. R. Wallace, J. R. Graham, K. B. Gallagher, J. P. Poole, and D. W. Binkley. Unravel: A CASE tool to assist evaluation of high integrity software volume 1: Requirements and design. Technical report, 1995.

[91] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *2nd International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[92] P. Malacaria. Assessing security threats of looping constructs. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 225–235, 2007.

[93] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. *SIGPLAN Not.*, 40:378–391, January 2005.

[94] H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 116–133, Berlin, Heidelberg, 2010. Springer-Verlag.

[95] H. Mantel, H. Sudbrock, and T. Kraußer. Combining different proof techniques for verifying information flow security. In *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin / Heidelberg, 2007.

[96] S. McCamant and M. Ernst. Quantitative information flow as network flow capacity. In *PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205, 2008.

[97] J. McLean. Security models and information flow. *IEEE Symposium on Security and Privacy*, 0:180, 1990.

[98] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[99] B. mo Chang and J. deok Choi. Thread-sensitive points-to analysis for multithreaded Java programs. In *In ISCIS'04: Proceedings of the 19th International Symposium on Computer and Information Sciences*, pages 945–954, 2004.

[100] D. P. Mohapatra, R. Mall, and R. Kumar. Computing dynamic slices of concurrent object-oriented programs. *Inf. Softw. Technol.*, 47:805–817, September 2005.

[101] M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311:325–388, January 2004.

[102] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 647–656, New York, NY, USA, 2001. ACM.

[103] A. C. Myers, L. Zheng, S. Zdancevic, S. Chong, and N. Nystrom. Jif: Java information flow. http://ww.cs.cornell.edu/jif.

[104] M. G. Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, Bombay, 2001.

[105] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 180–190, New York, NY, USA, 2000. ACM.

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

[107] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 338–354, London, UK, 1999. Springer-Verlag.

[108] B. Nürnberger. Slicing und Pfadbedingungen mit CodeSurfer. Master's thesis, Universität Passau, 2008.

[109] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[110] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, 1990.

[111] X. Qi and B. Xu. An approach to slicing concurrent Ada programs based on program reachability graphs. 2008.

[112] X. Qi, X. Zhou, X. Xu, and Y. Zhang. Slicing concurrent programs based on program reachability graphs. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 248–253, Washington, DC, USA, 2010. IEEE Computer Society.

[113] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16:1467–1471, September 1994.

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

[115] V. Ranganath, T. Amtoft, A. Banerjee, M. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures, 2004.

[116] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent Java programs. In *Proceedings of 13th International Conference on Compiler Construction (CC'04)*, volume 2985 of *LNCS*, pages 39–56, Mar. 2004.

[117] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent Java programs (extended version), 2006.

[118] V. P. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9:489–504, October 2007.

[119] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, 1994.

[120] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52. ACM Press, 1995.

[121] A. W. Roscoe. CSP and determinism in security modelling. In *SP '95: Proceedings of the 1995 IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 1995.

[122] P. Rousseau. A new approach for concurrent program slicing. *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, 4229:228–242, 2006.

[123] E. Ruf. Effective synchronization removal for Java. *SIGPLAN Not.*, 35(5):208–218, 2000.

[124] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 177–189, Washington, DC, USA, 2006. IEEE Computer Society.

[125] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proceedings of the 6th international Andrei Ershov memorial conference on Perspectives of systems informatics*, PSI'06, pages 474–480, Berlin, Heidelberg, 2007. Springer-Verlag.

[126] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[127] A. Sabelfeld and A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *In Proc. Andrei Ershov International Conference on Perspectives of System Informatics, volume 2890 of LNCS*, pages 260–273. Springer-Verlag, 2003.

[128] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW '00: Proceedings of the 13th IEEE workshop on Computer Security Foundations*. IEEE Computer Society, 2000.

[129] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[130] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.

[131] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, 1981.

[132] V. Simonet. The Flow Caml system. http://cristal.inria.fr/ simonet/soft/flowcaml.

[133] G. Smith. A new type system for secure information flow. In *CSFW '01: Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 115. IEEE Computer Society, 2001.

[134] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–13, 2003.

[135] G. Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *J. Comput. Secur.*, 14(6):591–623, 2006.

[136] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364. ACM Press, 1998.

[137] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis*, pages 332–348. Springer-Verlag London, UK, September 1996.

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

[139] G. Snelting and D. Wasserrab. A correctness proof for the volpano/smith security typing system. In *The Archive of Formal Proofs*. http://afp.sf.net/entries/VolpanoSmith.shtml, September 2008. Formal proof development.

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

[141] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[142] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical Report MIP-0011, Univerity Passau, November 2000.

[143] B. D. Sutter, L. Van Put, and K. D. Bosschere. A practical interprocedural dominance algorithm. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[144] R. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. In *Acta Informatica*, volume 19, pages 57–84, 1983.

[145] T. Terauchi. A type system for observational determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300, Washington, DC, USA, 2008. IEEE Computer Society.

[146] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 30:27:1–27:30, September 2008.

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

[148] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *ICSE '97: Proceedings of the*

19th international conference on Software engineering, pages 433–443, New York, NY, USA, 1997. ACM.

[149] A. Tsitovich, N. Sharygina, C. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011.

[150] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.*, 7(2-3):231–253, 1999.

[151] N. Walkinshaw. *Partitioning Object-Oriented Software for Inspections.* PhD thesis, The University of Strathclyde., 2006.

[152] N. Walkinshaw, M. Roper, M. Wood, and N. W. M. Roper. The Java System Dependence Graph. In *In Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 5–5, 2003.

[153] D. Wasserrab and D. Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th International Verification Workshop - VERIFY-2010*, 2010.

[154] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterference and its modular proof. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*, pages 31–44. ACM, June 2009.

[155] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

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

[157] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 185–195, 2007.

[158] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[159] R. Yokomori, F. Ohata, Y. Takata, H. Seki, and K. Inoue. An information-leak analysis system based on program slicing. *Information and Software Technology*, 44(15):903 – 910, 2002.

[160] S. Zdancewic and A. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, 2003.

[161] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 94–106, New York, NY, USA, 2004. ACM.

[162] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

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

[164] 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. IEEE Computer Society Press, 1996.

F. Index