Synthesizing Framework Uses from Program Behavior Data

by

Zhilei Xu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2011

Certified by .....
Armando Solar-Lezama
Assistant Professor
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Synthesizing Framework Uses from Program Behavior Data

by

Zhilei Xu

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis presents MATCHMAKER, a new synthesis tool that aims to help programmers use software frameworks by synthesizing source code needed to interact with the framework.

Software engineers of today are constantly faced with the task of using or extending large software code bases. This proves to be a challenging endeavor, as object-oriented frameworks tend to grow exceedingly intricate. Functionality is spread out among numerous classes and tailoring it for a specific need requires knowledge of exact components to extend and combine.

MATCHMAKER is presented to help programmers understand such complex behavior, especially, to help deal with one common task in using frameworks: connecting two classes so that they can interact with each other. Taking as input two classes that the programmer want to connect, MATCHMAKER observes many real runs of the framework, aggregates relevant execution traces in which similar connections are built by client-framework interactions, and synthesizes the necessary source code the user needs to write to make the connection possible.

MATCHMAKER relies on the hypothesis that the logical connection between two objects is fulfilled by a chain of pointer references linking them together, and the earliest possible pointer reference chain (called *Critical Chain*) is critical to the logical connection. MATCHMAKER employs a new program behavior data engine (called DELIGHT) to find the critical chain, uses a special slicing algorithm to dig out the relevant instructions which form the client-framework protocol from the critical chain, and synthesize the client code from the slices.

In this thesis we also demonstrate MATCHMAKER's capability on a range of programming tasks using complex software frameworks such as Eclipse, and evaluate MATCHMAKER's usability and its improvement to programming efficiency by comprehensive user study.

Thesis Supervisor: Armando Solar-Lezama
Title: Assistant Professor

Acknowledgments

I would like to thank my advisor, Professor Armando Solar-Lezama for his advice, inspiration, and encouragement. Without his help this thesis cannot come to what it is now.

I want to thank my colleague, Kuat Yessenov who have made every effort to help me with the project. The DeLight engine was mainly developed by him, and provides a rock hard foundation for MatchMaker.

My fellow CSAIL graduate students Jean Yang and Rishabh Singh have never failed to provide inspiring discussion and helpful feedback.

I also want to thank Professor Rastislav Bodik who gives valuable advice to the project since its very early stage.

Finally I would like to thank Bin Lian for being in my life for last nineteen years. I could never imagine a life without her.

Contents

List of Figures

Chapter 1

Introduction

Modern programming relies heavily on rich extensible frameworks that pack large amounts of functionality for programmers to draw upon. The advent of these frameworks has revolutionized software development, making it possible to write applications with rich functionality by simply piecing together pre-existing components. But the productivity benefits come at a price: a steep learning curve as programmers struggle to master a complex framework with tens of thousands of components and millions of lines of code. As this thesis will show, synthesis can alleviate this problem by leveraging the programmer's limited understanding of the system to generate code that uses the framework correctly.

In order for synthesis to succeed in this domain, however, the synthesizer needs to cope with the same challenges that stymie professional programmers: scale and dynamic behavior. The Eclipse ecosystem, for example, has over 33 million lines of code [1], and like many other frameworks, it achieves flexibility through aggressive use of dynamic dispatch and reflection. Together, scale and dynamic behavior make detailed semantic analysis impractical, limiting the applicability of traditional synthesis methodologies [13]. On the other hand, the reusable nature of frameworks implies that the same functionality is used in different combinations by different applications; this suggests a more empirical approach to synthesis, where one can discover the correct usage of a component by analyzing how it has been used in different contexts. This thesis explores this empirical approach to software synthesis in the context of a well-defined programming challenge: establishing an interaction between two classes in a framework.

1.1 The Matching Problem

The problem we are solving arises from the way object-oriented frameworks tend to atomize functionality into large numbers of classes, each dealing with a very specific aspect of a task. This atomization implies that functionality that should logically require the interaction of two objects also requires a number of auxiliary objects that have little intuitive meaning to the novice programmer. For example, a programmer trying to implement syntax highlighting for an editor written on top of the Eclipse

framework will quickly discover that he needs to use a RuleBasedScanner to identify the different lexical classes that need to be shown in different colors.

The problem, with matching the two classes is that the interaction between the editor and the scanner is mediated by a number of classes such as SourceViewerConfiguration, PresentationReconciler and – most confusingly – DefaultDamagerRepairer; classes whose function can only be understood if one understands something about the internal workings of Eclipse. As a result, even after the programmer knows she needs to write a RuleBasedScanner and make it interact with the text editor, it still takes her a long time to discover all the auxiliary classes that are needed to establish this interaction and to figure out how to use them. Therefore, we can significantly increase programmer productivity by automatically synthesizing the glue code that allows two classes to interact with each other.

1.2 Technical Approach

The focus of this thesis is MATCHMAKER, the synthesis tool that automatically generates the glue code to enable the interaction of two classes. The tool takes as input the names of two classes that the user wants to use and produces the necessary glue code, including any methods that need to be overridden, classes that need to be instantiated and APIs that need to be called.

MATCHMAKER is based on DELIGHT, a data collection, management and analysis engine. The data collector in DELIGHT allows us to build rich databases of program behavior in real time. The database contains detailed information about control transfer in the application, as well as information about the dynamic evolution of the heap. Using a novel technique that uses abstraction as a mechanism for data indexing, DELIGHT is able to efficiently answer the questions about the evolution of heap connectivity, thus provide the basis for MATCHMAKER, the synthesizer.

1.3 Contributions

The most important contribution of this thesis is to demonstrate how synthesis can be enabled by a new form of *data-driven* program analysis that is qualitatively different from more traditional dynamic analysis. At the practical level, maintaining a database of program behavior eliminates one of the main shortcomings of dynamic analysis: the need for the user to be able to run the program before the analysis can yield any useful results. Instead, programming tools that rely on the data driven model can offer push-button responsiveness by tapping into shared data store such as DELIGHT. More fundamentally, the feature that distinguishes data-driven analysis from traditional dynamic analysis is that instead of analyzing the runtime behavior "live" as the program executes, the execution data is stored and indexed to support deep queries about the program behavior over time and across runs. The distinction is somewhat akin to the difference between a journalist and a historian. Journalists provide a direct view of the events of the day, but historians are able to piece together the

records of the past to provide a more complete view of events. In a similar way, our system is able to use the program execution data to understand how connections between different kinds of objects arise in the heap and what programmers need to do to create them.

In order to achieve our goal, this thesis makes a number of important technical contributions:

- We empirically show that one can infer the proper way to establish an interaction between two objects by focusing on how chains of pointers are created between them, and introduce the concept of *Critical Chain*, as well as the method to find critical chain efficiently.
- We propose a slicing algorithm based on thin slicing to dig out relevant instructions enabling the client-framework interaction.
- We explore algorithms for normalizing the slices to get the interesting instructions that form the protocol between client and framework for the interaction, and propose a synthesis algorithm to generate necessary client code for the client-framwork interaction from the normalized slices.
- We present MATCHMAKER, a new synthesis tool that can synthesize necessary source code for using the framework to connect two classes together, and demonstrate its usefulness.
- We give results of user study to evaluate the usability and effectiveness of MATCHMAKER.

1.4 Organization of the thesis

In Chapter 2, we describe the overview of the MATCHMAKER through a running example. Chapter 3 describes the preliminaries of MATCHMAKER, especially DELIGHT, the data collection and query engine that MATCHMAKER based on. Chapter 4 introduces the concept of *critical chain*, and gives an algorithm to find it. Chapter 5 details the slicing algorithm that MATCHMAKER uses to dig out relevant instructions from the critical chain to fulfill the client-framework protocol. Chapter 6 presents the algorithms to normalize the slicing result and to synthesize the source code from the normalized slice. Chapter 7 discusses various related works. Chapter 8 evaluates the generality of MATCHMAKER. Chapter 9 presents user study results to demonstrate the usability and improvement to programmers' productivity of MATCHMAKER. Finally Chapter 10 concludes the thesis and discusses future work directions.

Chapter 2

Overview

In this chapter, we elaborate on the running example introduced earlier involving the syntax highlighting functionality in Eclipse. We use this example to illustrate the challenges introduced by object oriented frameworks and to describe how MATCHMAKER attacks this problem through a data driven approach to synthesis.

2.1 Example

Eclipse makes it easy to add syntax highlighting to a user-created editor by defining an ITokenScanner interface. The framework is designed to apply the user's implementation of the scanner to identify different kinds of tokens in a text file and highlight them in different colors. However, if you are new at creating Eclipse plugins, you may find yourself struggling to understand how to get the framework to take your scanner and use it in the context of your editor.

The problem is that the editor doesn't use the scanner directly; the editor actually interacts with a component called the SourceViewer, which manages add-ons to the editor. The SourceViewer in turn uses a PresentationReconciler to maintain a representation of the document in the presence of changes. The PresentationReconciler uses an IPresentationDamager to identify changes to a document, and an IPresentationRepairer to incrementally scan those changes, and it is these two classes that interact directly with the scanner. If the programmer wants to get these classes to use a different scanner, she actually has to write her own SourceViewerConfiguration class and override the getPresentationReconciler() method to return a PresentationReconciler whose IPresentationDamager and IPresentationRepairer reference the new scanner. She then has to register her SourceViewerConfiguration with the editor by calling setSourceViewerConfiguration (myConfiguration) in the initializer of the editor. The code to do this is illustrated in Figure 2-1.

If you found yourself struggling to follow the preceding paragraph, you are not alone. The sheer number of classes involved makes it difficult, but things are made worst because the names of auxiliary classes are outright cryptic to someone unfamiliar with Eclipse concepts (e.g. what does damage repairing have to do with syntax highlighting?). The underlying problem is a basic tension between flexibility and us-

```
// inside Framework
class TextEditor {
  SourceViewerConfiguration fConfiguration;
  ISourceViewer fSourceViewer;
  createPartControl() {
    fSourceViewer = createSourceViewer();
    fSourceViewer.configure(fConfiguration);
  }
  setSourceViewerConfiguration(configuration) {
    fConfiguration = configuration;
  }
}

class SourceViewer {
  IPresentationReconciler fPresentationReconciler;
  configure(SourceViewerConfiguration configuration) {
    fPresentationReconciler =
      configuration.getPresentationReconciler();
  }
}

// code by User
class MyConfiguration extends SourceViewerConfiguration {
  IPresentationReconciler getPresentationReconciler() {
    PresentationReconciler reconciler =
      new PresentationReconciler();
    RuleBasedScanner myScanner = new MyScanner();
    DefaultDamagerRepairer dr =
      new DefaultDamagerRepairer(myScanner);
    reconciler.setRepairer(dr, DEFAULT_CONTENT_TYPE );
    reconciler.setDamager(dr, DEFAULT_CONTENT_TYPE);
    return reconciler;
  }
}

class MyTextEditor extends TextEditor {
  MyTextEditor() {
    MyConfiguration mySourceViewerConfiguration = new MyConfiguration();
    setSourceViewerConfiguration(mySourceViewerConfiguration);
  }
}
```

Figure 2-1: Code required to let an editor use a specific scanner. MATCHMAKER is able to synthesize all the code above except for the part highlighted in gray.

ability. The design of Eclipse achieves flexibility by factoring functionality into large numbers of components, each responsible for a very specific aspect of the functionality. This is a challenge for usability because it means that interactions that look simple at the high level actually require the collaboration of large number of objects. Synthesis offers a way around this basic tension by bridging the gap between the programmer's high-level goal and the low-level component interactions that need to be established in order to achieve it. In doing so, synthesis reduces the usability burden created by highly modular framework designs.

2.2 The MatchMaker Approach

MATCHMAKER is able to synthesize code like the one in Figure 2-1 from a simple query of the form: "How do I get an editor and a scanner to interact with each other?". More generally, given two object types A and B, MATCHMAKER identifies what the user of the framework has to do in order for two objects of these types to work together.

In order for a tool to do this, the first problem that has to be addressed is to give semantic meaning to the query; *i.e.* what does it even mean for two objects to "work together" or to "interact with each other"? Our system gives semantic meaning to these concepts by exploiting a new hypothesis about the design of object oriented frameworks.

Hypothesis 1 *The* MATCHMAKER *hypothesis*

In order for two objects to interact with each other, there must be a chain of references linking them together. Therefore, the set of actions that led to the creation of the chain is the set of actions that need to take place to enable the interaction.

The MATCHMAKER hypothesis does not always hold; sometimes, for example, two objects can interact with each other by modifying the state of some globally shared object, without the need of having a chain of references that connects them. Nevertheless, we have experimental evidence to suggest that the hypothesis is true often enough that a tool based on this hypothesis can have a real impact on programmer productivity (see Chapters 8 and 9).

The hypothesis is useful because it suggests a relatively simple algorithm to answer the programmer's query. In the case of the running example, the algorithm works like this:

- Take a set of editors written on top of Eclipse that implement syntax highlighting.
- Identify code in the implementation of these editors that contributes to the creation of a chain of references from an editor to a scanner.
- Normalize the code to remove arbitrariness specific to individual example trace, and present to the user the most general way to achieve the goal.

This is the basic algorithm behind our synthesis tool; the rest of this chapter will elaborate more on the specific challenges that arise for each of the high-level steps

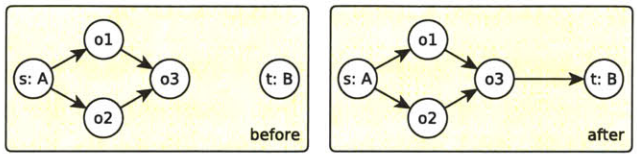

Figure 2-2: Event $o3.f \leftarrow t$ establishes the critical link between s and t and creates two critical chains.

outlined above. The biggest challenge presented by the algorithm is that it requires reasoning about the evolution of the heap with a level of precision that is arguably unachievable by static analysis, particularly given the scale of the frameworks involved and their aggressive use of reflection. The way around this problem is to follow a data-driven approach: rather than try to reason statically about existing implementations of different editors, we collect information about their execution, and organize it in a way that allows one to efficiently answer even deep queries about the evolution of the heap.

2.2.1 Finding Critical Chains with DeLight

The algorithm outlined above requires us to identify code in the implementation of existing editors that leads to the creation of a chain of references from one object, call it the *source*, to another object we call the *target*. Because we are following a data-driven approach, we are going to be identifying these pieces of code by using information collected from concrete executions of these editors.

The first step in this process is to find events in each execution where the source and target objects become linked by a chain of references. In the specific case of the editor and the scanner, we are looking for events in the execution such that before this event, the scanner cannot be reached from the editor, but after the event it can. We call the reference created by this event the *critical link* between the editor and the scanner, and any chain of references from the source to the target object that was created as a consequence of adding the critical link is called a *critical chain*. Note that while the critical link is unique, there can be many critical chains as illustrated in Figure 2-2.

Conceptually, we can find a critical link by running a depth-first-search on the heap after every memory write operation. When the critical link is formed, DFS will finally succeed in reaching the target object from the source and we will have found the event we seek. Unfortunately, this naïve strategy is as inefficient as it sounds; the cost of running DFS over the entire heap after every memory update would be prohibitively expensive for all but the shortest of traces. Instead, DeLight can compute critical chains very efficiently because it uses sophisticated abstractions to organize and index execution data, so MatchMaker just uses DeLight to compute the critical chain.

2.2.2 Turning Critical Chains into Code

The critical chain computation gives us a set of events that created a chain of references from the source to the target object, but these events are usually field updates deep inside the framework, and have no real significance for the programmer. What the synthesizer is looking for is the set of actions—including method overrides, API calls, class instantiations, etc.—that the *client* code needs to effect in order for those field updates deep inside the framework to take place.

Our algorithm for identifying the client code is described in Chapter 5 and is based on a dynamic form of *thin slicing* [14]. An important feature of thin slicing is that when a value is passed via a container—as is commonly done in a high-level language like Java—the slice only contains the event that placed the value in the container, but omits any events that simply manipulated the container. The resulting slice contains the set of events relevant to the creation of the critical chain. By separating events that took place in client code from those that took place inside the framework, the algorithm identifies those events that are relevant from the point of view of the user. Moreover, the slice provides information about relevant methods in the user's code that are called by the framework; these are methods that the user needs to override so the framework can use them to invoke the client code.

The slices that result from each trace contain information about how the interaction was established in that particular trace. Some of the information in the slices, however, may be too specific to a particular use. Our system copes with this by normalizing the traces to remove extraneous details about the structure of the different clients. This exposes a large degree of similarity between clients, and allows us to identify distinct patterns of interaction between the client and the framework.

After a small amount of post processing to make the code presentable and give names to any new subclasses, the synthesizer produces code like the one shown in Figure 2-1 (except for the highlighted portions; we explain why in Chapter 8).

Chapter 3

Preliminaries

This chapter gives the preliminaries of MATCHMAKER: the terminologies that we will use throughout the thesis and the DELIGHT engine and its data model that MATCHMAKER based on.

3.1 Terminology

This section gives some important terminologies that we will use hereafter.

3.1.1 Framework

An Object-oriented *Framework* (we will just call it Framework for short) is commonly considered to be an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code, thus providing specific functionality [2]. The common functionalities provided by the framework are implemented by a number of *components*, where each component is a class or a set of closely-related classes, and the user can select the components she wants and piece them together to form some feature, or extend and customize some components by overriding the corresponding classes.

Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined application programming interface (API), yet they contain some key distinguishing features that separate them from normal libraries:

Inversion of control In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.

Default behavior A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.

Extensibility A framework can be extended by the user usually by selective overriding or specialized by user code providing specific functionality.

Non-modifiable framework code The framework code, in general, is not allowed to be modified. Users can extend the framework, but not modify its code.

3.1.2 Client

Client (or *Client Code*) is the source code written by the user to use the framework. Usually the client contains two parts:

Task Code is the source code that overrides some components in the framework and specialize the component to fulfill some special needs of the user for a specific task.

Glue Code is the source code that pieces components in the framework together. Glue code does not implement any customized feature for the task, but specifies the way components interact with each other, and organizes the components to form some feature.

3.2 DeLight and its Data Model

DeLight is a program behavior database management system that collects and stores many execution traces of real world applications, and can answer queries related to the execution traces very efficiently. DeLight is mainly developed by Kuat Yessenov [3] in CSAIL.

The MatchMaker and the underlying DeLight data engine relies on three complementary views of execution data that can be used in tandem to answer complex questions about program behavior. The first is the *call tree presentation*, which directly models the sequence of instructions executed by each thread and the nesting of method calls. The advantage of this presentation is that for any point in the execution, it provides detailed information focused around that point in time. This representation is good for performing time-based analysis such as slicing, and will be used in Chapter 5.

The main drawback of the call tree presentation is that it is hard to answer global queries without looking at the entire trace. To address this problem, DeLight provides a complementary graph-based presentation that provides a global view of the evolution of the heap, called *heap series*. The two presentations are connected via time stamps that are assigned to every program instruction. The graph presentation makes it possible to answer heap connectivity queries by traversing objects via their fields.

The heap series graph is massive since it has every object ever created as a node and every reference ever updated as an edge. In order to make queries on this graph more tractable, DeLight employed another representation called *heap abstractions* that capture the essential domain information from the heap series and reduce its size, and can be used to speed up heap queries like critical chain computations.

Here we will only describe the call tree presentation, because the other two representations are solely used inside DELIGHT to answer critical chain computations and not the focus of this thesis.

Call Tree Presentation is essentially a sequence of events. An event is triggered for every state update and every transition across method boundaries:

Type	Description
$a \leftarrow b.f$	Read of value a from field f of object b.
$a \leftarrow f$	Read from a static field f.
$a \leftarrow b[i]$	Read of value a from array b.
$b.f \leftarrow a$	Write of value a into field f of object b.
$f \leftarrow a$	Write to a static field f.
$b[i] \leftarrow a$	Write of value a into array b.
call $m(\mathbf{p})$	Method enter.
return a	Normal exit of a method.
throw e	Exceptional exit of a method.

The sequence $\mathbf{p}$ in method enter events is the sequence of parameters to the method call, starting with **this** for non-static methods. Values $\mathcal{V}$ in our model consist of object instances, the special value **null**, and primitive values (integer, **void**, *etc.*) type(a) denotes Java type of value a. Each event is assigned a unique timestamp (or as we call it later, its time), which among other things allows us to assign a total order to events executed by different threads. Conceptually, method enter and exit events for a single thread form a call tree where the leaf nodes are the state reads and writes. This presentation provides a pre-order traversal of the call tree, allowing us to query information in the dynamic call scope of any given event.

Chapter 4

Critical Chain

This chapter formally defines the concept of *critical chain* introduced in Chapter 2.

Let $\mathcal{H}_t$ denote the heap at time point t, where each $\mathcal{H}_t$ is a directed graph in which nodes are objects and edges are pointer references. An edge $a \xrightarrow{f} b \in \mathcal{H}_t$ means in $\mathcal{H}_t$, two objects a and b are connected by a pointer reference ($a.f = b$). Let $\widehat{\mathcal{H}}$ denote the *heap series* derived by summing up all the heaps. $\widehat{\mathcal{H}}$ is a directed graph in which nodes are objects and edges are pointer references labeled with their living time intervals. An edge $a \xrightarrow{(f,T)} b \in \widehat{\mathcal{H}}$ (where T is a time interval) means that during time interval T, $a.f = b$.

A chain is a simple path in the heap connecting objects via directed edges labeled with fields. As the heap evolves over time, chains form and disappear, but with the entire sequence of heaps at hand, we have the power to answer the following question:

> *what is the earliest moment when the two given objects got connected by a chain?*

We call the event corresponding to this moment a *critical event*, and any chain between the two relevant objects created by the critical event is called a *critical* chain (recall from Figure 2-2 that there can be more than one). Formally, a critical event occurs at the minimal time t for which there is a chain in heap $\mathcal{H}_t$ connecting the two objects of interest. Any chain in $\mathcal{H}_t$ connecting the two objects will be a critical chain.

The critical event query can be formulated as a data flow equation on $\widehat{\mathcal{H}}$. Let us denote the time interval during which objects a and b are connected via some viable chain as $\mathsf{viable}(a, b)$. It is the union of lifetimes of all viable paths between a and b, as described by the following inductive definition:

$$\mathsf{viable}(a, b) = \bigcup_{c \xrightarrow{(f,T)} b \in \widehat{\mathcal{H}}} (\mathsf{viable}(a, c) \cap T)$$

(union is taken over incoming edges of b.)

The right hand side is monotonic in $\mathsf{viable}(a, \cdot)$ and, thus, could be used for the least fixed point computation with initial values $\mathsf{viable}(a, b) = \bot$ for $a \neq b$ and $\mathsf{viable}(a, a) = \top$. The minimal time in $\mathsf{viable}(a, b)$ is the critical time for a and b.

However, applying the equation directly to $\widehat{\mathcal{H}}$ is intractable due to the size of the graph (millions of objects), the required number of iterations (long paths in the heap), and complex time intervals (millions of field writes.)

Instead, MATCHMAKER builds on DELIGHT which can find critical events and critical chains very efficiently. DELIGHT provides the query interface as follows:

Find a critical chain between two classes The user gives DELIGHT the names of two endpoint classes, and pick the program execution trace to work on, then DELIGHT returns a critical chain (in the form of all the heap events that create all the links on the chain) between two objects of the two endpoint classes if there exists one, or returns NotFound if there is no chain between the two classes.

Chapter 5

Slicing algorithm

The critical chain computation produces a set of references that connect the two objects of interest. Each of these references has an associated event in the database corresponding to the field update that created it. So in principle, if we want to know what events led to the chain of references, these are the events. However, these events are usually low-level events deep inside the framework; what the user really wants to know is what code to write in the client in order to get the relationship between the two classes established.

Intuitively, the solution to this problem is slicing. The idea would be to compute a dynamic slice using as the slicing criterion each of the events that created the links in the critical chain. Some of the events in the slice will be internal to the framework, and some of them will be part of the client code. By focusing on the latter set, we will be able to tell which events in the client code led to the creation of the critical chain.

The main problem with this approach is that traditional slices tend to contain too much unnecessary information. This is problematic for two reasons; the first and most obvious is performance; the bigger the slice, the longer it takes to process. But most importantly, as Sridharan *et al.* [14] have observed, slices contain a lot of information that is not really useful for program understanding. For example, if we care about an element that came out of a data-structure, the slice will contain the event that inserted the element into the data-structure, but it will also contain many other events that modified the data-structure, including many that added and removed other elements.

The solution that Sridharan *et al.* offer to this problem is *thin slicing*. The basic idea behind thin slicing is to ignore value flows to base pointers of heap accesses; for example, if you have a field write $b.f \leftarrow a$, thin slicing only follows value flows to a, rather than also following value flows to b as a traditional slicing algorithm would. Also, thin slicing doesn't follow *control dependence*. In some cases, this may lose important information, so in the original application of thin slicing, the programmer was given the ability to explore some of these value flows to base pointers or follow some of the control dependence to get a better understanding of the program behavior (called *expansions* of thin slicing). In our case, we cannot rely on the programmer to decide when it might be useful to follow value flows to base pointers, so instead we

apply a set of simple heuristics to do *automatic expansion* based on the observation that we care primarily about client code and not so much about what happens in framework code:

- When we slice on value a and see $b.f \leftarrow a$ or $a \leftarrow b.f$, we also follow the base pointer b only if: (a) the statement is within a framework method, but the value is of some client class; (b) the statement is within a client method, and the producer statement and the consumer statement of the value are not in the same method.
- When we consider each statement, we examine the call stack and for each dynamically dispatched call $f(x, ...)$ where x is the *receiver object*, if the caller is a framework method and the callee is a client method (then x must be of a client class), then we should also follow the receiver object x.

Note that when applying these heuristics, to determine whether an object is of framework class or client class, the *dynamic* type information of the object at runtime is used, rather than the static type definition.

Figure 5-1 shows an example code piece. In this example, we assume that U and UX are client classes while F, M, N, FV, and FX are framework classes. The starting point of the slicing is "F.bar(x); //e16", which is supposed to create a critical link in a critical chain. We only show the relevant code pieces to illustrate the slicing algorithm and omit the full detail of the critical chain.

Figure 5-2 shows the instructions in the thin slice for the parameter x of "F.bar(x); //e16". Note the two places marked as "// ** do crazy things to m" in the source code: in each case, m is the base pointer for $m.n$ in the slice, but because n is of framework class, the complex instructions to get m were excluded from the slice thanks to our slicing heuristic.

```
class F {
  class N {
    FV v;
    N(FV u) {
      this.v = u; // e4: n.v ← u
    }
  }
  class M {
    N n;
  }
  M m;

  void main() {
    a(); // e1: call F.a(f)
    b(); // e9: call F.b(f)
  }
  void a(...) {
    FV u = ...    // e2: ...
    N n = new N(u);    // e3: call N.init(n, u)
    M m = ...    // ** do crazy things to m
    m.n = n;    // e5: m.n ← n
    u.noo();    // e6: call U.noo(u)
  }
  void b(...) {
    M m = ...    // ** do crazy things to m
    N n = m.n;    // e10: n ← m.n
    FV u = n.v;    // e11: u ← n.v
    u.foo();    // e12: call U.foo(u)
  }
  static void bar(FX x) { ... }
}
class U extends FV {
  private UX f;
  @Override void noo() {
    UX x = new UX(); // e7: call UX.init(x)
    this.f = x;          // e8: u.f ← x
  }
  @Override void foo() {
    UX x = moo(); // e13: call U.moo(u)
    F.bar(x);        // e16: call F.bar(x), this is the starting point for slicing.
  }
  private UX moo() {
    UX x = this.f; // e14: x ← u.f
    return x;       // e15: return x
  }
}
```

Figure 5-1: Example code.

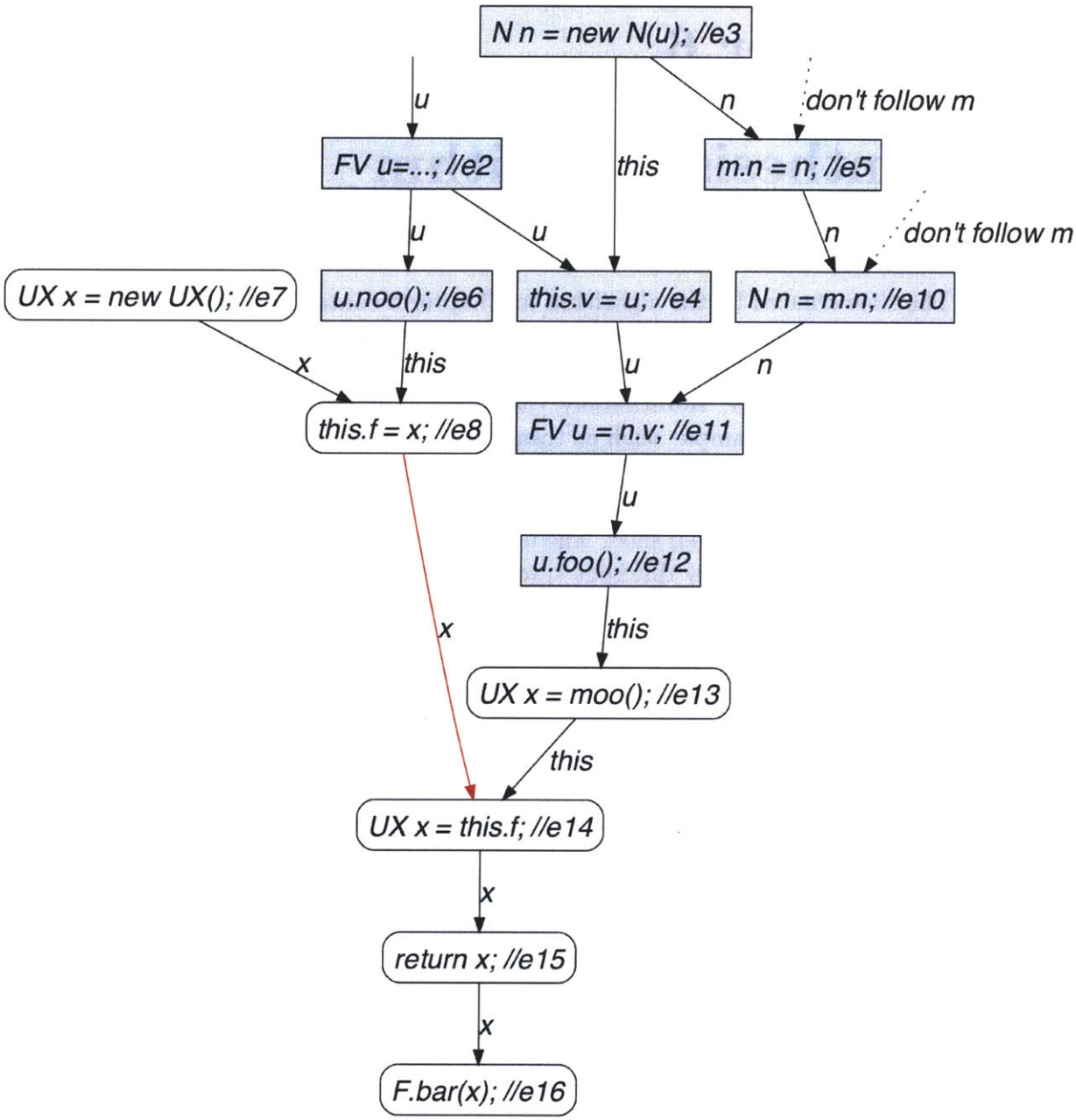

Figure 5-2: Thin slice on x in the statement F.bar(x) in Figure 5-1. Grey boxes show framework code, and white boxes show client code.

Chapter 6

Synthesis Algorithm

The slice created by the algorithm in Chapter 5 contains all the statements that were necessary to create the critical chain in a particular execution trace. However, a lot of the code in the slice is code that belongs to the framework, and is therefore of no interest to the user. Additionally, the slice contains many details that are too specific to a particular example, such as transitive copies of objects through internal fields, or calls to functions internal to the client code. MATCHMAKER addresses this problem by computing a *normalized slice* that eliminates framework code as well as superfluous details from the original slice. The normalized slice will also make it easier to compare the results from many different traces; after normalization, many slices will become identical, and those that remain different will correspond to different ways of using the framework.

We use the example code shown in Figure 5-1 to illustrate the process. Note that the instructions in the slice before normalization and the dependency relations among them are shown in Figure 5-2.

6.1 Calls between client and framework

The essence of the interaction between client and framework code is captured by calls that cross the framework-client boundary. Calls from the framework to the client are called FC-calls, and calls from the client to the framework are called CF-calls. In the example, e6 and e12 are FC-calls, while e16 is a CF-call; the rest of the calls, like e13 are termed L-calls, because they stay local to either the framework or the client (see Figure 6-1).

FC-calls and CF-calls describe the interaction between framework and client: FC-calls tell us which classes to extend and which method to override, while CF-calls tell us which APIs to call. The normalizing slice will preserve information about FC-calls and CF-calls as well as relevant client code, but will simplify away the complexity and arbitrariness of the specific sample code from which the program behavior database was built. For our example, the normalizing slice will indicate that noo() must write to f and foo() will pass the value of f to bar(), but the fact that this involves a call to moo() is only a detail of this particular example and not relevant in the normalizing

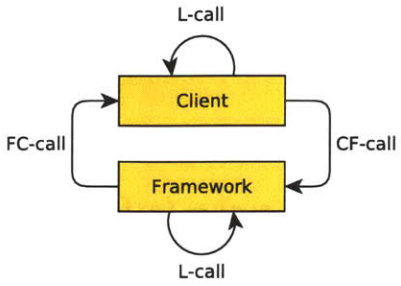

Figure 6-1: Client and framework interaction

slice.

6.2 Normalization algorithm

We first define some simple notation to allow us to describe the structure of calls. Consider a call event c (call m) and its enclosing call event c' (call m'), which we denote as caller(c). Then we say FC(c) (c is an FC-call) iff m' is in framework code and m is in client code, and we write CF(c) (c is a CF-call) iff m' is in client code and m is in the framework; otherwise c is an L-call and we write $c' \gg c$. When an event e happens in the dynamic scope of a call event c and all the calls on the call stack from c to e (except for c and e themselves) are all L-calls, we say that c *covers* e, or formally $c \gg^* e$:

$$c \gg^* e \iff c = \mathsf{caller}(e) \vee (\exists c' \cdot c \gg c' \wedge c' \gg^* e)$$

For example, in the sample program e6 $\gg^*$ e7 and e12 $\gg^*$ e15.

With this notation at hand, we are ready to define the notion of *normalized source* defined on a triple (c, e, o), were c is an FC-call, e is an event in client code that is covered by that FC-call and o is an object used by event e. The normalized source is the first event e' that is also covered by c and that produced object o. For example, in the context of the call e12, the normalized source for object x in event e16 would be event e14.

We will define a function $ns(c, e, o) = (e', E', T')$ that produces the normalized source e' in addition to a set of events E' and call-tree edges T' to be added to the normalized slice. The function is defined in terms of the dependence relation D from the original slice we are trying to normalize, where D is a set of triples of the form $e' \xrightarrow{o} e$, where event e' produced object o consumed by event e.[1]

1. If there is no $e' \xrightarrow{o} e \in D$, then

$$ns(c, e, o) = (e, \{e\}, \{(c, e)\})$$

[1]We use the notation from dependence analysis of drawing arrows from producer to consumer, rather than the less intuitive notation, sometimes used in the slicing literature, of drawing arrows backwards.

2. Otherwise there's some $e' \xrightarrow{o} e \in D$, then
   (a) If $c \gg^* e'$ then $ns(c, e, o) = ns(c, e', o)$.
   (b) Othwerise:
      i. if some other FC-call $c' \gg^* e'$ then

$$ns(c, e, o) = (e, \{e, e'\}, \{(c, e), (c', e')\})$$

      ii. Otherwise e' is in framework code, so it doesn't need to be returned: $ns(c, e, o) = (e, \{e\}, \{(c, e)\})$.

Case 1 will arise when event e is the origin of an object, in which case e is itself the normalized source. Case 2b is the most interesting; in both instances of 2b, the source of the object lies outside the function, so the normalized source is e itself. However, in 2bi, the origin is also in client code, but in an event that was covered by a different FC-call. This means that the source event together with the other FC-call will also have to be added to the slice, just like the case of e10 and e5 in the example code.

Given those definitions, the algorithm for the normalized slice itself is fairly simple. For a slice $s = (E, D, T)$, we generate its normalized slice $\hat{s} = (\hat{E}, \hat{D}, \hat{T})$ by the following process:

1. For each FC-call $c \in E$, for each CF-call event $e \in E$ covered by c, (i.e. $\text{FC}(c) \wedge \text{CF}(e) \wedge c \gg^* e$), put e in $\hat{E}$ and (c, e) in $\hat{T}$ (*inline* e into c). When c has a corresponding return event r, also put r in $\hat{E}$ and (c, r) in $\hat{T}$.
2. For each $e \xrightarrow{o} e' \in D$ where e' is in framework code and e is in client code, add e to $\hat{E}$ and (c, e') to $\hat{T}$, where $c \gg^* e'$.
3. For each event $e \in \hat{E}$ and $(c, e) \in \hat{T}$, for each dependency $\cdot \xrightarrow{o} e \in D$, let $(e', E', T') = ns(c, e, o)$, then put E' into $\hat{E}$, $e' \xrightarrow{o} e$ into $\hat{D}$ (when $e' \neq e$), as well as T' into $\hat{T}$.

6.3 Example

We show the whole process of normalizing the slice for the example code in Figure 5-1:

1. In the thin slice shown in Figure 5-2, there is only one CF-call "F.bar(x); //e16", which is covered by the FC-call "u.foo(); //e12". So as a starting point e16 is put into $\hat{E}$ and $(e12, e16)$ is put into $\hat{T}$, as shown in Figure 6-2(1) (solid arrow labeled with x or u indicates relation in $\hat{D}$; dashed arrows labeled with T indicates relation in $\hat{T}$).

2. $ns(e12, e16, x)$ is computed as follows:

$$\begin{aligned}
& ns(e12, e16, x) \\
= \; & ns(e12, e15, x) && \text{// Rule (2a) applied} \\
= \; & ns(e12, e14, x) && \text{// Rule (2a) applied} \\
= \; & (e14, \{e14, e8\}, \{(e12, e14), (e6, e8)\}) && \text{// Rule (2bi) applied}
\end{aligned}$$

So e14 and e8 are put into $\hat{E}$, $e14 \xrightarrow{x} e16$ is put into $\hat{D}$, and $(e12, e14)$ and $(e6, e8)$ are put into $\hat{T}$, as shown in Figure 6-2(2).

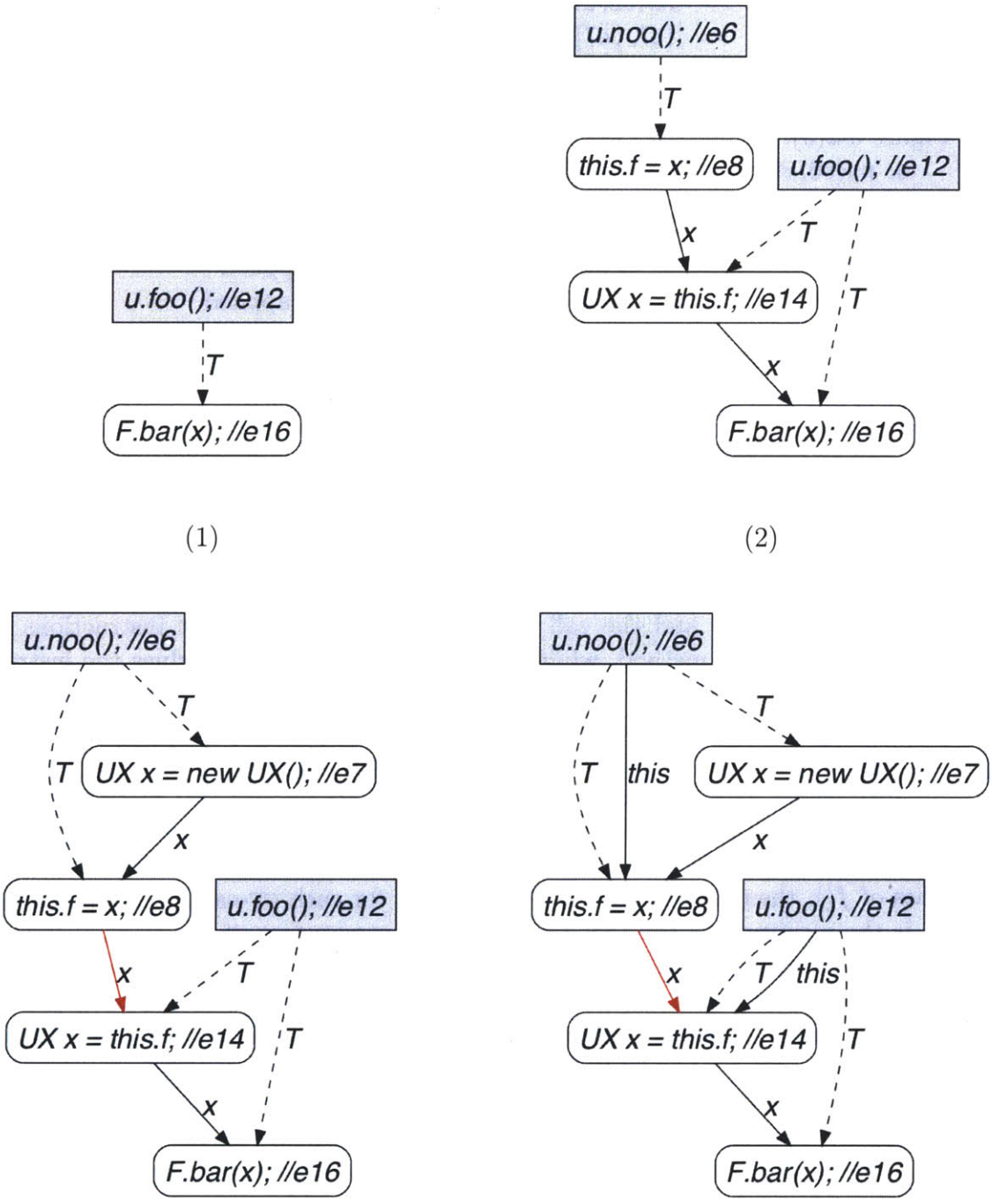

Figure 6-2: Normalized Slicing Steps for the example code in Figure 5-1

3. $ns(e6, e8, x)$ is computed as follows:

$$\begin{array}{lll} & ns(e6, e8, x) & \\ = & ns(e6, e7, x) & // \text{ Rule (2a) applied} \\ = & (e7, \{e7\}, \{(e6, e7)\}) & // \text{ Rule (1) applied} \end{array}$$

So e7 is put into $\hat{E}$, $e7 \xrightarrow{x} e8$ is put into $\hat{D}$, and $(e6, e7)$ is put into $\hat{T}$, as shown in Figure 6-2(3).

4. $ns(e6, e8, u)$ is computed as follows:

$$\begin{array}{lll} & ns(e6, e8, u) & \\ = & ns(e6, e6, u) & // \text{ Rule (2a) applied} \\ = & (e6, \{e6\}, \emptyset) & // \text{ Rule (2bii) applied} \end{array}$$

So e6 is put into $\hat{E}$ and $e6 \xrightarrow{u} e8$ is put into $\hat{D}$. Similarly, $ns(e12, e14, u) = (e12, \{e12\}, \emptyset)$, so e12 is put into $\hat{E}$ and $e12 \xrightarrow{u} e14$ is put into $\hat{D}$. The final normalized slice is shown in Figure 6-2(4).

6.4 Synthesis algorithm

It's easy to synthesize source code from normalized slice: the FC-calls tell us which method to override and the call tree relation $\hat{T}$ tells us what instructions to put in these overridden method, while the dependency relation $\hat{D}$ glues the instructions together by data dependency and gives a partial order on the instructions. In this sense, normalized slice is as expressive as source code, and it is normalized to elide low-level details such as reordering of independent instructions.

Just before synthesizing the code, MATCHMAKER has two more things to do. First, MATCHMAKER finds for each client class A the most generic framework class to extend from. For each FC-call method $A.f()$, let A_f indicate the original declaring class of $f()$, then A must be a subclass of A_f; for each CF-call $g(x_0, x_1, ..., x_k)$ where x_0 is the receiver and the rest x_i's are actual parameters, if the original type of the j-th formal parameter (including 0-th, the receiver) is A_j and $x_j = a$ where a is an instance of A, then A must be a subclass of A_j. MATCHMAKER extracts from FC-calls and CF-calls all these subclass constraints, and computes the join of these constraint to get a lower bound of class A, which is the most generic framework class for A to extend from. In fact a similar process is employed to determine for each used framework class B, the most generic class to use in place of B.

Second, MATCHMAKER gives each client class A a pretty name: if A is determined to extend from A', then A is called MyA'. MATCHMAKER names each variable of type X as x where x is obtained by making the first letter X lower case (we add numbers to avoid conflict names). MATCHMAKER also renames each field $X.somef$ of type Y as $X.fY$ (again add numbers to avoid conflict field names). Some variables might be unresolved to any value or instruction due to incompleteness of either the database or the slicing algorithm, and they are named ??, the *holes* of the synthesis result.

```
class MyFV extends FV { // the client class U in the example is renamed to MyFV
  private UX fUX; // the field f in the example is renamed to fUX
  @Override void noo() {
    UX ux = new UX();
    this.fUX = ux;
  }
  @Override void foo() {
    UX ux = this.fUX;
    F.bar(ux);
  }
}
```

Figure 6-3: Synthesized client code for the example in Figure 5-1.

Then, for each interesting event in each FC-call, MatchMaker generates the corresponding statement, and it collects the statements of FC-calls together and put them to the owner client classes, to form the final source code.

There can be multiple execution traces in the program behavior database that contain critical chains between the two queried endpoint classes, and MatchMaker produces multiple program slices from all the chains. Thanks to the normalization step, many seemingly different program slices become one identical normalized slice, so MatchMaker just output several representative synthesis results (which correspond to several different representative ways of connecting two classes together) to the end-user.

Figure 6-3 shows the synthesized client code for the example in Figure 5-1. Note that the client class U is renamed to MyFV because the most generic class for client classes U is computed to be FV; also the field U.f is renamed to MyFV.fUX.

Chapter 7

Related work

7.1 Learning framework uses from examples

The idea of extracting useful information from existing examples to help programmers learn and use large frameworks has been studied in various related works.

FUDA [5] is closely related in its goal of producing program templates from example traces. It is developed to automate the "*Monkey See/Monkey Do* rule", i.e. use existing framework applications as a guide to develop new applications'. Like our system, FUDA tries to turn the programmer's high level idea of the concept that needs to be implemented into low level template code (similar to the concept of glue code in the thesis).

Unlike our system, FUDA is not fully automated: it is based on dynamic analysis, and requires the user to find an existing example that implements the same concept she wants, and furthermore, manually select a set of interesting events (instructions) during the example execution as the starting point of information extraction, because there is a gap between the user's high level concept (implementing some feature in the framework) and the elements that can be handled by FUDA (instructions). In comparison, MATCHMAKER can automatically pick the interesting events because the MatchMaker hypothesis gives a way to assign semantic meanings (critical chain on the heap) to the high level concept (matching two classes) thus bridging the gap.

Starting from the interesting events picked by the user, FUDA uses a slicing algorithm to find other relevant instructions that are related to implementing the concept. The slicing algorithm used by FUDA is much more imprecise than ours: it uses shared objects in argument list of calls to detect data dependencies among instructions, because of the lack of heap updates history. This approach produces results that contain more unnecessary statements than ours, and may miss data dependencies that are not directly achieved by sharing objects in argument lists.

Like our system, FUDA also leverages the distinction between client and framework code to normalize slices, though they did not present a clearly defined rule to perform the normalization.

Framework usage comprehension tools such as Strathcona [6] and FrUiT [9] use static analysis to extract the relevant source code of sample applications and allow

retrieving code snippets or usage rules for a particular API element. These tools can be very helpful to understand API usage but require the user to at least know some of the APIs. They are less helpful if the developer has only a high level idea of the feature that needs to be implemented or if the implementation needs to deal with multiple classes.

7.2 Using program data to help program understanding

The idea of using large corpus of data for program understanding has seen many incarnations in the past few years.

Prospector [8], XSnippet [12], MAPO [16], PARSEWeb [15], and Strathcona [6] mine source code repositories to assist programmers in common tasks: finding call sequences to derive an object of one type from an object of another type, constructing complex initialization patterns, and digging out frequent API usage patterns. They do so by computing relevant code snippets as determined by the static program context and then applying heuristics to rank them. Since they primarily utilize static analysis, the context lacks heap connectivity information. These tools are geared towards code assistance and do not produce full templates of the program that may span multiple classes.

7.3 Concept location tools

Concept location concentrates on understanding how a certain concept or feature is implemented in the source code of an application. Whyline [7] combines source code analysis with dynamic analysis to assist debugging. It does so by tracing program execution and input/output events, and suggesting questions that relate external program observations to method calls. On the contrary, MATCHMAKER does not use external observations (GUI or input events) to locate points of interest in the trace, instead it uses internal heap configuration to identify important events in the trace. DELIGHT could potentially serve as a common framework for tools like Whyline and MATCHMAKER that need to query program executions.

7.4 Dynamic analysis to help program understanding and debugging

PQL [10] is a query language for analyzing program execution statically or dynamically. It is aimed at finding design defects in the code and as such requires full knowledge of the code details. It is not suitable for program understanding tasks. PTQL [4] uses its own relational query language to instrument a program and dynamically query live executions.

Tools that rely on light-weight dynamic analysis require manual effort in formulating queries in a specialized language or collecting traces specific to these queries. MATCHMAKER attempts to reuse existing databases for answering many queries, while keeping the query language very simple.

BugNet [11] and similar tools record the full program trace for deterministic replay debugging. MATCHMAKER does not collect entire execution data. It uses just enough information from the trace to be able to answer program synthesis-related queries effectively.

7.5 Other Program Synthesis

Program synthesis systems such as SKETCH [13] can produce program text from a slower version of the same program or its specification via a combinatorial search over ASTs. The level of deep static reasoning about program that is needed by SKETCH has not been achieved for the large scale software like Eclipse. Moreover, the dynamic features that are prevalent in Eclipse and its scale make it very hard to employ any static reasoning except for the very light-weight.

Chapter 8

Generality Evaluation

We want to evaluate the generality of MatchMaker hypothesis. During the process of developing MATCHMAKER we have used four motivating examples to test our tool: Editor to Scanner, Editor to ICompletionProposal (the auto-completion choice in Eclipse), Menu to Action, and Toolbar to Action. Usually the generated code is very similar to the code written by human expert, but it may have two kinds of imperfections:

1. Some method call parameters may not resolve to a known name, so they are left as "??" (holes). This can be because of two reasons: First, any primitive typed value is not collected in the program behavior database, so any dependence to a primitive value results in a hole; Second, there is incompleteness in the program behavior database and our slicing algorithm, which may cause some dependence to be unresolved, leaving a hole in the resulting synthesized code. The user will need to look up the documentation and tutorials to find out the exact value of these holes.
2. Some statements cannot be captured by the critical chain chosen by MATCHMAKER, because there can be multiple critical chains (see Figure 2-2) and currently MATCHMAKER only arbitrarily pick one. In the Editor-Scanner example, the critical chain chosen by MATCHMAKER goes through the link from reconciler to repairer, but not through the link from reconciler to damager, so MATCHMAKER was unable to find the call `reconciler.setDamager()`. The user needs to manually add the missing statements. Note that this is not an inherent limitation: we envision that once we improve MATCHMAKER to look at all critical chains, this problem should be fixed.

We think that the additional job is relatively easy for the user because she now has the knowledge of all the necessary classes and most API calls, and by using them as keywords to search the tutorials and documents, she can soon learn about the missing calls and fill in the holes. In Figure 8-1 we show for each class pair the extra changes the user needs to make.

In addition to the above mentioned four class pairs we also conduct an experiment as follows: We pick the eclipse.jdt.internal.ui plugin, treat everything in eclipse.jdt.* as client code, and extract all framework classes that are extended and used inside this plugin. There are more than two hundred of these classes. We examined these classes manually, and pick pairs of classes that seemed related with each other based on the names. Many of these classes serve for completely different features, and we

Source type	Target type	# holes	# missing statements
Editor	Scanner	1	1
Editor	ICompletionProposal	1	0
Menu	Action	1	0
Toolbar	Action	1	0
ITextEditor	IContentOutlinePage	0	1
MonoReconciler	IReconcilingStrategy	1	0
ITextEditor	QuickAssistAssistant	0	1
QuickAssistAssistant	IQuickAssistProcessor	0	0
ITextEditor	ITextHover	0	0
ISpellCheckEngine	ISpellChecker	1	1
ITextEditor	SelectionHistory	0	0
ITextEditor	SemanticHighlighting	1	1
IContentAssistProcessor	ContentAssistInvocationContext	1	0
ITextEditor	IAutoEditStrategy	0	0
ITextEditor	ContextBasedFormattingStrategy	1	2
ITextEditor	TextFileDocumentProvider	0	0

Figure 8-1: Generality evaluation table. First column gives the names of class pairs. "# holes" and "# missing" counts the number of holes and missing statements in the generated code.

were able to pick 16 pairs of classes that we thought were related to each other. Then we run MatchMaker on each of these pairs, and examine the generated code to see whether it's reasonable glue code to make the interaction between the pair of classes, and measure how far it is from the functionally correct version. For 12 out of these 16 pairs MatchMaker were able to generate reasonable code; for the other 4 pairs MatchMaker didn't generate code, but we have yet to find whether it's because of incompleteness of our program behavior database or because of failure of MatchMaker hypothesis.

From Figure 8-1 we can see the pairs for which MatchMaker did generate solutions, they are quite close to the correct answers. This suggests that the MatchMaker approach can be generally applied to a wider range and not restricted to the Editor-Scanner example we have shown.

Chapter 9

User Study

We have conducted a user study to measure the effect of MATCHMAKER on programmer productivity. This chapter gives the result of the user study. Detailed analysis of the behavior of the participants has provided us with some important insights about the effects of MATCHMAKER on programmer behavior and about the relative effort required to write glue code compared to the effort required to actually program a task. In the following sections we describe the basic setup of the user study, and provide a detailed analysis of the behavior of our subjects.

Task Description For our user study, subjects are asked to implement an editor with *Syntax Highlighting* for a new language—the SKETCH language developed by our group. Specificaly, subjects are asked to highlight two keywords in the language: **implements** and the operator ??. They are given an incomplete RuleBasedScanner that can already highlight the keyword **implements**, but they have to write *task code* to complete the scanner and connect it to the editor by writing *glue code* like the one in Figure 2-1.

Note that we have chosen this task to reveal different aspects of using MATCHMAKER, rather than to give a best show case. In fact MATCHMAKER may show much greater productivity improvement on other tasks than this one, because of two reasons:

First, there are many useful tutorials and examples on the web describing how to write editors with syntax highlighting feature using Eclipse. Control subjects relied heavily on the tutorials to finish the task. We could have chosen another task like matching editor and IContentOutlinePage, for which there are no good tutorials on the web. We believe that in that case most of the control subjects would not even finish the task, but MATCHMAKER users would be less affected because MATCHMAKER can still generate very good glue code.

Second, the code synthesized by MATCHMAKER for this task is imperfect: it contains one hole (in the statement reconciler.setRepairer(dr, ??)) and misses one statement (reconciler.setDamager(dr, DEFAULT_CONTENT_TYPE)), so the MATCHMAKER user needs to look up the documentation and tutorials on the web to fill the hole and add the missing statement. We observed that adding the missing statement is the most time-consuming work item for MATCHMAKER users. We could have chosen another

task like matching Toolbar and Action, for which MATCHMAKER synthesizes nearly perfect code, and the MATCHMAKER users just need to copy and paste the generated code and do simple editing. In that case, the MATCHMAKER users might perform even better than the framework expert.

Methodology We have been recruiting participants through mass advertising around campus with the promise of two free movie tickets for any participant that attends the study. When participants arrive, they are randomly assigned to one of two groups: those in the *control* group are simply given a description of the task and are told they can use any information available on the internet to help them complete the study; those in the *experimental* group are also given 15 minutes to review a tutorial on MATCHMAKER. The subjects in the experimental group are advised to consult both MatchMaker and the tutorials and documentation on the web because the result given by MatchMaker may contain holes or miss important statements. Subjects in the control group do not know they are the control group, or even of the existence of MATCHMAKER; they are led to believe that the purpose of the study is simply to analyze programmer's use of frameworks.

To help the subjects finish the task in reasonable time, we provide the two endpoint classes (TextEditor and RuleBasedScanner) to use. Moreover, we give partial implementation of RuleBasedScanner so that it can scan one of the keywords and color it correctly. The user just need to modify the partial implementation to scan the other keyword, which is much easier than implementing a RuleBasedScanner from scratch, but still not trivial because the user must modify two internal classes WordRule and WordDetector used by the RuleBasedScanner. Providing the partial implementation significantly reduced the time subjects spent writing task code (as can be seen in Section 9.1, the time spent on task code is much less than on glue code for almost every subject), but should not have much effect on the time spent on glue code. We know that in a real life development scenario, the user may need to figure out the two endpoint classes to fit her own needs by herself, and implement the customized endpoint classes completely by herself, but we guess that the way she connects the two endpoint classes should not differ too much from our user study.

The programmers work environment is a virtual machine created with VirtualBox that has been set up to do screen-captures at 1 frame per second; this, together with Eclipse's local history and the subject's browsing history give us a very complete picture of the programmer's actions during the user study.

Subjects Of the seven subjects who have responded our ads and completed the study, four were randomly assigned to the experimental group (we'll call them Alice, Chuck, David, and Earl), and three subjects (we'll call them Bob, Frank, and Gary) were assigned to the control group. Another two subjects responded to our ads and did part of the study, but were distracted by phone calls and had to quit the study before finishing, so we exclude them from the analysis: one is a control subject who spent around 40 minutes and still did not finish the glue code; the other one is a MATCHMAKER user who have established partial connection (see Section 9.2)

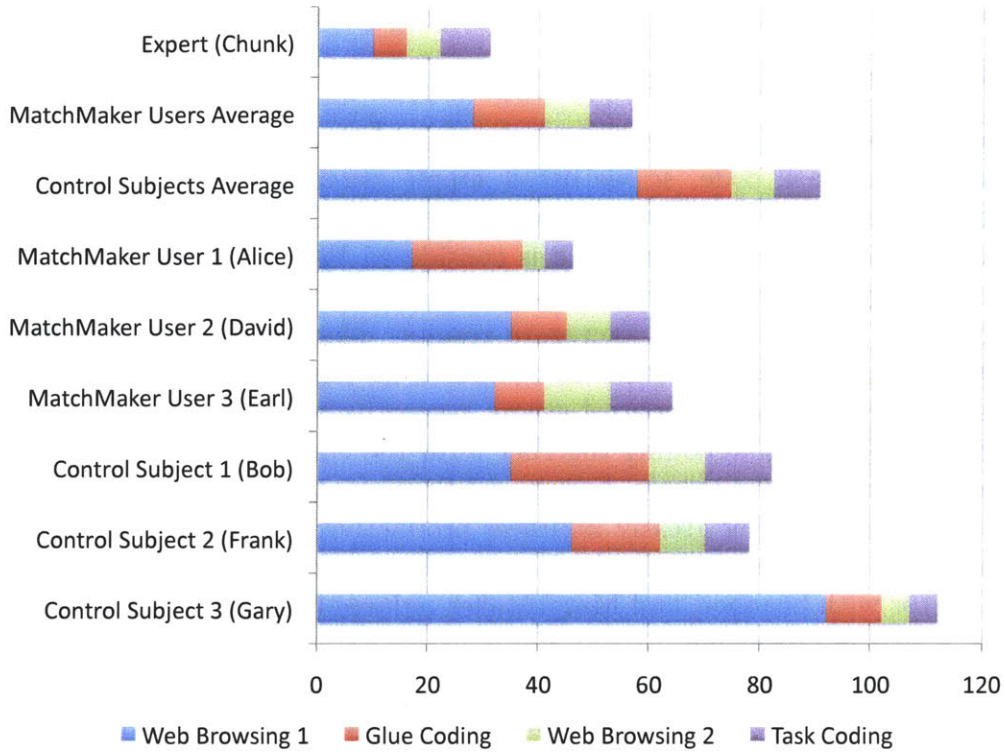

Figure 9-1: Time spent on the task, split to individual work items.

between editor and scanner within 20 minutes and then quit. So we guess that these two subjects who have quit wouldn't differ too much from their groups, and excluding them should not introduce much bias.

Chuck is a special case because he turned out to be highly experienced in writing Eclipse plugins; in fact, it is safe to say that he knows significantly more about the subject than we do. Thus, even though he was assigned to the experimental group and given the MATCHMAKER tutorial, he never used it, as he was extremely comfortable writing eclipse code without the aid of the tool.

David, on the other hand, had very limited Java experience, and had never written any code using big frameworks. The remaining five subjects have similar Java experiences and overall, all of them were very competent programmers.

9.1 Aggregate Results

Figure 9-1 shows for each subject time consumption on the task; the total time consumption is split to individual work items:

Glue Coding The time spent writing and debugging *glue code* to match the Editor class with the Scanner class.

Web Browsing 1 The time spent browsing the web (including searching and reading the documentation, tutorials, etc.) to find out how to write glue code.

Task Coding The time spent writing *task code* to complete the Scanner.

Web Browsing 2 The time for browsing the web to find out how to write task code.

It is hard to completely separate the web pages that are related to glue code from those related to task code, so we use a simple criterion here: we can easily tell whether the programmer was writing glue code or task code, so any web browsing immediately before writing glue code is considered Web Browsing 1, and any web browsing immediately before writing task code is considered Web Browsing 2.

Figure 9-1 also shows two aggregate results: MatchMaker Users Average and Control Subjects Average, which were computed by taking arithmetic average of the experimental group and the control group, respectively. The time consumptions on web browsing 1 and glue coding together form the time consumption on matching problem (connecting two classes, Editor and Scanner, together). From the figure we can see that MatchMaker greatly improves productivity by reducing the time spent on glue code: For users new to the framework, it takes 75 minutes on average to match Editor with Scanner, and MatchMaker reduces this number to 41 minutes on average, a 45% improvement.

We can also see that the difference of time consumptions on task code (web browsing 1 plus task coding) between experimental and control group is insignificant: both groups spent around 16 minutes on average to finish the task code, only slightly slower than the expert who spent 15 minutes. This suggests that:

1. MatchMaker can help with writing glue code, but not task code. Because task code only involves one particular component, not relations between components.

2. The expert knows more than novice users about the overall structure of the framework and the way components interact with each other, but not much on the details of how to use one specific component.

9.2 Observations from Representative Subjects

We pick a representative MATCHMAKER user Earl to observe in detail the whole process of using MATCHMAKER to solve the task. The process is naturally split into four phases:

Phase 1: Earl started by entering a query "TextEditor – RuleBasedScanner" into MATCHMAKER, and got the synthesized code after 1 minute. Then he just copied and pasted the code to his project, and organized them to several class files. Now he was facing a hole in the statement "reconciler.setRepairer(dr, ??)", and he didn't know that there is a missing statement "reconciler.setDamager" at this point.

Phase 2: Earl then spent around five minutes browsing the web to find out the value for the hole, and then he was able to connect the editor with the scanner and made the program run. Because "setDamager" was missing, this is only a *partial connection*: with only repairer but no damager set, the editor implements a static form of syntax highlighting; that is, it will highlight the keywords when it loads the document, but it will not change their color when you edit the document.

Phase 3: Earl quickly realized that his editor can only do *static syntax highlighting* when he was testing his program, and he spent a little more than 25 minutes browsing the web to find out why. After he found that this was because the missing setDamager, he fixed it immediately and got the glue code correct.

Phase 4: Earl spent another 25 minutes to get the task code, including browsing the web and writing and debugging the code.

This is a very typical usage of MATCHMAKER. The other users, Alice and David, both showed similar phases, with only slight differences: First, Alice and David both did Phase 4 (task code) immediately after Phase 2 (partial connection). After we pointed to them that the highlighting did not refresh with editing, they realized the problem of partial connection, and then they did Phase 3. Second, Alice and David both spent less than 15 minutes on Phase 4 (task code), much better than Earl. Actually Earl was the slowest to finish task code among all 7 subjects, but thanks to MATCHMAKER, he finished glue code very fast, so his overall performance still beats every control subject.

The control subjects, on the other hand, struggled with glue coding. For example, Bob browsed the web for 22 minutes before he started to write the first line of glue code, and spent a total of one hour to get the glue code correct. The other two control subjects Frank and Gary both spent more than an hour writing glue code.

It is worth pointing out that there are actually a number of tutorials on the web that describe how to implement syntax highlighting, several of which were visited by all three control subjects. However, these tutorials are either poorly written or contain too much or too little information, so it took them a significant amount of time to extract the relevant facts from these tutorials.

The framework expert Chuck, on the other hand, knew exactly what he was looking for on the web and found it very quickly, so he could finish glue code in 16 minutes.

9.3 Conclusions

Overall, comparison of the aggregate results and observations from individual subjects allow us to venture some preliminary conclusions.

- The matching problem is indeed a significant problem when writing code on top of complex frameworks. This is particularly true for people who are new to the framework, and less so for experts.

- Tutorials and documentation available online are not enough to close the gap between novices and experts. The class-by-class documentation available through JavaDoc is particularly unhelpful because it fails to describe multi-object interactions. Tutorials, in turns, can be unreliable because of errors and omissions, but the most important problem is the sheer amount of data that a novice has to read before beginning to understand the framework.

- MATCHMAKER has a significant impact on programmer productivity by showing programmers the object interactions that are necessary to achieve a task.

This is true even when the tool fails to give a complete solution to the task in question. It is worth pointing out that the missing setDamager() could actually have been found by MATCHMAKER simply by tuning the heuristics it uses to determine which critical chains to choose. However, not tuning the heuristics this way allowed us to observe how programmers cope with incomplete results.

Chapter 10

Conclusions and Future Work

This thesis presented a new approach to synthesis based on the analysis of very large amounts of program execution data. The approach is made possible by the DeLight system, which allows for the efficient collection, management and analysis of this data. DeLight uses abstraction to support detailed queries about how the heap evolves as the program executes, which are necessary to support our synthesis algorithm.

Our synthesis algorithm focuses on the problem of generating the glue code necessary for two classes to interact with each other. This glue code often involves instantiating new classes, making API calls, and even overriding methods in specific classes, and our tool MatchMaker can support all of these actions.

Our empirical evaluation shows that writing this glue code is especially time consuming for novice programmers, and that MatchMaker can significantly improve their productivity. It also shows that MatchMaker is general enough to handle many interesting queries, and produces code that can be used by programmers with very few changes.

As our future work, an interesting research problem is to generalize the MatchMaker Hypothesis so that it can capture other forms of the programmer's high level idea of using the framework. Another important research topic is how to combine the knowledge learned from the program behavior database to invent new knowledge not in the existing examples and help programmers devise completely new ways of using the frameworks.

Bibliography

[1] http://eclipse.org.

[2] http://en.wikipedia.org/wiki/software_framework.

[3] http://people.csail.mit.edu/kuat/index.html.

[4] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. *SIGPLAN Not.*, 40:385–402, October 2005.

[5] Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 344–368, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[7] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[8] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *SIGPLAN Not.*, 40:48–61, June 2005.

[9] Mira Mezini Marcel Bruch, Thorsten Schafer. Fruit: Ide support for framework understanding. 2006.

[10] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40:365–383, October 2005.

[11] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33:284–295, May 2005.

[12] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. *SIGPLAN Not.*, 41:413–430, October 2006.

[13] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40:404–415, October 2006.

[14] M. Sridharan, S.J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 112–122. ACM, 2007.

[15] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[16] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. *ECOOP 2009–Object-Oriented Programming*, pages 318–343, 2009.