A STATIC ANALYSIS TO EXTRACT DATAFLOW EDGES FROM OBJECT-ORIENTED PROGRAMS WITH OWNERSHIP DOMAIN ANNOTATIONS

by

SUHIB RAWSHDEH

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2011

MAJOR: COMPUTER SCIENCE

Approved by:

Advisor	Date

DEDICATION

To my family

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support of many people.

First and foremost, I thank God Almighty for his endless blessings he has bestowed upon me. I couldn't stop praising Him for all the good things He has done and brought in my life

I must thank my queen and my wife Ola for her endless love and support. You have been my inspiration for all of this. You were always next to me. My love for you is eternal and unconditional. I wish to express my love and gratitude to my beloved parents Nazem Rawshdeh and Najah Al-Masri. Without you non of this could have been possible. To my brothers Dr. Bilal and Mohammad Rawshdeh, thanks for all of your support and encouragement. To my beloved sister Dr. Sarah Rawshdeh, thanks for your kind words and prayers. You have always encouraged me to do my best. God could not have blessed me with a better family. To my parents in law, Jameel and Bassema, and their family, thanks all for your support.

I would like to thank my advisor, Prof. Abi-Antoun for his invaluable advice that walked me all the way long to finish this thesis. Thanks for the pair reviewing and for returning my drafts as soon as possible just so I could make it to the deadline. Indeed without your support this thesis would not have been possible. I'm also thankful to Dr. Marcus and Dr. Fisher, my thesis committee, for your guidance and comments. Any remaining faults in this thesis are mine alone.

Finally, I would like to thank my best friends who have become brothers to me over the years. Special thanks to Dr. Mohammad Albanna and Mahmoud Najeh. I could not list all my wonderful friends, but I would like to thank you all.

TABLE OF CONTENTS

Dedication ii

Acknowledgements iii

List of Figures vi

Chapter 1: Introduction 1

1.1 Background on Scholia 3

1.2 Requirements 3

1.3 Contributions 4

1.4 Thesis Statement 5

1.5 Outline 6

Chapter 2: Object Graph Extraction 7

2.1 Introduction 7

2.2 Dataflow Definition 7

2.3 Ownership Domain Annotations 11

2.4 OGraph Extraction 15

2.5 Advanced Features 28

2.6 Discussion 31

Chapter 3: Formalization of the Analysis 33

3.1 Introduction 33

3.2 OGraph formalization 35

3.3 Constraint-Based Specification 36

3.4 Soundness 42

3.5 Credits 43

Chapter 4: Evaluation 44

4.1 Implementation 44

4.2 Listeners System 44

4.3 Banking System 45

4.4 CourSys System 48
4.5 Information Flow Example 50
4.6 Summary 53
Chapter 5: Related Work 54
5.1 Static vs. Dynamic analysis 54
5.2 Applications of Dataflow Information 58
5.3 Information Flow Control 59
Chapter 6: Discussion 60
6.1 Validation of Thesis Statement 60
6.2 Satisfaction of the Requirements 60
6.3 Limitations 61
6.4 Future Work 63
6.5 Conclusion and Broader Impact 63
References 65
Abstract 71
Autobiographical Statement 73

LIST OF FIGURES

Figure. 2.1 Export and Import edges. Adapted from Spiegel's Pangaea [41] . 8

Figure. 2.2 Dataflow example. 9

Figure. 2.3 Dataflow edges. 10

Figure. 2.4 Listeners: code with ownership domain annotations. 12

Figure. 2.5 Listeners: code with concrete annotations. 13

Figure. 2.6 Datatype declarations for the OGraph. 15

Figure. 2.7 Notation for DfOOG. 16

Figure. 2.8 Abstractly interpreting the program. 17

Figure. 2.9 Listeners: class diagram. Retrieved by ObjectAid UML [35]. . . 18

Figure. 2.10 Abstractly interpreting the program (continued). 19

Figure. 2.11 Abstractly interpreting the program (continued). 21

Figure. 2.12 Listeners: object graph without dataflow edges. 22

Figure. 2.13 Abstractly interpreting the program (continued). 24

Figure. 2.14 Abstractly interpreting the program (continued). 25

Figure. 2.15 Abstractly interpreting the program (continued). 27

Figure. 2.16 Abstractly interpreting the program (continued). 28

Figure. 2.17 Abstractly interpreting the program (continued). 29

Figure. 2.18 Listeners: object graph with dataflow edges. 30

Figure. 2.19 QuadTree example with annotations. 30

Figure. 2.20 Abstractly interpreting the QuadTree class. 31

Figure. 2.21 QuadTree example OGraph. 32

Figure. 3.1 Simplified FDJ abstract syntax [8]. 34

Figure. 3.2 Datatype declarations for the OGraph. 34

Figure. 3.3 Constraint-based specification of the OGraph. 38

Figure. 3.4 Constraint-based specification of the OGraph (continued). 39

Figure. 3.5 Instrumented runtime semantics (core rules). 41

Figure. 4.1 Listeners: Dataflow vs. Points-to edges. 46

Figure. 4.2 BankingSystem DfOOG. 47

Figure. 4.3 BankingSystem: Spiegel's Pangaea flat object graph. 48

Figure. 4.4 CourSys DfOOG. 49

Figure. 4.5 Information Flow example. Adapted from Liu and Milanova [31] 51

Figure. 4.6 Information Flow example using annotations. 52

Figure. 4.7 Information Flow example OGraph. 53

Chapter 1: Introduction

Software maintenance accounts for 50% to 90% of the costs over the lifestyle of a software system. And program comprehension is a major activity during software maintenance, absorbing around half of the maintenance costs [10]. In order to support program comprehension, software researchers have produced many tools to visualize the structure of a system. The prevalent thinking is that diagrams help with program comprehension. For example, a high-level diagram can help a developer locate where to implement a change. Similarly, a visual inspection of the dependencies among entities may hint at the magnitude of the ripple effects of implementing a change [45].

One common design diagram for object-oriented code is the class diagram. A class diagram shows the type structure of the program. Class diagrams are widely adopted and well supported by tools [26]. Although class diagrams are useful, they do not explain the object structure of a program. For that purpose, we need an additional diagram, the *object diagram*.

In an *object diagram* or an *object graph*, the nodes represent objects, i.e., instances of the classes in a class diagram, and the edges correspond to various kinds of relations between objects. An object diagram makes explicit the structure of the objects instantiated by the program and their relations, facts that are only implicit in a class diagram. While in the class diagram a single node represents a class and summarizes the properties of all of its instances, an object diagram represents different instances as distinct nodes, with their own properties [45].

In object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For instance, in the Observer design pattern [19], understanding "what" gets notified during a change notification is crucial for the operation of the system, but "what" does not usually mean a class, "what" means a particular instance. Furthermore, a class diagram often shows several classes

depending on a single container class such as `ArrayList`. However, different instantiations of an `ArrayList` often correspond to different elements in the design. Hence, we need an instance-based view to *complement* a class diagram. For example, in the Design Patterns book [19], Gamma et al. used *both* class and object diagrams to explain several structural design patterns such as Proxy, Mediator and Composite.

Historically, object diagrams have had less mature tool support and less widespread use than class diagrams. Until recently, few tools extracted object diagrams, and the ones that did so extracted flat object graphs [25, 48], which do not scale to an entire program. Abi-Antoun and Aldrich had recently proposed the Scholia approach to extract hierarchical object graphs from an object-oriented code [4, 2]. But, the Scholia object graph showed only one kind of relation between objects, namely points-to edges due to field references.

For program comprehension, developers often require many sources of information such call graphs [28, 17], points-to graphs [33] or shape graphs [40], among others. For example, call graphs are widely used to enable developers to understand call interactions between different parts of the code [28]. However, most call graph tools do not track objects, to avoid an exponential blowup in the number of call paths to display.

In this thesis, we follow the same strategy as Scholia to extract hierarchical object graphs. However, our object graphs have a different kind of edge, namely dataflow edges, which we define to be usage edges which show the flow of objects in the program. Going back to the above example of the Observer pattern, we would be able to see not just "what" gets notified, but also "what kind" of notification is sent from the publisher of the notification to its subscribers.

1.1 Background on Scholia

In this section, we summarize Scholia [4, 2], on which we base our analysis. We then discuss the differences between our static analysis and Scholia.

Scholia is the state of the art in the static extraction of sound, hierarchical runtime architectures. Scholia statically extracts a hierarchical object graph from an object-oriented code with Ownership Domain (OD) annotations [8]. According to OD (See Section 2.3), an *ownership domain* is a conceptual group of objects. Developers use annotations to specify to which domain an object belongs by annotating the object's references with the name of its owner domain. All references of an object across the code must have the same annotation that refers to the domain containing that object. Developers use a typechecker to check and validate that the annotations are consistent with each other and with the code.

The object graph Scholia extracts is hierarchical in that it collapses low-level objects underneath more architecturally-relevant objects. To achieve this hierarchy, Scholia uses ownership domain annotations [8] in the code. However, Scholia's static analysis extracts only points-to edges on the object graph. While points-to edges are useful, developers need to obtain different types of relations between objects on the object graph such as dataflow relations [15]. Therefore, we propose a static analysis to extract a hierarchical object graph showing dataflow edges.

1.2 Requirements

We selected the following two quality attributes for our analysis, soundness and precision.

1.2.1 Soundness

Soundness in the context of our analysis consists of object soundness and edge soundness. Object soundness means that each runtime object must have exactly one representative object in the object graph. Edge soundness means that if there is a dataflow between two objects at runtime, then our analysis would show a corresponding edge between the representatives of these objects in the extracted object graph.

1.2.2 Precision

Our analysis should be precise and raise few false positives (i.e. showing dataflow edges that could never exist at runtime). One major problem with attempting to achieve precision and soundness simultaneously is that it is often almost impossible to maintain precision while achieving soundness. A sound static analysis often runs the risk of generating overly conservative approximations. For example, extracting a graph with one node or showing a fully connected graph would be sound, but not useful. Our aim is to increase precision while maintaining soundness.

1.3 Contributions

The contributions of this thesis are the following:

- A static analysis to extract a hierarchical object graph with dataflow edges from object-oriented programs with ownership domain annotations.
- A worked example of the static analysis, illustrating interesting cases.
- A formalization of the static analysis using constraint-based inference rules.
- An evaluation of the static analysis on realistic object-oriented code to evaluate its precision in practice.

1.4 Thesis Statement

The thesis statement is:

> A static analysis can extract sound and precise dataflow edges from an object-oriented program with ownership domain annotations.

We create three hypothesis subordinate to the main thesis statement. Since each hypothesis is smaller that the main thesis, each can be directly supported by evidence.

> H1: A static analysis can extract a sound hierarchical object graph with dataflow edges from an object-oriented program with ownership domain annotations.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include the following:

1. The analysis extracts dataflow edges that are consistent with dataflow-based communication occurring in the program as determined by a code inspection.

Evidence. We support this hypothesis with the following evidence:

1. A formal definition of the analysis using constraint-based inference rules.
2. A formal proof of soundness (outside the scope of this thesis).
3. An implementation of the analysis and an evaluation on realistic object-oriented code showing that extracted object graph make visually obvious all of the dataflow communication in a program.

> H2: A static analysis can extract precise dataflow edges from an object-oriented program with ownership domain annotations.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include the following:

1. The analysis extracts dataflow edges that are true positives.

Evidence. We support this hypothesis with the following evidence:

1. An implementation of the analysis and an evaluation on realistic object-oriented code showing that extracted object graph does not have a large number of edges that do not correspond to real dataflow communication occurring in the program.

2. The extracted object graphs are not conservative over-approximations.

1.5 Outline

The rest of this thesis is organized as follows: Chapter 2 describes our analysis informally. Chapter 3 formalizes our analysis. Chapter 4 presents an evaluation of our analysis on four realistic Java examples. Chapter 5 reviews related work. We conclude the thesis with a discussion (Chapter 6).

Chapter 2: Object Graph Extraction

2.1 Introduction

In this chapter, we informally describe our algorithm to extract hierarchial object graphs with dataflow edges. We follow SCHOLIA [4] in the way it extracts an object graph from a program annotated with ownership domain types. However, our algorithm produces different kind of edges to connect the extracted runtime objects on the object graph. While SCHOLIA uses points-to edges for this purpose, our algorithm connects graph objects using dataflow edges. Some applications require dataflow edges in addition to or instead of points-to edges that correspond to field reference relations. Dataflow edges indicate when some reference may propagate from one object to another.

In Section 2.2, we explain the dataflow algorithm our analysis uses and the dataflow edges it extracts. In Section 2.3, we briefly review the ownership domain annotations our analysis uses and the underlying type system. Section 2.4 presents our static analysis and explains the analysis on a realistic Java example. We discuss some advanced features (Section 2.5) and conclude this chapter with a discussion (Section 2.6).

2.2 Dataflow Definition

In this section, we explain the dataflow definition we use in our analysis.

Dataflow with annotations. The object aliasing policy given to us by using ownership domain annotations increases the precision of our extracted dataflow edges on the object graph. According to ownership domains, two objects that are in two different domains may not alias; so the analysis can keep them separate and consequently

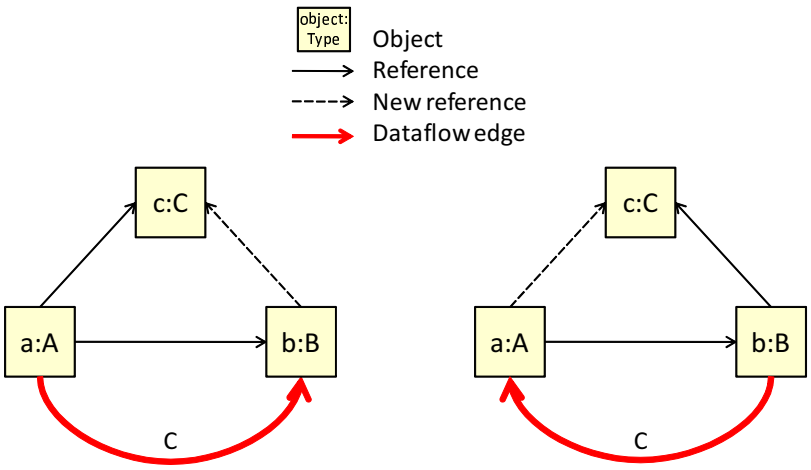


Figure 2.1: Export and Import edges. Adapted from Spiegel's Pangaea [41]

keep separate dataflow edges associated with them. Another analysis which does not assume the presence of ownership annotations is going to have to do additional analysis to separate these objects by some sort of inference. Failure to separate these objects leads to imprecision on the extracted dataflow edges.

Export and Import dataflow edges. Our analysis follows the state-of-art definition of dataflow. It shows dataflows between objects when object references are propagated from one object to another in the program. We identify two different kinds of dataflow scenarios and two kinds of dataflow edges that correspond to these scenarios (Fig. 2.1). The definition of export and import edges are adapted from Spiegel's Pangaea system [41].

In the object graph, export and import edges illustrate the scenarios of object references being propagated due to method invocations or field accesses[1].

Export edge definition An export scenario means that object a owns a reference to object c and passes it to object b. In the object graph, our analysis adds an export edge from a to b, annotated by the type of c, C if:

[1]Array access may also result in dataflow edges. We omit this from our discussion, but we handle array access in the implementation.

```
class Test<OWNER> {
  domain ADOM
  public A<ADOM> oA;
  public void test() {
    oA.ma();
  }
}
class A<OWNER> {
  domain BDOM, DDOM
  public B<BDOM,DDOM> oB;
  public D<DDOM> oD;
  public void ma() {
    oB.mb(oD);
  }
}
class B<OWNER,DDOM> {
 domain CDOM
 public C<CDOM,DDOM> oC;
 public D<DDOM> oD2;
 public void mb(D<DDOM> oD) {
   oD2 = oC.mc(oD);
 }
}
class C<OWNER,DDOM> {
 public D<DDOM> mc(D<DDOM> oD) {
    return oD;
 }
}
class D<OWNER> {}
```

Figure 2.2: Dataflow example.

1. Object a invokes one of b's methods and passes c as one of the invoked method's arguments.

2. Object a writes one of object b's fields by assigning c to it.

Import edge definition An import scenario means that object a owns a reference to object b in which it receives a reference to object c. In the object graph, our analysis adds an import edge from b to a, annotated with the type of c, C if:

1. Object a calls one of b's methods and receives a reference of object c that object a may not previously know about.

2. Object a reads one of object b's fields, and this field is object c.

We further illustrate these dataflow scenarios with a code example (Fig. 2.2). The invocation of the method mb on the receiver object (i.e., $recv$) oB and context object (i.e., O_{this}) oA causes oA to export object oD to object oB. Edge E_1 represents this dataflow (Fig 2.3). Object oB further exports oD to its final destination object oC by calling the method mc on oC (Edge E_2). The same call of mc then returns back oD to oB (Edge E_3). This code illustrates the different scenarios of objects export and import. Figure 2.3 illustrates the results of the three dataflow scenarios.

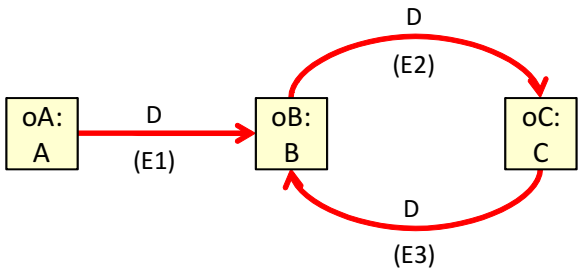


Figure 2.3: Dataflow edges.

Special Cases We are confident that our algorithm captures many important dataflow edges on our extracted object graph. However, there are some cases concerning library classes such as containers that we handle differently. Container classes such as List, Vector, Set, and others provide their clients with public methods to insert and extract objects in and out of them. A container insertion or lookup operation is responsible for a dataflow edge between the container object and its element objects. In the case of insertion, we encounter a dataflow edge from the object passed to the container's insert method (e.g. add, put, etc.) to the container object itself. When the container's extract method is used to retrieve an object that was previously inserted into the container, we encounter a dataflow edge from the container object to the object retrieved. These edges are hard to capture because they reflect the natural usage of these containers and not the way these containers' methods and fields are being invoked. For example, according to the definition of import and export edges, the invocation of the method `add` on an `ArrayList` collection object `l`, $l.add(o)$ does not result in any import or export edges between object l and object o.[2]

In order to handle these cases, we introduced the following special annotation:

$$@Dataflow(\{\text{“}o \mapsto recv\text{”}\})$$

The above annotation tells our analysis to capture dataflow edges associated with

[2]we assume that all the container's elements are ending up getting merged in one object in one domain.

the container. This annotation is lightweight and needs only to be assigned once on the container's method declaration. The annotation has only one parameter which identifies the dataflow edge to be added to the object graph. This annotation is similar to virtual dataflows or function summaries [3]. Notice that object *o* in the above annotation may have different meaning depending on the context in which the container's method is invoked (See Figures 2.13, 2.14 in Sec. 2.4).

2.3 Ownership Domain Annotations

Our static analysis assumes that the program to be analyzed has ownership domain annotations. These annotations must be added manually to the program before the analysis can begin. In this section, we review the annotations and explain why we chose the ownership domains type system and not some other ownership type system. For a more thorough discussion of Ownership Domains, please refer to Aldrich and Chambers [8].

Ownership domain. An *ownership domain* is a conceptual group of objects that may alias. In ownership domains, an object is owned by exactly one domain, and one object can have several domains to own its substructure. Developers use annotations to specify to which domain an object belongs by annotating the object's reference with the name of its owner domain. All references of an object across the code must have the same annotation that refers to the domain containing that object. It is the type system responsibility using its typechecker to check and validate these annotations.

Annotation syntax. In this thesis, we use the same annotation syntax as the one used in Aldrich and Chambers [8]. Each object reference is annotated with a list of domain parameters, where the first parameter indicates the owning domain (Fig. 2.4). The concrete syntax used in the implementation is different (Fig. 2.5) but tends to

```
1  abstract class Listener<OWNER> {  }
2  class List<OWNER, T<ELTS>> {
3    public add(T<ELTS> l) {  }
4  }
5  class Msg<OWNER> {  }
6  class MsgMtoV<OWNER> extends Msg<OWNER> {  }
7  class MsgVtoM<OWNER> extends Msg<OWNER> {  }
8  class BaseChart<OWNER,M> extends Listener<OWNER> {
9    // M is a domain parameter bounds to the domain DOCUMENT in class Main
10   private domain OWNED; // Declare private domain OWNED
11   public domain DATA;
12   // The first OWNED annotation is for the list object
13   // List has domain parameter ELTS for its elements
14   // Annotation M is bound to List's ELTS for the list elements
15   List<OWNED, Listener<M>> listeners = new List<OWNED, Listener<M>>();
16
17   // A public method CANNOT return a reference to listeners in private domain
18   // public List<OWNED, Listener> getListeners() { return listeners; }
19   public addListener(Listener<M> l) { listeners.add(l); }
20   public notifyObservers() {  }
21 }
22 class BarChart<V, M> extends BaseChart<V, M> {  }
23 class PieChart<V, M> extends BaseChart<V, M> {  }
24 class Model<D, V> extends Listener<D> {
25   private domain OWNED;
26   public domain DATA;
27   // listeners object is encapsulated
28   // Annotation V (i.e., VIEW) is bound to the
29   // List's ELTS for the list elements
30   List<OWNED, Listener<V>> listeners = new List<Listener<V>>();
31
32   public addListener(Listener<V> l) {
33     listeners.add(l);
34   }
35   public notifyObservers() {
36     MsgMtoV<DATA> mTOv = new MsgMtoV<DATA>();
37     Listener<V> l = listeners.value;
38     l.update(mTOv);
39   }
40 }
41 class Main<OWNER> {
42   public domain DOCUMENT, VIEW;
43   Model<DOCUMENT, VIEW> model = new Model<DOCUMENT, VIEW>();
44   BarChart<VIEW, DOCUMENT> barChart = new BarChart<VIEW, DOCUMENT>();
45   PieChart<VIEW, DOCUMENT> pieChart = new PieChart<VIEW, DOCUMENT>();
46
47   public void run() {
48     model.addListener(barChart);
49     model.addListener(pieChart);
50     barChart.addListener(model);
51     pieChart.addListener(model);
52
53     model.notifyObservers();
54     barChart.notifyObservers();
55     pieChart.notifyObservers();
56   }
57 }
```

Figure 2.4: Listeners: code with ownership domain annotations.

```
@Domains({"OWNED"})
@DomainParams({"M"})
class BaseChart extends Listener {
  @Domain("OWNED<M>") List<Listener> listeners = new List<Listener>();
}
```

Figure 2.5: Listeners: code with concrete annotations.

be more verbose [2].

Declaring a domain. In OD, a domain is declared at the class level. A developer can declare for one class several domains to convey architectural intent and to group the class's internal objects. A domain can be either *private* or *public* to govern different objects access policies. We follow the convention of using capital letters for domain names to distinguish them from other program identifiers which usually do not use capital letter names.

Private domains. A private domain provides *strict encapsulation* i.e., objects in a private domain are strongly encapsulated inside the object which owns that private domain. For example, the `listeners` object in the private domain `OWNED` is encapsulated inside class `BaseChart` (Fig. 2.4). A programmer cannot declare a public method that returns the list `listeners` in the private domain `OWNED`. Using a private domain is stronger than using the Java visibility modifier private which provides only a name-based protection and does not provide strict encapsulation. So, if we were to uncomment the method `getListeners` which returns a reference to the list `listeners` (Lines 17-18 in Fig. 2.4), the typechecker will produce a warning.

Public domains. A public domain provides *logical containment*. The right to access an object implies the right to access its public domains and the objects inside them. For example, inside the class `BaseChart`, we could have declared a public domain and put the object `listeners` inside it. Then, any object that has access to either `barChart` or `pieChart` objects, which are subclass objects of class `BaseChart`),

will also have access to their `listeners` objects. Additionally, if we were to uncomment the method `getListeners` (Lines 17-18 in Fig. 2.4), the typechecker will not produce a warning.

Our analysis distinguishes between the `listeners` object inside `barChart` from that inside `pieChart`, and as a result, shows them as two distinct objects in the object graph. See Section 2.4. In our approach, we selected ownership domains because it allows developers to express their design intent in code. Several other type systems [14, 13, 12] support representing ownership of objects in code. However, unlike ownership domains, they assume a single domain per object. Ownership domains allow multiple domains per object. Also, the ownership domains type system is more expressive than other ownership type systems because it supports both strict encapsulation and logical containment.

Top-level domains. Similarly to SCHOLIA [4], our analysis assumes that the program operates by creating a main object. The class of the main object declares no domain parameter. We refer to domains declared by this class as top-level domains.

Domain parameters. Domain parameters propagate ownership information of objects outside the current encapsulation. They allow objects to share state. For example, the class `BarChart` (Fig. 2.4) needs to access objects in the `DOCUMENT` domain declared in the class `Main`. To do so, class `BarChart` declares the domain parameter `M`. When class `Main` instantiates the object `barChart`, it binds the domain `DOCUMENT` to the domain parameter `M` in class `BarChart`. As a result, instances of the class `BarChart` can access objects of type `Listener` i.e., `model`, inside the domain `DOCUMENT` (line 15).

Domain parameters are inherited. For example, each of `BarChart` and `PieChart`, which are subclasses of `BaseChart`, binds its domain parameter `M` to the domain parameter `M`, inherited from `BaseChart`.

$$
\begin{array}{ll}
G \in \mathsf{OGraph} & ::= \langle\ \textbf{Objects},\ \textbf{Domains},\ \textbf{Edges}\ \rangle \\
 & ::= \langle\ DO,\ DD,\ DE\ \rangle \\
D \in \mathsf{ODomain} & ::= \langle\ \textbf{Id} = D_{id},\ \textbf{Domain} = C{::}d\ \rangle \\
 & ::= \langle\ D_{id}, C{::}d\ \rangle \\
O \in \mathsf{OObject} & ::= \langle\ \textbf{Id} = O_{id},\ \textbf{Type} = C{<}\overline{D}{>}\ \rangle \\
 & ::= \langle\ O_{id}, C{<}\overline{D}{>}\ \rangle \\
E \in \mathsf{OEdge} & ::= \langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst},\ \textbf{DataType} = C\ \rangle \\
 & ::= \langle\ O_{src},\ O_{dst},\ C\ \rangle
\end{array}
$$

Figure 2.6: Datatype declarations for the OGraph.

2.4 OGraph Extraction

Our static analysis extracts an object graph (OGraph) that approximates any possible runtime object graph due to different program executions. An OGraph has two types of nodes, OObjects and ODomains. OObjects correspond to runtime objects, and ODomains correspond to runtime domains. The analysis subsequently extracts and adds dataflow edges (OEdges) between runtime objects. The datatype declarations are in Fig. 2.6. An OGraph may have cycles[3].

Graphical notation. Our analysis extracts Dataflow Ownership Object Graph (DfOOG), we graphically distinguish between objects and domains by using a yellow-filled rectangle-shape to represent an object and a white-filled rectangle-shape to represent a domain. We further distinguish between public and private domains using a thin dashed border for a public domain and a bold dashed border for a private domain. A thick and red arrow represents a dataflow edge. In all cases, we label each rectangle with the name of the object or domain that corresponds to it (Fig. 2.7).

[3]Our analysis handles recursive types which produce cycles on the OGraph (Sec. 2.5)

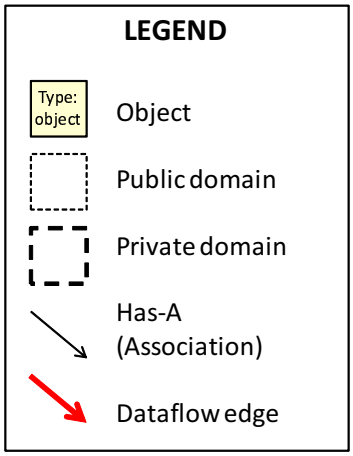


Figure 2.7: Notation for DfOOG.

Aliasing strategy. The analysis does not require an alias analysis and relies instead on ownership domain annotations to control object aliasing. In our analysis, an OObject type is a pair of its class and list of domain parameters (Fig. 2.6). The analysis distinguishes between objects in different domains and merges objects of compatible types if they are in the same domain. This means that objects in different domains cannot alias, whereas objects in the same domain may alias.

2.4.1 Abstract interpretation

The static analysis abstractly interprets the program statements and produces OObjects, ODomains, and OEdges. We illustrate the analysis in a small example, Listeners (Fig. 2.4). We use the following notation to fully qualify objects and domains. This notation is based on Abi-Antoun's dissertation [2].

Notation. We use the following notation in the abstract interpretation:

1. *obj*.*DOM* refers to either a public or a private domain *DOM* inside object *obj*, e.g., `main.DOCUMENT`. It effectively treats a domain as a field of an object;

2. *obj*1.*DOM*.*obj*2 refers to the object *obj*2 inside the domain *DOM*, e.g., `main.DOCUMENT.model`;

```
OObject(main, Main<SHARED>) (O0) 1
Main<SHARED> main = new Main();
analyze(main, [Main::OWNER ↦ SHARED])
recv ↦ main, O_this ↦ main 2
class Main<OWNER> {
  public domain DOCUMENT, VIEW;
  ODomain(main.DOCUMENT, Main::DOCUMENT) (D1) 3
  ODomain(main.VIEW, Main::VIEW) (D2) 4

  OObject(main.VIEW.barChart, BarChart<main.VIEW, main.DOCUMENT>) (O1) 5
  BarChart<VIEW, DOCUMENT> barChart = new BarChart...();
  analyze(barChart, [BarChart::M ↦ main.DOCUMENT, BarChart::OWNER ↦ main.VIEW],
  recv ↦ main.VIEW.barChart, O_this ↦ main) 6

  OObject(main.VIEW.pieChart, PieChart<main.VIEW, main.DOCUMENT>) (O2) 13
  PieChart<VIEW, DOCUMENT> pieChart = new PieChart...();
  analyze(pieChart, [PieChart::M ↦ main.DOCUMENT, PieChart::OWNER ↦ main.VIEW],
  recv ↦ main.VIEW.pieChart, O_this ↦ main) 14
  // The analysis is similar to barChart, omitted for brevity

  OObject(main.DOCUMENT.model, Model<main.DOCUMENT, main.VIEW>) (O3) 15
  Model<DOCUMENT, VIEW> model = new Model...();
  analyze(model, [Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT],
  recv ↦ main.DOCUMENT.model, O_this ↦ main) 16
  ...
}
```

Figure 2.8: Abstractly interpreting the program, starting with the root class `Main`.

3. *obj*....*DOM* refers to a public domain. The ownership domain type system allows path-dependent annotations that are of the form `obj1.obj2...DOM`, where `obj1`, `obj2`, ..., are chains of final fields or variables, and `DOM` is a public domain declared on the type of the last object in the path;

4. *C*::*d* refers to a domain *d* qualified by the class *C* that declares it.

2.4.2 Listeners Example

We illustrate our analysis on a Listeners system. The class diagram is in Fig. 2.9. The system consists of one `Model` class and two chart classes, class `BarChart` and

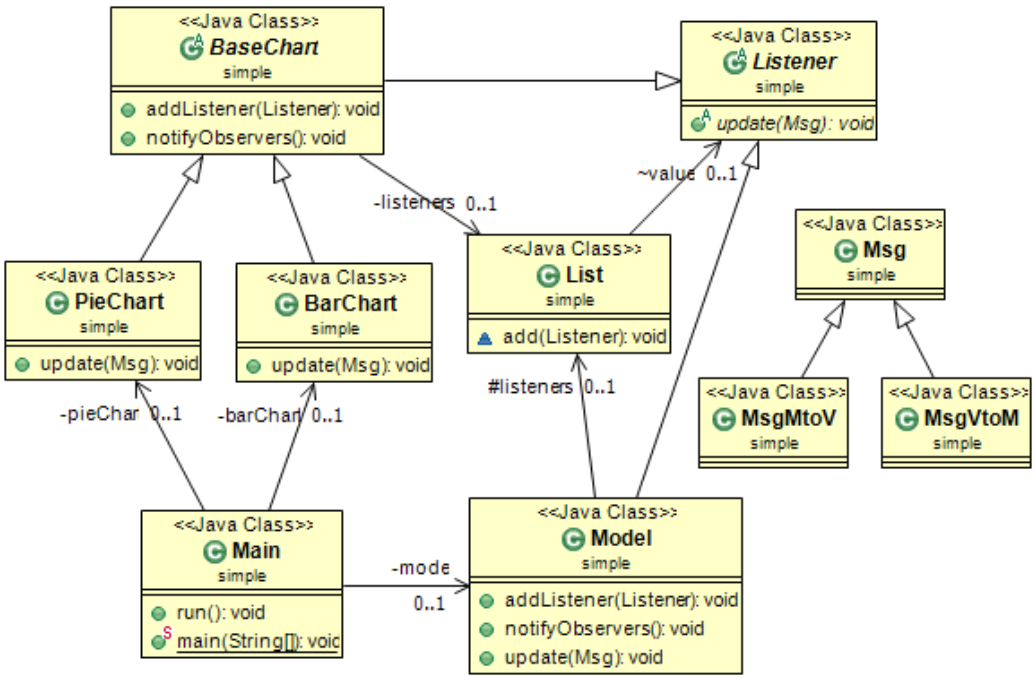


Figure 2.9: Listeners: class diagram. Retrieved by ObjectAid UML [35].

`PieChart`, both extend class `BaseChart`. The Listeners system implements the Observer design pattern [19] where both `BaseChart` and `Model` classes extend class `Listener`. An object of type `Model` (i.e., the subject) may register objects of type `BaseChart` (i.e., the observers), and vice versa.

In the following discussion, we numbered the steps of the analysis on the right hand side of the program statements. The elements highlighted in rectangles represent analysis elements and they are not part of the program code. Our analysis starts with the user selecting a root type, in this case, the `Main` class (Fig 2.8). First, the analysis creates OObject (O0) for the root object allocation, main. Then, it analyzes class `Main` in the context of OObject (O0) (Fig 2.8).

Before the analysis analyzes class `Main`, it binds all formal domain parameters, if any, to their corresponding ODomains in the OGraph. In this case, the analysis binds the special owner domain parameter `MAIN::OWNER` on class `Main` to the global domain `SHARED`. The analysis keeps track of two important pieces of information which are the `recv` OObject and the context OObject O_{this}. These two OObjects are important

```
[BarChart::M ↦ main.DOCUMENT, BarChart::OWNER ↦ main.VIEW]
recv ↦ main.VIEW.barChart
O_this ↦ main
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  analyze(barChart, [BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW],
  recv ↦ main.VIEW.barChart, O_this ↦ main) 7

  public void update(Msg<LENT> msg) {...}
}
[BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW]
recv ↦ main.VIEW.barChart
O_this ↦ main
class BaseChart<OWNER, M> extends Listener<OWNER> {
  public domain OWNED;
  ODomain(main.VIEW.barChart.OWNED, BaseChart::OWNED) (D3) 8

  private domain DATA;
  ODomain(main.VIEW.barChart.DATA, BaseChart::DATA) (D4) 9

  OObject(main.VIEW.barChart.OWNED.listeners,
  List<main.VIEW.barChart.OWNED, Listener<M>) (O4) 10
  List<OWNED, Listener<M>> listeners = new List...();
  analyze(listeners,
  [List::ELTS ↦ main.DOCUMENT, List::OWNER ↦ main.VIEW.barChart.OWNED],
  recv ↦ main.VIEW.barChart.OWNED.listeners, O_this ↦ main.VIEW.barChart) 11
}
[List::ELTS ↦ main.DOCUMENT, List::OWNER ↦ main.VIEW.barChart.OWNED]
recv ↦ main.VIEW.barChart.OWNED.listeners
O_this ↦ main.VIEW.barChart
T = Listener
class List<OWNER, T<ELTS>> {

  private domain OWNED;
  ODomain(main.VIEW.barChart.OWNED.listeners.OWNED, List::OWNED) (D5) 12
  T<ELTS> value;
  ...
}
```

Figure 2.10: Abstractly interpreting the program (continued): `BarChart`, `BaseChart` and `List`.

because they help the analysis identify the type of dataflow edges to add to OGraph (Sec 2.2).

The analysis continues inside class `Main` and finds that the first statement is a domain declaration. The analysis processes this statement by creating two ODomains (D1) and (D2) that correspond to domains `DOCUMENT` and `VIEW`, respectively. The analysis then encounters an object allocation statement for object `barChart` inside domain `VIEW`. It creates OObject `barChart` (O1) and then proceeds to analyze class `BarChart` in the context of the OObject `barChart`. Here, the analysis binds the formal domain parameter `BarChart::M` to the ODomain `main.DOCUMENT` in class `Main`, and binds the special domain parameter `BarChart::OWNER` to the ODomain `main.VIEW` in class `Main`, which is the owning domain of `barChart`.

The analysis analyzes the class `BarChart` and its superclass `BaseChart` in the context of the OObject `barChart` (Fig. 2.10). Since class `BarChart` is empty, the analysis proceeds into the superclass `BaseChart`. Inside `BaseChart`, the analysis encounters two domain declarations `OWNED` and `DATA`. As a result, it creates the two ODomains (D3) and (D4). Then, the analysis creates OObject `main.VIEW.barChart.OWNED.listeners` (O4) inside `barChart`'s `OWNED` domain for `List<Listener>`.

While analyzing the class `List<Listener>` in the context of the receiver (i.e., `recv`) OObject `listeners` and O_{this} `barChart`, the analysis encounters only one domain declaration statement. Therefore, it creates the ODomain `main.VIEW.barChart.OWNED` (D5), then returns back to the `Main` class to continue analyzing the rest of the program statements.

The analysis of class `PieChart`, its superclass `BaseChart`, and its `List` is similar to that of `BarChart`, so we omitted it for brevity. The resulting OObjects and ODomains from the analysis are shown in Fig. 2.12.

In reference to class `Main` (Fig. 2.8), the analysis encounters an object allocation

```
[Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT]
recv ↦ main.DOCUMENT.model
O_this ↦ main
class Model<OWNER, V> extends Listener<OWNER> {
  private domain OWNED;
  ODomain(main.DOCUMENT.model.OWNED, Model::OWNED) (D9) 17

  public domain DATA;
  ODomain(main.DOCUMENT.model.DATA, Model::DATA) (D10) 18

  OObject(main.DOCUMENT.model.OWNED.listeners,
  List<main.DOCUMENT.model.OWNED, Listener<V>>) (O6) 19
  List<OWNED, Listener<V>> listeners = new List...();
  analyze(listeners,
  [List::ELTS ↦ main.VIEW, List::OWNER ↦ main.DOCUMENT.model.OWNED],
  recv ↦ main.DOCUMENT.model.OWNED.listeners, O_this ↦ main.DOCUMENT.model) 20
}
[List::ELTS ↦ main.VIEW, List::OWNER ↦ main.DOCUMENT.model.OWNED]
recv ↦ main.DOCUMENT.model.OWNED.listeners
O_this ↦ main.DOCUMENT.model
T = Listener
class List<OWNER, T<ELTS>> {

  private domain OWNED;
  ODomain(main.DOCUMENT.model.OWNED, Model::OWNED) (D11) 21

  T<ELTS> value;
  ...
}
```

Figure 2.11: Abstractly interpreting the program (continued): `Model` and `List`.

statement for object `Model` inside domain `DOCUMENT`. Therefore, it creates OObject `model` (O3) and then proceeds to analyze class `Model` in the context of OObject `model` (Fig. 2.11). While analyzing the class `Model`, the analysis creates two ODomains (D9) and (D10) corresponding to the declared domains `OWNED` and `DATA` in class `Model`, respectively (Fig. 2.12). Then it creates OObject `listeners` (O6) and analyzes the class `List<Listener>` in the context of the OObject `main.DOCUMENT.model.OWNED.listeners`.

In Fig. 2.13, the analysis encounters the first method invocation statement in

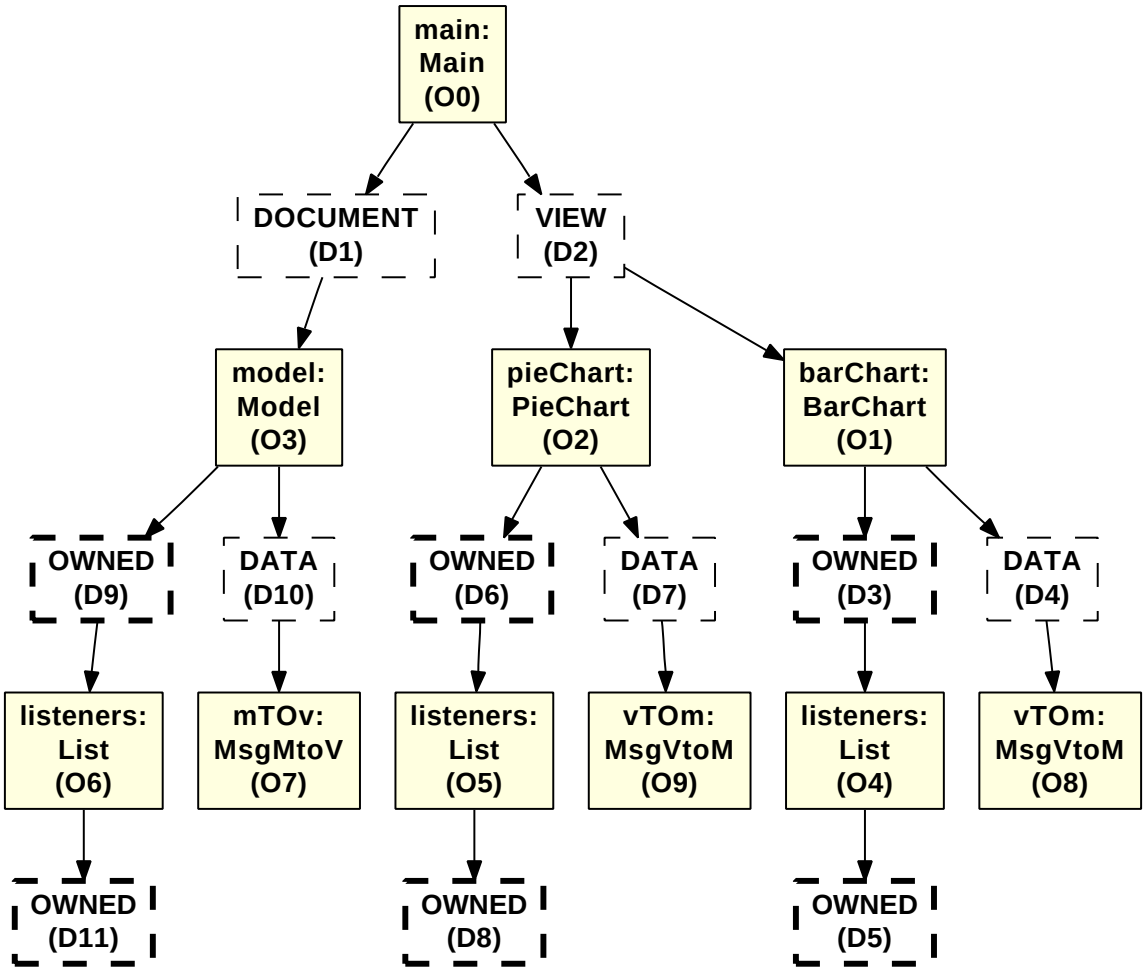


Figure 2.12: Listeners: object graph without dataflow edges.

the program, namely the `run` method. The user can select this root method when selecting the root class. However, for simplicity, we can select the `run` method by default. Later, when processing this `run` method or any other method, the analysis tracks the method's receiver and context OObjects. In this case, both `recv` and O_{this} correspond to OObject `main`.

In Fig. 2.13, inside the `run` method, the analysis encounters the first method invocation statement `model.addListener(barChart)`. It analyzes this method invocation in the context of O_{this} OObject `main` and the receiver OObject `main.DOCUMENT.model`. According to our dataflow rules (Sec. 2.2), this method invocation introduces an export dataflow edge OEdge (E1) from OObject `main` to

OObject `model` because OObject `main` exports OObject `barChart` to OObject `model` as a method argument.

Inside the method `addListener` in class `Model` (Fig. 2.14), the method invocation `listeners.add(l)` introduces another dataflow edge from the context OObject `model` to the method receiver OObject `main.DOCUMENT.model.OWNED.listeners`.

While analyzing method `listeners.add(l)` on the collection class `List`, the analysis looks if there is a method declaration dataflow annotation `@DataFlow` on the method `add`. Indeed, the analysis finds one, namely `@Dataflow({"value ↦ recv"}` that introduces dataflow edges between OObjects, which correspond to the `add` method argument `value`, and the receiver OObject `main.DOCUMENT.model.OWNED.listeners`. As a result, the analysis looks up any OObjects of type `Listener` in the domain `main.VIEW`. It finds two such `OObjects`, namely `barChart` and `pieChart`. It then adds an OEdge from the OObject `barChart` to `listeners`, and another OEdge from `pieChart` to `listeners`.

Similarly, the analysis processes the method invocation statement `model.addListener(pieChart)` and completes steps similar to those performed when analyzing `model.addListener(barChart)`. We omit them for brevity.

The analysis then encounters the method invocation `barChart.addListener(model)`. It analyzes this method in the context of OObject `main` and the receiver OObject `main.VIEW.barChart`. According to our dataflow rules (Sec. 2.2), this method invocation introduces a dataflow edge OEdge (E5) from OObject `main` to OObject`barChart` due to the export of a method argument.

The analysis processes the method invocation `listeners.add(l)` in class `BaseChart` in a similar way to the analysis performed for the method invocation `listeners.add(l)` in class `Model` (Fig. 2.15). However, when the analysis looks up OObjects of type `Listener` in the domain `main.DOCUMENT`, it finds only one OObject,

```
main.run();
analyze(Main::run,
[Main::DOCUMENT ↦ main.DOCUMENT, Main::VIEW ↦ main.VIEW, Main::OWNER ↦ SHARED],
recv ↦ main, O_this ↦ main) 22

[Main::DOCUMENT ↦ main.DOCUMENT, Main::VIEW ↦ main.VIEW, Main::OWNER ↦ SHARED],
recv ↦ main
O_this ↦ main
public class Main<OWNER> {
  ...
  public void run() {

    model.addListener(barChart);
    analyze(Model::addListener, [Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT],
    recv ↦ main.DOCUMENT.model, O_this ↦ main) 23
    OEdge(main, main.DOCUMENT.model) (E1) 24
    // continue to Fig 2.14

    model.addListener(pieChart);
    // The analysis is similar to model.addListener(barChart)
    // Omitted for brevity

    barChart.addListener(model);
    analyze(BaseChart::addListener, [BaseChart::M ↦ main.DOCUMENT,
    Model::OWNER ↦ main.VIEW], recv ↦ main.VIEW.barChart, O_this ↦ main) 29
    OEdge(main, main.VIEW.barChart) (E5) 30
    // continue to Fig 2.15

    pieChart.addListener(model);
    // The analysis is similar to barChart.addListener(model)
    // Omitted for brevity
    OEdge(main, main.VIEW.pieChart) (E8) 35

    model.notifyObservers();
    analyze(Model::notifyObservers, [Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT],
    recv ↦ main.DOCUMENT.model, O_this ↦ main) 36
    // continue to Fig 2.16

    barChart.notifyObservers();
    analyze(BaseChart::notifyObservers,
    [BaseChart::M ↦ main.DOCUMENT, Model::OWNER ↦ main.VIEW],
    recv ↦ main.VIEW.barChart, O_this ↦ main) 41
    // continue to Fig 2.17

    pieChart.notifyObservers();
    // The analysis is similar to barChart.notifyObservers()
    // Omitted for brevity
  }
}
```

Figure 2.13: Abstractly interpreting the program, class `Main`.

```
[Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT]
recv ↦ main.DOCUMENT.model
O_this ↦ main
class Model<OWNER, V> extends Listener<OWNER> {
  ...
  public void addListener(Listener<V> l) {
    listeners.add(l);
    analyze(List::add, [List::ELTS ↦ main.VIEW],
    recv ↦ main.DOCUMENT.model.OWNED.listeners,
    O_this ↦ main.DOCUMENT.model]) 25
    OEdge(main.DOCUMENT.model, main.DOCUMENT.model.OWNED.listeners) (E2) 26
  }
}
[List::ELTS ↦ main.VIEW, Map::OWNER ↦ main.DOCUMENT.model.OWNED]
recv ↦ main.DOCUMENT.mode.OWNED.listeners
O_this ↦ main.DOCUMENT.model
T = Listener
class List<OWNER, T<ELTS>> {

    T<ELTS> value;

    @Dataflow(value ↦ recv)
    public void add(T<ELTS> value) {...}
    OObject(main.VIEW.barChart, BarChart<main.VIEW, main.DOCUMENT>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.VIEW.barChart, main.DOCUMENT.model.OWNED.listeners) (E3) 27

    OObject(main.VIEW.pieChart, PieChart<main.VIEW, main.DOCUMENT>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.VIEW.pieChart, main.DOCUMENT.model.OWNED.listeners) (E4) 28
    ...
}
```

Figure 2.14: Abstractly interpreting the program (continued): `Model` addListener method.

namely `model`. As as a result, it adds an OEdge from the OObject `model` to the OObject `main.VIEW.barChart.OWNED.listeners`.

In Fig. 2.16, the analysis processes the method invocation `model.notifyObservers()`. When it locates the object allocation statement for object `mTOv`, it creates the OObject `mTOv` (O7) and analyzes the class `MsgMtoV`. However, this class has no statements to analyze, so the analysis proceeds to the next method invocation statement in the method body which is `l.update(mTOv)`. This method invocation produces an export dataflow edge OEdge from the context OObject `model` to OObjects that have the same type as the receiver variable `l` (i.e., `Listener<main.VIEW>`). The analysis finds two such OObjects, namely `barChart` and `pieChart`. So it adds an OEdge from the OObject `model` to both of them.

Finally, the analysis processes the method invocation `barChart.notifyObservers` (Fig. 2.17). It analyzes the superclass `BaseChart` and performs the same steps previously discussed. However, when the analysis looks up OObjects that share the same type as the local variable `l` in the method invocation `l.update(vTOm)`, it finds only one OObject in domain `main.DOCUMENT` (i.e. `model`). Therefore, it creates one dataflow edge OEdge (E13) from OObject `barChart` to OObject `model`. It should be noted that our analysis does not add an OEdge from `pieChart` to `model`, even though the types of both `barChart` and `pieChart`, `BarChart` and `PieChart`, extend from the `Listener` abstract baseclass. This illustrates how our dataflow edges are more precise than those shown by transferring information from the type structure onto an object graph. The ownership domain annotations enable the analysis to distinguish between objects in the object graph and increases the overall precision of the analysis.

```
[BarChart::M ↦ main.DOCUMENT, BarChart::OWNER ↦ main.VIEW]
recv ↦ main.VIEW.barChart
O_this ↦ main
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  analyze(barChart, [BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW],
  recv ↦ main.VIEW.barChart, O_this ↦ main) 31

  public void update(Msg<LENT> msg) {...}
}
[BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW]
recv ↦ main.VIEW.barChart
O_this ↦ main
class BaseChart<OWNER, M> extends Listener<OWNER> {
  ...
  public void addListener(Listener<M> l) {
    listeners.add(l);
    analyze(List::add, [List::ELTS ↦ main.DOCUMENT],
    recv ↦ main.VIEW.barChart.OWNED.listeners, O_this ↦ main.VIEW.barChart]) 32
    OEdge(main.VIEW.barChart, main.VIEW.barChart.OWNED.listeners) (E6) 33
  }
}
[List::ELTS ↦ main.DOCUMENT, List::OWNER ↦ main.VIEW.barChart.OWNED]
recv ↦ main.VIEW.barChart.OWNED.listeners
O_this ↦ main.VIEW.barChart
T = Listener
class List<OWNER, T<ELTS>> {
  ...
  @Dataflow(value ↦ recv)
  public void add(T<ELTS> value) {...}
  OObject(main.DOCUMENT.model, Model<main.DOCUMENT, main.VIEW>) ∈
  lookup(Listener<main.DOCUMENT>)
  OEdge(main.DOCUMENT.model, main.VIEW.barChart.OWNED.listeners) (E7) 34
  ...
}
```

Figure 2.15: Abstractly interpreting the program (continued): `BaseChart` addListener method.

```
[Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT]
recv ↦ main.DOCUMENT.model
O_this ↦ main
class Model<OWNER, V> extends Listener<OWNER> {
  ...
  public void notifyObservers() {
    OObject(main.DOCUMENT.model.DATA.mTOv,
    MsgMtoV<main.DOCUMENT.model.DATA>) (O7) 37
    MsgMtoV<DATA> mTOv = new MsgMtoV();
    analyze(mTOv, [MsgMtoV::OWNER ↦ main.DOCUMENT.model.DATA],
    recv ↦ main.DOCUMENT.model.DATA.mTOv, O_this ↦ main.DOCUMENT.model) 38

    Listener<V> l = listeners.value;
    l.update(mTOv);
    OObject(main.VIEW.barChart, BarChart<main.VIEW, main.DOCUMENT>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.DOCUMENT.model, main.VIEW.barChart) (E11) 39

    OObject(main.VIEW.pieChart, PieChart<main.VIEW, main.DOCUMENT>) ∈
    lookup(Listener<main.VIEW>)
    OEdge(main.DOCUMENT.model, main.VIEW.pieChart) (E12) 40
    }
  }
}
```

Figure 2.16: Abstractly interpreting the program (continued): `Model` notifyObservers method.

2.5 Advanced Features

Recursion. Our analysis handles recursive types which can cause the OGraph to grow arbitrarily deep and the analysis to not terminate. For example, consider the class QuadTree, which is adapted from Abi-Antoun and Aldrich [4] (Fig. 2.19). The QuadTree class declares a field of type QuadTree in its `OWNED` domain. If we were not to handle recursion, our analysis would keep creating QuadTree OObjects and ODomains and would never terminate.

Our analysis handles recursive types by unifying domains [4]. It creates a cycle whenever the analysis encounters a previously visited context. More specifically, when the same runtime ODomain appears as the child of two OObjects. Figure 2.20

```
[BarChart::M ↦ main.DOCUMENT, BarChart::OWNER ↦ main.VIEW]
recv ↦ main.VIEW.barChart
O_this ↦ main
class BarChart<OWNER, M> extends BaseChart<OWNER, M> {
  analyze(barChart, [BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW],
  recv ↦ main.VIEW.barChart, O_this ↦ main)

  public void update(Msg<LENT> msg) {...}
}
[BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW]
recv ↦ main.VIEW.barChart
O_this ↦ main
class BaseChart<OWNER, M> extends Listener<OWNER> {
  ...
  public void notifyObservers() {

    OObject(main.VIEW.barChart.DATA.vTOm,
    MsgVtoM<main.VIEW.barChart.DATA>) (O8) 42
    MsgVtoM<DATA> vTOm = new MsgVtoM();
    analyze(vTOm, [MsgVtoM::OWNER ↦ main.VIEW.barChart.DATA],
    recv ↦ main.VIEW.barChart.DATA.vTOm, O_this ↦ main.VIEW.barChart)

    Listener<M> l = listeners.value;
    l.update(vTOm);
    OObject(main.DOCUMENT.model, Model<main.DOCUMENT, main.VIEW>) ∈
    lookup(Listener<main.DOCUMENT>)
    OEdge(main.VIEW.barChart, main.DOCUMENT.model) (E13) 43
  }
}
```

Figure 2.17: Abstractly interpreting the program (continued): `BaseChart` notifyObservers method.

illustrates our analysis abstract interpretation on the QuadTree example. When the analysis reaches step 9 to analyze the QuadTree class of the newly created OObject `nwQT`, it encounters the same ODomain `OWNED` which tells the analysis to stop further analyzing the class and to create a cycle on the OGraph between the OObject `nwQT` and the ODomain `OWNED`. Fig. 2.21 shows the resulting OGraph for the same example.

Primitive Types Although we formalized our analysis using Featherweight Java (FJ) (Chapter 3), which is a pure object-based language, our implementation handles

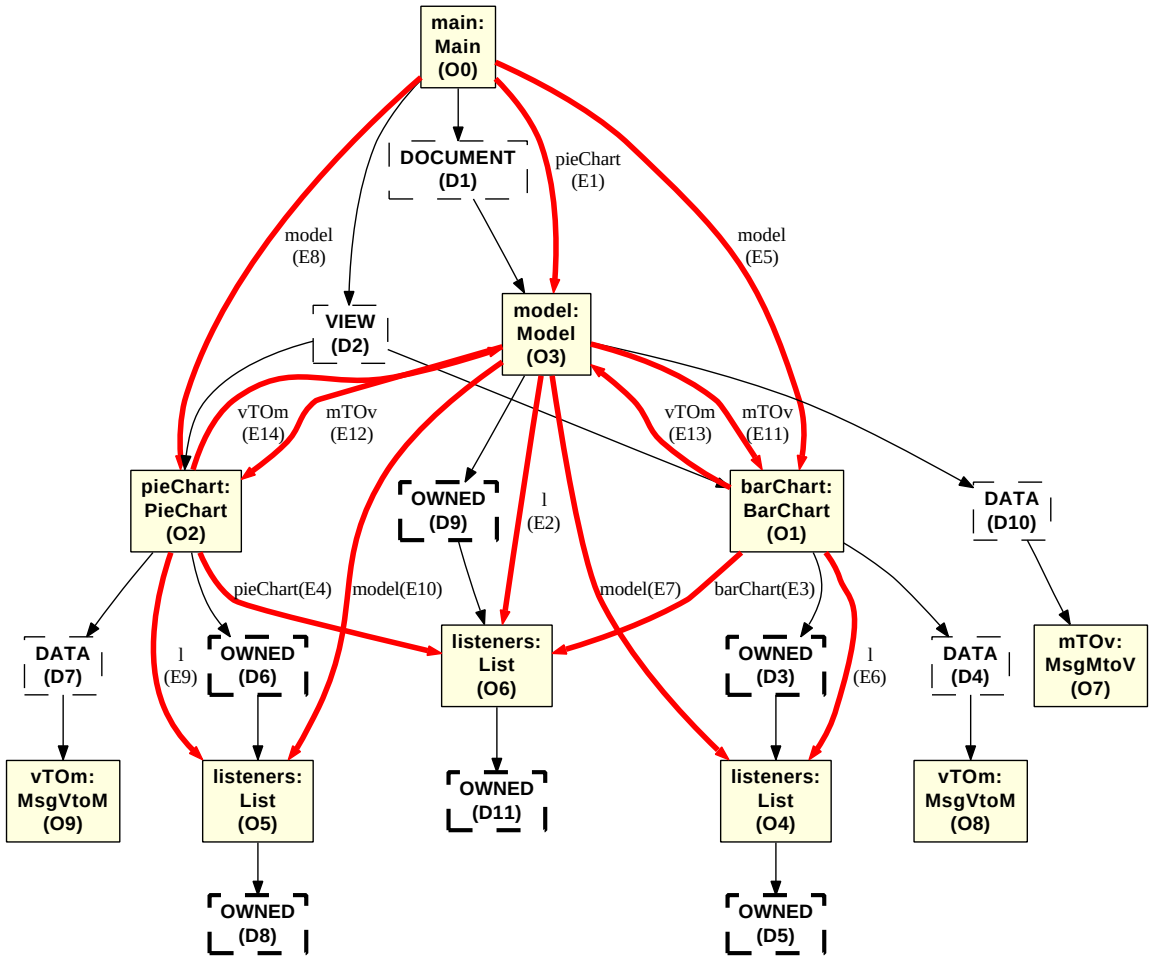


Figure 2.18: Listeners: object graph with dataflow edges.

```
class Main<OWNER> {
  private domain OWNED;
  QuadTree<OWNED> aQT = new QuadTree<OWNED>();
}
class QuadTree<M> {
  domain OWNED;
  QuadTree<OWNED> nwQT = new QuadTree<OWNED>();
}
```

Figure 2.19: QuadTree example with annotations.

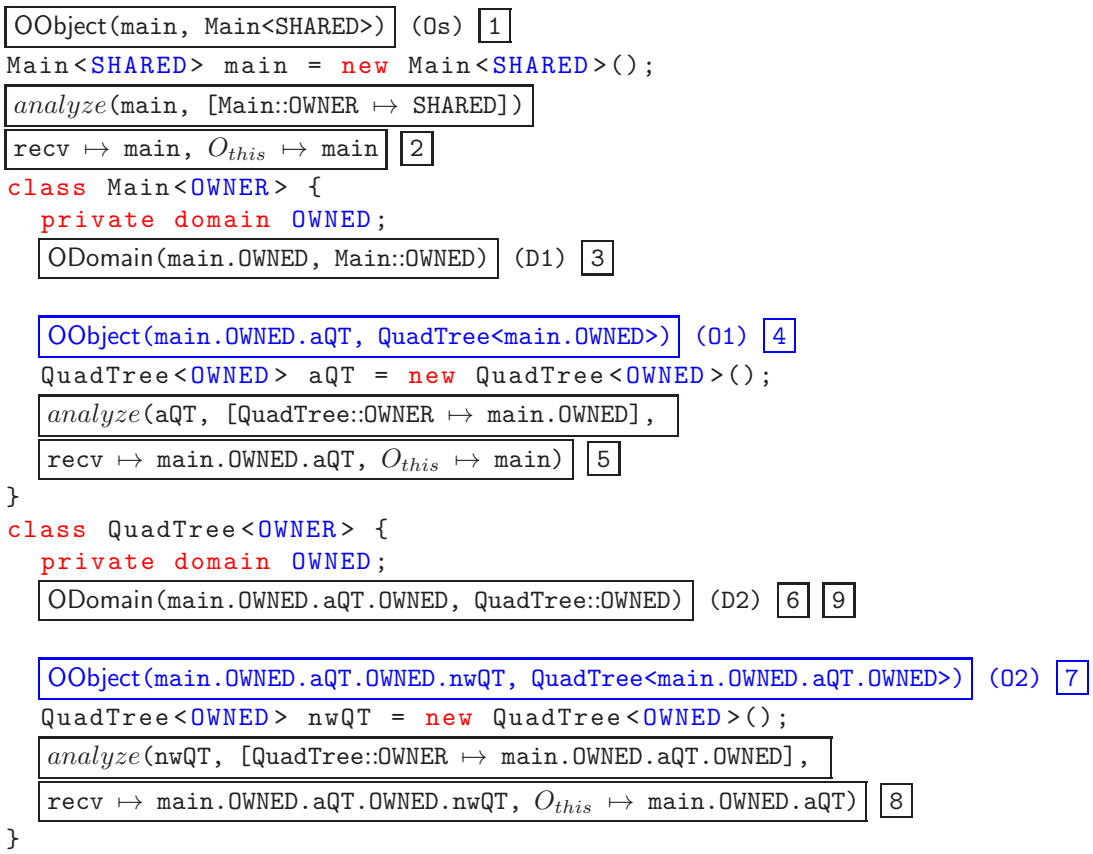


Figure 2.20: QuadTree abstract interpretation with cycle detection.

primitive datatypes and considers the flow of both object and primitive types. In Fig. 2.2 in Section 2.2, the type of oD does not necessarily need to be an object type, it could be a primitive type (i.e., integer, double, etc) and still the analysis would generate the same dataflow edges between the objects oA, oB, and oC.

2.6 Discussion

Object-sensitivity vs. domain-sensitivity Our analysis is considered domain-sensitive because it distinguishes between objects in different domains even if they are created at the same allocation site in the source code. Since there are fewer domains than objects in any program, our analysis is considered to be more scalable than an object-sensitive analysis. The state-of-art analysis is object-sensitive [31] because it is more precise. However, our analysis is object-insensitive like SCHOLIA and suffers

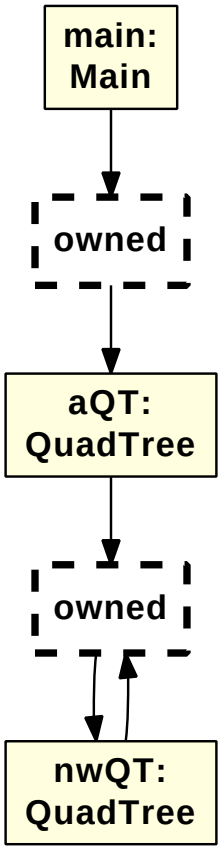


Figure 2.21: QuadTree example OGraph.

from some imprecisions that an object-sensitive analysis handles (See Abi-Antoun's dissertation [2, Section 2.6.3]).

Chapter 3: Formalization of the Analysis

3.1 Introduction

In Chapter 2, we described our analysis informally. In this chapter, we formally describe our static analysis using Featherweight Domain Java (FDJ) [8]. FDJ uses Featherweight Java (FJ) [24]. FJ is a core language that makes a number of simplifications to the full Java language. It ignores complex Java language constructs such as statics, interfaces, and reflection among others, since these constructs can be rewritten in terms of more fundamental ones. Our formalization captures all dataflow scenarios we explained earlier in Section 2.2.

In the following section, we introduce the formalization of the Object Graph (OGraph). In the formalization, we adopt a simplified FDJ abstract syntax (Fig. 3.1). The metavariable C ranges over class names; T ranges over types; f ranges over fields; v ranges over values; e ranges over expressions; x ranges over variable names; n ranges over domain and variable names; S ranges over stores; ℓ ranges over locations in the store, α and β range over formal ownership domain parameters, and m ranges over method names. As a shorthand, an overbar is used to represent a sequence, and we use $\bullet$ to denote an empty sequence. In FDJ, classes are parameterized by a list of ownership domains, the first domain parameter of a class denotes its owning domain. A class can extend another class that has a subsequence of its domain parameters.

A store S maps locations ℓ to their contents. Each location in the store consists of the class of the object, the actual ownership domain parameters, and the values stored in its fields. The expression form $\ell \triangleright e$ represents a method body e executing with a receiver ℓ.

$$
\begin{array}{rcl}
cdef & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} \\
 & & \{\ \overline{dom};\ \ \overline{T\ f};\ \ \overline{md}\ \} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; \\
md & ::= & T_R\ m(\overline{T}\ \overline{x})\ T_{this}\ \{\texttt{return}\ e_R;\} \\
e & ::= & x\ \mid\ \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}()\ \mid\ e.f\ \mid \\
 & & e.f = e'\ \mid\ e.m(\overline{e})\ \mid\ \ell\ \mid\ \ell \triangleright e \\
n & ::= & d\ \mid\ v \\
p & ::= & \alpha\ \mid\ n.d\ \mid\ \texttt{shared} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} \\
v, \ell & \in & \mathit{locations} \\
S & ::= & \ell\ \rightarrow\ C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \\
\Sigma & ::= & \ell\ \rightarrow\ T \\
\Gamma & ::= & x\ \rightarrow\ T
\end{array}
$$

Figure 3.1: Simplified FDJ abstract syntax [8].

$$
\begin{array}{lll}
G \in \mathsf{OGraph} & ::= \langle\ \textbf{Objects} = DO,\ \textbf{Domains} = DD,\ \textbf{Edges} = DE\ \rangle & \\
 & ::= \langle\ DO,\ DD,\ DE\ \rangle & \\
D \in \mathsf{ODomain} & ::= \langle\ \textbf{Id} = D_{id},\ \textbf{Domain} = C{::}d\ \rangle & \\
 & ::= \langle\ D_{id}, C{::}d\ \rangle & \\
O \in \mathsf{OObject} & ::= \langle\ \textbf{Id} = O_{id},\ \textbf{Type} = C\texttt{<}\overline{D}\texttt{>}\ \rangle & \\
 & ::= \langle\ O_{id}, C\texttt{<}\overline{D}\texttt{>}\ \rangle & \\
E \in \mathsf{OEdge} & ::= \langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst},\ \textbf{DataType} = C\ \rangle & \\
 & ::= \langle\ O_{src},\ O_{dst}, C\ \rangle & \\
DD & ::= \emptyset\ \mid\ DD \cup \{\ (O,d) \mapsto D\ \} & \text{Dataflow Domain} \\
DO & ::= \emptyset\ \mid\ DO \cup \{\ O\ \} & \text{Dataflow Object} \\
DE & ::= \emptyset\ \mid\ DE \cup \{\ E\ \} & \text{Dataflow Edge} \\
\Upsilon & ::= \emptyset\ \mid\ \Upsilon \cup \{\ C\texttt{<}\overline{D}\texttt{>}\ \} & \text{Visited objects} \\
H & ::= \emptyset\ \mid\ H \cup \{\ \ell \mapsto O\ \} & \text{Object map} \\
K & ::= \emptyset\ \mid\ K \cup \{\ \ell.d \mapsto D\ \} & \text{Domain map}
\end{array}
$$

Figure 3.2: Datatype declarations for the OGraph.

3.2 OGraph formalization

The OGraph is a graph composed of three different sets: OObjects, ODomains, and OEdges. OObjects and ODomains represent the nodes of the OGraph, whereas OEdges represent dataflow edges between OObjects (Fig. 3.2).

The aliasing invariant adopted by our system requires two objects to be the same only if they share the same owning domain and the same set of ownership domain parameters. This is why the datatype declaration of OObject is $C<\overline{D}>$ instead of only `C`. We follow the FDJ convention and consider an OObject's owning domain ODomain to be the first ODomain D_1 in $\overline{D}$. The root OObject of OGraph has no domain parameters and its owned by the global `shared` domain which is represented by ODomain D_{shared}.

Although a domain d is declared at the level of a class `C` in a program, each instance of class `C` gets its own runtime domain $\ell.d$ because class `C` instances are different if they are declared in different domains. For example, if there are two distinct runtime objects ℓ and ℓ' of class `C`, and class `C` declares a domain `d` in code, then the analysis will distinguish between the runtime domains $\ell.d$ and $\ell'.d$. This is why in the domain map DD (Fig 3.2), we associate an ODomain `D` to a pair consisting of an OObject `O` and a declared domain `d`.

To deal with recursive types (Sec. 2.5), an ODomain can have multiple parent OObjects, so an ODomain does not have an owning OObject in its datatype declaration (Fig. 3.2). However, it is qualified by the name of the class that declares it.

Each OEdge E represents a dataflow edge from a source OObject to a destination OObject. The third argument in the OEdge datatype declaration is the class[1]of the object that flows because of this edge.

The last two lines in Figure 3.2 show two maps, H and K that the instrumented

[1]The class of an object is only a part of its type i.e., the C part. The full type includes as well the list of the object's domain parameters $C<\overline{D}>$

runtime semantics uses to evaluate program expressions. We use the map H to look up the OObject that corresponds to a program value ℓ, and we use the map K to look up the ODomain from a runtime domain $\ell.d$. Υ records the combinations of class and domain parameters analyzed in the call stack to avoid non-termination in the analysis due to recursive calls.

3.3 Constraint-Based Specification

We formalized our analysis using a constraint-based specification instead of transfer functions. As a result we do not need to worry about the order in which program expressions are evaluated. Constraint-based specification formalizes the static analysis as a set of inference rules and makes it easier to prove soundness[2]. The constraint system requires adding OObject, ODomain, and OEdge to OGraph. The constraint system is solved once we can no longer add to these sets. The analysis of a program P is the least solution $G = \langle DO, DD, DE \rangle$ of the following constraint system:

$$\emptyset, \emptyset, DO, DD, DE \vdash P = (CT, e_{root}.m_{root}())$$

The judgement form for expressions is as follows:

$$\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this},\ H} e$$

The O_{this} subscript on the turnstile captures the context-sensitivity. It represents the object in which the statement being currently analyzed is executing. H is part of the instrumentation we described earlier, but we omit it from the rules that do not use it. The context Γ is the FDJ typing context, and Υ is the call stack context to avoid cycles.

[2]The analysis proof of soundness is out side the scope of this thesis and is left for future work.

Inference rules. In FDJ, a program P is a tuple (CT, S, e) that consists of a class table CT which maps classes to their definitions, a store S and an expression e. Our analysis starts with a method invocation m_{root} on the root expression e_{root}. Our analysis requires a root OObject O_{root} to start from. The root OObject has a single ODomain D_{shared} that corresponds to the global domain `shared`. We qualify a domain d by the class that declares it, as C::d. Because no class declares the `shared` domain, we qualify it as ::`shared`. The OObject O_{root} does not correspond to an actual runtime object. It is only required by our analysis to start with, as a dummy receiver for top-level code.

$$D_{shared} = \langle\ D_s,\ \texttt{::shared}\ \rangle$$
$$O_{world} = \langle\ O_{world},\ \texttt{Object<}D_{shared}\texttt{>}\ \rangle$$

The analysis starts by abstractly interpreting the method invocation on the root expression $e_{root}.m_{root}$ in the O_{root} context,

$$\emptyset, \emptyset, DO, DD, DE \vdash_{O_{root}} e_{root}.m_{root}()$$

In Df-New, the analysis interprets a `new` object allocation in the context of the receiver OObject O_{this}. The analysis first ensures that DO contains an OObject O_C for the newly allocated object. If DO does not contain O_C, the analysis adds O_C to DO. Then, Df-New ensures that DD has a representative ODomain D_i for each domain parameter p_i passed to the creation of an instance of class C. And within class C, DD maintains the mapping of each formal parameter α_i to a corresponding D_i in the context of the receiver O_C by having a map between the pair (O_C, α_i) and D_i (Fig. 3.3).

Then, Df-New uses the auxiliary judgement Df-Dom to ensure that DD has an

$$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}$$

$$CT(\texttt{Object}) = \texttt{class}\ \texttt{Object<}\alpha_o\texttt{>}\ \ \{\ \ \}$$

$$\frac{\begin{array}{c}\forall i \in 1..|\overline{p}| \qquad D_i = DD[(O_{this}, p_i)] \qquad params(C) = \overline{\alpha} \\ O_C = \langle\ O_{id},\ C\texttt{<}\overline{D}\texttt{>}\ \rangle \qquad \{O_C\} \subseteq DO \qquad \{(O_C, \alpha_i) \mapsto D_i\} \subseteq DD \\ DO, DD, DE \vdash_{O_{this}} ddomains(C, O_C) \qquad DO, DD, DE \vdash_{O_{this}} dfields(C, O_C) \\ \forall m.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, DO, DD, DE \vdash_{O_C} e_R \\ \forall k \in 1..|\overline{e}| \qquad e_k : C_{e_k}\texttt{<}\overline{p}\texttt{>} \qquad \Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_k \\ \overline{e} \neq \bullet \Longrightarrow \{\langle O_{this}, O_C, C_{e_k}\rangle\} \subseteq DE\end{array}}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e})}[\textsc{Df-New}]$$

$$\frac{\begin{array}{c}\forall(\texttt{domain}\ d_j) \in \overline{dom} \qquad D_j = \langle D_{id_j},\ C\texttt{::}d_j\rangle \qquad \{(O_C, d_j) \mapsto D_j\} \subseteq DD \\ DO, DD, DE \vdash_{O_{this}} ddomains(C', O_C)\end{array}}{DO, DD, DE \vdash_{O_{this}} ddomains(C, O_C)}[\textsc{Df-Dom}]$$

$$\frac{\begin{array}{c}\forall(T_k\ f_k) \in \overline{T}\ \overline{f} \qquad owner(T_k) = p'_k \qquad D_k = DD[(O_C, p'_k)] \\ DO, DD, DE \vdash_{O_{this}} dfields(C', O_C)\end{array}}{DO, DD, DE \vdash_{O_{this}} dfields(C, O_C)}[\textsc{Df-Fields}]$$

$$\frac{}{DO, DD, DE \vdash_{O_{this}} ddomains(\texttt{Object}, O_C)}[\textsc{Df-Obj1}]$$

$$\frac{}{DO, DD, DE \vdash_{O_{this}} dfields(\texttt{Object}, O_C)}[\textsc{Df-Obj2}]$$

$$\frac{}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} x}[\textsc{Df-Var}] \qquad \frac{}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} \ell}[\textsc{Df-Loc}]$$

Figure 3.3: Constraint-based specification of the OGraph.

ODomain corresponding to each domain that the class C locally declares. Df-Dom does this recursively to include inherited domains from superclasses as well. Df-New then uses the auxiliary judgement Df-Fields to recursively include inherited fields from superclasses.

Df-Dom and Df-Fields are recursive judgments. Df-Obj1 and Df-Obj2 are the base cases for Df-Dom and Df-Fields, respectively, to deal with the root class, `Object`. They do not need to do anything because according to FDJ [8], the class `Object` has no fields, domains, or methods.

Df-New then proceeds and obtains each expression e_R in each method m in

$$\frac{\begin{array}{c}\exists(T_k\ f_k) \in \overline{T}\ \overline{f} \qquad T_k = C_{f_k}\!<\!\overline{p}\!> \qquad e_0 : T_{e_0} \\ \forall i\ DO, DD, DE \vdash_{O_{this}} lookup\ (T_{e_0}) = O_i \quad \{\langle O_i, O_{this}, C_{f_k}\rangle\} \subseteq DE \\ \Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_0\end{array}}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_0.f_k}[\textsc{Df-Read}]$$

$$\frac{\begin{array}{c}\exists(T_k\ f_k) \in \overline{T}\ \overline{f} \qquad T_k = C_{f_k}\!<\!\overline{p}\!> \qquad e_0 : T_{e_0} \\ \forall i\ DO, DD, DE \vdash_{O_{this}} lookup\ (T_{e_0}) = O_i \qquad \{\langle O_{this}, O_i, C_{f_k}\rangle\} \subseteq DE \\ \Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_0 \qquad \Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e'\end{array}}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_0.f_k = e'}[\textsc{Df-Write}]$$

$$\frac{\begin{array}{c}O_k = \langle\ O_{id}, C\!<\!\overline{D}\!>\ \rangle \in DO \qquad T' = C'\!<\!\overline{p'}\!> \qquad C <: C' \\ \forall i \in 1..|\overline{p'}| \qquad D'_i = DD[(O_{this}, p'_i)] \qquad D'_i = D_i\end{array}}{DO, DD, DE \vdash_{O_{this}} lookup\ (T') = O_k}[\textsc{Df-Lookup}]$$

$$\frac{\begin{array}{c}(\overline{x} : \overline{T},\ e_R) \in\ mbody(m, C\!<\!\overline{p}\!>) \qquad e_R : C_{e_R}\!<\!\overline{p'}\!> \qquad e_0 : T_{e_0} \\ \forall i\ DO, DD, DE \vdash_{O_{this}} lookup\ (T_{e_0}) = O_i \quad \{\langle O_i, O_{this}, C_{e_R}\rangle\} \subseteq DE \\ \forall k \in 1..|\overline{e}| \qquad e_k = C_{e_k}\!<\!\overline{p}\!> \qquad \Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_k \\ \overline{e} \neq \bullet \Longrightarrow \{\langle O_{this}, O_i, C_{e_k}\rangle\} \subseteq DE \\ \Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_0 \qquad \Gamma, \Upsilon, DO, DD, DE \vdash_{O_i} e_R\end{array}}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}} e_0.m(\overline{e})}[\textsc{Df-Invk}]$$

$$\frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, DO, DD, DE \vdash_{O_C} e}{\Gamma, \Upsilon, DO, DD, DE \vdash_{O_{this}, H} \ell \triangleright e}[\textsc{Df-Context}]$$

$$\frac{\begin{array}{c}\forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{p}\!> \\ H[\ell] = O_{this} = \langle O_{id}, C\!<\!\overline{D}\!>\rangle \in DO \\ \forall m.\ mbody(m, C\!<\!\overline{p}\!>) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\!<\!\overline{p}\!>\}, \emptyset, DO, DD, DE \vdash_{O_{this}} e_R\end{array}}{DO, DD, DE \vdash_{CT,H} \Sigma}[\textsc{Df-Sigma}]$$

Figure 3.4: Constraint-based specification of the OGraph (continued).

C, and processes e_R in the context of the new receiver OObject which is now O_C. Df-New checks these expressions recursively. However, before Df-New analyzes any new allocation expressions of OObject, it checks if the expression's combination of the class type and parameters have been previously analyzed by looking for this combination in Υ to avoid infinite recursion. If this combination does not exists, Df-New adds the current combination of a type and actual domain parameters to Υ and then proceeds to analyze the new OObject. It is important to mention here that Υ only keeps track of previously analyzed OObjects at the call stack level. It does

not do this globally across the program because similar combinations of class and domain parameters are allowed in different contexts. Df-New further analyzes the arguments expressions $\overline{e}$ to the constructor of class C. Finally, if there are argument expressions $\overline{e}$ to the constructor of class C, Df-New adds a dataflow edge from the receiver context OObject O_{this} to the newly allocated object O_C.

In Df-Read, the analysis adds a dataflow edge from OObject O_i, that has the type of the expression e_0, to the receiver context OObject O_{this} of the field write expression. Df-Read represents an import edge due to the propagation of a field reference from the expression receiver OObject O_i to the receiver context OObject O_{this}.

Df-Write is similar to Df-Read. However, it adds an export edge in the opposite direction, i.e., from the receiver context OObject O_{this} to OObject O_i. The edge has the class type of the expression e_0. Df-Write represents dataflow due to object export due to the propagation of a field reference from the receiver context OObject O_{this} to the expression receiver OObject O_i in the field write expression.

Both Df-Write and Df-Read analyze the receiver of the field access e_0. Df-Write, however, analyzes the expression e' on the right hand side of the assignment.

Df-Invk is interesting because it is responsible for two dataflow scenarios which are due to object export and import on a method invocation (See Sec 2.2). The first dataflow scenario is due to the arguments being passed to the method invocation. This argument passing introduces a dataflow edge from the receiver context OObject O_{this} to each OObject O_i, one which has the type of the expression e_0. The type of the data passed is at least the type of one of the actual parameters passed to the method invocation. In FJ, every method body consists of a return statement, which causes a dataflow from the method's receiver OObject O_i to the receiver context OObject O_{this}. The type of the data returned is the type of the returned object. Finally, Df-Invk analyzes the receiver and the actual arguments for the method invocation, each with

$$\frac{\begin{array}{c}\boxed{\ell \notin dom(S) \qquad S' = S[\ell \mapsto C\!<\!\overline{p}\!>\!(\overline{v})]} \\ G = \langle DO, DD, DE\rangle \\ \overline{p} = \overline{\ell'.d} \qquad D_i = K[\ell'_i.d_i] \\ O_C = \langle O_{id}, C\!<\!\overline{D}\!>\rangle \qquad O_C \in DO \qquad H' = H[\ell \mapsto O_C] \\ \forall(\texttt{domain}\ d_j) \in domains(C\!<\!\overline{p}\!>) \qquad D_j = DD[(O_C, d_j)] \qquad K' = K[\ell.d_j \mapsto D_j] \\ O = H[\theta] \qquad E = \langle O, O_C, ..\rangle \in DE\end{array}}{\theta \vdash \boxed{\texttt{new}\ C\!<\!\overline{p}\!>\!(\overline{v}); S}; H; K \rightsquigarrow_G \boxed{\ell; S'}; H'; K'}\text{[IR-New]}$$

$$\frac{\begin{array}{c}\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad fields(C\!<\!\overline{p}\!>) = \overline{T}\ \overline{f}} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad E = \langle O_\ell, O, ..\rangle \in DE\end{array}}{\theta \vdash \boxed{\ell.f_i; S}; H; K \rightsquigarrow_G \boxed{v_i; S}; H; K}\text{[IR-Read]}$$

$$\frac{\begin{array}{c}\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad fields(C\!<\!\overline{p}\!>) = \overline{T}\ \overline{f}} \\ \boxed{S' = S[\ell \mapsto C\!<\!\overline{p}\!>\!([v/v_i]\overline{v})]} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad E = \langle O, O_\ell, ..\rangle \in DE\end{array}}{\theta \vdash \boxed{\ell.f_i = v; S}; H; K \rightsquigarrow_G \boxed{v; S'}; H; K}\text{[IR-Write]}$$

$$\frac{\begin{array}{c}\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad mbody(m, C\!<\!\overline{p}\!>) = (\overline{x}, e_R)} \\ O = H[\theta] \qquad O_\ell = H[\ell] \qquad E = \langle O, O_\ell, \overline{x}\rangle \in DE \\ E' = \langle O_\ell, O, ..\rangle \in DE\end{array}}{\theta \vdash \boxed{\ell.m(\overline{v}); S}; H; K \rightsquigarrow_G \boxed{\ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S}; H; K}\text{[IR-Invk]}$$

$$\frac{}{\theta \vdash \boxed{\ell \triangleright v; S}; H; K \rightsquigarrow_G \boxed{v;\ S}; H; K}\text{[IR-Context]}$$

Figure 3.5: Instrumented runtime semantics (core rules).

the appropriate context $\mathsf{OObject}\ O_{this}$.

Each of Df-Read, Df-Write, and Df-Invk use the auxiliary judgment Df-Lookup, which is used to search for $\mathsf{OObject}$s in DO that match the type of the expression passed to it.

All the rules are responsible for dataflow scenarios, and eventually add edges to DE, except for Df-Var and Df-Loc. The rules Df-Var and Df-Loc for variables and locations exist to complete our formalization and make the induction go through. In the case of Df-Loc, the store constraint Df-Sigma enforces any necessary conditions on each location ℓ.

The last two rules are DF-CONTEXT and DF-SIGMA. The purpose of DF-CONTEXT is to analyze expressions of the form $\ell \triangleright e$, where O_C is the OObject we lookup for the receiver ℓ. DF-SIGMA is required for the induction and to ensure that method bodies have been analyzed for all objects in the store.

For completing the formalization of our analysis, we instrumented the runtime semantics (Fig. 3.5). The instrumentation is safe since discarding it produces exactly the same semantics as FDJ [8] (the common parts of the rules are highlighted).

3.4 Soundness

An OGraph is a *sound* approximation of a Runtime Object Graph (ROG), represented by a well-typed store S, if the OGraph relates to the ROG informally as follows:

- **Object soundness:** Each object ℓ in the ROG has exactly one representative OObject in the OGraph. Similarly, each domain in the ROG has exactly one representative ODomain in the OGraph.
- **Edge soundness:** If there is a dataflow from object $\ell 1$ to object $\ell 2$ in a ROG, then the OGraph has an OEdge between the OObjects $O1$ and $O2$ that are the representatives of $\ell 1$ and $\ell 2$, respectively.

Formal Definition of Soundness

$$\forall G = \langle DO, DD, DE \rangle \vdash P = (CT, e) \quad CT, e \;\; well - typed$$
$$\forall e; \emptyset, \emptyset, \emptyset \rightsquigarrow e; S; H; K$$
$$\forall \Sigma \vdash S$$
$$DO, DD, DE \vdash_{CT,H} \Sigma$$
$$(S, H, K) \sim (DO, DD, DE)$$

This formal definition states that given a well-typed store S, an OGraph produced

from the same program P, there exists a map H that maps each location ℓ in the store to a unique OObject, and a map K that maps each runtime domain in the store to a unique ODomain. The symbol $\sim$ denotes an approximation relation between the state (S, H, K) and the analysis result (DO, DD, DE) (See Abi-Antoun's dissertation [2, Section 3.3.2]). A formal proof of soundness is left for future work. Since these rules are very similar to the previous rules in SCHOLIA [2], we conjecture, but do not prove, that the analysis is sound. Soundness proof is by induction on the inference rules and will rely on the instrumented runtime semantics (Fig. 3.5) and congruence rules (not shown).

3.5 Credits

These rules follow the same style of formalization as Abi-Antoun [2], which includes a formal proof of soundness.

Chapter 4: Evaluation

In this chapter, we implement our analysis and evaluate it on four interesting examples, Listeners, BankingSystem, CourSys, and InfoFlow.

4.1 Implementation

We implemented the analysis as a plugin to the Crystal 3.3 static analysis framework [38] in Eclipse. The implementation is a whole-program analysis, which uses the Java JDT libraries [1] and a Visitor on the Java Abstract Syntax Tree (AST). We currently export the extracted object graph to GraphViz DOT [20].

The analysis supports a subset of the Java language, the same one in Featherweight Java. For example, our implementation does not handle interfaces, static fields or methods, or external libraries. The object graphs in this document were generated using our implementation[1].

4.2 Listeners System

In Section 2.4, we illustrate our analysis on the Listeners example, and we show the Listeners dataflow object graph (DfOOG) (Fig. 2.4). In this Section, we compare the extracted dataflow edges with the points-to edges which Scholia extracts on the same Listeners example (Fig. 4.1a). Both graphs use a similar notation, except for the edges. In PtOOG, a thin and black arrow represents a points-to edge. However, in DfOOG a thick and red arrow represents a dataflow edge.

[1] Our implementation handles arrays.

4.2.1 Dataflow vs. points-to edges

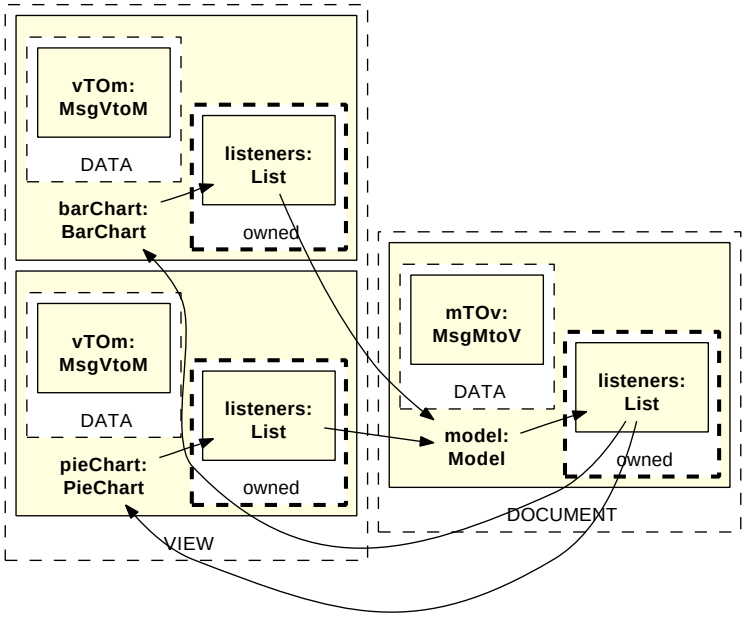
A points-to edge corresponds to a field reference relation. Points-to edges do not reflect all communication scenarios. For example, the points-to edge from the `model` object to its `listeners` object does not mean that `model` object is using or communicating with its `listeners`. Moreover, a missing points-to edge from `model` to each of `barChart` and `pieChart` does not mean that these objects do not communicate with the `model` (Fig. 4.1a).

The dataflow edges our analysis extracts show that the `model` object is indeed communicating with each of the `barChart` and `pieChart` objects by sending them `mTOv` messages (Fig. 4.1b). Similarly, both chart objects are sending back `vTOm` messages to the `model` object. Our dataflow edges seem consistent with dataflow-based communication occurring in a program as determined by code inspection. These edges are missing from the PtOOG (Fig. 4.1).

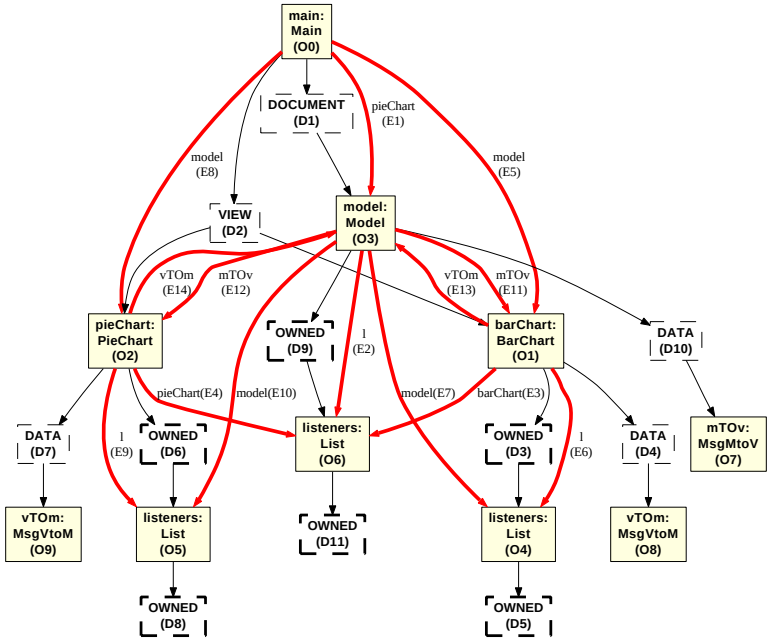
The extracted dataflow edges on Fig. 4.1b are also useful in helping understand the Listeners system. For example, our analysis made visually obvious interesting dataflow communications between the `model` object and each of `BaseChat` objects, `barChart` and `pieChart`. The object graph illustrates the Observer design pattern [19] by clearly showing how the subjects exchange messages with their observers (i.e., listeners).

4.3 Banking System

BankingSystem is a small banking application, adapted from Aldrich and Chambers [8]. A Bank has customers and branches. Every branch has a set of tellers and vaults. Every customer has a customer agent that may access any of the branch's tellers. The customer agent cannot access vaults directly. Only a branch teller can access the branch's vaults (Fig. 4.2).



(a) Points-to OOG (PtOOG)



(b) Dataflow OOG (DfOOG)

Figure 4.1: Listeners: Dataflow vs. Points-to edges.

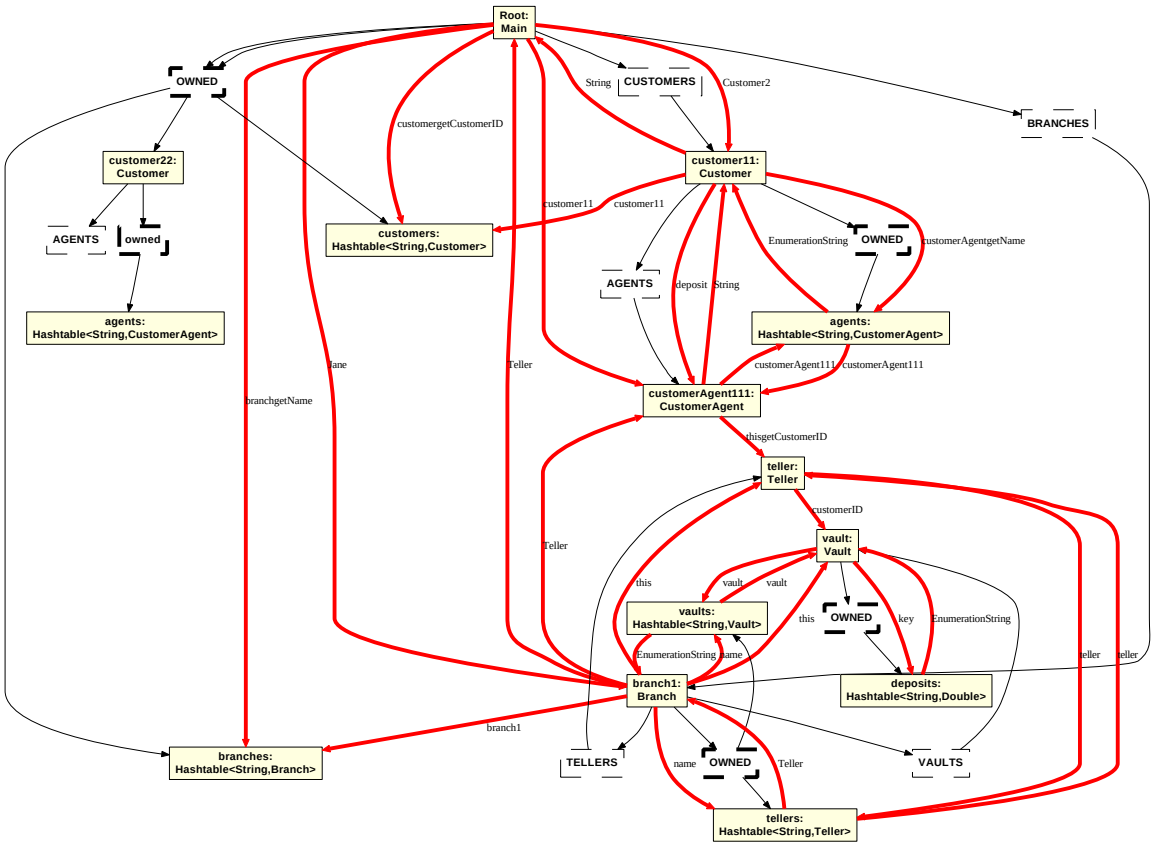


Figure 4.2: BankingSystem DfOOG.

The extracted DfOOG shows the two top-level domains `BRANCHES` and `CUSTOMERS` descending from the root object (Fig 4.2).

4.3.1 Dataflow edges on DfOOG vs. dataflow edges on a flat object graph.

We compare our analysis of dataflow edges with dataflow edges shown on a flat object graph (Fig. 4.3). Spiegel's Pangaea object graph extraction algorithm extracts dataflow edges on a flat object graph [41]. The dataflow edges our analysis extracts are more precise than those of Pangaea because our analysis has access to ownership domain annotations. Ownership domains give more precision on aliasing. Our analysis keeps data flowing to different objects apart if the objects are in separate domains. For example, our analysis treats object `customer22` in domain OWNED as a

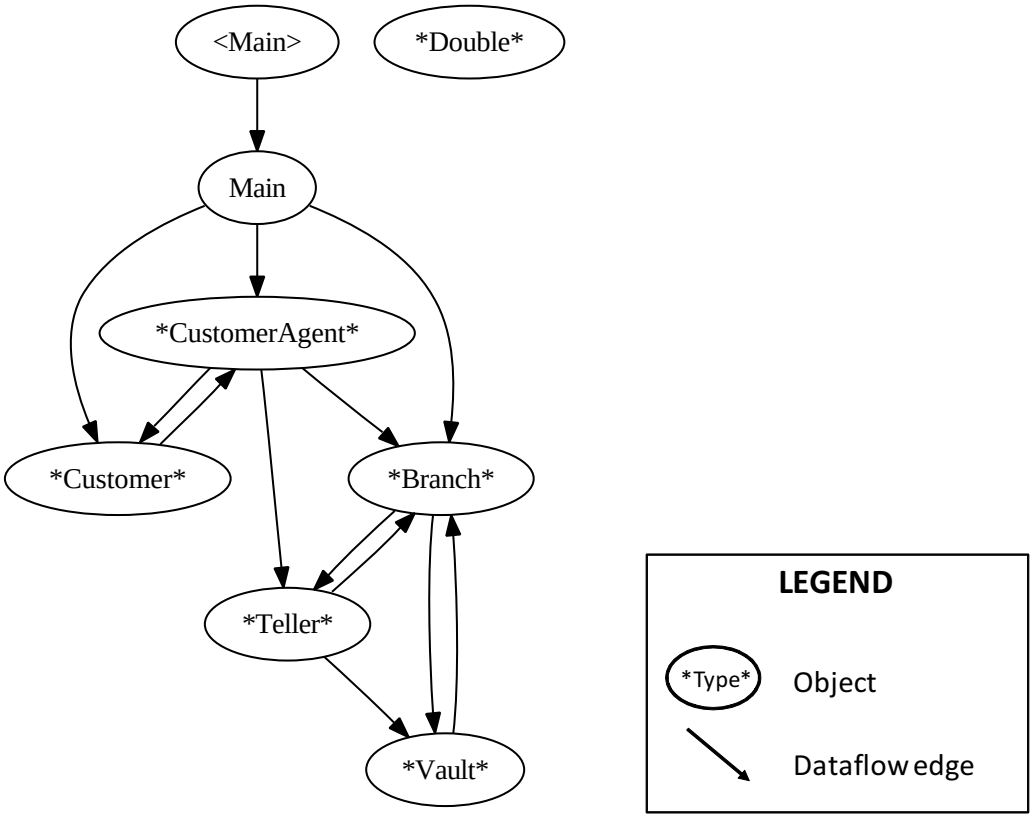


Figure 4.3: BankingSystem: Spiegel's Pangaea flat object graph.

distinct object that has its own dataflow edges, i.e., there are not any in this example. Pangaea analysis will treat the two customer objects similarly which is imprecise.

4.3.2 Discussion

The extracted BankingSystem DfOOG showed no dataflow edges between the customer agent object and the branch's vaults (Fig. 4.2). This is consistent with the design intent of allowing only the branch teller to access the branch's vaults.

4.4 CourSys System

CourSys is a prototypical simple course registration system. It follows the three-tiered architectural style. We annotated the CourSys source code using three *top-level*

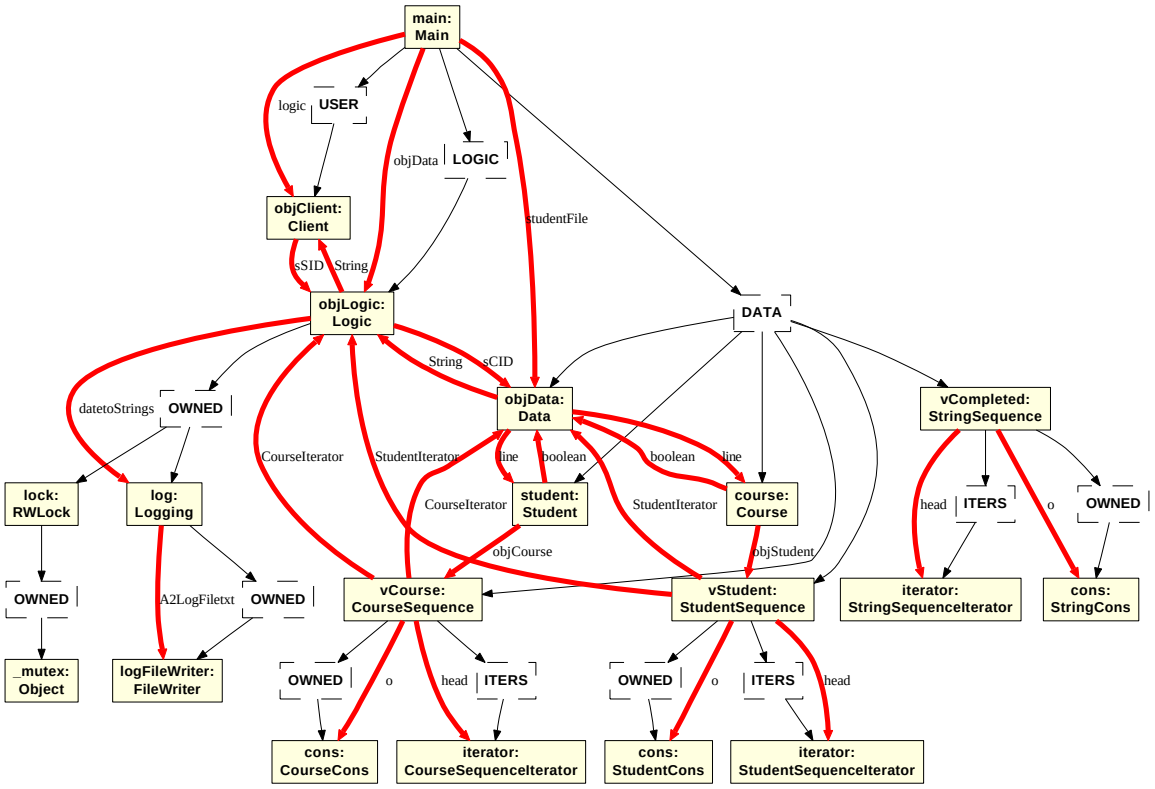


Figure 4.4: CourSys DfOOG.

domains: `USER`, `DATA`, and `LOGIC`. We extracted the DfOOG in (Fig 4.4).

The extracted DfOOG showed the three top-level domains `USER`, `DATA`, and `LOGIC` descending from the root object (Fig 4.4). Inside domain `USER`, we have the object `objClient`, and inside domain `LOGIC`, we have object `objLogic` which has the two objects `lock` and `log` in its owned domain. The thick, red edges represent dataflow. We annotate these edges with the name of the data that is propagating between every two objects.

4.4.1 Discussion

The extracted CourSys DfOOG showed that there was no direct communication between the `Client` object, `objClient` and the `Data` object, `objData`. Instead all communications went through the `Logic` object, `objLogic` (Fig. 4.4). These dataflow edges confirmed that CourSys indeed follows the three-tiered architectural style.

4.5 Information Flow Example

According to Liu and Milanova [31], the flow of information can be categorized as either explicit (flow of information arises due to the use of assignment statements) or implicit (flow of information is the result of using conditional statements). To illustrate these two categories, we use the same example by Liu and Milanova [31] (Fig. 4.5). The example is a simple web application that originally discussed by Chong et al. [11]. The user of this application has a number of tries to guess a number between 1 and 10 and wins if her guess matches the true number (i.e. the field `secret`). The field `tries` shows the number of guesses attempted by the user. Methods `this.finishApp` and `message.setText` output messages and are considered untrusted outputs. The goal is to reason about data confidentiality by ensuring the absence of information flow from `secret` to any of the untrusted outputs at lines 11, 15, 17, and 20. Liu and Milanova's [31] analysis infers no explicit information flow from the field `secret` to any of these untrusted outputs. However, their analysis infers an implicit information flow from `secret` to lines 11, 15, and 17 because of the conditional at line 9 (the value of `secret` is checked at this line). This means that the outputs at lines 11, 15, and 17, although not necessarily printing the value of `secret`, disclose information about the `secret`.

Because our analysis tracks dataflow between objects, we adapt the example from Liu and Milanova [31] and convert primitive types to object types. e.g., we use `Integer` instead of `int` (Fig. 4.5). We declared two top-level domains, `OWNED` and `MSG`. We put `secret` in `OWNED`, and `message` in `MSG` (Fig. 4.6). The resulted DfOOG is shown in Fig. 4.7.

```
1  class GuessANumber {
2    int secret;
3    int tries;
4    ...
5    void makeGuess ( Integer num ) throws NullPointerException {
6      int i = 0;
7      if ( num != null ) i = num.intValue();
8      if ( i >= 1 && i <= 10 ) {
9        if ( tries > 0 && i == secret ) {
10         tries = 0;
11         this.finishApp("You win!");
12       } else {
13         tries--;
14         if ( tries > 0 )
15           message.setText("Try again");
16         else
17           this.finishApp("Game over!");
18       }
19     } else
20     message.setText("Out of range");
21   }
22   finishApp(String msg) {}
23 }
```

Figure 4.5: Information Flow example. Adapted from Liu and Milanova [31]

4.5.1 Discussion

Our analysis is not intended to capture the implicit dataflow between `secret` and `message` objects. This dataflow is absent because the field `secret` never appears as an argument to the output method `message.setText`. However, our analysis was able to point out an indirect flow from `secret` to `message` through the `main` object. Information flow techniques are more precise and provide stronger guarantees on how secret information flows cross the system, than the dataflow technique we use in our analysis. However, our analysis can still reveal interesting dataflow scenarios due to objects propagation.

```
public class Main {

  domain OWNED, MSG
  Secret<OWNED> secret = new Secret();
  int tries;
  Message<MSG> message = new Message();

  public static void main(String<SHARED[SHARED]> args[]) {
    Main<SHARED> m = new Main();
    m.run(null);
  }
  void run(Integer<SHARED> num) throws NullPointerException {
    int i = 0;
    if (num != null)
      i = num.intValue();
    if (i >= 1 && i <= 10) {
      if (tries > 0 && i == secret.getInt()) {
        tries = 0;
        finishApp("You win!");
      } else {
        tries--;
        if (tries > 0)
          message.setText("Try again");
        else
          finishApp("Game over!");
      }
    } else
      message.setText("Out of range");
  }
  private void finishApp(String<SHARED> string) {
    System.out.println(string);
  }
}
class Message<OWNER> {
  domain OWNED
  void setText(String<SHARED> msg) {
    StringBuffer<OWNED> text;
    text = new StringBuffer("GuessNum");
    text.append(msg);
  }
}
class Secret<OWNER> {
  public int getInt() {}
}
class Integer<OWNER> {
  int val;
  public int intValue() {
    return val;
  }
}
```

Figure 4.6: Information Flow example using annotations.

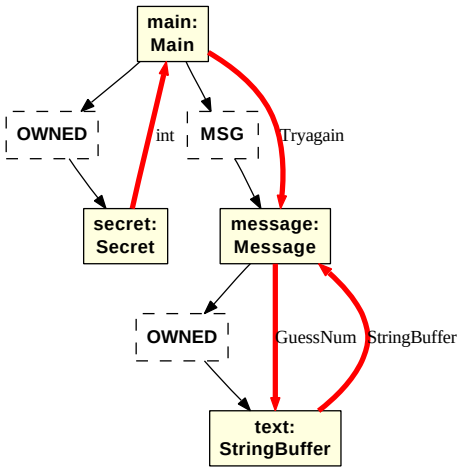


Figure 4.7: Information Flow example OGraph.

4.6 Summary

In this Chapter, we evaluated our analysis on four realistic Java examples. In one example, we compared our extracted dataflow edges with points-to edges Abi-Antoun and Aldrich [4] extracts using the SCHOLIA analysis. Points-to edges show reference relations between objects due to field references. However, our dataflow edges are usage edges. Using our dataflow edges, we visualized the messages exchanged between a subject and its observers in the Listeners example, which implements the Observer design pattern [19]. In another example, we compared our extracted dataflow edges with dataflow edges extracted on a flat object graph. Because our analysis can distinguish between objects in different domains, our analysis was able to extract more precise dataflow edges. Finally, we compared our analysis with another analysis that infers explicit and implicit information flow scenarios. We concluded that our analysis does not directly capture implicit information flow scenarios between objects. However, it did capture explicit information flow scenarios.

Chapter 5: Related Work

5.1 Static vs. Dynamic analysis

5.1.1 Static analyses

Fully automated static analyses Several static object graph analyses do not require any annotations [41, 25, 36, 18, 16, 49], but also extract flat, non-hierarchical object graphs which do not scale because they grow larger in size as a function of the size of the program. In our approach, we follow SCHOLIA in requiring annotations and extracting hierarchical object graphs, which can scale because the number of objects at the top level stays relatively small [2]. This thesis adopts the same definition of dataflow [41]. However, we show dataflow edges on a hierarchical object graph instead of a flat object graph. The work by Debzani et al. [16] assumes all dataflow scenarios are the result of method invocation statements. In addition to method invocations, our analysis handles dataflow due to field access statements. Because our analysis uses ownership domain annotations which give precision about aliasing, our analysis is able to distinguish between objects in different domains and therefore produces more precise dataflow edges than other analysis that do not use ownership domain annotations [41, 16, 49, 18]. These analysis works by first building a type graph from the type structure then unfolding it into an object graph. Finally, unlike Spiegel's Pangaea and its equivalent analyses [41, 16, 49, 18], we formalized our analysis with the goal of proving its soundness.[1]

Liu and Milanova [30] proposed a static analysis and object graph extraction algorithm. The objects on the object graph are connected using four kinds of edges; *create edges* due to object creation in allocation statements; *in edges* due to arguments passing in method invocations; *out edges* due to return statements; and *self edges* due

[1]While we conjecture but do not prove soundness, Spiegel's Pangaea is unsound with respect to aliasing [2].

to the passing of `this`. Both in and out edges cause objects to flow on the object graph. They are similar to export and import dataflow edges we used in this thesis.

Reverse engineering object graphs Tonella and Potrich [45] proposed a static analysis to extract an Object Flow Graph (OFG) from object-oriented source code. OFG is a result of a reverse engineering static analysis traces the flow of information from one object to another in object assignment statements; actual arguments substitution for formal method or constructor parameters in method invocations and allocation statements; and object returns from method invocations. Their approach requires rewriting the program source code into a simplified abstract language that maintains features related to object flow and omits other complex features of the language. Like our analysis, their approach deals with dataflows due to container objects differently.

Nodes in the OFG are represented by objects references. OFG requires a points-to analysis [9, 39, 33, 32] to obtain a static approximation of the points-to set of objects that a given reference may point to. Points-to analysis can either be context/flow sensitive or context/flow insensitive. While context/flow sensitive analysis produces accurate results, it is computationally expensive and does not scale to large programs. On the other hand, context/flow insensitive analysis has lower complexity, but it does not distinguish the invocation context of program statements. Our analysis does not require points-to analysis and instead relies on the aliasing precision provided to us by the ownership domain annotations. This enables our analysis to distinguish between objects in different domains and merge objects of compatible types if they are in the same domain.

Womble [25] is a lightweight static analysis tool that extracts object models from Java bytecode. Womble produces an unsound object model which consists of nodes and edges. Nodes represent Java classes and edges represent either subclassing or

associations. Unlike traditional class diagrams, Womble's object model shows association edge multiplicity for array fields and infers appropriate associations in handling container classes. For example, a container class such as vector is not appearing as a node in the object model. Instead, the correct association between the class that contains this vector and vector elements is appearing.

Annotation-based static analyses Lam and Rinard [27] proposed a type system and a static analysis whereby developer-specified annotations guide the static abstraction of an object model by merging objects based on tokens. Their approach supports a fixed set of statically declared global tokens, and their analysis shows a graph indicating which objects appear in which tokens. Since there is a statically fixed number of tokens, all of which are at the top level, an extracted object model is a top-level architecture that does not support hierarchical decomposition, thus limiting the scalability of the approach to large systems. The SCHOLIA approach extended Lam and Rinards both to handle hierarchical object graphs and to support object-oriented language constructs such as inheritance. Moreover, Lam and Rinard extract an "object model", a "subsystem access model", a "call/return interaction model", and a "heap interaction model", but do not particularly track the kind of dataflow information we do.

Slicing Program slicing is a technique to extract program parts with respect to a particular computation [51]. According to Weiser [50], for a given slicing criterion $< p, V >$, where p is a program point and V is a set of variables, a program slice with respect to the point p consists of the set of all statements that might affect the value of an arbitrary variable v in V at p. Alternative definitions of a program slice have been proposed in the literature [23]. Horwitz et al. [23] argue that one is often interested in how statements in a slice might affect a specific variable v in the set V at a program point p. Sridharan et al. [42] proposed thin slicing technique, which

includes only statements relevant to human in a slice and discard other unrelevant statements. For example, in graph construction thin slicing ignores edges due to flow or control dependencies that traditional slicing algorithm usually consider.

Procedural Dependence Graphs In order to compute program slices, procedural dependence graphs are used. A Procedural Dependence Graph (PDG) represents how information flows in a program [23]. A PDG is a graphical representation of a procedure where nodes are program statements and edges represent the dependence between the statements. In the PDG, a call statement is shown by a call node and the collection of the actual-out and actual-in nodes. Orso et al. [37] present an incremental slicing technique that initially starts with a slice that contains only statements as a result of stronger data dependencies, and incrementally augment the slice if weaker data dependencies are to be considered in the analysis. Slicing has been used for program understanding and to isolate the computation among threads in a multi-threaded program [50].

Liang and Harrold [29] refined the slicing definition for object-oriented programs by introducing the concept of object slicing. In object slicing, the analysis looks for statements of a particular object's method that might affect the slicing criterion. They argue that object slicing can be useful for debugging and impact analysis. Unlike object graphs, where the main focal point is the object, object slicing is more fine-grained because it considers dependencies between different objects' statements. Generally speaking, in object slicing graph construction, if there are missing edges between two objects' slices, then there will be definitely no dataflow edges between these two objects as well.

5.1.2 Dynamic analysis

Dynamic analysis constructs object graph by tracing a target program's execution on a set of test cases [45]. Each program execution must be associated with an execution trace. This trace should include information about the context object identifier. Execution traces can be obtained using tracing tools or by program instrumentation. One obvious limitation of dynamic analysis is that test cases may not cover all possible program executions. This leaves some object flows scenarios that exists in code but the analysis fails to capture.

Dynamic Tainting Dynamic taint analysis is quickly becoming a staple technique when conducting security analysis. The aim of dynamic taint analysis is to track information flow between a source and a sink. In order to analyze a program, a taint policy is used in determining exactly how a taint will flow during the execution, what operation could trigger a taint, and what kind of checks should be carried out on taints. Tainting has been used to prevent integrity-compromising attacks [34, 52], SQL injection [22, 46], enforcing data confidentiality [47], and recently detecting vulnerabilities in Web applications [46]. Dynamic tainting requires instrumentation during program execution and it has the limitations of other dynamic analysis techniques previously mentioned. Our analysis, however, is static and is completed at compile time and before program execution. It can be used to capture security vulnerability if a conformance approach used to expose the flow of objects across the program. Unlike the dynamic tainting approach, our analysis does not taint objects in order to track their propagation cross the program.

5.2 Applications of Dataflow Information

Checking conformance. One application of dataflow information is Data Flow Diagrams (DFDs). A DFD is an architectural view which visualizes the flow of

data in a system [44, 7]. Extracting dataflow information could enable checking the conformance of an implementation to a target architecture [6].

Program comprehension. Dataflow information can be potentially useful for developers performing code modification tasks. Objects dependencies that are discovered by dataflow edges can direct developers to the right decisions on code optimization and code refactoring. Unfortunately we cannot support this argument with evidences, but we expect to see future studies similar to [5] that investigate this application further.

5.3 Information Flow Control

Information Flow Control (IFC) refers to the procedure of ensuring that transfer of information in a given system does not occur from an object with high security information to one with a lower security level [21]. This reduces the possibility of information disclosure between security levels. Liu and Milanova [31] categorize the flow of information into explicit (flow of information which arises due to the use of assignment statements) or implicit (flow of information which is the result of using conditional statements). Liu and Milanova [31] proposes a new static information flow inference analysis that infers explicit and implicit information flows. The analysis is context sensitive and can be applied to detect security violations in a client application. Sun et al. [43] proposes a specification of a modular algorithm to infer security types for a sequential, class-based, and object-oriented language. Both classes and methods are parametrized by security levels (e.g., Low, High). Unlike Liu and Milanova [31], the inference algorithm is proven sound. Although our analysis is not as precise in detecting security flaws as IFC, it still can capture interesting dataflows between objects.

Chapter 6: Discussion

6.1 Validation of Thesis Statement

The goal of this thesis was to introduce a static analysis to extract sound and precise dataflow edges from object-oriented programs with ownership domain annotations. We divided our thesis statement into two hypotheses. In the following sections, we discuss how we satisfied these hypotheses.

6.1.1 H1: Sound dataflow edges

We presented a formal definition of the analysis using constraint-based inference rules to ease a formal proof of soundness. We implemented and evaluated the analysis on realistic Java code. Finally, we showed that the extracted object graph indeed made visually obvious all of the dataflow communication in a program.

6.1.2 H2: Precise dataflow edges

We confirmed that the analysis extracts precise dataflow information (Chap. 4). We evaluated our analysis on realistic Java programs and confirmed that the extracted dataflow edges were all true positives.

6.2 Satisfaction of the Requirements

In Chapter 1.2, we outlined a set of requirements for our analysis. In this section, we discuss how these requirements were satisfied.

6.2.1 Soundness of the Extracted Dataflow Edges

We formalized the analysis using constraint-based inference rules (Chapter 3). Although we did not formally prove soundness, we conjecture that the analysis is sound. We evaluated our analysis on realistic Java code and showed that our analysis did not miss a dataflow scenario that exists in code without showing it on the object graph. The formal proof of soundness is left for future work.

6.2.2 Precision of the Extracted Dataflow Edges

Our analysis confirms this requirement by an evaluation on realistic Java programs (Sec. 2.6). An example of highly imprecise dataflow edges would be to show a dataflow edge between every two objects on the object graph (i.e., every object flows to every other object). In practice, we demonstrated that our analysis extracts object graphs that have few false positives on small, but realistic programs.

6.3 Limitations

Our current implementation is a proof-of-concept prototype which is not integrated with the Scholia tool set.

6.3.1 Missing features in the implementation

Our prototype implementation currently does not handle all the features of the Java language, such as interfaces or static code. Also, our prototype does not handle library code. In principle, we could rely on external files to add annotations to the portions of external libraries that are in use, as in the Scholia approach [2, Appendix A].

6.3.2 Scholia's visualization of object graphs

Our prototype implementation extracts only an ObjectGraph or OGraph. In particular, we do not deal with the DisplayGraph or DGraph in SCHOLIA [2, Section 3.4]. In SCHOLIA, the DGraph is the object graph that the tool displays to a developer, and with which the developer interacts. Integrating our analysis with the SCHOLIA toolset will enable:

- Using the same style of visualization based on box nesting as in SCHOLIA; we would be able to generate more compact object graphs, by collapsing all the objects except the ones in the top-level domains;
- Allowing the user to control the unfolding depth of the ObjectGraph into a DisplayGraph, to expand or collapse the sub-structure of individual objects, while the tool automatically adds lifted edges [2, Fig. 2.27:];
- Using the abstraction by types feature on the DisplayGraph, which enables collapsing objects in a domain further based on their declared types. Abstraction by types is often necessary to extract meaningful OOGs for larger programs.

6.3.3 Scholia's conformance analysis

Integrating our analysis with the SCHOLIA toolset will enable also us to follow the same *extract-abstract-check* strategy as SCHOLIA to analyze conformance. The steps would be as follows:

1. Add annotations to the code and type-check them;
2. *Extract* a *sound* object graph that conveys architectural abstraction by hierarchy and by types;
3. *Abstract* an extracted object graph into an as-built runtime architecture;
4. Document the target, as-designed runtime architecture;
5. *Check* the conformance between the as-built and the as-designed architectures.

One special kind of a target architecture is a security architecture which shows dataflow edges. When both the as-built and the as-designed architectures show dataflow edges, it is possible to analyze the conformance of an implementation to a Data Flow Diagram (DFD) used in security threat modeling [6].

6.4 Future Work

Future work could include three main directions. First, we need a formal proof of soundness. Second, we will consider making the analysis sensitive to the program's control flow to minimize the number of false positives, thus increasing the precision of the extracted dataflow edges. Finally, we would like to conduct a thorough evaluation of the analysis on real and large Java programs.

6.5 Conclusion and Broader Impact

This work is a novel application of ownership types. We assume the presence of ownership domain annotations in the code. Our static analysis leverages these annotations to extract a hierarchical object graph with dataflow edges.

We informally described the analysis on one example and formalized it using constraint-based inference rules and conjectured that the analysis is sound. We implemented the analysis and tested it on several examples that showed that our analysis achieved better results than other static analyses that perform on top of plain Java programs that do not have annotations. The extracted graphs were useful because they made certain dataflows in the program visually obvious.

Our analysis extracts additional information from code that could potentially be useful for developers. Another application would involve analyzing conformance between an implementation and a target security architecture which shows dataflow information. Additionally, the dataflow edges shown on our object graphs can be

potentially useful for developers performing code modification tasks. They can assist in locating where to implement a code change and what program's entities may be directly and indirectly affected by the implemented change.

REFERENCES

[1] Eclipse Java Development Tooling (JDT) core. `http://www.eclipse.org`, 2006.

[2] Abi-Antoun, M. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure.* PhD thesis, Carnegie Mellon University, 2010. Available as Technical Report CMU-ISR-10-114.

[3] Abi-Antoun, M., and Aldrich, J. Ownership Domains in the Real World. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)* (2007).

[4] Abi-Antoun, M., and Aldrich, J. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009).

[5] Abi-Antoun, M., and Ammar, N. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *Workshop on Human Aspects of Software Engineering (HAoSE), co-located with SPLASH/OOPSLA* (2010).

[6] Abi-Antoun, M., and Barnes, J. M. Analyzing Security Architectures. In *Automated Software Engineering* (2010).

[7] Abi-Antoun, M., Wang, D., and Torr, P. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security (Short Paper). In *Automated Software Engineering* (2007).

[8] Aldrich, J., and Chambers, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)* (2004).

[9] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[10] Bennett, K. H., Rajlich, V., and Wilde, N. Software evolution and the staged model of the software lifecycle. *Advances in Computers 56* (2002), 3–55.

[11] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. Secure web applications via automatic partitioning. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (2007), pp. 31–44.

[12] Clarke, D., and Drossopoulou, S. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2002).

[13] Clarke, D. G., Noble, J., and Potter, J. Simple Ownership Types for Object Containment. In *European Conference on Object-Oriented Programming (ECOOP)* (2001).

[14] Clarke, D. G., Potter, J. M., and Noble, J. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1998).

[15] Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.

[16] Deb, D., Fuad, M. M., and Oudshoorn, M. J. Towards autonomic distribution of existing object oriented programs. In *International Conference on Autonomic and Autonomous Systems (ICAS'06* (2006), pp. 17–23.

[17] Détienne, F. *Software design—cognitive aspects*. Springer-Verlag, 2002.

[18] Diaconescu, R. E., Wang, L., Mouri, Z., and Chu, M. A Compiler and Runtime Infrastructure for Automatic Program Distribution. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2005).

[19] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[20] Gansner, E. R., and North, S. C. An Open Graph Visualization System and its Applications to Software Engineering. *Software: Practice & Experience 30*, 11 (2000), 1203–1233.

[21] Genaim, S., and Spoto, F. Information flow analysis for java bytecode. In *Verification, Model Checking, and Abstract Interpretation*, vol. 3385 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 346–362.

[22] Halfond, W. G. J., Orso, A., and Manolios, P. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Foundations of Software Engineering (FSE)* (2006), pp. 175–185.

[23] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In *Programming Language Design and Implementation (PLDI)* (1988), pp. 35–46.

[24] Igarashi, A., Pierce, B., and Wadler, P. Featherweight Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1999).

[25] Jackson, D., and Waingold, A. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering 27*, 2 (2001).

[26] Kollman, R., Selonen, P., Stroulia, E., Systä, T., and Zundorf, A. A Study on the Current State of the Art in Tool-Supported UML-Based Static

Reverse Engineering. In *Working Conference on Reverse Engineering (WCRE)* (2002).

[27] Lam, P., and Rinard, M. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *European Conference on Object-Oriented Programming (ECOOP)* (2003).

[28] LaToza, T. D., and Myers, B. A. Developers ask reachability questions. In *International Conference on Software Engineering (ICSE)* (2010), pp. 185–194.

[29] Liang, D., and Harrold, M. J. Slicing objects using system dependence graphs. In *International Conference on Software Maintenance (ICSM)* (1998), pp. 358–367.

[30] Liu, Y., and Milanova, A. Practical Static Ownership Inference. Tech. Rep. RPI/DCS-09-04, Rensselaer Polytechnic Institute, 2009.

[31] Liu, Y., and Milanova, A. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *European Conference on Software Maintenance and Reengineering (CSMR)* (2010), pp. 146–155.

[32] Milanova, A. Light Context-Sensitive Points-To Analysis for Java. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (2007).

[33] Milanova, A., Rountev, A., and Ryder, B. G. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM Transactions on Software Engineering and Methodology 14*, 1 (2005).

[34] Newsome, J., and Song, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In ACM Network and Distributed System Security Symposium* (2005).

[35] ObjectAid. The ObjectAid UML Explorer for Eclipse. `www.objectaid.com/`.

[36] O'Callahan, R. W. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, CMU, 2001.

[37] Orso, A., Sinha, S., and Harrold, M. J. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Trans. Softw. Eng. Methodol. 13* (April 2004), 199–239.

[38] PLAID Research Group. The Crystal Static Analysis Framework, 2009. `http://code.google.com/p/crystalsaf`.

[39] Rountev, A., Milanova, A., and Ryder, B. G. Points-to Analysis for Java using Annotated Constraints. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2001).

[40] Sagiv, M., Reps, T., and Wilhelm, R. Parametric Shape Analysis via 3-Valued Logic. In *Principles of Programming Languages (POPL)* (1999).

[41] Spiegel, A. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.

[42] Sridharan, M., Fink, S. J., and Bodik, R. Thin slicing. In *Programming Language Design and Implementation (PLDI)* (2007), pp. 112–122.

[43] Sun, Q., Banerjee, A., and Naumann, D. A. Modular and constraint-based information flow inference for an object-oriented language. In *In Proc. of the Eleventh International Static Analysis Symposium (SAS* (2004), pp. 84–99.

[44] Swiderski, F., and Snyder, W. *Threat Modeling*. Microsoft Press, 2004.

[45] Tonella, P., and Potrich, A. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[46] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. Taj: effective taint analysis of web applications. In *Programming Language Design and Implementation (PLDI)* (2009), pp. 87–97.

[47] Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M., and August, D. I. Rifle: An architectural framework for user-centric information-flow security. In *In MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture* (2004), IEEE Computer Society, pp. 243–254.

[48] Waingold, A., and Lee, R. SuperWomble Manual. `http://sdg.lcs.mit.edu/womble/`, 2002.

[49] Wang, L., and Franz, M. Automatic Partitioning of Object-Oriented Programs for Resource-Constrained Mobile Devices with Multiple Distribution Objectives. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)* (2008).

[50] Weiser, M. Program slicing. In *International Conference on Software Engineering (ICSE)* (1981), pp. 439–449.

[51] Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes 30* (2005), 1–36.

[52] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (2006).

ABSTRACT

A STATIC ANALYSIS TO EXTRACT DATAFLOW EDGES FROM OBJECT-ORIENTED PROGRAMS WITH OWNERSHIP DOMAIN ANNOTATIONS

by

SUHIB RAWSHDEH

May 2011

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science

Degree: Master of Science

For program comprehension, developers often require complementary sources of information to understand a software system. They use information about the code structure (class diagrams), points-to field references, control flow (call graphs), and dataflow. Using static analysis to extract dataflow information from object-oriented code is challenging because the analysis must deal with language features such as inheritance, recursion, and aliasing. Existing analyses extract flat graphs that show a large number of objects and lack architectural abstraction. Furthermore, some existing analyses suffer from imprecision. An adoptable analysis should aim for a judicious tradeoff between precision and scalability.

In order to extract information that conveys design intent, we rely on annotations in the code. The annotations implement the Ownership Domains type system, by Aldrich and Chambers. An *ownership domain* is a conceptual group of objects where each object belongs to one domain that does not change at runtime. The developer annotates each object reference in the program with the domain that owns it.

In this thesis, we leverage the ownership domain annotations that are present in a program and propose a static analysis to extract a hierarchical object graph that

shows objects and edges that convey dataflow information. We informally describe the analysis on a Listener example, formalize it using constraint-based inference rules and conjecture that the analysis is sound. We implement the analysis and evaluate it on real object-oriented code to evaluate its precision in practice. We confirm that the analysis extracts precise dataflow information. The extracted graphs make certain dataflows in the program visually obvious. Such diagrams could be potentially useful for developers performing code modification tasks. We also compare the analysis with another analysis that extracts flat object graphs and we show that our analysis is more precise.

AUTOBIOGRAPHICAL STATEMENT

SUHIB RAWSHDEH

EDUCATION

- Master of Science (Computer Science), May 2011
  Wayne State University, Detroit, MI, USA
- Bachelor of Engineering (Computer Engineering), June 2007
  Jordan University of Science and Technology, Jordan