



Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

**DATAFLOW ANALYSIS FOR SOFTWARE
PRODUCT LINES**

Táris Wanderley Tolêdo

DISSERTAÇÃO DE MESTRADO

Recife - PE

21 de fevereiro de 2013

Universidade Federal de Pernambuco
Centro de Informática

Táris Wanderley Tolêdo

DATAFLOW ANALYSIS FOR SOFTWARE PRODUCT LINES

*Trabalho apresentado ao Programa de Pós-graduação em
Ciência da Computação do Centro de Informática da Uni-
versidade Federal de Pernambuco como requisito parcial
para obtenção do grau de Mestre em Ciência da Com-
putação.*

Orientador: *Paulo Henrique Monteiro Borba*

Recife - PE
21 de fevereiro de 2013

ACKNOWLEDGEMENTS

À minha família pelo apoio nos momentos em que precisei e pela compreensão nos momentos em que precisou e não pude estar lá para apoiá-la.

Ao professor Paulo Borba pela indiscutível competência com que orienta seus alunos.

A todos os amigos que contribuíram para a conclusão deste trabalho: Henrique Rebêlo, Jean Melo, Jefferson Almeida, Laís Neves, Leopoldo Teixeira, Paola Accioly, Rodrigo Andrade e em especial Márcio Ribeiro, com quem pude dividir tantas conquistas.

Agradeço ainda aos professores Claus Brabrand e Eric Bodden pela brilhante forma de construir e compartilhar conhecimento.

RESUMO

Linhas de produto de software (LPS) são frequentemente construídas com o uso de diretivas de pré-processador como `#ifdefs`, por exemplo, para implementar a variabilidade das *features*. Estas diretivas impedem o uso de técnicas de análise de fluxo de dados e forçam os desenvolvedores a gerar de forma explícita todos os produtos da linha a fim de analisá-los individualmente. Devido à natureza combinatorial do número de produtos em uma LPS, isto rapidamente se torna impraticável. Neste trabalho, demonstra-se como dois *frameworks* de análise de fluxo de dados podem ser transformados para dar suporte à *análise de fluxo de dados sensível a features* sem a necessidade de utilizar-se de força-bruta e gerar todos os produtos da linha explicitamente. Especificamente, descreve-se quatro maneiras diferentes de se implementar análises de fluxo de dados intraprocedural sensíveis a *features* dentro do framework de Killdall para análise estática; e também como análises interprocedurais pode ser transformadas em análises sensíveis a *features* dentro do *framework* IFDS. O desempenho das técnicas propostas é avaliado através de experimentos que envolvem aplicar análises sensíveis à *features* em quatro LPS. Através destes experimentos, apresenta-se evidência de que é possível conseguir um ganho significativo no desempenho das análises ao utilizar as técnicas propostas.

Palavras-chave: Linhas de Produtos de Software, Pré-processadores, Análise de fluxo de dados.

ABSTRACT

Software product lines (SPLs) that use preprocessor directives such as `#ifdefs` to define code-level variability suffer from the lack of techniques of dataflow analyses. Developers are forced to explicitly generate all products from the SPL in order to apply dataflow analyses. That quickly becomes prohibitive as the number of possible products increases due to the combinatorial nature of SPLs. In this work, we describe how two different dataflow analysis frameworks, Kildall’s and IFDS, are lifted to support *feature-sensitive dataflow analyses* without using brute-force to explicitly generate all possible products in the product line. Specifically, we describe four different ways of implementing intraprocedural analyses within Kildall’s framework that are feature-sensitive; and how interprocedural dataflow analyses can also be lifted to feature-sensitivity within the IFDS framework. We experimented with the performance of the proposed techniques by applying feature-sensitive dataflow analyses to four different SPLs. We found that it is possible to significantly speed up intraprocedural and interprocedural analyses, the latter by several orders of magnitude.

Keywords: Software Product Lines, Preprocessors, Dataflow analysis

CONTENTS

Chapter 1—Introduction	1
1.1 Outline	2
Chapter 2—Background	3
2.1 Software Product Lines	3
2.2 Preprocessors	4
2.3 Intraprocedural Dataflow Analysis	6
2.3.1 Control flow graph	6
2.3.2 Lattice	6
2.3.3 Transfer functions	7
2.3.4 Reaching definitions	8
2.3.5 Intraprocedural dataflow analysis with Soot	9
2.4 Interprocedural dataflow analysis with IFDS	12
Chapter 3—Feature-sensitive intraprocedural dataflow analysis	17
3.1 Motivation	17
3.2 Consecutive	18
3.2.1 Implementation	22
3.3 Simultaneous	26
3.3.1 Implementation	27
3.4 Shared simultaneous	30
3.4.1 Implementation	30
3.5 Reversed shared simultaneous	35
3.5.1 Implementation	35
3.6 Evaluation	40
3.6.1 Study settings	40

3.6.2	Previous experiments	41
3.6.3	Revisiting the experiments	43
3.6.4	Results discussion	44
3.6.5	No JIT results discussion	48
3.6.5.1	Benchmark 1: GPL	52
3.6.5.2	Benchmark 2: MobileMedia08	55
3.6.5.3	Benchmark 3: Lampiro	57
3.6.5.4	Benchmark 4: BerkeleyDB	58
3.6.6	Synthetic benchmarks	61
3.6.7	Evaluation summary	67
3.7	Threats to validity	67
Chapter 4	Feature-sensitive interprocedural dataflow analysis for SPL	69
4.1	Motivation	69
4.2	Lifting IFDS-based analyses	70
4.3	Evaluation	74
4.3.1	Study settings	74
4.3.2	Results discussion	76
4.4	Threats to validity	78
Chapter 5	Related Work	81
Chapter 6	Concluding Remarks	83
6.1	Summary of contributions	84
6.2	Limitations	84
6.3	Future work	85

LIST OF FIGURES

2.1	A simple feature diagram.	4
2.2	A toy program and its CFG.	7
2.3	A Hasse diagram representing a lattice with all the possible assignments in our toy program.	7
2.4	The effect of applying a transfer function of an assignment to the \perp lattice value.	8
2.5	The assignment $x = 0$ does not reach the skip statement. All paths from the assignment to the skip statement contains new assignments to x	8
2.6	An interprocedural control flow graph.	12
2.7	Transfer function encoding in IFDS.	13
2.8	An exploded super-graph showcasing the taint analysis.	15
3.1	Partial CFG with one node instrumented.	19
3.2	A statement with a nested complex conditional expression.	19
3.3	Result of the reaching definitions analysis using the consecutive approach. .	21
3.4	Result of the reaching definitions analysis using the simultaneous approach.	27
3.5	Result of the reaching definitions analysis using the shared approach. . . .	31
3.6	Result of the reaching definitions analysis using the reversed shared approach.	36
3.7	Histogram of the number of configurations of methods in the GPL benchmark.	41
3.8	Histogram of the number of configurations of methods in the MobileMedia08 benchmark.	42
3.9	Histogram of the number of configurations of methods in the Lampiro benchmark.	42
3.10	Histogram of the number of configurations of methods in the BerkeleyDB benchmark.	43
3.11	Median (over the 10 runs) of the sums of the analysis time of methods in each benchmark	46

3.12	Time measurements of the consecutive analysis on one method in the GPL benchmark. This is the method with the largest number of configurations in GPL: 106.	47
3.13	Time measurements of the consecutive analysis on one method in the GPL benchmark with the JIT compiler disabled. This is the method with the largest number of configurations in GPL: 106.	49
3.14	Time measurements of all approaches on a method with 2 configurations and 5 statements. Variations are easily spotted in the measurements of all approaches.	49
3.15	Sum of medians of analysis time in the benchmarks, with the JIT compiler disabled.	50
3.16	Number of times each approach performed the fastest on methods with 2 configurations in the GPL benchmark, averaged over the 10 runs.	52
3.17	Number of times each approach performed the fastest on methods with 4 configurations in the GPL benchmark, averaged over the 10 runs.	54
3.18	Number of times each approach performed the fastest on methods with more than 4 configurations in the GPL benchmark, averaged over the 10 runs.	54
3.19	Number of times each approach performed the fastest on methods with 2 configurations in the MobileMedia08 benchmark, averaged over the 10 runs.	55
3.20	Number of times each approach performed the fastest on methods with 4 configurations in the MobileMedia08 benchmark, averaged over the 10 runs.	56
3.21	Number of times each approach performed the fastest on methods with more than 4 configurations in the MobileMedia08 benchmark, averaged over the 10 runs.	58
3.22	Number of times each approach performed the fastest on methods with 2 configurations in the Lampiro benchmark, averaged over the 10 runs.	59
3.23	Number of times each approach performed the fastest on methods with 2 configurations in the BerkeleyDB benchmark, averaged over the 10 runs.	59
3.24	Number of times each approach performed the fastest on methods with 4 configurations in the BerkeleyDB benchmark, averaged over the 10 runs.	61
3.25	Number of times each approach performed the fastest on methods with more than 4 configurations in the BerkeleyDB benchmark, averaged over the 10 runs.	62

3.26	Methods in the synthetic benchmark.	62
3.27	A beanplot with time measurements on method m.	64
3.28	Medians over the 50 runs of each method in the synthetic benchmark. . .	65
3.29	Methods in the modified synthetic benchmark with a higher number of assignments.	66
3.30	Medians of each method in the modified synthetic benchmark.	66
4.1	All cases of transfer function lifting by SPL^{LIFT} , from [17, Figure 4, page 4].	72
4.2	Application of the lifted flow functions of the taint analysis to the example in Listing 4.1. An insecure flow of information is highlighted in thick, red arrows. Figure taken from [17, Figure 5].	74

LIST OF TABLES

3.1	A summary of key characteristics in the benchmarks used in the experiments.	41
3.2	A demonstration of the measurements taken by the experiments.	44
3.3	For methods with 4 configurations, p-values output by the Wilcoxon test for the GPL benchmark on 3 of the 10 executions.	45
4.1	Number of features, total number of configurations and valid configurations reached by the solver on each benchmark. The highlighted cells indicate discrepancies from the original experiment performed by Bodden et al. .	76
4.2	Performance of the SPL^{LIFT} approach vs. the naive, brute-force approach. Highlighted values are estimates based on partial progress.	77
4.3	The effects in time performance of regarding/ignoring the feature model in each benchmark.	78

CHAPTER 1

INTRODUCTION

A software product line (SPL) describes a set of related products which are based on the same reusable code. Variations in this code, called *features*, are pieces of code that build on top of the common code base to provide different functionalities to the products.

Many important applications implement their variations using conditional compilation constructs, like `#ifdefs`. Conditional compilation has the property of being independent of the underlying language. Because of this, developers can mix common, optional and even conflicting behavior in the same code asset. In order to generate individual products, the code assets must be preprocessed before compilation can occur, and only then developers can apply their static analyses techniques in search for bugs, for example.

Dataflow analysis is a form of static analysis used to perform optimizations [38] and to support developers by providing useful and precise information about the program being developed or maintained [48]. In general, standard dataflow analysis cannot be applied to code assets that are not valid with respect to the underlying language. Until recently, in order to make use of dataflow analysis in SPLs, developers were required to explicitly generate products from the line so that dataflow analysis could be applied to them. This changed with the introduction of intraprocedural and interprocedural *feature-sensitive* dataflow analysis by Brabrand et al. [18] and Bodden et al. [17], respectively.

In this dissertation, we explain our participation and build upon these two works and provide a deeper explanation to the challenges of the implementations, as well as a statistically-based empirical evaluation of these techniques. Specifically, we present the implementation of four different approaches to intraprocedural feature-sensitive dataflow analysis, which are not covered in the publication by Brabrand et al. Then we evaluate these implementations through a series of performance experiments and explore the data in greater depth. We found further evidence that each approach behaves differently depending on several characteristics of the method being analyzed. In the interprocedural context, we discuss the proposal of Bodden et al. and present our own evaluation data.

We found evidence that the feature-sensitive approaches clearly outperforms the

feature-oblivious, brute-force approaches, in all cases. Additionally, we provide evidence that each of the four intraprocedural feature-sensitive approaches performs differently depending on various characteristics of the method being analyzed, like size and number of features. By examining their implementations and the data acquired with our executions of the experiments, we discuss aspects such as ease of implementation and raw performance.

1.1 OUTLINE

The remainder of this work is organized as follows:

- In Chapter 2 we present the key concepts covered throughout this dissertation;
- We present the concepts of intraprocedural feature-sensitive dataflow analysis in Chapter 3 and present our contributions;
- In Chapter 4 we describe interprocedural feature-sensitive analysis and our empirical evaluation;
- In Chapter 5 we discuss other works that are related to the topics discussed in this dissertation; and
- Lastly, in Chapter 6, we present our final remarks.

CHAPTER 2

BACKGROUND

In this chapter we present the main concepts we focus and extend in this dissertation.

2.1 SOFTWARE PRODUCT LINES

A Software Product Line (SPL) describes a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [21]. Core assets are artifacts we use to instantiate more than one product [28]. Examples of potential core assets are: requirements, binary files, test cases, image files, and so on. A feature is a prominent and visible aspect, quality, or characteristic of a software system or systems [30]. Features can describe the commonalities and variabilities within the SPL scope. In this context, reuse plays an important role since products frequently share commonalities between them. In fact, when adopting a SPL approach to development, we may benefit from the following characteristics:

- Reduction of development costs: due to intensive reuse of artifacts, individual products are not developed from scratch, which leads to cost reduction. Nevertheless, it is important to note we need to design and implement the core assets beforehand so that we can reuse them when building the other products in the future. Empirical studies reveal that this initial investment pay off when having three products [21];
- Enhancement of quality: because we reuse the core assets between products, we have more opportunities to test and validate such assets, increasing the chance of finding bugs and correcting them earlier;
- Mid-term reduction of time-to-market: In initial stages, the time-to-market is high since we first need to develop the core assets. As time passes and core assets are consolidated, the time-to-market drops because we are able to deliver more products faster due to the aforementioned reuse.

To enjoy these benefits, we must manage the product line features accordingly. Feature-Oriented Domain Analysis (FODA) is a domain analysis that describes the SPL commonalities and variabilities by means of feature diagrams. To better illustrate this approach, Figure 2.1 shows an example of simple feature diagram. This diagram defines a hierarchical decomposition of features with the following relationships: mandatory (filled circle), optional (open circle) and alternative (open arc) relationships. Notice that this model defines not only the features and their relationships, but also feature constraints. For example, this feature diagram states (at the bottom) that A implies E . That is, an instantiated product cannot have feature A without feature E .

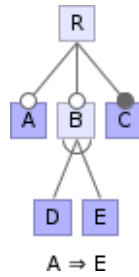


Figure 2.1: A simple feature diagram.

Several techniques exist to support the development of SPL: preprocessors [35, 48, 22, 41], aspect-oriented programming [10, 34], design patterns [11], programming transformations [45, 9] etc. In this work, we focus on preprocessors since they are common in industrial practice. We detail preprocessors in next section.

2.2 PREPROCESSORS

Preprocessors are a widely used mechanism to deal with software variabilities. Examples of real systems that use preprocessors are: the Linux and FreeBSD operating systems, the VIM text editor, and the gcc compiler. In general, preprocessors require that the compiler preprocesses the code and decides—based on directive tags—which parts of the source code should be compiled or not. Preprocessors are also known as *conditional compilation*. Listing 2.1 depicts part of the code of a feature from the Lampiro product line, called BT_PLAIN_SOCKET. Notice that we encompass the feature code by using an `#ifdef` preprocessor directive. To build a product with the BT_PLAIN_SOCKET feature, we define the BT_PLAIN_SOCKET tag and let the compiler consider the code for compilation. Otherwise, the compiler ignores the code, which means we are building a product without

the `BT_PLAIN_SOCKET` feature.

```

1 xmlStream = new SocketStream();
2 ...
3 // #ifdef BT_PLAIN_SOCKET
4 Channel connection = new SocketChannel("socket://" + cfg.getProperty(Config
    .CONNECTING_SERVER), xmlStream);
5 // #endif
6 ...
7 ((SocketChannel) connection).KEEP_ALIVE = Long.parseLong(cfg.getProperty(
    Config.KEEP_ALIVE));

```


Listing 2.1: Code snippet from the Lampiro product line. Lampiro uses preprocessors to implement features.

Despite their widespread usage [41, 49], developers using preprocessors face several problems [53, 25, 40, 33]. Depending on the number of `#ifdef` directives, it can be difficult to read and understand the code, specially if they are nested. Consequently, developers are prone to subtle errors, like when opening a bracket within the `#ifdef` scope and closing it outside the `#ifdef` scope, or the variable that is defined in line 4 of Listing 2.1 and used outside of the `#ifdef` block. In this case, the developer might only detect this problem if he eventually compile the code for the problematic feature combination. Because the number of possible products is combinatorial, finding such errors is expensive. Last but not least, preprocessors do not provide separation of concerns. The directives are tangled with the code base, which means that mandatory and optional features may reside in the same code asset, which leads to maintainability and traceability problems.

To minimize these problems, Kästner et al. proposed the Colored IDE (CIDE) [33]. Instead of `#ifdef` directives, CIDE uses background colors to encompass feature code. This avoids code pollution and decreases the system size in terms of source lines of code. Also, CIDE only allows *disciplined annotations*. According to Liebig et al. [42], disciplined annotations are:

Annotations on one or a sequence of entire functions are disciplined. Furthermore, annotations on one or a sequence of entire statements are disciplined. All other annotations are undisciplined.

Additionally, CIDE implements the Virtual Separation of Concerns (VSoC) concept. When using VSoC, developers can hide feature code not relevant to the current task. Thus, developers can focus on a feature without the distraction caused by the code of other features. The approach is called “virtual” because there is no physical separation. The tool simply collapses the feature code, hiding it from the user, even though it is still there in the original place. Listing 2.2 illustrates the BT_PLAIN_SOCKET feature hidden according to the CIDE tool and the VSoC approach.

```
xmlStream = new SocketStream();
...

...
((SocketChannel) connection).KEEP_ALIVE = Long.parseLong(cfg.getProperty(
    Config.KEEP_ALIVE));
```

Listing 2.2: The rectangle indicates that there is hidden feature code.

2.3 INTRAPROCEDURAL DATAFLOW ANALYSIS

In this section we review some key dataflow analysis concepts that are required later in Chapter 3 and 4. The key concepts are the control flow graphs, lattices, and transfer functions. Together they form the Kildall’s framework for dataflow analysis [38]. In what follows we detail each of them.

2.3.1 Control flow graph

The control flow graph (CFG) is an abstract representation of the flow of control of a program procedure. A CFG is a directed graph in which nodes are statements and edges represents the flow of control between these statements. This flow must conform with the language semantics. Figure 2.2 illustrates a toy program and its corresponding CFG.

2.3.2 Lattice

We use lattices to represent the information calculated during an analysis. A lattice can be visualized as a Hasse diagram. Figure 2.3 shows a lattice for analyzing the value of the variable x , whose possible values are 0 and 1. The $x = 0$ value represents the fact that x is

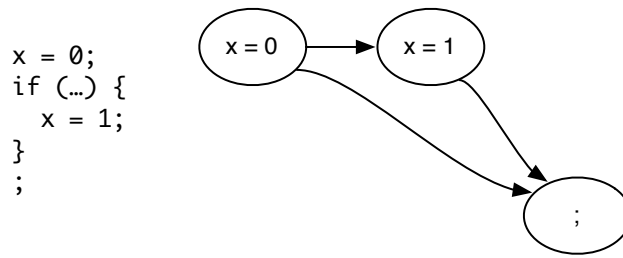


Figure 2.2: A toy program and its CFG.

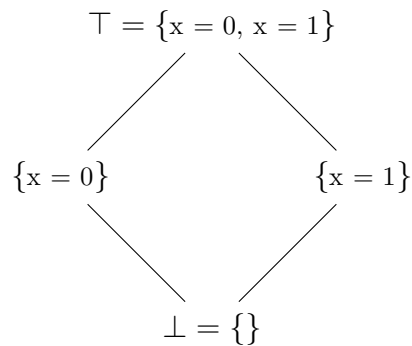


Figure 2.3: A Hasse diagram representing a lattice with all the possible assignments in our toy program.

assigned to 0. Analogously, $x = 1$ represents the fact that the x is assigned to 1. Notice that the lattice contains two additional elements: \perp , which means “unknown”; and \top , which, in this case, means that x can be assigned to either 0 or 1. For example, unless we execute the program, we do not know the value of the x variable at the skip command; it might be either 0 or 1.

2.3.3 Transfer functions

Each statement in the program being analyzed is associated with a transfer function. A transfer function simulates the execution of a given statement in the domain defined by the lattices. To understand how these functions work, consider the statement $x = 0$ in Figure 2.4. The lattice value “flowing” into the transfer function t is \perp which represents the fact we know nothing about x right before the assignment at hand. This transfer functions computes that the assignment to x results in the lattice value $x = 0$, representing

the fact that we now know that the value of x is 0.

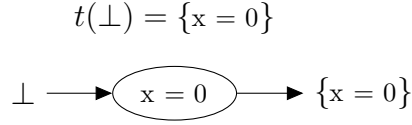


Figure 2.4: The effect of applying a transfer function of an assignment to the \perp lattice value.

2.3.4 Reaching definitions

The reaching definitions analysis computes which variables definitions can reach a given program point p . Hence, every assignment to any variable represents a definition d . A definition d reaches a point p if there exists a path from d to p such that d is not redefined (killed) along that path [8]. In other words, if along this path there is an assignment to the variable corresponding to d , this definition does not reach the point p .

To better illustrate how this analysis works, consider the CFG illustrated in Figure 2.5. Consider the definition $x = 0$. Notice that this definition does not reach the skip statement. All paths from the definition to such a statement contains new assignments to x . On the other hand, $x = 2$ reaches the skip statement: there is no new assignments to x along this path.

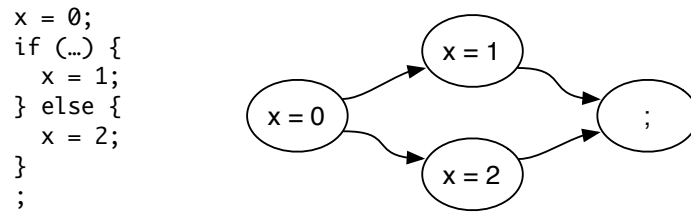


Figure 2.5: The assignment $x = 0$ does not reach the skip statement. All paths from the assignment to the skip statement contains new assignments to x .

When executing the reaching definition analysis, the lattice arranges information with respect to assignments, as illustrated in Figure 2.3. In this context, when simulating the executing of a statement corresponding to a definition d , say, $x = 0$, we “generate” the new

definition of the x variable. Additionally, we “kill” the other definitions of x . Therefore, the transfer function of an assignment in the reaching definitions analysis is defined as:

$$f_d(in) = gen_d \cup (in - kill_d)$$

In this case, gen_d corresponds to the set of definitions generated by the definition d ; $kill_d$ is the set of definitions killed by d ; and in is the set of definitions that flows into d .

2.3.5 Intraprocedural dataflow analysis with Soot

Soot [2] is a framework for analyzing and transforming Java programs. Soot uses intermediate representations of programs, with the most prominent being *Jimple*, a typed 3-address representation designed for optimizations.

Intraprocedural dataflow analyses in Soot with the Jimple representation are typically implemented by extending the `ForwardFlowAnalysis <U,L>` class, if the analysis being implemented is a forward-analysis, e.g. reaching definitions, or `BackwardsFlowAnalysis <U, L>` if the analysis is a backwards-analysis, e.g. live variables. The type parameters U and V are for the type of statements in the CFG of the method being analyzed and the type of lattice implementation, called `FlowSets` in Soot, respectively. Listing 2.3 shows an implementation for a reaching definitions analysis. This class extends `ForwardFlowAnalysis <Unit, FlowSet>`. In Soot, the interface `Unit` represents a generic statement or command in any of the available intermediate representations, including Jimple. The contract behind the `ForwardFlowAnalysis` class requires that the client implement the following template methods that are called by the analysis solver:

- **copy**: this method is called when the analysis solver wants to copy the contents of the value from the source argument to the dest argument. We delegate this operation to the homonymous lattice operation, which copies the content of source into dest;
- **merge**: also known as the *least upper bound* operation. This method is called when contents of lattices `source1` and `source2` must be combined in a confluence point. Because this method returns no value, i.e. it is a **void** method, the contract for this method expects the result of the merge operation to side-effect into the `dest` argument. According to the `FlowSet` contracts, some lattice operation op between two `FlowSets`, l_1 and l_2 , is called with a third argument, l_3 that receives the result of the operation. That is: $l_1.op(l_2, l_3)$ results in $l_3 = l_1 op l_2$. Because reaching

definitions is a *may* analysis, we delegate the merge operation to the union of the FlowSets;

- `newInitialFlow`: the return of this function is used to initialize all points with a starting lattice. In the reaching definition analysis, the starting lattice is $\perp = \{\}$. Thus we return a new empty FlowSet of type `ArraySparseSet`;
- `entryInitialFlow`: similar to the `newInitialFlow` method, except the returned value is used only at the entry point of the method. In the case of the reaching definitions, it is also $\perp = \{\}$; and
- `flowThrough`: this method represents the application of the transfer function relative to the unit parameter. The source argument is the lattice flowing into the transfer function. This method expects the client to side-effect the resulting lattice into the dest argument. In our reaching definitions analysis, if the statement for which we are applying the transfer function is *not* an assignment (`AssignStmt` in Soot), we simply copy the source lattice into the dest lattice untouched (cf. line 29). However, if the statement *is* an assignment, then we must update the lattice accordingly. We implement this with `kill` and `gen` functions shown in lines 33 – 48. In the `kill` function, we store all preexisting assignments to the same variable that the assignment parameter assigns to inside the `kills` FlowSet. We then put all the assignments that are not in the `kills` FlowSet in the `dest` parameter by means of the difference operation (cf. line 43). Lastly, the `gen` function simply adds the assignment parameter to the `dest` parameter.

```

1 public class ReachingDefinitions extends ForwardFlowAnalysis<Unit, FlowSet>
  {
2   public ReachingDefinitions(DirectedGraph<Unit> cfg) {
3     super(cfg);
4     super.doAnalysis();
5   }
6
7   protected void copy(FlowSet source, FlowSet dest) {
8     source.copy(dest);
9   }
10
11  protected void merge(FlowSet source1, FlowSet source2, FlowSet dest) {
```

```

12     source1.union(source2, dest);
13 }
14
15 protected FlowSet newInitialFlow() {
16     return new ArraySparseSet();
17 }
18
19 protected FlowSet entryInitialFlow() {
20     return new ArraySparseSet();
21 }
22
23 protected void flowThrough(FlowSet source, Unit unit, FlowSet dest) {
24     if (unit instanceof AssignStmt) {
25         AssignStmt assignment = (AssignStmt) unit;
26         kill(source, assignment, dest);
27         gen(dest, assignment);
28     } else {
29         source.copy(dest);
30     }
31 }
32
33 private void kill(FlowSet source, AssignStmt assignment, FlowSet dest)
34 {
35     FlowSet kills = new ArraySparseSet();
36     for (Object earlierAssignment : source.toList()) {
37         if (earlierAssignment instanceof AssignStmt) {
38             AssignStmt stmt = (AssignStmt) earlierAssignment;
39             if (stmt.getLeftOp().equivTo(assignment.getLeftOp())) {
40                 kills.add(earlierAssignment);
41             }
42         }
43     }
44     source.difference(kills, dest);
45 }
46
47 private void gen(FlowSet dest, AssignStmt assignment) {
48     dest.add(assignment);
49 }

```

Listing 2.3: Implementation of an intraprocedural reaching definitions analysis in Soot

The constructor of the `ReachingDefinitions` class, shown in line 2, takes as parameter a `DirectedGraph` representing the CFG of the method to be analyzed. The constructor then delegates the computation of the fixed to the solver by calling the `doAnalysis()` method of its superclass.

2.4 INTERPROCEDURAL DATAFLOW ANALYSIS WITH IFDS

An Interprocedural Control Flow Graph (ICFG) is a CFG in which method calls are represented by the flow of control from the call-site to the CFG of the called method and back again to the call-site. Figure 2.6 shows an example of such an ICFG. There are two calls to the function `f` in it. The flow of control is transferred to the method in both calls, whose solo content is the `return z` statement. In a CFG, all paths are valid ones, but in an ICFG this is not always the case. For example, the path `int x = 0 → f(x) → return z → int y = f(3)` is not a feasible one. This is known as the problem of *lack of context sensitivity*, where interprocedural paths do not respect the context in which functions calls were made. Considering infeasible paths leads to imprecise solution in dataflow problems. For example, a taint analysis [26] works by defining a set of variables that are initially tainted, e.g it holds sensitive data. Other variables get tainted if an assignment to them uses a tainted variable. The infeasible path will cause the analysis to imprecisely conclude that the `y` variable is tainted by `x`.

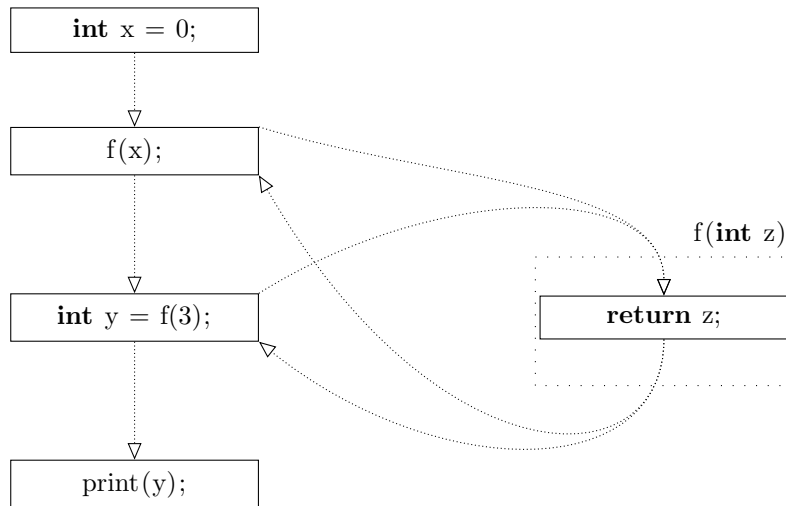


Figure 2.6: An interprocedural control flow graph.

The IFDS framework [47] defines graph-based solution to interprocedural dataflow flow analysis that is context-sensitive. The main idea behind this framework is to reduce a

program-analysis problem into a graph-reachability problem that ignores infeasible paths. The IFDS framework works on a so-called *exploded super-graph*, where nodes have the form (s, d) , where s is a statement and the d is dataflow fact. The dataflow fact d at a statement s only holds if there is a path from the special starting node $(s_0, \mathbf{0})$ to (s, d) . To accomplish this, the framework requires that transfer functions be represented as nodes and edges.

Figure 2.7 shows how transfer functions in the IFDS framework are encoded. The identity transfer function, depicted in Figure 2.7a, shows how all dataflow facts, $\mathbf{0}$, x and y , are mapped to themselves. The nodes at the top of the transfer function represent the facts that hold *before* the transfer function execution and the values at the bottom represent the facts that hold *after* the transfer function. The $\mathbf{0}$ fact is a special dataflow fact unrelated to the domain of the analysis that represents a fact that is always true. Figure 2.7b shows a typical gen/kill transfer function, where some values are killed and others generated. The generation of the y value is represented by the arrow from $\mathbf{0}$ to y ; and the killing of x is represented by the lack of arrows from other nodes to it.

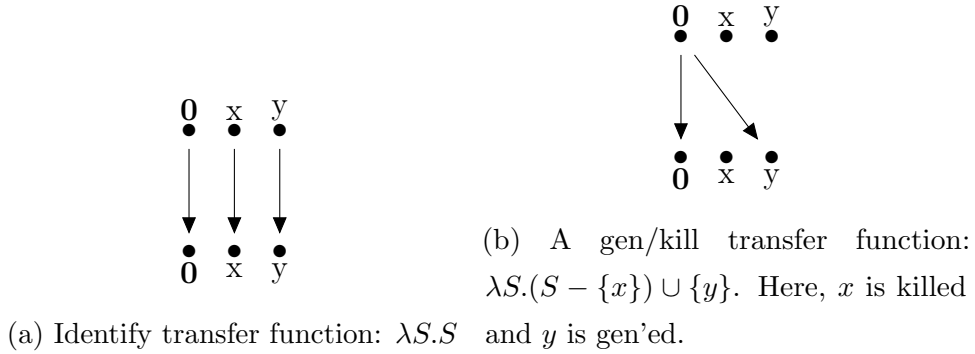


Figure 2.7: Transfer function encoding in IFDS.

As an interprocedural dataflow analysis framework, IFDS represents interprocedural flow of data by subdividing transfer functions into four classes of *flow functions*:

- *normal* flow functions model intraprocedural flow, i.e. flow that is not in a function call or return;
- *call* flow functions model flow of data from the call-site to the entry statement of the called function;
- *return* are flow functions that model the flow of data from the exit node of the called functions to the successor of the call-site statement; and

- *call-to-return* which models a summary-like intraprocedural flow of data from the antecessor of the call-site to the return-site.

In a taint analysis, we generate a variable as dataflow fact in order to represent that variable as tainted. A tainted variable t_1 taints another variable t_2 when t_1 is used in the right hand side of an assignment to t_2 . Conversely, we kill the variable if it is assigned to another variable that is not tainted. Figure 2.8 shows an example of this analysis applied to the exploded super-graph generated from the program in Figure 2.6. The analysis begins at the top statement, `int x = 0;`. To illustrate the analysis, we consider the assignment to 0 as the tainting of the variable x . Thus the flow function generates this fact, which is represented by the arrow connecting the topmost **0** to the x below it. The next statement is the call to $f(x)$. The call flow function carries the fact that x is tainted to the formal parameter z , in the f function. Because the statement $f(x)$ does not assign the result value from f , which simply returns its parameter, the return flow function does not connect the dataflow fact z to the program point immediately after the function call $f(x)$. However, x is still tainted, which is represented by the call-to-return flow function not killing the dataflow fact x . In the next statement there is a new call to f , `int y = f(3)`. Unlike the other call to f , this one does not pass the tainted value x as a parameter to f . Thus, in this case, the data flow fact x does not connect to the formal parameter z in f . The return flow function from this specific call to f connects the formal parameter z to y . Again, x is still tainted at this point because there is no assignment to the x variable that could kill the fact that it is tainted. The IFDS problem solver would solve the equations for these transfer function and compute that, in fact, although the y dataflow fact is reachable from the starting node, the *paths in which that happens are infeasible paths*, because the call and return edges do not match.

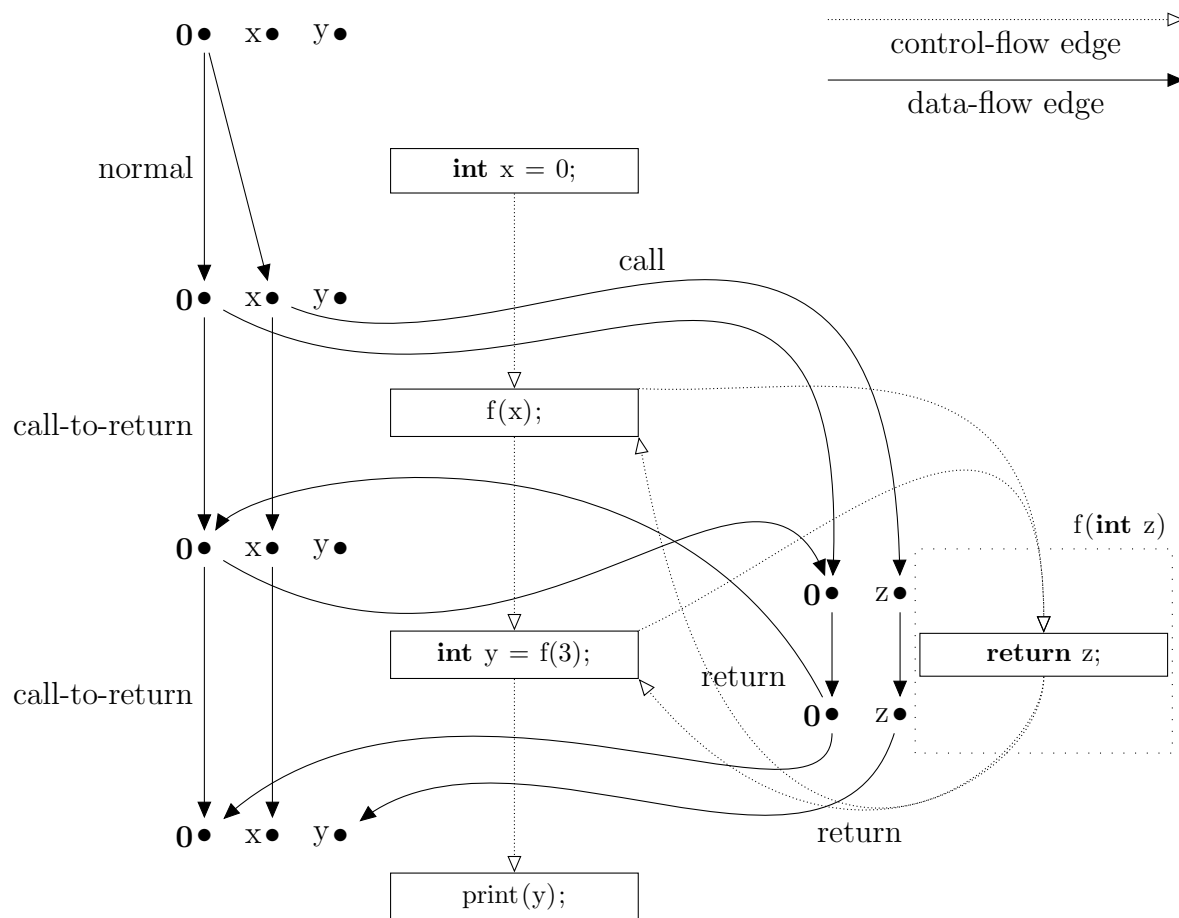


Figure 2.8: An exploded super-graph showcasing the taint analysis.

CHAPTER 3

FEATURE-SENSITIVE INTRAPROCEDURAL DATAFLOW ANALYSIS

As we discussed in Section 2.3, one can use Kildall’s dataflow analysis framework to define dataflow analyses that can compute a set of properties from procedures in a given input program. When trying to analyze the products of a software product line, however, developers were required to generate all programs explicitly so that its procedures could be analyzed. Brabrand et al. [18] showed that doing so could be prohibitive for a program family with a large amount of products. They also show how an intraprocedural dataflow analysis can be *lifted* to perform the same analysis on all procedure without explicitly having to generate all possible products.

In this chapter we discuss Brabrand and colleague’s proposal of *feature-sensitive dataflow analysis*. In the following sections we review their proposals and present the actual implementation, which is not discussed in the aforementioned paper by Brabrand et al. Then we present a new, deeper evaluation of their proposal.

3.1 MOTIVATION

Consider the method fragment in Listing 3.1, written in a language similar to C++, where memory is allocated with the `new` operator, in which memory for the `obj` is only deallocated if the preprocessor variable `F` is defined. For a standard dataflow analysis to even try to figure out that this could lead to a memory leak, the method that contains this code fragment would have to be preprocessed first in order to generate its *variants*: one that has the `delete obj` instruction, and the other that does not, and then analyze each one individually. We henceforth denominate this the *brute-force* approach.

The number of method variants is exponentially proportional to the number of preprocessor variables present in method, considering there is no constraints between the features imposed by a feature model. This means that a method with 10 different variables has up to 1024 different variants. Generating each possible configuration so that they can be

```

Object obj = new Object();
// ...
#ifdef F
delete obj;
#endif
// ...

```

Listing 3.1: Hypothetical procedure fragment with potential memory leak.

compiled and analyzed can quickly become intractable due to this exponential nature.

Even though the term *configuration* is more commonly used to refer to an individual product in a product line, from now on we use the same term when referring to a method variant as well.

Brabrand et al. [18] describe theoretically how one can implement feature-sensitive dataflow analysis in four ways. In the next sections we progressively discuss each of them in greater detail.

3.2 CONSECUTIVE

The *consecutive* approach to feature sensitive analysis analyzes every configuration of a method individually. However, we must somehow encode the variability present in form of `#ifdefs` in order to avoid generating all possible variants of a method. We call this encoding the *instrumentation* process. It revolves on attaching information about the variability to the CFG nodes of the method. Figure 3.1 displays the instrumented CFG from Listing 3.1.

The instrumentation process consists of annotating the CFG nodes that are encompassed by an `#ifdef F`, where `F` could be any preprocessor variable or expression of variables, with the boolean expression $[[F]]$.

More complex boolean expressions are also allowed and nested `#ifdef` are handled by conjunction. For example, the code fragment in Listing 3.2 can be represented by the CFG in Figure 3.2.

In general, `#ifdefs` can be used in any part of the source because they are not bound to the syntax of the underlying language. In this work and that of Brabrand et al., however, we only consider SPLs that use a special case of disciplined annotations [33].

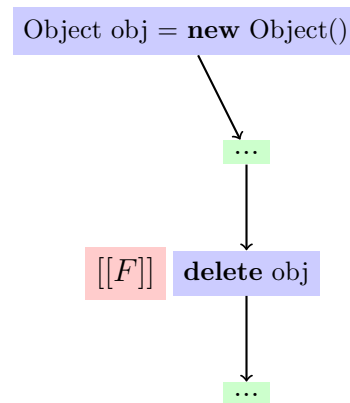


Figure 3.1: Partial CFG with one node instrumented.

```

// ...
#ifdef F || G
#ifdef H
delete obj;
#endif
#endif
// ...

```

Listing 3.2: A statement encompassed by several conditional compilation directives.

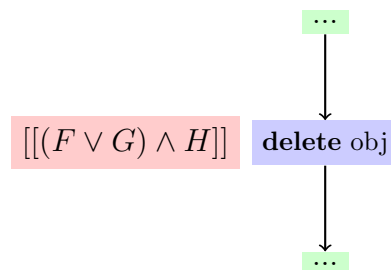


Figure 3.2: A statement with a nested complex conditional expression.

More generally, the expression $[[\psi]]$ represents the set of configurations that a statement is bound to. We assume that nodes in the CFG that have no conditional constraint are annotated with a special constraint **true**. The annotations are representations of the configuration sets associated with each statement, i.e. the configurations in which that statement is always present. The special **true** means that the annotated statement should be present in all possible configurations.

Thus the code variability denoted by the **#ifdefs** is encoded in the CFG of a method. That alone, however, is not enough to allow dataflow analysis to be feature-sensitive; we must address the other components of our analysis: the lattices and the transfer functions. How we define these is what differentiates the approaches to intraprocedural feature-sensitive dataflow analysis, as they all required an instrumented CFG.

We mentioned that the consecutive approach computes the analysis by analyzing each configuration individually. This means that the analysis is instantiated for each configuration c . So, before applying the transfer function relative to some statement, the analysis checks for applicability of that transfer function against the associated configuration set, $[[\psi]]$, of that statement by deciding if $c \in [[\psi]]$. For example, the transfer function associated with the statement **delete** *obj* in Figure 3.2 would not be applied for the configuration $c = \{H\}$, that is, a configuration in which the feature H is enabled and F and G are disabled, because $c \notin [(F \vee G) \wedge H]$.

To illustrate this, consider the code fragment in Listing 3.3, where the **#ifdef** F encompasses the definition of the variable *obj*. Figure 3.3 shows the result of the reaching definitions analysis for that listing. In this figure the **#ifdef** statement has been removed and the associated configuration set, $[[F]]$, is represented in the leftmost column, meaning that the statement Object *obj* = **new** Object(); has the associated configuration set $[[F]]$. In the middle column, the analysis is instantiated with the configuration $c = \{F\}$. The result of the transfer function of the Object *obj* = **new** Object(); command is that the applicability test, $c \in [[F]]$, succeeds so the assignment is stored in the out-flowing lattice $\{\text{Object } \textit{obj} = \text{new Object}()\}$. On the rightmost column, however, the same applicability test for the transfer function is not successful, so the assignment is *not* stored in the lattice, which gets copied below untouched. The result of this analysis could be used to prove that for configuration $c = \{\}$, the **delete** operator is being called on a object that was not defined.

```

#ifdef F
Object obj = new Object();
#endif
// ...
delete obj;

```

Listing 3.3: Possibly undefined variable

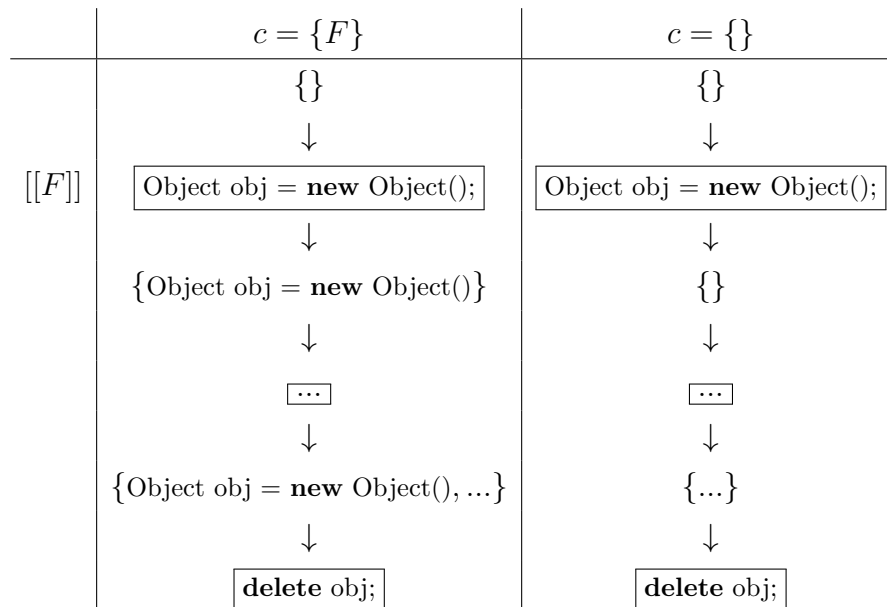


Figure 3.3: Result of the reaching definitions analysis using the consecutive approach.

3.2.1 Implementation

Recall from Section 2.3.5 that statements in Soot’s intermediate representations have a common interface, `Unit`. Soot also provides a tagging mechanism that allows one to attach arbitrary information to Units. We exploit this to attach feature information to individual Jimple statements. And since the CFG is composed of Jimple statements, the CFG itself is said to be instrumented. We use CIDE’s API to query what features are color a given portion of code.

We instrument each method by iterating over all Jimple statements and attaching feature information to it. The container for this information is the `FeatureTag` class, shown in Listing 3.4. The remainder of the class is omitted for brevity, as this class is merely a container for the object of type `IFeatureRep`, which is a representation of a set of features.

```
public class FeatureTag implements Tag {
    private IFeatureRep rep;
    // ...
}
```

Listing 3.4: The container class for the instrumented information.

CIDE uses a restricted form of conditional compilation in which only conjunction is permitted. Under this restriction, we implement an efficient representation for a set of configurations as a bit vector. Given a set of features that occur in a method, we incrementally map each of them to powers of 2. For example:

$$\mathcal{F} = \begin{cases} A \rightarrow 1 \\ B \rightarrow 2 \\ C \rightarrow 4 \end{cases}$$

Thus we can represent a single configuration as a bit vector; the configuration $\{A, B\}$, for instance, can be represented as the bit vector $[1\ 1\ 0]$, or with the integer $1 \oplus 2 = 3$ ¹.

Moreover, we can represent a set of configurations while preserving this compact representation. For example, the bit vector $[0\ 0\ 0\ 1\ 0\ 0\ 1\ 0]$ can be seen as the configuration set $\{\{A, B\}, \{B, C\}\}$. The indices set to true are 3 and 6, which correspond to $\mathcal{F}(A) \oplus \mathcal{F}(B) = 1 \oplus 2 = 3$ and $\mathcal{F}(B) \oplus \mathcal{F}(C) = 2 \oplus 4 = 6$. The fact that each feature

¹ \oplus is the bitwise AND operator.

is mapped to a power of 2 guarantees the uniqueness of the representations of this set. We can also exploit this representation to avoid generating configurations that are invalid according to the feature model. For example, if the configuration represented by the bit with index 6 bit in this bit vector is deemed invalid by the feature model, we set it to 0.

The classes `BitVectorFeatureRep` and `BitVectorConfigRep`, shown in Listing 3.5, are implementations of the `IFeatureRep` and the `IConfigRep` interfaces. These are interfaces for representing a set of features and a set of configurations, respectively. The attribute `atoms` in both classes is the mapping \mathcal{F} we just discussed above. It is a bidirectional map from `Strings` (feature names) to `Integers`. The `BitVector` `bits` attributes are used to represent a configuration in the `BitVectorFeatureRep` class and sets of configurations in the `BitVectorConfigRep`, as discussed above. The bit vector representation allows for an efficient implementation of the of operations like union and intersection (cf. lines 10 and 11) over the sets of configurations.

The separation of these concepts in terms of interfaces allows for greater flexibility; should we chose to provide a new representation for these sets using binary decision diagrams [24], for example, the analyses that rely on them would continue working seamlessly, because they rely solely on the interfaces, not the actual implementations.

```
public class BitVectorFeatureRep implements IFeatureRep {
    private BidiMap atoms;
    private BitVector bits;
    // ...
}

public class BitVectorConfigRep implements IConfigRep {
    private BidiMap atoms;
    private BitVector bits;
    public IConfigRep union(IConfigRep other) { /* omitted */ }
    public IConfigRep intersection(IConfigRep other) { /* omitted */ }
}
```

Listing 3.5: An implementation of the `IFeatureRep` interface using a bit vector.

Concretely, the instrumentation process consists of attaching a `FeatureTag` containing an `IFeatureRep` to every `Jimple` statement in the method. The fragment in Listing 3.6 shows part of the implementation, with some parts omitted. The integer `idGen` in line 2

```

1 // ...
2 int idGen = 1;
3 Map<String , Integer> allFeaturesSoFar = new HashMap();
4 for (Unit nextUnit : body.getUnits()) {
5     Set<String> nextUnitFeatures = currentColorMap.get(nextUnit);
6     if (nextUnitFeatures != null) {
7         for (String featureName : nextUnitFeatures) {
8             if (!allFeaturesSoFar.containsKey(featureName)) {
9                 allFeaturesSoFar.put(featureName, idGen);
10                idGen = idGen << 1;
11            }
12        }
13        IFeatureRep featureRep = featureRep(nextUnitFeatures, allFeaturesSoFar);
14        nextUnit.setTag(new FeatureTag(featureRep));
15    }
16 }
17 // ...

```

Listing 3.6: Main loop for the instrumentation process.

will be used to generate the powers of 2 to uniquely identify every feature encountered (cf. \mathcal{F} above). The identifiers generated are stored in the Map defined in line 3. Lines 4 through 16 comprise the main loop that iterates over all the statements (objects of type Unit) in the body of the method being instrumented. In line 5 we retrieve the features that nextUnit belongs to, according to CIDE. Because CIDE implements this operation inefficiently, we pre-fetch this information and store it in currentColorMap. Lines 7 through 12 handle the generation and storage of the generated identifiers for the features. Lastly, lines 13 and 14 instantiates the IFeatureRep, wrap it in a FeatureTag instance and then add it to the Unit. The information attached to the Jimple statements can later be retrieved by using Soot’s Tag facilities.

Like in the case of sets of features/configurations, the judicious use of interfaces shield the analyses from currently used preprocessor technology, CIDE, giving us greater flexibility over the implementation.

Making use of the instrumentation process, Listing 3.7 shows part of the code for the consecutive reaching definitions analysis. The class ConsecutiveReachingDefinitions extends Soot’s default forward dataflow analysis base implementation, ForwardFlowAnalysis. The

constructor in line 3 has two parameters: the `DirectedGraph<Unit>` `graph`, which is the CFG where the analysis will execute, standard to other Soot analysis; and the `IConfigRep` configuration, which represents the individual configuration the instance will analyze. The transfer function in lines 8–24 is very similar to the one described in Section 2.3.5, Listing 2.3, except for the applicability test in line 15. If the test succeeds then the base kill and gen functions, which operate on normal lattices, are executed. Otherwise, the lattice is simply copied untouched (line 19) by the transfer function.

```

1 public class ConsecutiveReachingDefinitions extends ForwardFlowAnalysis<
    Unit, FlowSet> {
2     private IConfigRep configuration;
3     public ConsecutiveReachingDefinitions(DirectedGraph<Unit> graph,
        IConfigRep configuration) {
4         this.configuration = configuration;
5         // ...
6     }
7
8     protected void flowThrough(FlowSet source, Unit unit, FlowSet dest) {
9         if (unit instanceof AssignStmt) {
10             AssignStmt assignment = (AssignStmt) unit;
11
12             FeatureTag tag = (FeatureTag) assignment.getTag(FeatureTag.
                FEAT_TAG_NAME);
13             IFeatureRep featureRep = tag.getFeatureRep();
14
15             if (featureRep.belongsToConfiguration(configuration)) {
16                 kill(source, assignment, dest);
17                 gen(dest, assignment);
18             } else {
19                 source.copy(dest);
20             }
21         } else {
22             source.copy(dest);
23         }
24     }
25 }

```

Listing 3.7: A consecutive implementation of the reaching definitions analysis.

The IConfigRep is an interface that represents one or a set of configurations by using single integer or a bit vector, respectively. While we just showed the application of the former in the consecutive reaching definitions implementation above, the latter is specially useful in the other feature-sensitive approaches we discuss in Sections 3.4 and 3.5.

3.3 SIMULTANEOUS

The simultaneous analysis, unlike the consecutive one, is capable of analyzing all configurations in a single analysis computation. To do so, it requires what we call a *lifted lattice*: a lattice that is adjusted to contain information about all configurations. Thus, a lifted lattice, L , is a mapping between configurations, $[[\psi]]$, and base lattices, l , shown below.

$$L = [[\psi]] \rightarrow l$$

The mapping between configurations and base lattices allows one to represent information regarding many configurations in a single lattice. These lattices represent dataflow facts relative to each configuration they are mapped by. Each transfer function must iterate over each of these configurations in order to decide whether the lattice should be updated or not. This way, the dataflow facts with respect to many configurations are propagated as one.

Thus the transfer functions must also be *lifted* to operate on the mappings between configurations and base lattices instead of only operating on base lattices. To do so, the transfer functions must operate in a point-wise manner by i) taking every configuration-lattice pair in the lifted lattice, $c \rightarrow l$; ii) checking the applicability of c against the associated configuration set of the statement; iii) either applying the transfer function to l if the applicability test succeeds or copying the lattice untouched to the corresponding configuration in the output otherwise.

To better illustrate this, consider Listing 3.3 and the analysis result for the reaching definition analysis in Figure 3.4. Again we use the leftmost column to denote the associated configuration set of statement `Object obj = new Object();`. In the first line of the rightmost column is the lifted lattice $\{\{\neg F\} \rightarrow \{\}, \{F\} \rightarrow \{\}\}$. We use $\{\neg F\}$ to denote the empty configuration to keep it distinct from the empty base lattice $\{\}$. This lifted lattice represents the fact that before the statement `Object obj = new Object();`, no other variable has been defined in the configurations we are analyzing, $\{\neg F\}$ and $\{F\}$. The next lattice value, $\{\neg F\} \rightarrow \{\}, \{F\} \rightarrow \{\text{Object obj} = \text{new Object}()\}$, represents the fact that the variable `obj` was defined only for configuration $\{F\}$.

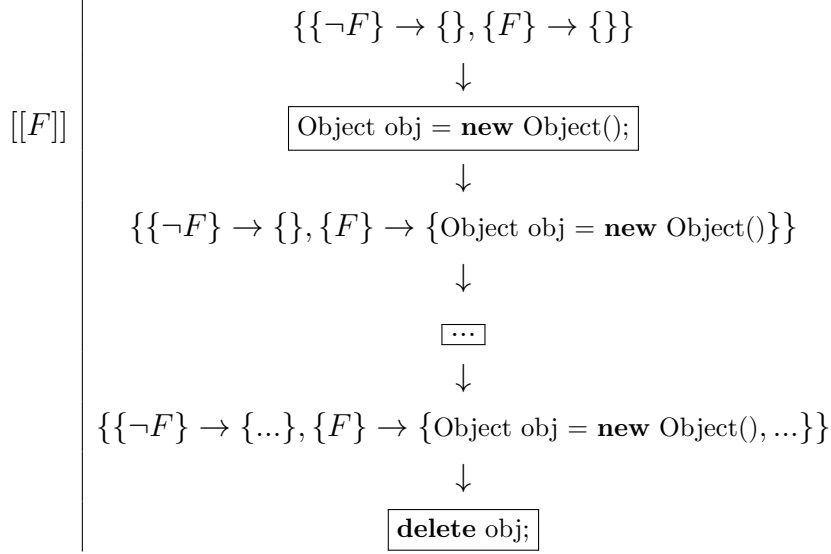


Figure 3.4: Result of the reaching definitions analysis using the simultaneous approach.

3.3.1 Implementation

We implemented the lifted lattice for the simultaneous approach by using a combination of inheritance and aggregation. To plug our feature-sensitive components in the existing infrastructure of Soot, we must abide to the existing contracts. That means, for instance, that lifted lattices must implement the `FlowSet` interface. The lifted lattice we just discussed is in fact a mapping from configurations to base lattices, thus we use aggregation to implement a lifted lattice.

Listing 3.8 shows the lifted lattice used in the simultaneous implementation. The `AbstractMapLiftedFlowSet` in line 1 is the superclass of the `EagerMapLiftedFlowSet`, shown in lines 8–28. The abstract class has the `Map<IConfigRep, FlowSet> map` attribute, which represents the mapping between configurations and lattices (`FlowSets`). The `FlowSet` interface requires the implementation of multiple methods like `intersection` (line 3) and `union` (line 4), which are implemented by the subclasses, in this case, `EagerMapLiftedFlowSet`. The implementation of the `union` method from lines 9 through 26 show how the operation is delegated in a point-wise manner to the inner lattices, inside the map, according to our informal definition in the beginning of this section.

```

1 public abstract class AbstractMapLiftedFlowSet extends AbstractFlowSet {
2     protected Map<IConfigRep, FlowSet> map;
3     public abstract void intersection(FlowSet aOther, FlowSet aDest);

```

```

4  public abstract void union(FlowSet other, FlowSet dest);
5  // ...
6  }
7
8  public class EagerMapLiftedFlowSet extends AbstractMapLiftedFlowSet {
9      public void union(FlowSet aOther, FlowSet aDest) {
10         EagerMapLiftedFlowSet otherLifted = (EagerMapLiftedFlowSet) aOther;
11         EagerMapLiftedFlowSet destLifted = (EagerMapLiftedFlowSet) aDest;
12
13         Set<Entry<IConfigRep, FlowSet>> entrySet = map.entrySet();
14         // for each configuration-lattice mapping...
15         for (Entry<IConfigRep, FlowSet> entry : entrySet) {
16             IConfigRep config = entry.getKey();
17             FlowSet thisNormal = entry.getValue();
18             FlowSet otherNormal = otherLifted.map.get(config);
19
20             /* destNewFlowSet contains the result of the intersection between
21                thisNormal and otherNormal. */
21             ArraySparseSet destNewFlowSet = new ArraySparseSet();
22             thisNormal.intersection(otherNormal, destNewFlowSet);
23             // the result is added to the destLifted lattice
24             destLifted.map.put(config, destNewFlowSet);
25         }
26     }
27     // ...
28 }

```

Listing 3.8: A simultaneous lifted lattice implementation and its base class.

Listing 3.9 shows the implementation for a simultaneous reaching definitions analysis, the `SimultaneousReachingDefinitions` class. It is a `ForwardFlowAnalysis`, but instead of using regular `FlowSets`, it uses `EagerMapLiftedFlowSets` (cf. line 1). The constructor in line 3 accepts as parameters a graph, i.e. the CFG, and a set of configurations, `Set<IConfigRep> configurations`. Each of these configurations will be used to map base lattices when the analysis starts. The transfer function, shown in lines 9–30 operates on instances of the lifted simultaneous lattice implementation, `EagerMapLiftedFlowSet`. The main work of the transfer function is done in the loop shown in lines 17 through 26. There, the analysis iterates over all the configurations present in the lifted lattice and, if the applicability test succeeds (cf. line 20), the `kill` and `gen` functions are executed, otherwise the base lattice is

copied untouched.

```

1 public class SimultaneousReachingDefinitions extends ForwardFlowAnalysis<
  Unit, EagerMapLiftedFlowSet> {
2   private Set<IConfigRep> configurations;
3   public SimultaneousReachingDefinitions(DirectedGraph<Unit> graph, Set<
    IConfigRep> configurations) {
4     super(graph);
5     this.configurations = configurations;
6     super.doAnalysis();
7   }
8   // ...
9   protected void flowThrough(EagerMapLiftedFlowSet source, Unit unit,
    EagerMapLiftedFlowSet dest) {
10    if (unit instanceof AssignStmt) {
11      AssignStmt assignment = (AssignStmt) unit;
12
13      FeatureTag tag = (FeatureTag) assignment.getTag(FeatureTag.
        FEAT_TAG_NAME);
14      IFeatureRep featureRep = tag.getFeatureRep();
15
16      Collection<IConfigRep> configs = source.getConfigurations();
17      for (IConfigRep config : configs) {
18        FlowSet sourceFlowSet = source.getLattice(config);
19        FlowSet destFlowSet = dest.getLattice(config);
20        if (config.belongsToConfiguration(featureRep)) {
21          kill(sourceFlowSet, assignment, destFlowSet);
22          gen(destFlowSet, assignment);
23        } else {
24          sourceFlowSet.copy(destFlowSet);
25        }
26      }
27    } else {
28      source.copy(dest);
29    }
30  }
31  // ...
32 }

```

Listing 3.9: A concrete implementation of a simultaneous reaching definitions.

3.4 SHARED SIMULTANEOUS

The shared simultaneous analysis, just like the simultaneous, can analyze all possible configurations of a method but represents exactly the same lattices in a more compact manner. Note the first simultaneous lifted lattice, $\{\{\neg F\} \rightarrow \{\}, \{F\} \rightarrow \{\}\}$, in Figure 3.4. Both configurations map to the same lattice value, the empty lattice $\{\}$. We can merge both lattices into one lifted lattice that share the base lattice value by their configurations: $\{\{\neg F \vee F\} \rightarrow \{\}\}$. The lattice is thus mapped by a *set of configurations*, instead of only by a configuration like in the simultaneous approach. More generally, the effect of a transfer function f on a statement S and its associated configuration set $[[\phi]]$, over a *shared lifted lattice* can be seen as follows:

$$\begin{array}{c}
 \{[[\psi]] \rightarrow l, \dots\} \\
 \downarrow \\
 \boxed{[[\phi]]: S} \\
 \downarrow \\
 \{[[\psi \wedge \phi]] \rightarrow f(l), [[\psi \wedge \neg \phi]] \rightarrow l, \dots\}
 \end{array}$$

The lifted lattice key, the set of configurations $[[\psi]]$, is “split” into two disjoint sets parameterized by the associated configuration set $[[\phi]]$ of the statement. The transfer function f is only applied to one of the sets, with the lattice value l simply being copied over untouched to the other set. Much like the simultaneous transfer function, these transfer functions must operate in a point-wise manner, but taking pairs of set of configurations and lattices and splitting them into disjoint parts as the analysis progresses.

Figure 3.5 shows, again, the result of the reaching definitions analysis using the shared simultaneous approach. The end result is similar to the one in Figure 3.4, except for the fact that the first lifted lattice is sharing the same base lattice between the configuration set $[[\neg F \vee F]]$.

3.4.1 Implementation

Listing 3.10 shows the class that implements the shared lifted flow set, `LazyMapLiftedFlowSet`. Just like the `EagerMapLiftedFlowSet`, it inherits from `AbstractMapLiftedFlowSet`. And because the `IConfigRep` interface can either represent a single configuration or a set thereof, the structure of the `LazyMapLiftedFlowSet` is nearly identical to that of the `EagerMapLiftedFlowSet`, except for the implementation of the required operations. Shown in lines 9–38 is the

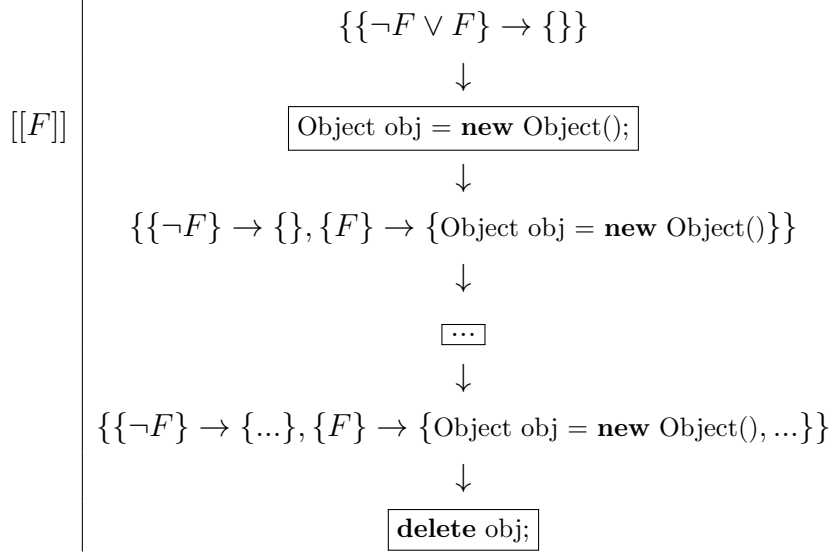


Figure 3.5: Result of the reaching definitions analysis using the shared approach.

implementation for the union operation. Two distinct instances of `EagerMapLiftedFlowSet`, L_1 and L_2 , are guaranteed to have the same number of configurations pointing to base lattices:

$$\begin{array}{c}
L_1 = \{c_1 \rightarrow l_1, c_2 \rightarrow l_2 \dots, c_k \rightarrow l_k\} \\
\downarrow \\
\boxed{[[\psi]] : S} \\
\downarrow \\
L_2 = \{c_1 \rightarrow l'_1, c_2 \rightarrow l'_2 \dots, c_k \rightarrow l'_k\}
\end{array}$$

On the other hand, because of the sharing and splitting, two distinct instances of `LazyMapLiftedFlowSet` do not necessarily have the same size. Thus a different implementation of the union method is needed.

```

1 public abstract class AbstractMapLiftedFlowSet extends AbstractFlowSet {
2     protected Map<IConfigRep, FlowSet> map;
3     public abstract void intersection(FlowSet aOther, FlowSet aDest);
4     public abstract void union(FlowSet other, FlowSet dest);
5     // ...
6 }
7
8 public class LazyMapLiftedFlowSet extends AbstractMapLiftedFlowSet {
9     public void union(FlowSet aOther, FlowSet aDest) {

```

```

10 LazyMapLiftedFlowSet other = (LazyMapLiftedFlowSet) aOther;
11 LazyMapLiftedFlowSet dest = (LazyMapLiftedFlowSet) aDest;
12
13 Set<Entry<IConfigRep, FlowSet>> entrySet = this.map.entrySet();
14 Set<Entry<IConfigRep, FlowSet>> otherEntrySet = other.map.entrySet();
15
16 Map<IConfigRep, FlowSet> destMap = new HashMap<IConfigRep, FlowSet>();
17
18 // for every 2 sets of configurations in every lattice...
19 for (Entry<IConfigRep, FlowSet> entry : entrySet) {
20     for (Entry<IConfigRep, FlowSet> otherEntry : otherEntrySet) {
21         ILazyConfigRep key = (ILazyConfigRep) entry.getKey();
22         ILazyConfigRep otherKey = (ILazyConfigRep) otherEntry.getKey();
23
24         // take the intersection between the sets of configurations
25         ILazyConfigRep intersection = (ILazyConfigRep) key.intersection(
26             otherKey);
27
28         // if the intersection is not empty, i.e.  $\neq \emptyset$ 
29         if (intersection.size() != 0) {
30             FlowSet otherFlowSet = otherEntry.getValue();
31             ArraySparseSet destFlowSet = new ArraySparseSet();
32             // take the lattice intersection and store it in destMap
33             entry.getValue().union(otherFlowSet, destFlowSet);
34             destMap.put(intersection, destFlowSet);
35         }
36     }
37 }
38 dest.map = destMap;
39
40 public void intersection(FlowSet aOther, FlowSet aDest) {
41     /* Omitted. The intuition behind this method is the as the method above
42        , except it delegates the point-wise operation to the intersection of
43        the FlowSets instead of union. */
44 }
45 }

```

Listing 3.10: An implementation for a shared lifted lattice and its base class.

The implementation of the shared reaching definitions, class `SharedReachingDefinitions` is shown in Listing 3.11. This class also extends `ForwardFlowAnalysis`, but it uses the shared version of the lifted lattice, `LazyMapLiftedFlowSet`. Its constructor, shown in line 4, in addition to the usual `DirectedGraph<Unit>` graph representing the CFG, takes the single set of configurations, `ILazyConfigRep configs`, representing all possible configurations for this method. This is the configuration set that will be split along the way as the analysis progresses. The main work done in the transfer function is carried out by the loop in lines 19–53.

```

1 public class SharedReachingDefinitions extends ForwardFlowAnalysis<Unit,
  LazyMapLiftedFlowSet> {
2   private ILazyConfigRep configurations;
3
4   public SharedReachingDefinitions(DirectedGraph<Unit> graph,
    ILazyConfigRep configs) {
5     // ...
6   }
7
8   protected void flowThrough(LazyMapLiftedFlowSet source, Unit unit,
    LazyMapLiftedFlowSet dest) {
9     // pre-copy the information from source to dest
10    source.copy(dest);
11    if (unit instanceof AssignStmt) {
12      AssignStmt assignment = (AssignStmt) unit;
13      IFeatureRep featureRep = ... // retrieve tag from Unit
14      Map<IConfigRep, FlowSet> destMapping = dest.getMapping();
15
16      // iterate over all entries of the source lattice
17      Map<IConfigRep, FlowSet> sourceMapping = source.getMapping();
18      Iterator<Entry<IConfigRep, FlowSet>> iterator = sourceMapping.
        entrySet().iterator();
19      while (iterator.hasNext()) {
20        Entry<IConfigRep, FlowSet> entry = iterator.next();
21        ILazyConfigRep lazyConfig = (ILazyConfigRep) entry.getKey();
22        FlowSet sourceFlowSet = entry.getValue();
23        FlowSet destFlowSet = destMapping.get(lazyConfig);
24
25        /* split the configuration set being currently iterated into a pair
          of configurations sets. the transfer function is applied only to

```

```

    the first set of the pair, the other is mapped to the untouched
    lattice. */
26 Pair<ILazyConfigRep, ILazyConfigRep> split = lazyConfig.split(
    featureRep);
27 ILazyConfigRep first = split.getFirst();
28
29 // in case the first configuration set is empty, do nothing
30 if (first.size() != 0) {
31     /* if the first configuration set is "everything", simply apply
        the transfer function... */
32     if (first.size() == lazyConfig.size()) {
33         kill(sourceFlowSet, assignment, destFlowSet);
34         gen(assignment, destFlowSet);
35     } else {
36         ILazyConfigRep second = split.getSecond();
37         FlowSet destToBeAppliedLattice = new ArraySparseSet();
38
39         // apply point-wise transfer function
40         kill(sourceFlowSet, assignment, destToBeAppliedLattice);
41         gen(assignment, destToBeAppliedLattice);
42         if (second.size() != 0) {
43             destMapping.put(second, destFlowSet);
44         }
45
46         // add the new lattice
47         destMapping.put(first, destToBeAppliedLattice);
48
49         // remove config rep that has been split
50         destMapping.remove(lazyConfig);
51     }
52 }
53 }
54 }
55 }
56 // ...
57 }

```

Listing 3.11: The implementation of the shared version of the reaching definitions analysis.

3.5 REVERSED SHARED SIMULTANEOUS

A different approach to sharing is to map base lattices to configurations, effectively doing the representational opposite of the previously described shared simultaneous approach. That means that the base lattice can be seen as a map from base lattices, l , to a set of configurations, ϕ . The effect of a transfer function f of a statement S and its associated configuration set $[[\phi]]$, over a *reversed shared lattice* can be seen as follows:

$$\begin{array}{c}
 \{l \rightarrow [[\psi]]\} \\
 \downarrow \\
 \boxed{[[\phi]]: S} \\
 \downarrow \\
 \{f(l) \rightarrow [[\psi \wedge \phi]], l \rightarrow [[\psi \wedge \neg\phi]]\}
 \end{array}$$

Unlike the “split” from the shared simultaneous approach, this one is not guaranteed to keep the left-hand side of the mapping, the base lattices, disjoint. If $f(l) = l$, for instance, then there will be two lattice values mapping to different configuration sets. To address this matter, we adopt the convention that only a minimal and unique representation of a lifted lattice is used.

Figure 3.6 shows the result of using the reversed shared simultaneous to execute a reaching definitions analysis. The first lifted lattice, $\{\{\} \rightarrow \{\neg F \vee F\}\}$ represents the fact that the base empty lattice, $\{\}$, is shared by the configuration set $\{\neg F \vee F\}$. The lifted lattice after the statement `Object obj = new Object();` is composed of two parts: $\{\{\} \rightarrow \{\neg F\}\}$ and $\{\text{Object obj} = \text{new Object}()\} \rightarrow \{F\}$, which is the result of splitting the resulting lattice.

3.5.1 Implementation

The lifted lattice of the reversed shared approach, shown in Listing 3.12, has a single attribute, `BiMap<FlowSet, IConfigRep> map`, which is a bidirectional mapping between base lattices and sets of configurations. We use this specific Map interface because we frequently use the inverse of this Map to get a reference to a FlowSet from a given IConfigSet. The implementation of the **union** operation is shown from line 4 through 15. The *merge* operation we discussed in Section 3.5 is used by the **union** method and is implemented by the method `putAndMerge` in lines 18–41. The `putAndMerge` function has to handle three cases that arise when inserting something in BiMap map which are described as follows.

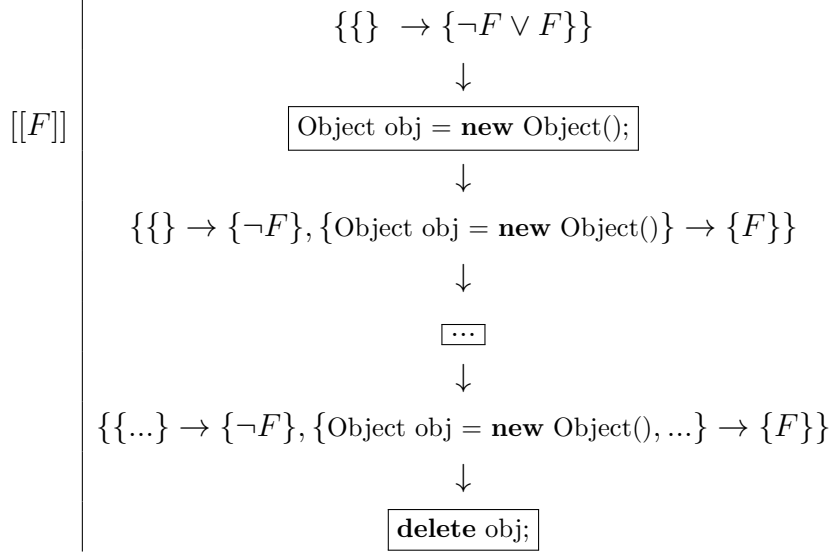


Figure 3.6: Result of the reaching definitions analysis using the reversed shared approach.

Keep in mind that we wish to add the association $l \leftrightarrow c$, where l is a base lattice and c is a configuration set, to the map.

- There is no FlowSet in map that is equal to the one being added and there is no IConfigRep equal to the one being added (cf. line 21). In this case, just add the association $l \leftrightarrow c$ to the map;
- There is a FlowSet in map (cf. line 25). In this case we try to replace the preexisting association $l \leftrightarrow c'$ with $l \leftrightarrow c \cup c'$;
- There is an IConfigRep already in the map that is equal to the one being added (cf. line 32). In this case, we try to replace the preexisting association $l' \leftrightarrow c$ with $l \sqcup l' \leftrightarrow c$.

In all the cases outlined above, recursion is necessary. For example, when trying to replace $l \leftrightarrow c'$ with $l \leftrightarrow c \cup c'$, as stated by the second case, there might already be an entry such as $l' \leftrightarrow c \cup c'$. These situations are solved by recurring into the function once again.

```

1 public class ReversedMapLiftedFlowSet extends AbstractFlowSet {
2     protected BiMap<FlowSet, IConfigRep> map;
3     // ...
4     public void union(FlowSet aOther, FlowSet aDest) {

```

```

5   ReversedMapLiftedFlowSet other = (ReversedMapLiftedFlowSet) aOther;
6   ReversedMapLiftedFlowSet dest = (ReversedMapLiftedFlowSet) aDest;
7
8   Set<Entry<FlowSet, IConfigRep>> otherEntrySet = other.map.entrySet();
9   // replaces dests inner map with a new BiMap
10  dest.map = HashBiMap.create(this.map);
11  for (Entry<FlowSet, IConfigRep> otherEntry : otherEntrySet) {
12      FlowSet key = otherEntry.getKey();
13      dest.putAndMerge(key, otherEntry.getValue());
14  }
15  }
16
17  /* Add an Entry in the map attribute from flowSet to config. A merge
   might be necessary if map already has a key equal to flowSet or a
   configuration set equal to config */
18  public void putAndMerge(FlowSet flowSet, IConfigRep config) {
19      boolean containsKey = map.containsKey(flowSet);
20      boolean containsVal = map.containsValue(config);
21      if (!containsKey && !containsVal) {
22          map.put(flowSet, config);
23          return;
24      }
25      if (containsKey) {
26          IConfigRep inConfig = map.get(flowSet);
27          IConfigRep union = inConfig.union(config);
28          map.remove(flowSet);
29          putAndMerge(flowSet, union);
30          return;
31      }
32      if (containsVal) {
33          BiMap<IConfigRep, FlowSet> inverse = map.inverse();
34          FlowSet inFlowSet = inverse.get(config);
35          ArraySparseSet unionFlowSet = new ArraySparseSet();
36          inFlowSet.union(flowSet, unionFlowSet);
37          inverse.remove(config);
38          putAndMerge(unionFlowSet, config);
39          return;
40      }
41  }

```

42 | }

Listing 3.12: The implementation for the reversed shared lifted lattice.

Listing 3.13 shows the implementation for the reversed shared reaching definitions analysis. It extends the usual `ForwardFlowAnalysis` and uses the reversed version of the mapped lattice (cf. line 1). The main work of the transfer function, implemented in method `flowThrough`, is done in lines 23 through 57. It is very similar to the shared transfer function described in Listing 3.11, with the main difference being the use of the `putAndMerge(FlowSet, IConfigRep)` to put associations in the lattice.

```

1 public class ReversedSharedReachingDefinitions extends ForwardFlowAnalysis<
  Unit, ReversedMapLiftedFlowSet> {
2   private ILazyConfigRep configurations;
3
4   public LazyLiftedReachingDefinitions(DirectedGraph<Unit> graph,
    ILazyConfigRep configs) {
5     super(graph);
6     this.configurations = configs;
7     super.doAnalysis();
8   }
9
10  protected void flowThrough(ReversedMapLiftedFlowSet source, Unit unit,
    ReversedMapLiftedFlowSet dest) {
11    if (unit instanceof AssignStmt) {
12      AssignStmt assignment = (AssignStmt) unit;
13
14      // clear the destination lattice to insert new ones
15      dest.clear();
16
17      FeatureTag tag = (FeatureTag) assignment.getTag(FeatureTag.
        FEAT_TAG_NAME);
18      IFeatureRep featureRep = tag.getFeatureRep();
19
20      // iterate over all entries of the source lattice
21      BiMap<FlowSet, IConfigRep> sourceMapping = source.getMapping();
22      Iterator<Entry<FlowSet, IConfigRep>> iterator = sourceMapping.
        entrySet().iterator();
23      while (iterator.hasNext()) {

```



```

24     Entry<FlowSet , IConfigRep> entry = iterator.next();
25     ILazyConfigRep lazyConfig = (ILazyConfigRep) entry.getValue();
26
27     FlowSet sourceFlowSet = entry.getKey();
28
29     Pair<ILazyConfigRep , ILazyConfigRep> split = lazyConfig.split(
30         featureRep);
31     ILazyConfigRep first = split.getFirst();
32     // in case the first configuration set is not empty
33     if (first.size() != 0) {
34         /* if the first configuration set is "everything", apply the
35            transfer function */
36         if (first.size() == lazyConfig.size()) {
37             FlowSet destFlowSet = new ArraySparseSet();
38             kill(sourceFlowSet , assignment , destFlowSet);
39             gen(assignment , destFlowSet);
40             dest.putAndMerge(destFlowSet , first);
41         } else {
42             FlowSet destFlowSet = sourceFlowSet.clone();
43
44             ILazyConfigRep second = split.getSecond();
45             if (second.size() != 0) {
46                 dest.putAndMerge(destFlowSet , second);
47             }
48
49             // apply base transfer function
50             FlowSet destToBeAppliedLattice = new ArraySparseSet();
51             kill(sourceFlowSet , assignment , destToBeAppliedLattice);
52             gen(assignment , destToBeAppliedLattice);
53
54             dest.putAndMerge(destToBeAppliedLattice , first);
55         }
56     } else {
57         dest.putAndMerge(sourceFlowSet , lazyConfig);
58     }
59 }
60 // ...

```

61 | }

Listing 3.13: The reversed shared implementation of the reaching definitions analysis.

3.6 EVALUATION

In this section we describe the design and execution of our evaluatory experiments. We reexecuted the experiments performed by Brabrand et al. in [18] in a effort to gain a deeper understanding of the performance of each approach. In doing so, we discovered that the data contains variations that affect both our statistical analysis and our reasoning over the results. Later, we discovered that disabling the JIT compiler in the JVM diminished the amount of variability in the data we gathered. Thus we reexecuted the experiments again with no JIT compiler in search of more stable data. In the following sections we report our findings as we trailed that path.

Henceforth, when we talk about any of the approaches presented in the previous sections, we are actually referring to our specific the *implementation* of said approaches.

3.6.1 Study settings

All the experiments we executed ourselves and the one in [18] by Brabrand et al. were done so in the same computer with the following hardware platform: an Intel[®] Core[™] i7-3820 processor at 3.6GHz with 32GB of RAM running a Linux distribution with the 3.2.0-23-generic kernel. The JVM is an OpenJDK, version 1.6.0_24, configured with a maximum heap of 20GB.

Brabrand et al. evaluates the feature-sensitive approaches by applying the consecutive, simultaneous, shared and reversed shared reaching definitions analysis on four different SPLs. Table 3.1, adapted from [18, Figure 10] shows some size metrics that characterize each SPL. The metrics shown are: #units, which represents the number of statements in the Jimple intermediate representation of the methods in the SPL; $|2^F|$ is the method with the largest number of configurations in the benchmark; #methods is the number of methods in the benchmark; and #configurations as the total number of method configurations in the benchmark. The last column points to respective figure with the histogram of the number of configurations of the methods. We leave out of this histogram the methods with no features, i.e., methods with only 1 configurations.

Each benchmark has different feature usage profiles. GPL, for example, is the benchmark with the least number of Jimple statements but also has the method with the most number of configurations. MM08 has more methods and units than GPL, but has a more modest feature usage. Lampiro has almost no feature usage. BerkeleyDB is the largest of the benchmarks in terms of units and number of methods.

Benchmark name	#units	$ 2^F $	#methods	#configurations	Config. histogram
GPL	2107	106	156	536	Figure 3.7
MM08	6393	24	286	523	Figure 3.8
Lampiro	81064	4	2003	2029	Figure 3.9
BerkeleyDB	79040	40	3609	5909	Figure 3.10

Table 3.1: A summary of key characteristics in the benchmarks used in the experiments.

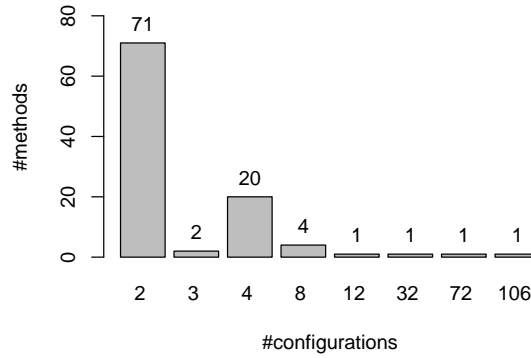


Figure 3.7: Histogram of the number of configurations of methods in the GPL benchmark.

3.6.2 Previous experiments

In their experiments, Brabrand et al. measures the time to analyze all methods of all the benchmark using all of the proposed feature-sensitive approaches with the exception of the reversed shared, which, as they report, at the time of the writing, did not have a reasonable implementation. They repeat this procedure 10 times and then report the sum of the medians (over the 10 runs) of the analysis time of all methods.

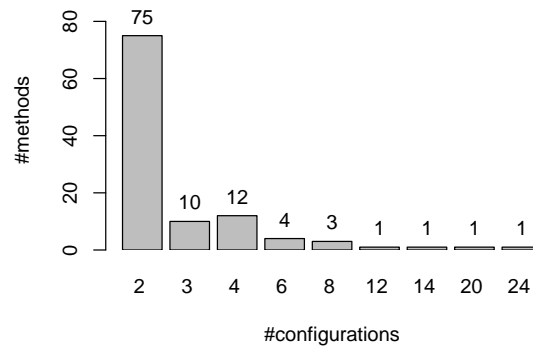


Figure 3.8: Histogram of the number of configurations of methods in the MobileMedia08 benchmark.

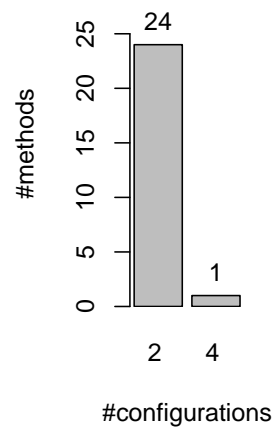


Figure 3.9: Histogram of the number of configurations of methods in the Lampiro benchmark.

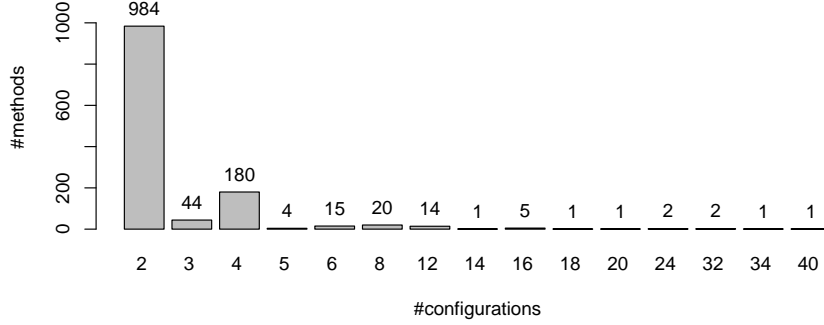


Figure 3.10: Histogram of the number of configurations of methods in the BerkeleyDB benchmark.

The experiments of Brabrand et al. show evidence that all proposed feature-sensitive approaches can outperform the brute-force one. This is mainly due to the fact that the source code has to be recompiled for each configuration and then analyzed, as opposed to the one-off compilation of the feature-sensitive analysis. They also compare the performance of the feature-sensitive implementation between themselves and show that they behave differently, depending on several characteristics of the method being analyzed. In this work, our main focus is on the difference between the features-sensitive approaches.

We reexecuted the experiments of Brabrand et al. on the same benchmarks but under different environments and provide, in the following sections, a more fine-grained explanation for the data obtained in the experiments.

3.6.3 Revisiting the experiments

Brabrand et al. discuss the results of their experiment by looking at the median between ten runs. Although the results are convincing that the approaches do in fact behave differently, there is not much statistical rigor in their analysis of the data. Additionally, we show that environmental aspects in the computer that executed the experiments have major influence in the obtained results.

Instead of focusing only on the median of the analysis time, we study each execution of the experiment individually and report our findings about consistency. Specifically, we are interested in checking under which circumstances there is (or not) a statistically significant difference between the approaches. To do so, we grouped the methods of each

Method	Consecutive (ns)	Simultaneous (ns)	Shared (ns)	Rev. shared (ns)
m_1	200	170	180	250
m_2	420	300	200	400
\vdots	\vdots	\vdots	\vdots	\vdots

Table 3.2: A demonstration of the measurements taken by the experiments.

benchmark into three groups:

1. with exactly two configurations;
2. with exactly four configurations; and
3. with more than four configurations.

We grouped the methods in this manner because it would yield reasonable samples sizes for the majority of the benchmarks. Also, since we are interested in comparing only the feature-sensitive approaches, we do not consider methods with no features. We use the R project [7] for statistical computing to execute the Wilcoxon signed-rank test [57] on the grouped methods to check if there is statistically significant difference between a pair of approaches. We chose to use this test because we cannot assume that the data is normally distributed.

To better illustrate, suppose we took the measurements shown in Table 3.2 from a hypothetical benchmark with methods that belong to a same group. We test the difference between the approaches in a pair-wise manner. For example, to compare the Consecutive and the Simultaneous approach, we execute the Wilcoxon test on the ratio of their measurements, $\{200/170, 420/300, \dots\}$. The test yields a p-value, which is used to reject or not the null hypothesis, \mathcal{H}_0 , that states that both approaches are indistinguishable from each other.

3.6.4 Results discussion

We present the results of our experiment, but this time including the measurements from the implementation of the reversed shared approach, which is not covered in their publication. Like the one performed by Brabrand et al., our experiment is comprised of 10 runs.

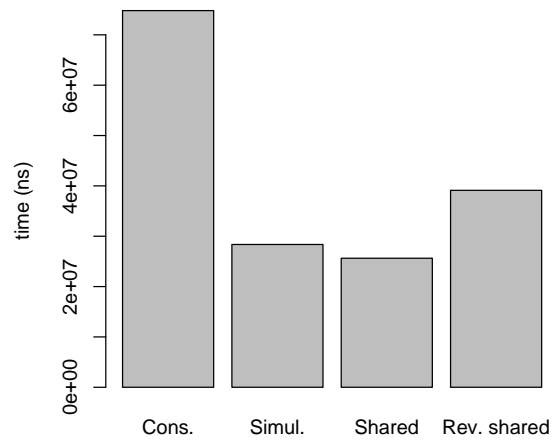
		Simultaneous	Shared	Rev. shared
1st execution	Consecutive	5.72×10^{-6}	2.61×10^{-4}	1.65×10^{-1}
	Simultaneous		5.22×10^{-1}	3.81×10^{-6}
	Shared			2.61×10^{-4}
		Simultaneous	Shared	Rev. shared
2nd execution	Consecutive	3.81×10^{-6}	4.41×10^{-2}	5.96×10^{-1}
	Simultaneous		1.91×10^{-6}	1.91×10^{-6}
	Shared			1.43×10^{-1}
		Simultaneous	Shared	Rev. shared
3rd execution	Consecutive	3.81×10^{-6}	6.29×10^{-5}	1.02×10^{-3}
	Simultaneous		1.99×10^{-3}	7.08×10^{-4}
	Shared			2.02×10^{-1}

Table 3.3: For methods with 4 configurations, p-values output by the Wilcoxon test for the GPL benchmark on 3 of the 10 executions.

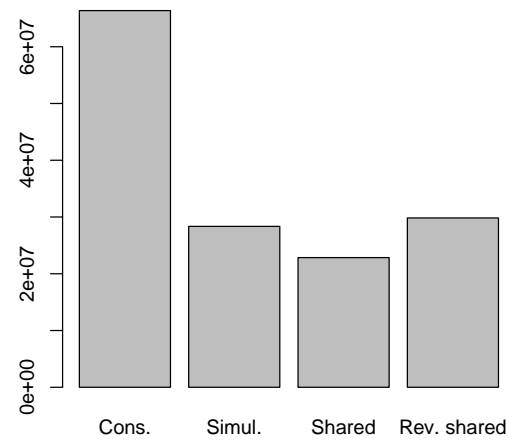
Figure 3.11 shows the median of the sum of the analysis time of each benchmark, for all methods that have more than one configuration. We can visualize a trend in these graphs. The consecutive performs slower than all the others in 3 of the 4 benchmarks; the exception happens on Lampiro, the benchmarks with the least feature usage. We can also see that in 2 of the 4 benchmarks the simultaneous and the reversed shared approach are close to each other. We can also see that the shared approach performs the best on all benchmarks, except on Lampiro, where there it seems to tie with the simultaneous approach.

We found that the p-values output by the Wilcoxon tests vary from each execution of the experiment, regardless of the benchmark. To illustrate this, Table 3.3 shows the p-values obtained for the first 3 executions (out of the total of 10) of the experiment on the group of methods with exactly four configurations from the GPL benchmark. The highlighted values are p-values that are > 0.05 and thus we cannot reject the null hypothesis for the two approaches.

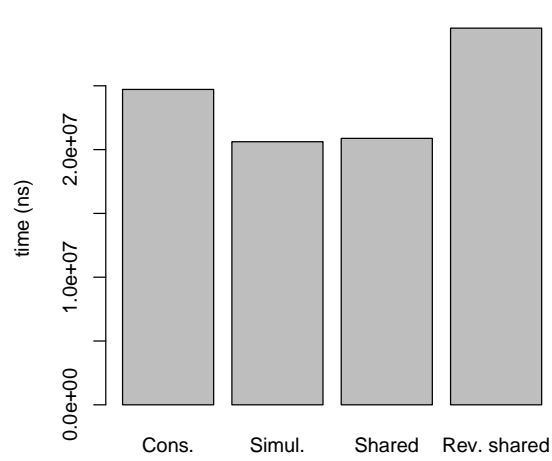
A closer look at the data for the methods individually shows a great deal of variation in the measurements. Figure 3.12 shows the ten measurements of the time taken to analyze the method with the largest number of features in the GPL benchmark using the



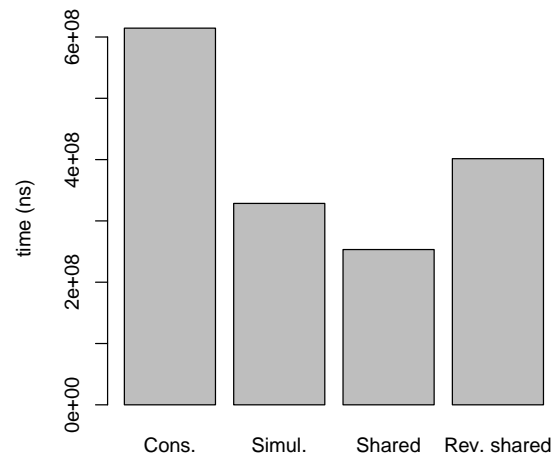
(a) GPL



(b) MM08



(c) Lampiro



(d) BerkeleyDB

Figure 3.11: Median (over the 10 runs) of the sums of the analysis time of methods in each benchmark

consecutive approach. In almost every method of every benchmark, the measurements in the first iteration is usually much slower. In the specific case of this GPL method, the standard deviation is approximately 2×10^7 , quite high for the numbers provided. There are several elements that can help explain the variation between each execution.

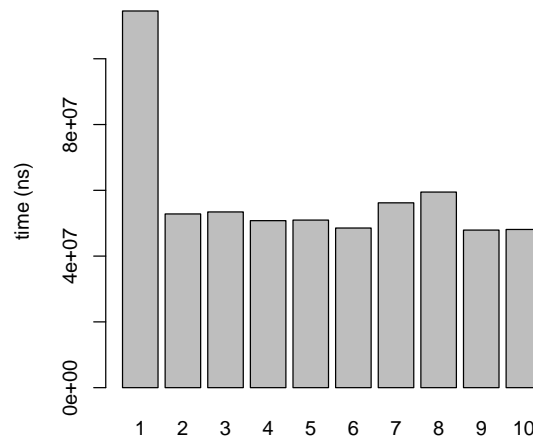


Figure 3.12: Time measurements of the consecutive analysis on one method in the GPL benchmark. This is the method with the largest number of configurations in GPL: 106.

The experiment works by starting a JVM, analyzing all methods of one benchmark ten times and then shutting down the JVM. Thus a new JVM is started for each benchmark. Between the start and shut down of the JVM, a lot more is going on behind the scenes where the code for the experiment is running and at least three of them can cause the variation seen in the measurements: the garbage collector, the thread scheduler and the Java compiler.

The garbage collector used by the JVM is the parallel compacting collector [54] which is a multi threaded garbage collector that from time to time needs to stop the world, that is, to completely stop the execution of the program to run the collector. Should this stop-the-world-event happen while we are measuring the time performance, then the time measurement will be higher than the actual spent doing the analysis.

Because our implementation relies on CIDE, which is built as an Eclipse plug-in, there are several other threads spawned while the experiment is taking place.

Last but certainly not least, the JVM uses Just-In-Time (JIT) compilation techniques

that try to compile the source code at runtime instead of just interpreting. To do so it relies on data collected at runtime and heuristics to identify portions of source code that are worth the effort to compile. This makes time measurements more difficult because, for example, in our case, the JIT might decide that only code for the consecutive analysis should be compiled. While this is fine for real world applications, we are trying to understand and interpret the results, and to do so, we need a more controlled environment.

Fortunately, the JIT compiler in the JVM can be easily disabled. We exploited this and reexecuted the experiment with the same parameters, but with JIT disabled and discuss the results in the next section.

3.6.5 No JIT results discussion

The performance hit taken by disabling the JIT compiler might make it unrealistic, but disabling it allow us to more comfortably reason about our implementation of the analysis and approaches.

We found that disabling the JIT compiler improves the consistency of the measurements, but with an added cost to runtime performance. Figure 3.13 displays the ten measurements of the time taken to analyze the same method for which we show the measurements in Figure 3.12. This time it only has slight variations across the executions; the standard deviation is approximately 3×10^6 , much lower compared to the one from Figure 3.12, but is about one order of magnitude slower overall. We noticed that our measurements are still tainted by variations, but on a smaller scale, in all methods of our benchmarks. Less variations on our measurements means that our observations are more consistent.

That is not the case, however, with methods that take a smaller amount of time to analyze. Figure 3.14 shows the measurements on a method in GPL that has only 2 configurations and only 5 statements, where the variations are visually perceivable. The measurements in Figure 3.14 are up to 3 orders of magnitude lower than those in Figure 3.13.

In our evaluation that follows, we consider how many times a given approach was the fastest of the four when the methods of a benchmark. Although we did not investigate this in depth, it seems plausible that this variation, specially in methods where the absolute time measured is low, can be the cause of the noise we perceived in this specific topic of our investigation.

Before we proceed with the statistical significance tests, we lay out some of the most

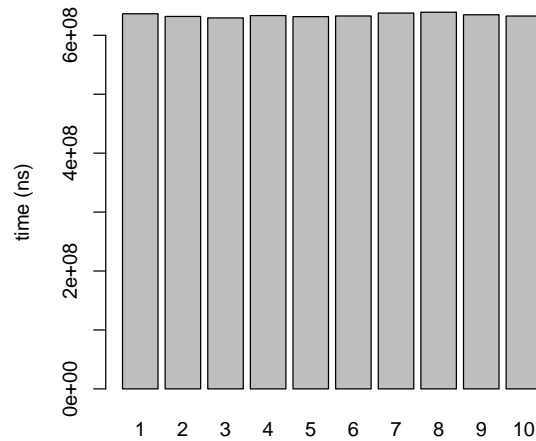


Figure 3.13: Time measurements of the consecutive analysis on one method in the GPL benchmark with the JIT compiler disabled. This is the method with the largest number of configurations in GPL: 106.

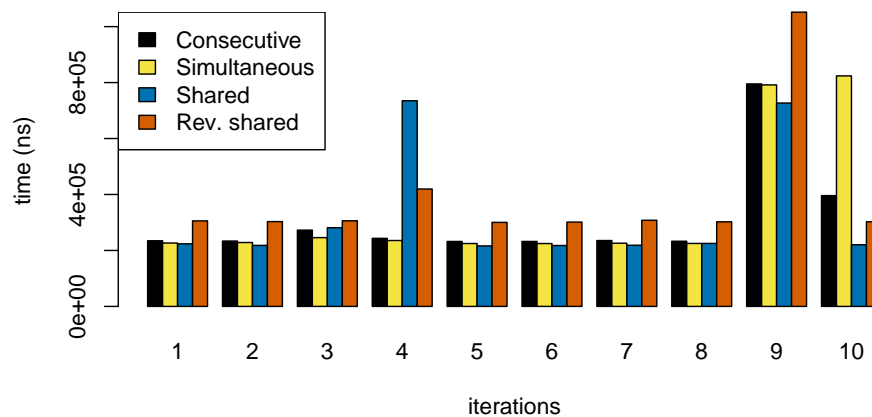
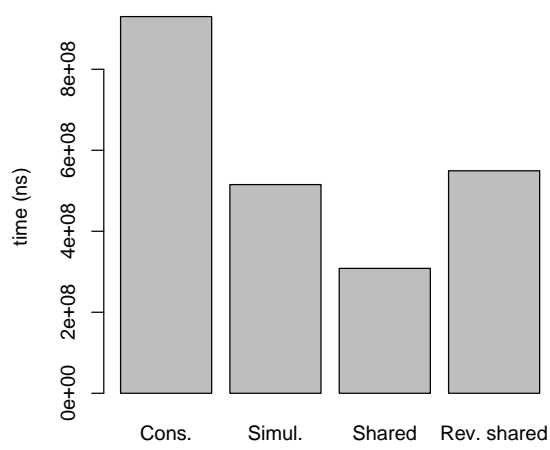
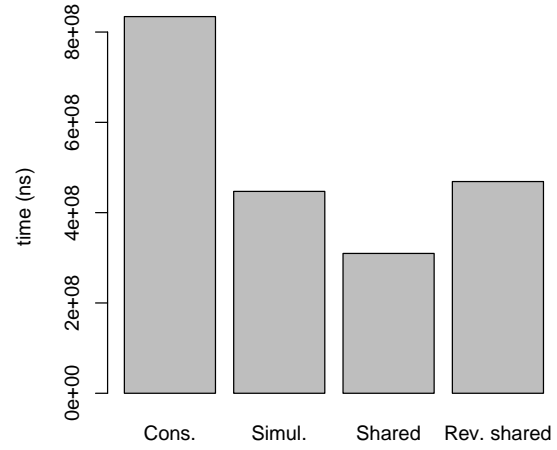


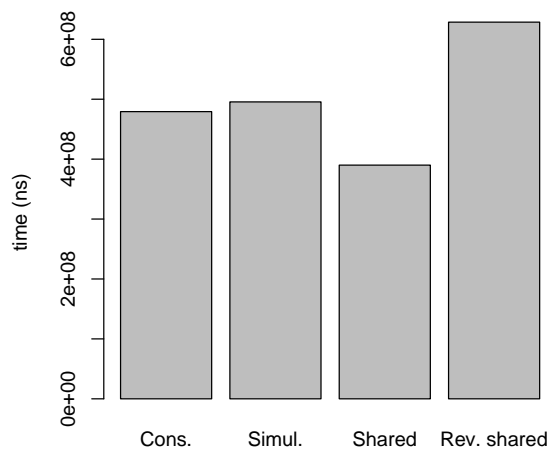
Figure 3.14: Time measurements of all approaches on a method with 2 configurations and 5 statements. Variations are easily spotted in the measurements of all approaches.



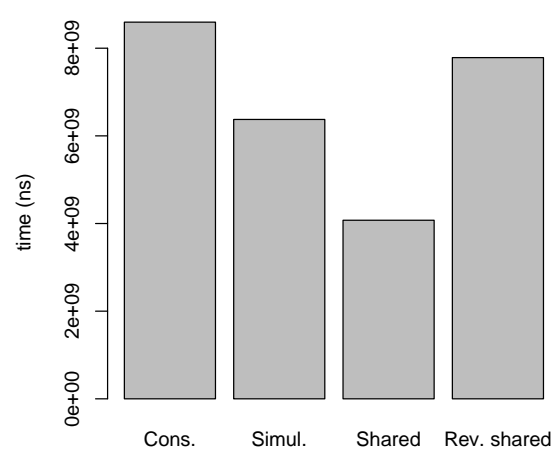
(a) GPL



(b) MM08



(c) Lampiro



(d) BerkeleyDB

Figure 3.15: Sum of medians of analysis time in the benchmarks, with the JIT compiler disabled.

important characteristics of each approach implementation. We use this information to guide our reasoning about the statistical tests and overall performance outcome.

- **Consecutive:** This approach requires one analysis computation for each configuration of the method. This means that many lattice operations and transfer functions will be executed redundantly proportional to the overlap existing between the results of the analysis for the configurations. On the other hand, the lattice used is the normal one so there is no representational overhead involved compared to the lifted lattices.
- **Simultaneous:** this approach requires a single analysis computation to analyze all configurations. Compared to the consecutive, it displaces the burden of the iterations over the configurations into the lifted lattice. A symptom of this is that there might be redundancies inside the lifted lattice, as opposed to the transfer function applications. If, for instance, there are more than several configurations mapping to the same lattice value, other operations such as union or creating a copy of the lattice will pay the price for the redundancy.
- **Shared:** this approach also requires a single analysis computation to analyze all configurations but tries to reduce the lifted lattice sizes by merging together configurations that point to the same lattice. This, in turn, reduces the cost of operations like copy, union and so on when compared to the other approaches. The performance efficiency of this approach depends on the sharing potential of the method being analyzed. That means that methods with large lattices that are shared frequently so as to avoid redundant operations will be the main beneficiaries. This approach also heavily depends on having an efficient representation for sets of configurations.
- **Reversed shared:** much like the shared approach, this one only requires a single analysis computation to analyze all configurations and tries to avoid storing redundant information by sharing equal lattices that map the different sets of configurations by merging them. Thus, the efficiency of this approach is also connected to the sharing potential of the methods. Unlike the shared approach, however, it carries the additional cost of the merge operation (cf. Section 3.5).

In the following Sections we discuss the statistical significance for each benchmark

individually. In addition to that, we also discuss the average number of methods each approach performed the best in the 10 runs of the experiment.

3.6.5.1 Benchmark 1: GPL

- Group: methods with exactly 2 configurations.
- Size: 71 methods

Figure 3.16 shows the number of times each approach was the fastest on the methods of this group. The shared approach was by far the fastest on most methods, 56, with the consecutive and simultaneous close to each other with 9 and 7, respectively. The reversed shared did not make it any faster than the others.

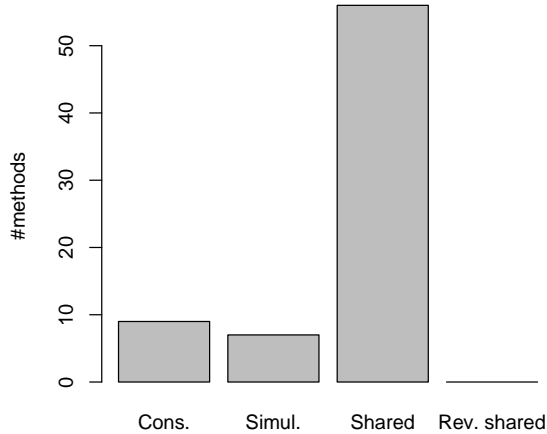


Figure 3.16: Number of times each approach performed the fastest on methods with 2 configurations in the GPL benchmark, averaged over the 10 runs.

The p-values for the Wilcoxon tests report that, on all ten executions, there is no statistically significant difference for the consecutive-simultaneous pair only. We take this as an indication that computing the analysis two times and computing it only once using the simultaneous approach is indistinguishable, time-wise.

Even for methods with only 2 configurations, the shared approach outperforms all other approaches both in occurrences, i.e., the number of times it performs better and

significantly so, i.e., there is a statistically significant difference between it and the others. We also take this as an indication that in fact, the overhead of our representation of set of configurations is low enough.

Lastly, the reversed shared approach was outperformed by all others. We take this as evidence that the reversed sharing is not beneficial enough for it to balance its shortcomings, like the necessary merge operation.

- Group: methods with exactly 4 configurations.
- Size: 20 methods

Figure 3.17 shows the average number of times each approach was the fastest. The shared approach maintains the lead as it still performs better on an average of 18 of the methods in this group. The consecutive approach was the fastest in only method on average, with the other two approaches not even making an appearance.

In this group, however, the p-values indicate that there is a statistically significant difference between the consecutive and simultaneous, which did not happen in the previous group. In fact, a closer look reveals that, the simultaneous outperforms the consecutive approach, on average, in 19 out of the 20 methods in this group. We take this as an indication that avoiding multiple computations of the analysis and using a simultaneous lifted lattice might start to pay off at four configurations.

The p-values also indicate that there is no statistically significant difference between the simultaneous and the reversed shared approach on all 10 executions. Previously outperformed by the simultaneous approach, the reversed shared shows signs of improvements as the number of configurations increases. We take this as an indication that the sharing strategy for the reversed shared implementation starts to pay off later when compared to the shared approach.

- Group: methods with > 4 configurations.
- Size: 8 methods

Figure 3.18 shows the average number of times each approach was the fastest. The shared approach outperforms all other approaches in this group, and all the reported p-values are > 0.05 .

The number of methods that fit this group for this benchmark is very small. so we must take the reported p-values with a grain salt. We refrain from building up evidence about this specific group/benchmark here.

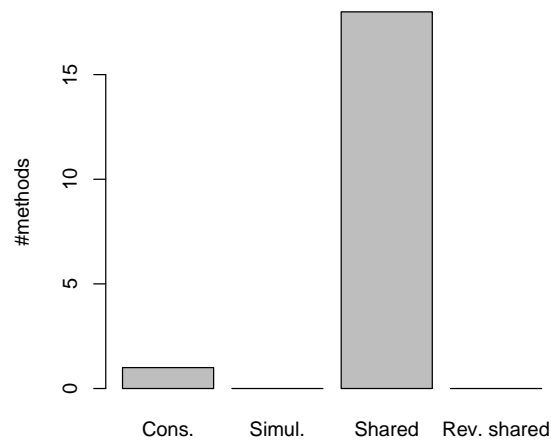


Figure 3.17: Number of times each approach performed the fastest on methods with 4 configurations in the GPL benchmark, averaged over the 10 runs.

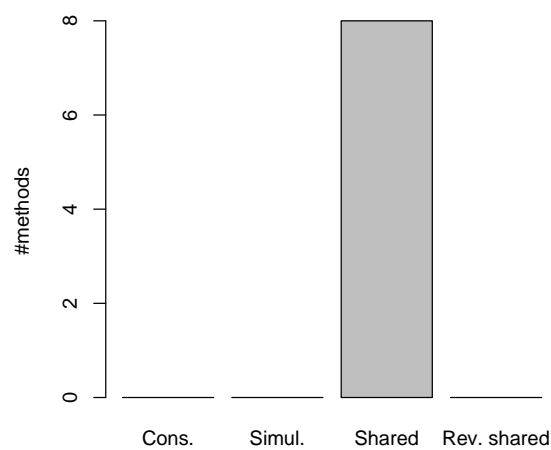


Figure 3.18: Number of times each approach performed the fastest on methods with more than 4 configurations in the GPL benchmark, averaged over the 10 runs.

3.6.5.2 Benchmark 2: MobileMedia08

- Group: methods with exactly 2 configurations.
- Size: 75 methods

Figure 3.19 shows the number of times each approach was the fastest on the methods of this group, on average. Similar to the GPL benchmark, the shared approach was again the fastest on most methods, 64, followed by the simultaneous approach being the fastest on only 6 methods and by the consecutive approach on 5. Also similar to the GPL benchmark, the reversed shared did not make it any faster than the others.

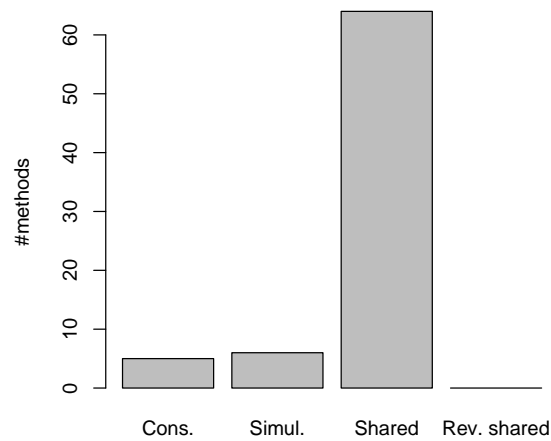


Figure 3.19: Number of times each approach performed the fastest on methods with 2 configurations in the MobileMedia08 benchmark, averaged over the 10 runs.

The p-values for the Wilcoxon tests are consistent with the GPL benchmark where the test produced a p-value > 0.05 when comparing the consecutive and the simultaneous approaches on all 10 executions of the experiment.

We take the fact that the shared approach significantly outperforms all other approaches on this group in this benchmark as further evidence that the overhead of representing sets of configurations is low and that the sharing does indeed pay off as soon as possible.

- Group: methods with exactly 4 configurations.
- Size: 12 methods

Figure 3.20 shows the average number of times each approach was the fastest. The shared approach performs better, on average, on all 12 of the methods in this group. Although a bit more drastic, this is consistent with what we see in the GPL benchmark, with the shared approach performing significantly better than the other approaches.

In this group, however, the p-values indicate that there is a statistically significant difference between the consecutive and simultaneous, similar to what happened in the GPL benchmark. Once again, a closer look reveals that, the simultaneous outperforms the consecutive approach, on average, in all 12 methods in this group. This adds up to the evidence that avoiding multiple analysis computations and using a simultaneous lifted lattice starts to pay off at four configurations.

Something similar to what happened in the GPL benchmark also happens in this one. The test indicates that there is no statistically significant difference between the simultaneous and the reversed shared approach. Again, on methods with four configurations, the reversed shared approach evens up with the simultaneous one. We also take this as further evidence that the reversed sharing technique in our implementation starts to pay off at methods with four or more configurations.

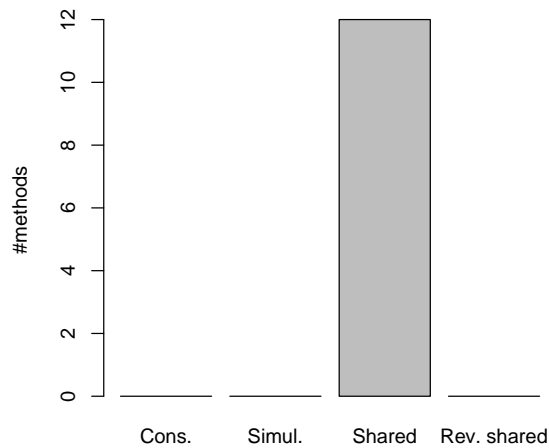


Figure 3.20: Number of times each approach performed the fastest on methods with 4 configurations in the MobileMedia08 benchmark, averaged over the 10 runs.

- Group: methods with > 4 configurations.

- Size: 11 methods

Figure 3.21 shows the average number of times each approach was the fastest. The shared approach continues to dominate, being the fastest, on average, on 10 methods, followed by the simultaneous as the fastest on one method.

The p-values indicate that there is statistically significant difference between the shared approach and all the other ones. This group has some unexpected p-values that are > 0.05 : in the simultaneous vs. the reversed shared on all ten executions and the shared vs the reversed shared on three executions.

While it is reasonable that the simultaneous and reversed shared approaches continue to be even in this group, like they were in the group with only 2 configurations, the analysis pinpoints three executions in which the shared approach evens with reversed shared seems a bit odd to say the least. A closer inspection reveals that, in all these three executions that the p-values were > 0.05 when comparing the shared and the reversed shared, the shared approach was the fastest on between 9 and 11 of the methods in this group. This is consistent with the results from the other seven executions. We find it reasonable to consider that the shared approach is superior to the reversed shared in this group because there are other seven p-values that state otherwise, and the number of occurrences in which the shared approach supersedes is consistent with these other seven executions.

When comparing only the simultaneous with the reversed shared, we found that the reversed shared approach was the fastest, on average, on 6 methods while the simultaneous was the fastest on 5.

3.6.5.3 Benchmark 3: Lampiro Lampiro only has one method with more than two configurations, so we only look at the group of methods with 2 configurations to compare the approaches. Figure 3.22 shows the average number of times each approach was the fastest on the 24 methods of this group. The shared approach is, on average, the fastest on 20 methods, followed by the simultaneous and consecutive with 2 and 1 respectively. The reversed shared approach, once again, did not make it as the fastest in any of the methods.

The only p-value reported as > 0.05 is the consecutive-simultaneous case. We take this as further evidence that, on methods characterized by this group, these two approaches are indistinguishable in terms of performance. With respect to the shared approach, the

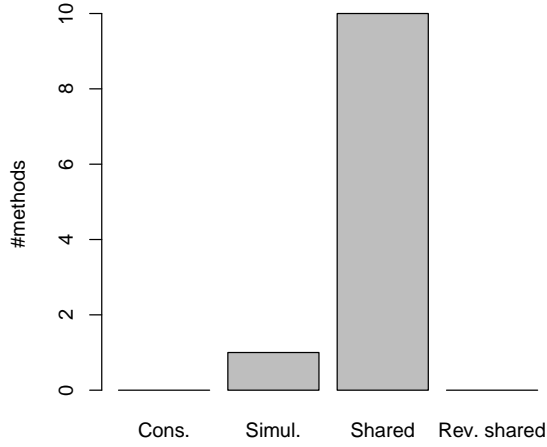


Figure 3.21: Number of times each approach performed the fastest on methods with more than 4 configurations in the MobileMedia08 benchmark, averaged over the 10 runs.

statistical tests for this benchmark add up to the evidence that the shared approach is, overall, the superior implementation, as all the other approaches performed marginally in this group of methods.

3.6.5.4 Benchmark 4: BerkeleyDB

- Group: methods with exactly 2 configurations.
- Size: 984 methods

Figure 3.23 shows the average number of methods each approach is the fastest on. The shared approach once again is by far the approach with the highest average here, being the fastest on average on 835 methods. The simultaneous, along with the consecutive are far behind, being the fastest on average on 86 and 61 methods, respectively. The reversed shared approach is on average the fastest on only 1 method.

In this group, all the p-values were > 0.05 , including the consecutive vs. simultaneous case, which had not happened in any of the other benchmarks. This provides evidence against our hypothesis that there the consecutive and the simultaneous approach are indistinguishable from each other in this group of methods, so we take a closer look at the data.

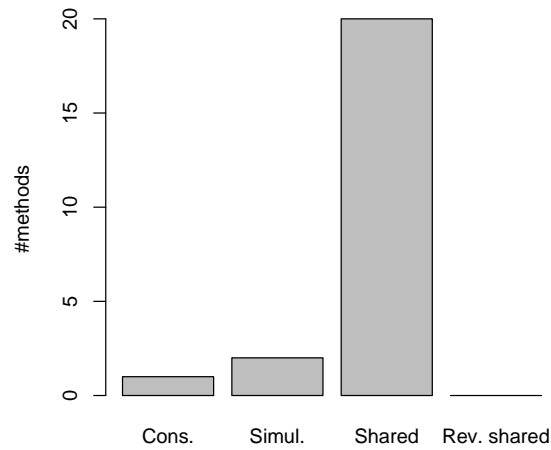


Figure 3.22: Number of times each approach performed the fastest on methods with 2 configurations in the Lampiro benchmark, averaged over the 10 runs.

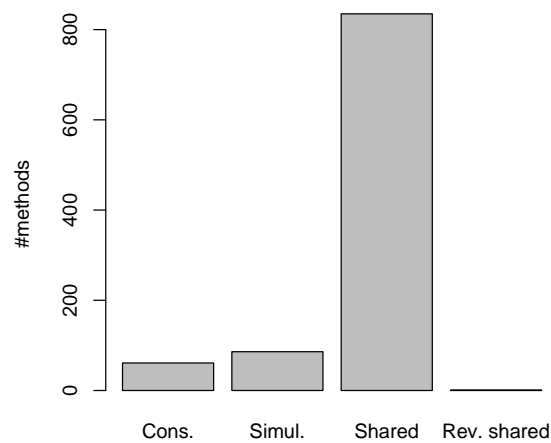


Figure 3.23: Number of times each approach performed the fastest on methods with 2 configurations in the BerkeleyDB benchmark, averaged over the 10 runs.

When comparing only the consecutive and simultaneous approach, we see that the simultaneous is the fastest, on average, on 613 methods, compared to the 371 of the consecutive approach. This fact drew our attention to the other benchmarks. Still comparing the two approaches with the methods of the same group, we found that for the GPL benchmark, the simultaneous approach is the fastest on 41 of the methods and the consecutive on 30 methods, on average. These same numbers for the MobileMedia08 benchmark are 40 and 35. Lastly, in the Lampiro benchmark the figures are 15 and 9. Thus we see that the simultaneous approach only appear to be slightly better in terms of occurrences than the consecutive approach, but not enough for the Wilcoxon test to output a p-value > 0.05 for the other benchmarks. We believe the difference between the approaches becomes significant because the BerkeleyDB benchmark has the largest pool of methods in this group, 984.

- Group: methods with exactly 4 configurations.
- Size: 180 methods

Figure 3.24 the average number of times each approach performs best on the methods of this group. The shared approach consistently outperforms all the others by a large margin, being the fastest on average on 169 methods, against 10 of the simultaneous, 1 for the reversed shared and 0 for the consecutive approaches.

In this group, all p-values are > 0.05 , including the simultaneous-reversed shared case, which also had not happened in the other benchmarks so far. When comparing only these 2 approaches for the methods in this group, we found that, on average, the simultaneous approach is the fastest on 112 methods, compared with the 68 methods for the reversed shared. This too drew our attention to the other benchmarks, and we present the numbers for them as well. They are 11 and 9 for the GPL benchmark, 8 and for the MobileMedia08 benchmark. The Lampiro benchmark only has one method in this same group. Thus we see that the simultaneous is consistently superior, although sometimes by a small margin, to the reversed shared approach. The BerkeleyDB benchmark has the largest pool of methods in this group compared to the other benchmarks. We believe that, by looking at the samples for the GPL and MobileMedia08 benchmarks, the simultaneous approach is from nothing to slightly better than the reversed shared approach for methods in this group.

- Group: methods with > 4 configurations.

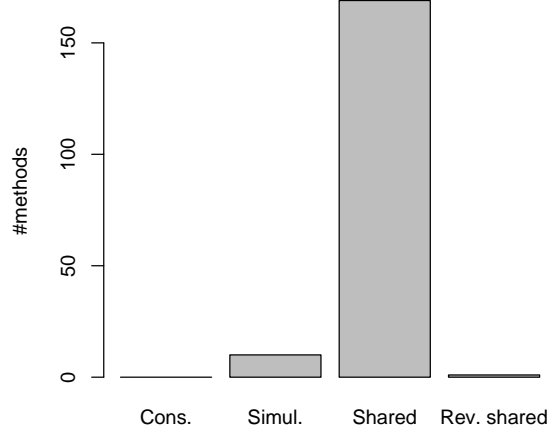


Figure 3.24: Number of times each approach performed the fastest on methods with 4 configurations in the BerkeleyDB benchmark, averaged over the 10 runs.

- Size: 180 methods

Figure 3.25 shows the average numbers of methods each approach performs best. The figures are 64 for the shared, 2 for the reversed shared, 1 for the simultaneous and 0 for the consecutive approach.

All the reported p-values are < 0.05 , in accordance to what we have seen so far in the other benchmarks, with the exception of the simultaneous-reversed shared case in the MobileMedia08 benchmark where the p-values were > 0.05 for the methods in this same group. A close look at this case reveals that the reversed shared is, on average, faster on 43 methods, compared with the 24 of the simultaneous approach.

3.6.6 Synthetic benchmarks

To further investigate the performance difference between the implementation of the approaches, we created a small benchmark that contains one class with five methods. Each of these five methods have the exact same statements that are increasingly associated with up to five distinct features. Figure 3.26 shows these methods. A method m_i has each of the i -th first statements of its body associated with i distinct features. All the features in this benchmark are optional.

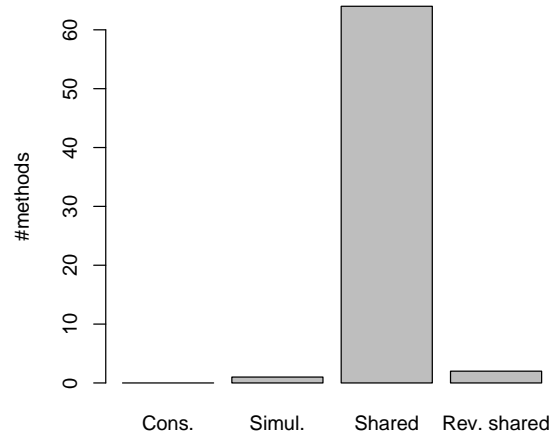


Figure 3.25: Number of times each approach performed the fastest on methods with more than 4 configurations in the BerkeleyDB benchmark, averaged over the 10 runs.

```

public class Main {
    public void m() {
        int a = 1;
        int b = a + 2;
        int c = b / 3;
        int d = b - (2*a*c);
        System.out.println(d);
    }

    public void m2() {
        int a = 1;
        int b = a + 2;
        int c = b / 3;
        int d = b - (2*a*c);
        System.out.println(d);
    }
}

    public void m3() {
        int a = 1;
        int b = a + 2;
        int c = b / 3;
        int d = b - (2*a*c);
        System.out.println(d);
    }

    public void m4() {
        int a = 1;
        int b = a + 2;
        int c = b / 3;
        int d = b - (2*a*c);
        System.out.println(d);
    }

    public void m5() {
        int a = 1;
        int b = a + 2;
        int c = b / 3;
        int d = b - (2*a*c);
        System.out.println(d);
    }
}

```

Figure 3.26: Methods in the synthetic benchmark.

We created this small benchmark so that we could focus on the number of configurations as the main factor in our performance study and because it would be hard to find a set of methods with precisely these characteristics. Because this benchmark is so small, we configured the experiment to execute 50 iterations instead of 10, like we did for the other benchmarks.

The beanplots [29] in Figure 3.27 shows what the number for the measurements of a single method, m , look like. In all cases, the highest density of values is within the smaller range of values, indicated by the bottom part of the plot. We know that there is a chance that the garbage collector, for example, might interrupt the execution of the program to perform a sweep over the objects allocated in memory. Thus, although we do not know for sure, we assume that the values indicated in the top of each plot is the result of measurements that were “tainted” by something alien to the code executing. With that in mind, we consider the medians, displayed as the thick lines in each bean, therefore, to be a reasonable choice of a summary metric for the data we see in Figure 3.27 because despite the presence of such tainted measurements, it still points to a value that is within the lower values region.

By looking at this beanplot, we can see one of the reasons the Wilcoxon test outputs different p-values for the same group of methods in the benchmarks we studied. Suppose, for example, that a measurements of the consecutive approach that are tainted never coincide with a tainted measurement of the simultaneous approach. If the sample size is small enough, this variation can build up so that the reported p-values also vary, causing it to jitter around our convention of 0.05.

Figure 3.28 shows the median of the 50 executions of each approach on the 5 methods of the synthetic benchmark. The median values for the method m show that all approaches have similar performance, with the reversed shared falling slightly behind. The median values for the $m2$ method, on the other hand, show that the reversed shared approach performed better than the consecutive one, but is still behind the shared and the simultaneous. On the method with 8 configurations, $m3$, and the method with 16 configurations, $m4$, we see that the consecutive approach does not scale nearly as good as the other ones as the number of configurations increases exponentially. Lastly, in the method $m5$, with its 32 configurations, we see that the reversed shared finally surpasses the simultaneous approach.

By fixating other variables in a method profile, such as number of statements or assignments, cyclomatic complexity etc., in this benchmark and varying only the number

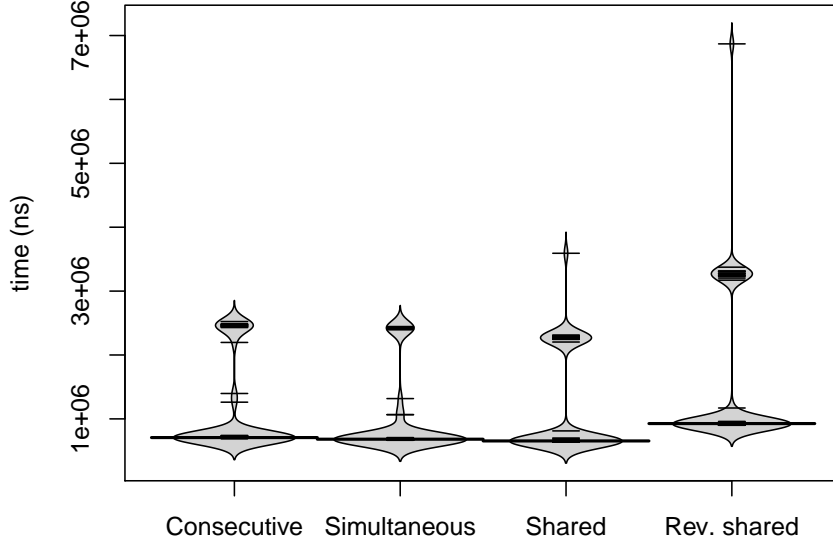


Figure 3.27: A beanplot with time measurements on method *m*.

of configurations, we were able to identify a strong relationship between the number of configurations and the performance of our implementations. Specifically, we found that, as the number of configurations increases, the shared approaches tend to perform even better, as is the case of the shared approach, or display significant improvements, in the case of the reversed shared approach. The non-shared approaches, in contrast, do not scale so well, specially the consecutive one.

Not all observations of this synthetic benchmark coincide with the benchmarks from Sections 3.6.5.1 – 3.6.5.4, however. We saw that the p-values for the consecutive vs. reversed shared case, both in GPL and MobileMedia08 benchmarks, did not allow us to reject the null hypothesis for method with 4 configurations. The same happened for the group of methods with > 4 configurations on MobileMedia08. This is not unexpected to happen as we have only been analyzing data with respect to one factor, the number of configurations. There are other characteristics that can impact the performance of the analysis, such as the number of assignments (statements for which transfer functions are defined in the reaching definitions analysis) and the number of confluence points (relates to the application of the union operation for the case of the reaching definitions analysis).

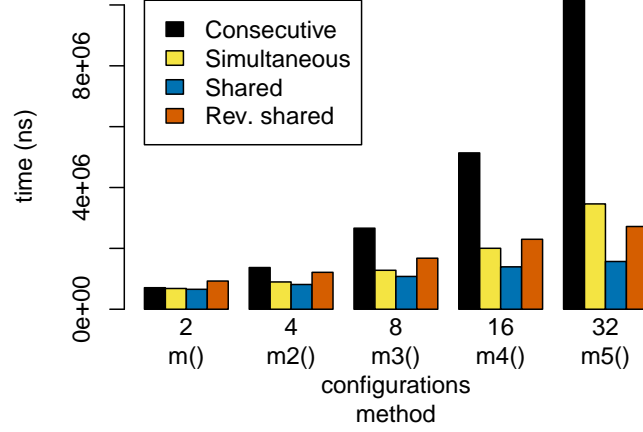


Figure 3.28: Medians over the 50 runs of each method in the synthetic benchmark.

We executed the experiment with the synthetic benchmark once again, but equally increasing the number of assignments on all methods. Figure 3.29 shows the methods of the modified synthetic benchmark. We replicated the same statements (altering local variable names) 5 times in each method, along with their respective feature association. In the Jimple representation of these methods they have 68 assignments, up from 10 in the original synthetic benchmark. For comparison, the methods in the MobileMedia08 benchmark have methods ranging from 0 to 244 assignments and the GPL benchmark from 0 to 111.

Figure 3.30 shows the medians of the 50 executions in the modified synthetic benchmark. In the *m* method, the figures are similar to that of Figure 3.28's, but the absolute values are higher across all of the approaches. We also see the difference between the simultaneous and the reversed shared approach in the *m2*, *m3* and *m4* methods increasing as the number of configuration increases. Perhaps most importantly, we see the simultaneous approach outperform the reversed shared by a large margin in the method with 32 configurations, *m5*. Additionally, the difference between the shared approach and the simultaneous is not as great compared to what we see in Figure 3.28.

```

public class Main {
    public void m(){
        int a0 = 1;
        int b0 = a0 + 2;
        int c0 = b0 / 3;
        int d0 = b0 - (2*a0*c0);
        System.out.println(d0);
        // ===
        int a1 = 1;
        int b1 = a1 + 2;
        int c1 = b1 / 3;
        int d1 = b1 - (2*a1*c1);
        System.out.println(d1);
        // ...
        int a5 = 1;
        int b5 = a5 + 2;
        int c5 = b5 / 3;
        int d5 = b5 - (2*a5*c5);
        System.out.println(d5);
        // ===
    }

    public void m2(){
        int a0 = 1;
        int b0 = a0 + 2;
        int c0 = b0 / 3;
        int d0 = b0 - (2*a0*c0);
        System.out.println(d0);
        // ===
        int a1 = 1;
        int b1 = a1 + 2;
        int c1 = b1 / 3;
        int d1 = b1 - (2*a1*c1);
        System.out.println(d1);
        // ...
        int a5 = 1;
        int b5 = a5 + 2;
        int c5 = b5 / 3;
        int d5 = b5 - (2*a5*c5);
        System.out.println(d5);
        // ===
    }

    ...

    public void m5(){
        int a0 = 1;
        int b0 = a0 + 2;
        int c0 = b0 / 3;
        int d0 = b0 - (2*a0*c0);
        System.out.println(d0);
        // ===
        int a1 = 1;
        int b1 = a1 + 2;
        int c1 = b1 / 3;
        int d1 = b1 - (2*a1*c1);
        System.out.println(d1);
        // ...
        int a5 = 1;
        int b5 = a5 + 2;
        int c5 = b5 / 3;
        int d5 = b5 - (2*a5*c5);
        System.out.println(d5);
        // ===
    }
}

```

Figure 3.29: Methods in the modified synthetic benchmark with a higher number of assignments.

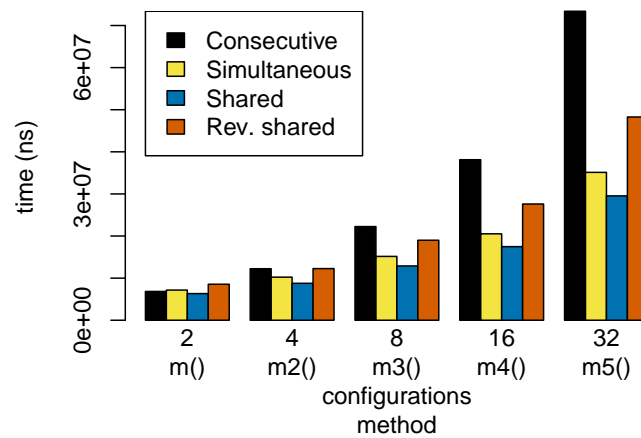


Figure 3.30: Medians of each method in the modified synthetic benchmark.

3.6.7 Evaluation summary

In this chapter we discussed the experiments performed by Brabrand et al. and report our findings as we dove deeper into the data generated by the experiments. We disabled the JIT compiler in the JVM where we executed the experiments to try and have a more controlled environment for our experiments. We found that disabling the JIT was crucial to our statistical analysis of the data, and made our exploration more consistent.

We found that the shared approach performed excellently on all scenarios we investigated, namely methods with varying number of configurations and sizes. We also found evidence that the number of configurations, although an important characteristic, is not the single definitive characteristic that one should consider when choosing which approach to use when analyzing a given method. The number of assignments in a method, as shown by the experiments with the synthetic benchmark, can have profound effects on the performance of the approaches for this specific analysis.

Of all the approaches presented, the shared one performed the best across all methods we examined in the experiments. If one must choose a single approach to implement with performance in mind, then the shared one is his choice. The consecutive analysis only managed to compete with the other approaches on methods with a small number of configurations, but has the most straightforward implementation, as it only requires the instrumented CFG and a simple form of lifted transfer functions.

3.7 THREATS TO VALIDITY

In our experiments with the four benchmarks, we separated the methods into 3 groups: methods with 2, 4 and more than 4 configurations. This allowed us to control the number-of-configurations variable in these methods. We decided to begin our exploration with this particular variable because of its exponential nature. However, in our experimental set up, there are many other variables that were not controlled, simply because the methods in the benchmarks are different. Two methods that belong to the same group can be very different methods in terms of number of statements, control flow statements, assignments and so on. In fact, we have reasons to believe that even if we take two methods that have exactly the content, that is, the same body, but change only the places where the `#ifdefs` are, the performance of the shared approach will differ on them because one of the methods might have a higher *sharing degree*, that is, the lattices might end up sharing more (or less) elements per configuration than the other. These are threats to the internal

validity of our experiments, and, because of that, we cannot assert with 100% confidence the cause-and-effect relationship between the number of configurations and the observed performance. Nevertheless, we argue that the total amount of data analysis points to the number of configurations as being one of the most influencing variable.

In addition to that, we observed that the data from experiments with JIT compiler enabled possessed variations that caused our statistical tests to produce varying results, even for the data collected from the same group of methods from the same benchmark. We tried to mitigate this problem by disabling the JIT compiler. In doing so, we achieved a lower degree of variation. Still, we can observe some variations on our data and on the reported p-values, and that is a threat to the statistical validity of our experiments. Despite of that, we argue we ameliorated this threat by running the experiment 10 times, and exploring every execution individually.

Our implementations for the feature-sensitive approaches are directly linked to our design choices. Many of these were made based on personal choices of style, others with certain goals in mind like raw performance or extensibility. On several instances we saw two approaches perform very similar to each other. It might be that a different group of engineers that makes a different set of design decisions end up with implementations that perform very different from our own. Therefore, The experiments discussed here are limited to, and only to, our own specific implementations and design choices.

Finally, we limit our hypotheses and evidence on the experiments to the single dataflow analysis which we experimented upon: the reaching definitions. We cannot generalize our observations to other analysis at this point.

CHAPTER 4

FEATURE-SENSITIVE INTERPROCEDURAL DATAFLOW ANALYSIS FOR SPL

We discussed in Section 2.4 how an important class of interprocedural dataflow problems can be defined in the IFDS or the more general IDE framework. Performing an interprocedural dataflow analysis involves analyzing (possibly) the entire program. In this chapter we discuss how the same class of dataflow problems can be lifted to perform feature-sensitive interprocedural dataflow analysis on an entire SPL.

4.1 MOTIVATION

Consider the problem of identifying an insecure flow of information, in which we want to find whether some value in a program has “leaked”. Listing 4.1, adapted from [17, Figure 1(a)], shows our target program. The standard naive approach requires that the whole SPL be preprocessed for all valid configurations. By repetitively applying the IFDS based analysis to each product we would eventually find out that for product $\neg F \wedge G \wedge \neg H$ the secret x would leak.

In the spirit of mitigating the problem of exponential blow up by features, Bodden et al. [17] proposes SPL^{LIFT} , an approach for transparently and automatically lifting IFDS-based dataflow analysis by transforming it to a more general framework. We contributed to the implementation of SPL^{LIFT} by providing our instrumentation infrastructure, discussed in Section 3.2, and assisting in the implementation of two of the three analyses used in their evaluation. In this work, however, we focus on our contribution to the evaluation of SPL^{LIFT} by reexecuting and analyzing their evaluatory experiment [5] in a different hardware configuration.

In the intraprocedural feature-sensitive dataflow analysis we discuss in Chapter 3, we exploited the fact that we only need to analyze the configurations yielded by taking the power set of the feature that are inside a method because we need not care about features that are outside the method. In interprocedural analysis, however, we cannot

```

void main() {
  int x = secret();
  int y = 0;
  #ifdef F
    x = 0;
  #endif
  #ifdef G
    y = foo(x);
  #endif
  print(y);
}

int foo(int p) {
  #ifdef H
    p = 0;
  #endif
  return p;
}

```

Listing 4.1: The target program for the insecure flow problem.

make that assumption since the analysis spans over (possibly) all methods in the SPL. This makes the interprocedural case even more interesting for feature-sensitive analysis, as performance becomes a central point.

4.2 LIFTING IFDS-BASED ANALYSES

A problem formulated within the IFDS framework can be formulated as special case of an IDE problem. The IDE framework also models flow of data through an exploded super graph. However, the IDE framework requires that a dataflow fact d map to a value v from a domain different than that of d . This domain is called the *value domain*. Thus, the transfer functions are actually *transformers* of *environments* such as $\{d \rightarrow v\}$.

A straightforward value domain is a binary one, comprised of $\{\top, \perp\}$. Each dataflow fact thus associates with one of the two values from the domain, for example $\{d \rightarrow \top\}$ or $\{d' \rightarrow \perp\}$, where the first one represents the fact that d holds at a given statement and the other represents the fact that d' does not hold.

The main goal of SPL^{LIFT} is to harness this expressiveness power of the IDE framework to associate the computation of a dataflow fact d to a feature constraint, effectively turning the value domain of an analysis into a domain which is comprised of *feature constraints*. In its essence, this idea is very similar to the one used of to build the lifted lattices of the shared approach to intraprocedural feature-sensitive dataflow analysis that is based on Kildall’s framework (cf. Section 3.4). The representational aspect in the case of the IFDS-based analysis, however, is different.

As Bodden et al write [17, page 3]:

Assume a statement s that is annotated with a feature constraint F (i.e., $\#ifdef (F) s \#endif$). Then s should have its usual data-flow effect if F holds, and should have no effect if F does not hold.

Bodden et al. describe a lifted transfer function as a combination of two functions, one that represents the *usual* dataflow effect when the feature F is enabled and the other that represents the *no effect* when the feature F is disabled. Unlike the transfer functions in the standard Kildall’s framework, there are different classes of dataflow functions used to represent intra- and interprocedural dataflow edges. The effective meaning of what usual and no effect depends on the class of the transfer function. The combination of these two functions gives rise to a lifted transfer function that is denoted as $f^{\text{LIFT}} := f^F \vee f^{-F}$, where f^F is the transfer function applied when F is enabled and f^{-F} when F is disabled.

Figure 4.1, taken from [17, Figure 4, page 4] summarizes the representations for all classes of flow functions. For *normal flow functions*, the case where F is enabled is shown in the leftmost column of Figure 4.1(a). It has the same effect as the original transfer function, but its edge is annotated with the feature F . This represents the action of generating the dataflow fact b conditionalized by the feature F . That is, the dataflow fact b will only hold when F is true. The case where F is disabled, shown in the middle column of Figure 4.1(a), represents the fact that no value is generated or killed by a non-branching statement conditionalized by a feature F if F is not enabled. This is equivalent to the identity transfer function. The resulting lifted flow function is shown in the rightmost column of the same sub-figure. Edges from the zero value, $\mathbf{0}$, to the other zero value are conditionalized by F and $\neg F$. The disjunction of both flow functions causes this edge to be conditionalized by **true**. This is represented by the solid arrow connecting the zero values. The same lifted flow function is for *call-to-return* types of flow edges.

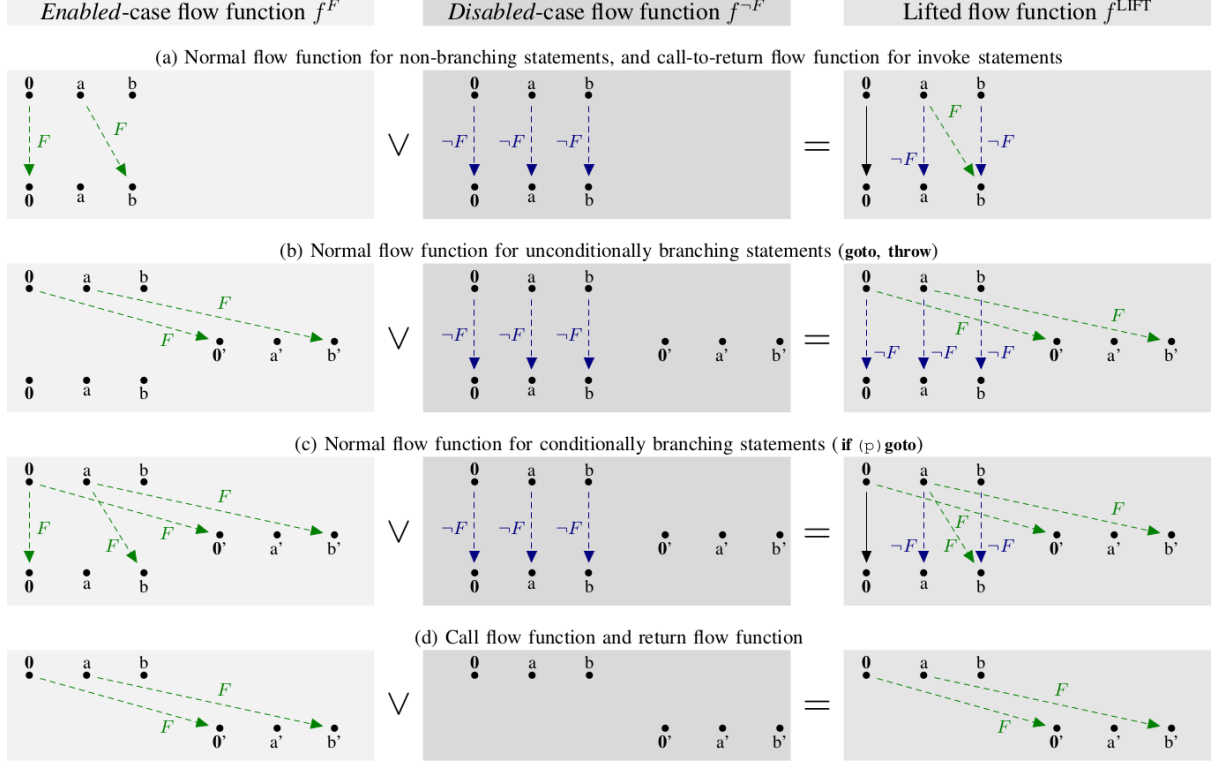


Figure 4.1: All cases of transfer function lifting by SPL^{LIFT} , from [17, Figure 4, page 4].

The flow functions for *branching statements* such as `goto` or `throw` is shown in Figure 4.1(b). The transfer function when F is enabled is shown once again in the leftmost column. Branching statements such as `goto` or `throw` are unconditional statements (not to be confused with the feature conditional statements like `#ifdef`) have no alternate flow like the usual `if` construction, where the flow of control can happen through the then-body or the else-body. Because of that, a `goto` or a `throw` statement that is conditionalized by a feature F have its only target flow conditionalized by F . The flow function when F is disabled ignores the branching to the target, and the facts flow along the fall-through branch, because the branching statements will not be executed. Thus, the resulting flow function only allows data to flow through the branching statements if when F is enabled, or the data falls through otherwise.

The case for the flow function of a *conditionally branching statement*, such as `if`, is shown in Figure 4.1(c). When F is enabled, the respective flow function, shown in the leftmost column, must allow for data to flow into the branched statement and into the fall-through statement. When F is disabled, however, the data must only flow into the fall-through. The resulting lifted flow function can be seen is a combination of the flow

functions for the non-branching statements and unconditionally branching statements, from Figures 4.1(a) and 4.1(b), respectively.

Last but certainly not least, the interprocedural call and return flow functions are shown in 4.1(d). These flow functions model interprocedural flow of data through the call and return of functions. The enabled case is modeled as the flow of facts from the call site to the statements inside the called function. However, in the disabled case, no data fact must flow into the called function because the call to that function will not execute when F is disabled.

This formulation allows for perhaps the most impressive feature of SPL^{LIFT} : the transparent reuse of the implementations of analysis. The programmer only have to implement a simple IFDS-based analysis and the SPL^{LIFT} implementation takes care of lifting it to feature-sensitivity. The only restriction for a feature-sensitive analysis is that the statements in the Jimple intermediate representation must be tagged with feature constraints.

We reproduce here, in Figure 4.2, the same Figure from [17, Figure 5], which shows the solution of the lifted taint analysis for the program shown in Listing 4.1. The topmost dataflow facts, $\mathbf{0}$, x and y , are labeled with **true**, **false** and **false**, respectively. These labels are boolean constraints that represent the environment value to which that dataflow fact is associated. The starting point of the analysis is initially associated the **true** constraint. As the analysis progresses, these starting constraints are conjoined with the feature constraints of the lifted flow functions. This ultimately means that a path between in the exploded super graph has a boolean constraint associated in which that path is feasible. This is exemplified by the possible flow of information in which the secret x would leak. By conjoining the features constraints along the path in red, we get $(\mathbf{true} \wedge \neg F \wedge G \wedge \neg H \wedge G) = \neg F \wedge G \wedge \neg H$. There also a different path that reaches the dataflow fact y , which is $(\mathbf{false} \wedge \neg G) = \mathbf{false}$. Combining these two paths we get $(\neg F \wedge G \wedge \neg H) \vee \mathbf{false} = \neg F \wedge G \wedge \neg H$. This ultimately means that under this specific constraint, the secret of x might leak.

To take the feature model into consideration when lifting this type of analysis, an elegant solution exists: translate the feature model into a boolean formula [15] m and use it as the starting value domain in entry point of the analysis, replacing the **true** constraint in the topmost dataflow fact $\mathbf{0}$ in Figure 4.2. This way, the constraints are propagated along as the flow functions are applied by the solver. However the taking-feature-model-into-consideration element it is not implemented like this in SPL^{LIFT} , because, as the

authors report, this wastes performance. Instead, they implemented this element by implicitly assuming that the feature constraint labels f in the flow functions are instead labeled as the conjunction of that feature constraint with the feature model constraint, $m \wedge f$, which ultimately yields the same analysis results and performs faster.

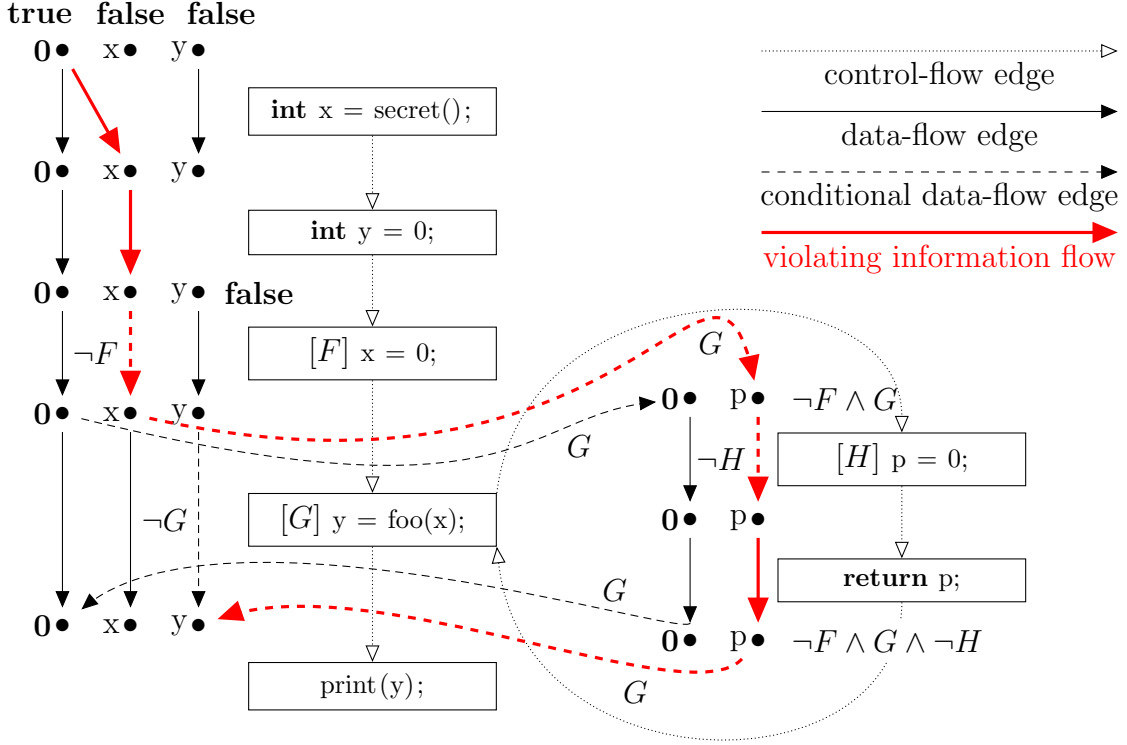


Figure 4.2: Application of the lifted flow functions of the taint analysis to the example in Listing 4.1. An insecure flow of information is highlighted in thick, red arrows. Figure taken from [17, Figure 5].

4.3 EVALUATION

In this session we present our own execution and evaluation of the experiment designed in Bodden et al. [17] to study the performance of the SPL^{LIFT} implementation.

4.3.1 Study settings

The evaluation of SPL^{LIFT} by Bodden et al. [17] focuses on three aspects: (i) correctness, (ii) efficiency and the (iii) use of the feature model in the analysis. The first aspect is

about whether the proposed approach computes a sound solution for the analyses. The second one, efficiency, relates to performance improvements with respect to the naive, feature-oblivious approach. Last but not least, the use of feature model, which concerns the cost of using the feature model in the propagation of dataflow facts during the analysis.

The correctness evaluation is addressed by comparing the results of the three dataflow problems against another IFDS-based feature-sensitive implementation. This implementation is, in essence, comparable to the consecutive approach for intraprocedural analysis that we discussed in Section 3.2. The approach implementation is simplistic and consists on iterating over all possible configurations in the SPL and applying the applicability test for each encoded transfer function; if the test fails, then the *identity* transfer function is used instead. Like the consecutive approach, this interprocedural version has a simple implementation that the authors considered foolproof. Every dataflow fact, d , computed by SPL^{LIFT} to hold under a constraint, c , must also hold in the result of the consecutive approach for that specific constraint c . The consecutive implementation is also used as a baseline for the performance comparison because the true brute force approach would require to generate and compile all possible configurations on all benchmarks. This quickly becomes intractable, as the estimates show. Even for the consecutive implementation, a cutoff-time for the experiment is required. Bodden et al. used a time limit of 10 hours per benchmark whereas we used 5 hours as our limit. We chose this limit because when we first executed the experiment as a pilot with the cutoff-time as 10 hours, all instances of our measurements that finished before the cutoff-time of 10 hours also finished before 4 hours.

The experiment evaluates the performance of the implementations by applying the interprocedural version of the uninitialized variables, possible types analyses and reaching definitions to the already known four product lines: GPL, MobileMedia08, BerkeleyDB and Lampiro. The versions of the benchmarks used in this experiment are not necessarily the same used in the evaluation of the intraprocedural implementations. For example, MobileMedia08 and Lampiro are J2ME applications and so they do not necessarily have main method. In these cases, a custom synthetic main method was used. Table 4.1 show some key interprocedural characteristics of the benchmarks. The data we obtained by reexecuting the experiment states that the number of reached features in the MobileMedia08 benchmark is different then the one reported by Bodden et al. In our execution, MobileMedia08 reaches a total of 14 different features, whereas in Bodden and colleagues' the number of reached features is 9. Consequently, this impacts the number of

Benchmark	Features		Configurations	
	total	reachable	reachable	valid
BerkeleyDB	56	39	$55 \cdot 10^{10}$	unknown
GPL	29	19	524,288	1,872
Lampiro	20	2	4	4
MobileMedia08	34	14	16,384	52

Table 4.1: Number of features, total number of configurations and valid configurations reached by the solver on each benchmark. The highlighted cells indicate discrepancies from the original experiment performed by Bodden et al.

configurations, valid or not, reached by the solver. We speculate that this might be due to how Soot unsoundly models native calls in different JRE versions, but at the time of the writing of this work, this issue has not yet been addressed.

Conceptually, in order to take the feature model into consideration, one would simply have to

The computer used to execute the experiment is the same that was used in the intraprocedural experiments (cf. Section 3.6.1): a Intel[®] Core[™] i7-3820 processor at 3.6GHz with 32GB of RAM running a Linux distribution with the 3.2.0-23-generic kernel. The JVM, however, is different: it is a Hotspot server 1.7.0_05 with 24GB maximum heap space. We use this specific JVM in this case, and not the one used in the intraprocedural experiments, because Bodden et al. packages, along with the code for executing the experiment, this specific JVM.

The hardware used by Bodden et al. is different from our own. Thus we cannot *directly* compare the our data with theirs. We expect, however, to see a similar trend.

4.3.2 Results discussion

In Table 4.2 we report the data we collected by reexecuting the experiment. The highlighted values are estimates. Due to the cutoff-time, when only a fraction of the possible configurations are iterated, the experiment instead takes average between the time taken to compute a solution for a configuration with all features enabled and the time taken to compute a solution for a configuration with all features disabled and multiply that average by the number of configurations. For BerkeleyDB, for which the number of valid

configurations is unknown, they extrapolate the results obtained within the cutoff-time. This happened on all three consecutive versions of the analysis used in BerkeleyDB and on two in GPL. Both SPL^{LIFT} and the consecutive use the same technique for constructing the call graph. In all cases, SPL^{LIFT} is faster than the consecutive implementation. The relative gain vary from benchmark to benchmark and from analysis to analysis. For example, SPL^{LIFT} is approximately 48 times faster than the consecutive in the MobileMedia08 benchmark, and approximately 1368 times faster in GPL. This is expected because both benchmarks have different characteristics, some of which directly affect the performance of the implementation we are measuring, like number of configurations. GPL has around 36 times more valid configurations than MobileMedia08. Even if we consider the unrealistic scenario where the consecutive time is equal to the cutoff-time we adopted, 5 hours, in the cases the execution was interrupted, SPL^{LIFT} still has the advantage. Most importantly, we have evidence that this trend that SPL^{LIFT} is faster than its consecutive counterpart is consistent with Bodden et al. findings.

Benchmark	Soot/CG	Possible types		Reaching definitions		Uninitialized variables	
		SPL^{LIFT}	Consecutive	SPL^{LIFT}	Consecutive	SPL^{LIFT}	Consecutive
BerkeleyDB	0:01:47.1	0:00:08.8	months	0:03:59.8	years	0:03:19.5	years
GPL	0:01:07.2	0:00:09.2	3:25:15.9	0:03:19.3	days	0:02:23.4	days
Lampiro	0:00:34.9	0:00:01.1	0:00:04.9	0:00:15.2	0:01:04.0	0:00:17.5	0:01:09.4
MobileMedia08	0:00:23.0	0:00:01.6	0:00:48.2	0:00:13.7	0:09:32.9	0:00:14.6	0:10:40.6

Table 4.2: Performance of the SPL^{LIFT} approach vs. the naive, brute-force approach. Highlighted values are estimates based on partial progress.

Disregarding the feature model when executing a feature-sensitive analysis means that work must be done on configurations that are not valid in the first place. Thus, using the feature model is essential for precision in feature-sensitive analyses, but this is not without overhead, as the feature model must be modeled and embedded in the analysis so that analyses can keep track of which configurations are valid or invalid. The experiment is programmed to report data that shows the effect of using (or not) the feature model in the analyses of the benchmarks. This is done by effectively executing the experiment with a synthetic configuration that allows for all possible configurations. Table 4.3 shows the data about this specific issue we obtained in our execution of the experiment in question. Only in Lampiro, it is possible to see the a speed up by using the feature model. In all

other benchmarks, using the feature incurs a certain overhead. In at least 3 cases, the time is virtually the same.

Benchmark	Feature model	Possible types	Reaching definitions	Uninitialized variables
BerkeleyDB	regarded	0:00:08.8	0:03:59.8	0:03:19.5
	ignored	0:00:08.8	0:03:34.3	0:03:22.5
GPL	regarded	0:00:09.2	0:03:19.3	0:02:23.4
	ignored	0:00:09.0	0:03:01.6	0:02:18.6
Lampiro	regarded	0:00:01.1	0:00:15.2	0:00:17.5
	ignored	0:00:01.7	0:00:15.1	0:00:18.5
MobileMedia08	regarded	0:00:01.6	0:00:13.7	0:00:14.6
	ignored	0:00:01.3	0:00:13.7	0:00:14.6

Table 4.3: The effects in time performance of regarding/ignoring the feature model in each benchmark.

We don’t know the variance of the data presented here because we only executed the experiment only once due to time constraints. Thus we cannot make use of statistical tools and make a more profound exploration of the data, like we do in Section 3.6. Doing multiple executions of this experiment with a corrected version of the reaching definitions analysis is scheduled for future work.

Unfortunately, at the time of the writing of this work, we discovered an error in the implementation of the reaching definitions analysis. The bug we found is a flaw in the design of its lattice. Consequently, the analysis does not compute what a reaching definitions analysis is supposed to. We still report the data we acquired from this analysis in the experiment because we believe that, for performance comparison purposes only, the performance measurements for this analysis is still useful.

4.4 THREATS TO VALIDITY

The experiments we present in this chapter are limited to a single execution, as already stated. This precludes us from applying any statistical test and strengthen the statistical threats in them. In this dissertation, we provide further data for the empirical evaluation of SPL^{LIFT} , but our data comes from the execution of the experiment on a different hardware platform and thus cannot be directly compared [44] to the data in the original

paper by Bodden et al. In this case, we argue that the difference in performance we observed with SPL^{LIFT} is enormous and therefore very convincing, in both our execution of the experiment and the one by Bodden et al.

The experiments with SPL^{LIFT} share some threats with the ones from the intraprocedural feature-sensitive dataflow analysis discussed in Chapter 3. In special, the threat to generalization over the implementation holds: the experiments presented in this chapter pertain only to this specific implementation and cannot be generalized to other IFDS-based feature-sensitive implementations that share a different set of design decisions.

The experiments with SPL^{LIFT} uses the same set of benchmarks from the experiments with the intraprocedural dataflow analyses in Chapter 3. Thus they share the same limitation as far as real-world projects go. On the bright side, the experiments on SPL^{LIFT} uses three different dataflow analyses, compared to only one in the experiments with the intraprocedural dataflow analysis in Chapter 3, a substantial increase.

CHAPTER 5

RELATED WORK

The original motivation for the concept of feature-sensitive dataflow analysis was proposed by Ribeiro et al. [48]. In this work, they highlight the need for dataflow analysis to identify dependencies between code elements, like definitions and uses of variables, that belong to different features [49]. The main goal is to help developers achieve parallel, independent and modular development of features by means of *Emergent Interfaces* (EI). EI are the results of the calculation of dependencies between code elements that emergent on-demand for the developers, to help them understand and maintain variability in the source code. Later, our collaboration with Ribeiro et al. [50, 51] resulted in the implementation of *Emergo*, a tool that implements the concept of EI as an Eclipse IDE [1] plug-in.

Predicated dataflow analysis [43] is a technique where propositional logic predicates are associated with dataflow values to generate what the authors call optimistic dataflow values. These predicates are used to keep several versions of analysis distinct from each other during analysis. This is, in essence, similar to the intraprocedural shared simultaneous we discuss in Chapter 3. The predicates, however, are over dynamic state, whereas the feature-sensitive approach uses feature constraints, which are all known at analysis time.

The concept of tagging statements that may or may not execute in a static analysis is related to path-sensitive dataflow analysis. The tagged information is then used to compute possibly diverging information along different paths of the statically simulated execution. This allows the analysis to ignore unrelated or irrelevant information from infeasible paths [14]. The same idea of tagging statements was explored by [56] in an attempt to improve precision in constant propagation optimizations. The main idea in this work is to identify and exclude dead statements that would otherwise hinder the performance of such analysis.

The work on SPL^{LIFT} , which we discuss in Chapter 4, is an extension to the exploratory ideas by Bodden [16]. SPL^{LIFT} improves on this earlier work by providing full support for IFDS analysis, instead of only taint analysis. Additionally, the work on SPL^{LIFT}

introduced the lifting on all classes of flow functions, which also had not be done, along with a more efficient BDD-based implementation for feature constraints. Last but not least, the work that proposed SPL^{LIFT} provides an experimental evaluation on four benchmarks, which we also used to evaluate our intraprocedural feature-sensitive analyses proposal.

TypeChef [36, 37] is a tool that can parse C source-code with variability encoded in `#ifdef` without actually preprocessing the source-code. Notably, it can parse the entire Linux kernel, without requiring a restricted form of `#ifdefs`, which we used throughout this work. Based on our work on feature-sensitive dataflow analysis, a subproject of TypeChef has recently implemented intraprocedural dataflow analysis for SPLs written in C. They experimented with their implementation on two real-world case studies: the Busybox tool suite and the Linux kernel. In their experimentation, however, they found mixed results, some feature-sensitive analysis were faster on one case study and slower on the other.

Thüm et al. performed a survey on analysis strategies for SPLs [55], that encompassed type checking [12, 31], parsing [36], model checking [20, 19], and verification [46, 39, 13]. Although the surveyed work does not include feature-sensitive dataflow analysis for SPL, it shares with the works discussed in Chapters 3 and 4 the general goal of efficiently computing properties about a SPL and mitigating the exponential blow-up problem commonly found in SPLs.

Kim et al. explores the idea of using dataflow analysis to identify features that are (not) reachable from a given test case [27]. This allows for developers to only execute the test cases for products that contain these features. The dataflow analysis used in this work, however are not feature-sensitive. Instead they use a custom analysis that relies on having part of the variability represented using conditional statements, not conditional compilation.

Safe composition is a body of work that focuses on the safe generation of products and verification of properties of SPLs. Its purpose is to provide some guarantees about generated products. Kästner et al. [32] proposed the Color Featherweight Calculus, later improved by Delaware et al. [23], which aims at guaranteeing that no products can be generated with typing errors. Dataflow analysis can also be used to infer properties about the type of variables. In this sense, the feature-sensitive approaches to dataflow analysis discussed in this work relates to that of type checking in the essence of computing properties about all products in a SPL without the need to explicitly generating all of them.

CHAPTER 6

CONCLUDING REMARKS

In this work we described how two frameworks for performing dataflow analysis can be *lifted* to perform what we call *feature-sensitive dataflow analysis*. Feature-sensitivity is a characteristic of a dataflow analysis that is able to take variability encoding statements, such as `#ifdefs`, into consideration when performing the analysis. This allows for the computation of dataflow facts that are associated with feature constraints. The major benefit of doing feature-sensitive analysis is that we can avoid the explicit generation of the variability. In the case of `#ifdefs`, we can avoid having to use brute-force to preprocess the source-code for each possible configuration of preprocessor variables.

In Chapter 3 we discuss how the Kildall-based [38] standard intraprocedural dataflow analysis can be lifted in four different ways to perform our feature-sensitive dataflow analysis [18]. In our work, presented the implementations and reevaluated the performance of these approaches by lifting the ubiquitous reaching definitions analysis and applying it to four qualitatively different benchmarks. We provide evidence that the number of possible configurations and the size of the body of the method being analyzed can have a profound impact on the performance of our approaches. We also provide evidence that one of the implementations performs generally better than the others in the situations we investigated. We learned that some implementations are more complex than others. Specifically, the implementation that we found to perform the worst on method with high number of configurations also has the most straightforward implementation. We used statistical analysis to back up our claims and evidences, although not all scenarios were fully explored.

In Chapter 4, we discuss SPL^{LIFT} [17], an implementation of an IFDS/IDE [52] that can transparently lift any IFDS-based dataflow analysis to feature-sensitivity. The IDE framework, which is generalization of the IFDS framework, allows for precise interprocedural dataflow analysis. In our work, we reexecuted the evaluatory work in [17] in a different hardware platform and present our observations. We provide further evidence that SPL^{LIFT} indeed outperforms the traditional brute-force approach by several orders of

magnitude.

6.1 SUMMARY OF CONTRIBUTIONS

In this work, we presented the following main contributions that are based on the work of Brabrand et. al (cf. Chapter 3):

- Implementation of the feature instrumentation process; and Soot/CIDE integration;
- Implementation of all four feature-sensitive approaches to intraprocedural dataflow analysis;
- Idealization of the consecutive feature-sensitive intraprocedural dataflow analysis;
- Implementation and (re)execution of the performance experiments;
- Statistical testing and in-depth exploratory study in the performance data; and
- Experimentation on the synthetic benchmarks.

In Chapter 4 we present our contributions to the work of SPL^{LIFT}, namely:

- Support and implementation for the Soot/CIDE integration;
- Implementation of two out of the three interprocedural dataflow analysis: reaching definitions and uninitialized variables;
- Reexecution and analysis of the evaluatory experiments;

6.2 LIMITATIONS

The proposals by Brabrand et al. [18] and Bodden et al. [17] in which we build upon in this work are not without limitations. In both cases the tool chain is limited to using CIDE as the preprocessor technology. CIDE is not a currently active project. At the time of the writing of this work, the last commit is almost one year old [4]. Also in both cases, the implementations are limited to the implementations within Soot and bound to the Jimple intermediate representation. The authors of SPL^{LIFT}, however, have made an effort to turn the IFDS/IDE solver into generic, Jimple-free implementation [6]. Soot, on the other hand, is has a very active community. In fact, during the development of this

```

Set<T> s;
#ifdef F
s = new HashSet<T>();
#endif
#ifdef F'
s = new TreeSet<T>();
#endif

```

Listing 6.1: The code in listings results in an imprecise call graph in SPL^{LIFT}.

work, we found and reported a bug in the implementation of the equals/hashCode contract in the lattice implementation of Soot. The bug fix was discussed in the mailing list and applied within a matter of days [3].

Our evaluation of the intraprocedural feature-sensitive dataflow analysis is based on a single dataflow analysis: reaching definitions. Because different analysis have different lattices domains and methods have different number of elements of these domains, we cannot generalize our findings to other kinds of analysis. Specifically, we see in Section 3.6.6 that the increase the number of assignments in a method can cause variations in the performance of the analysis.

Another limitation of SPL^{LIFT} is that the call graph is not constructed in a feature-sensitive manner. This yields sounds results, but not necessarily precise solutions to dataflow facts. For example, consider the code snippet in Listing 6.1. The variable `s` is initialized to the `HashSet` iff F is enabled and to `TreeSet` iff F' is enabled. Because the call graph construction itself is not feature-sensitive, this case would be modeled as having exactly two unconditionalized calls to the each of the constructors.

6.3 FUTURE WORK

We evaluated our implementations of intraprocedural feature-sensitive dataflow analysis using only the reaching definitions analysis. We intend to reexecute the experiments on different analysis, such as uninitialized variables, busy expressions and so on. In the experimental evaluation, we focused on the number of configurations as our main variable. We briefly explored the number of statements in a method and immediately saw that this variable can also affect the performance of our implementations in a significant manner.

The empirical evaluation of the interprocedural feature-sensitive dataflow analyses consists of a single execution of the experiment. Because of this, we do not know what is, for example, the variance between several runs. So far, we have two executions of the experiment, but on different computers with different hardware platforms. We intent reexecute the same experiment many times in the future in order to be able to apply meaningful statistical analysis on the data.

We learned during the writing of this work that the implementation of the interprocedural feature-sensitive reaching definitions analysis is flawed. We will investigate this matter in the near future and reexecute the experiment with a correct implementation of the reaching definitions analysis.

Last but not least, we performed the interprocedural analysis on a synthetic main method on two of the four benchmarks. In the future, we intend to explore different main methods in an attempt to increase the coverage of methods analyzed.

BIBLIOGRAPHY

- [1] Eclipse.org Home, January 2008. <http://www.eclipse.org/>.
- [2] Soot: a Java Optimization Framework, April 2010. <http://www.sable.mcgill.ca/soot/>.
- [3] AbstractFlowSet equals/hashCode contract issues, 2012. <https://github.com/Sable/soot/issues/11>.
- [4] CIDE GitHub page, 2013. <https://github.com/ckaestne/CIDE>.
- [5] Download materials related to SPLlift, February 2013. <http://www.bodden.de/research/current/spl/spllift/>.
- [6] Heros: Multi-threaded, language-independent IFDS/IDE solver, 2013. <http://sable.github.com/heros/>.
- [7] The R Project for Statistical Computing, February 2013. <http://www.r-project.org/>.
- [8] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 2nd edition, 2006.
- [9] Vander Alves. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Federal University of Pernambuco, Recife, Brazil, March 2007.
- [10] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *LNCS*, pages 70–81. Springer-Verlag, September 2005.
- [11] Michalis Anastasopoulos and Cristina Gacek. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*, pages 109–117, New York, NY, USA, 2001. ACM Press.

- [12] Sven Apel, Christian Kästner, Armin Grösslinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Eng.*, 17(3):251–300, September 2010.
- [13] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proc. 2011 26th IEEE/ACM int. conf. on Automated Software Engineering, ASE '11*, pages 372–375, 2011.
- [14] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proc. 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '01*, pages 97–103, 2001.
- [15] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines (SPLC'05)*, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.
- [16] Eric Bodden. Position Paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines. In *Proc. ACM SIGPLAN 7th Workshop on Programming Languages and Analysis for Security (PLAS 2012)*, June 2012. To appear.
- [17] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift - transparent and efficient reuse of ifds-based static program analyses for software product lines. In *Proc. of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Seattle, USA, 2013. To appear.
- [18] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development X*, 2013. To appear, awaiting publication.
- [19] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proc. 33rd int. conf. on Software Engineering, ICSE '11*, pages 321–330, 2011.
- [20] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal

- properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10) - Volume 1*, pages 335–344, New York, NY, USA, 2010. ACM.
- [21] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [22] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.
- [23] Benjamin Delaware, William R. Cook, and Don S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 243–252. ACM, 2009.
- [24] R. Drechsler and B. Becker. *Binary decision diagrams: theory and implementation*. Springer, 1998.
- [25] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, December 2002.
- [26] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proc. 2011 int. symp. on Software Testing and Analysis, ISSTA '11*, pages 177–187, 2011.
- [27] Chang Hwan, Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proc. 10th int. conf. on Aspect-oriented Software Development (AOSD'11)*, pages 57–68, 2011.
- [28] Pedro Matos Jr. Analyzing techniques for implementing product line variabilities. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2008. To be finished.
- [29] Peter Kampstra. Beanplot: A Boxplot Alternative for Visual Comparison of Distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, October 2008.

- [30] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.
- [31] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proc. 2008 23rd IEEE/ACM int. conf. on Automated Software Engineering, ASE '08*, pages 258–267, 2008.
- [32] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*, pages 258–267. IEEE Computer Society, September 2008.
- [33] Christian Kästner and Sven Apel. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.
- [34] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.
- [35] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 311–320, New York, NY, USA, 2008. ACM.
- [36] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. 2011 ACM int. conf. on Object oriented programming systems languages and applications, OOPSLA '11*, pages 805–824, 2011.
- [37] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. 2011 ACM int. conf. on Object oriented programming systems languages and applications, OOPSLA '11*, pages 805–824, 2011.
- [38] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '73*, pages 194–206, New York, NY, USA, 1973. ACM.

- [39] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing configurations to monitor in a software product line. In *1st International Conference on Runtime Verification (RV)*, volume 6418 of *LNCS*. Springer, November 2010.
- [40] Duc Le, Eric Walkingshaw, and Martin Erwig. `#ifdef` confirmed harmful: Promoting understandable software variation. In *IEEE International Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, pages 143–150, 2011.
- [41] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 105–114, New York, NY, USA, 2010. ACM.
- [42] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceeding of the 10th International Conference on Aspect Oriented Software Development (AOSD'11)*, pages 191–202, New York, NY, USA, 2011. ACM.
- [43] Sungdo Moon, Mary W. Hall, and Brian R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proc. 12th int. conf. on Supercomputing*, ICS '98, pages 204–211, 1998.
- [44] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009.
- [45] Thomas Patzke and Dirk Muthig. Product Line Implementation Technologies. Technical Report 057.02/E, Fraunhofer Institut Experimentelles Software Engineering, October 2002.
- [46] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. 2008 23rd IEEE/ACM int. conf. on Automated Software Engineering, ASE '08*, pages 347–350, 2008.
- [47] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. 22nd ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, POPL '95, pages 49–61, 1995.

- [48] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent Feature Modularization. In *Onward!, affiliated with ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'10)*, pages 11–18, New York, NY, USA, 2010. ACM.
- [49] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE'11)*, pages 23–32, Portland, Oregon, USA, 2011. ACM.
- [50] Márcio Ribeiro, Társis Toledo, Paulo Borba, and Claus Brabrand. A tool for improving maintainability of preprocessor-based product lines. In *Tools Session of the 2nd Brazilian Congress on Software (CBSOFT'11)*, 2011.
- [51] Márcio Ribeiro, Társis Toledo, Johnni Winther, Claus Brabrand, and Paulo Borba. Emergo: A tool for improving maintainability of preprocessor-based product lines. In *Proceedings of the 11th International ACM Conference on Aspect-Oriented Software Development (AOSD'12), Companion, Demo Track*, pages 23–26. ACM, 2012.
- [52] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT '95*, pages 131–170, 1996.
- [53] Henry Spencer and Geoff Collyer. `#ifdef` considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.
- [54] Sun Microsystems. Memory Management in the Java HotSpot Virtual Machine, April 2006. <http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>.
- [55] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, April 2012.
- [56] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.

- [57] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.