Data-Driven Synthesis for Object-Oriented Frameworks

Kuat Yessenov Zhilei Xu Armando Solar-Lezama *
Massachusetts Institute of Technology
{kuat,timxu,asolar}@csail.mit.edu

Abstract

Software construction today often involves the use of large frameworks. The challenge in this type of programming is that object-oriented frameworks tend to grow exceedingly intricate; they spread functionality among numerous classes, and any use of the framework requires knowledge of many interacting components.

We present a system named MatchMaker that from a simple query synthesizes code that interacts with the framework. The query consists of names of two framework classes, and our system produces code enabling interaction between them. MatchMaker relies on a database of dynamic program traces called DeLight that uses novel abstraction-based indexing techniques to answer queries about the evolution of heap connectivity in a matter of seconds.

The paper evaluates the performance and effectiveness of MatchMaker on a number of benchmarks from the Eclipse framework. The paper also presents the results of a user study that showed a 49% average productivity improvement from the use of our tool.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Computer-aided software engineering; I.2.2 [*Automatic Programming*]: Program synthesis

General Terms Human Factors

Keywords Program Synthesis, Dynamic Instrumentation, Thin Slicing, Software Engineering

1. Introduction

Modern programming relies heavily on extensible frameworks that pack large amounts of functionality. These frameworks make it possible to write rich applications by piecing together pre-existing components, but the productivity benefits come at a price: a significant learning curve as programmers master a complex framework with thousands of components. Synthesis [14, 16, 18] can alleviate this problem by leveraging the novice programmer's limited understanding of the system to generate code that uses the framework. Specifically, this paper demonstrates the potential impact of synthesis through a new tool, MatchMaker, that addresses a concrete programming challenge: establishing an interaction between two framework classes.

The problem arises from the way object-oriented frameworks factor functionality into a multitude of components. This factoring makes the framework flexible, but it also implies that interactions that look simple at a high level require collaboration of a number of auxiliary objects. As a consequence, the user must write *glue code* whose sole purpose is to coordinate these auxiliary objects.

To illustrate the problem, consider the following running example: extending an Eclipse [3] editor with syntax highlighting. This is done by defining a subclass of RuleBasedScanner with the code to identify and color tokens in a file. However, the editor does not use the scanner directly. Instead, the interaction is mediated by five additional classes. First, the editor interacts with a component called SourceViewer (see Fig. 1), which manages its add-ons. SourceViewer in turn uses PresentationReconciler to maintain a representation of the document in the presence of changes. 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. To glue these classes together, the programmer must extend SourceViewerConfiguration and override the getPresentationReconciler method to return an instance of the PresentationReconciler class. This instance must have its IPresentationDamager and IPresentationRepairer reference the new scanner. Finally, the new SourceViewerConfiguration must be registered with the editor by calling setSourceViewerConfiguration in the constructor of the editor (see Fig. 2).

* All authors are primary authors, listed in reverse alphabetical order.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

```
class AbstractTextEditor {
  SourceViewerConfiguration fConfiguration;
  ISourceViewer fSourceViewer;
  createPartControl() {
    fSourceViewer = createSourceViewer();
    fSourceViewer.configure(fConfiguration);
  }
  setSourceViewerConfiguration(config) {
    fConfiguration = config;
  }
}

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

Figure 1. Eclipse code in AbstractTextEditor (2950 LOC) and SourceViewer (537 LOC) relevant to interaction with a scanner.

```
class UserConfiguration extends SourceViewerConfiguration {
  IPresentationReconciler getPresentationReconciler() {
    PresentationReconciler reconciler =
      new PresentationReconciler();
    RuleBasedScanner userScanner = new UserScanner();
    DefaultDamagerRepairer dr =
      new DefaultDamagerRepairer(userScanner);
    reconciler.setRepairer(dr, DEFAULT_CONTENT_TYPE);
    reconciler.setDamager(dr, DEFAULT_CONTENT_TYPE);
    return reconciler;
  }
}

class UserEditor extends AbstractTextEditor {
  UserEditor() {
    userConfiguration = new UserConfiguration();
    setSourceViewerConfiguration(userConfiguration);
  }
}

class UserScanner extends RuleBasedScanner {...}
```

Figure 2. User code required to let an editor use a scanner. MatchMaker automatically synthesizes the text in black.

MatchMaker automatically synthesizes the glue code in Fig. 2 from a simple query of the form: "How do I get AbstractTextEditor and RuleBasedScanner to interact with each other?". More generally, given two types A and B, MatchMaker identifies what the user of the framework has to write in order for two objects of these types to work together. In order for MatchMaker to do this, the first problem that has to be addressed is to give semantic meaning to the query; *i.e.* what does it mean for two objects to "interact with each other"? We assign semantic meaning by exploiting a new hypothesis about the design of object-oriented frameworks.

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 having necessarily a chain of references that connects them. Nevertheless, we have experimental evidence to suggest that the hypothesis is true for a number of pairs of classes in Eclipse (see the generality evaluation in Sec. 6.3) and that a tool based on this hypothesis can have a real impact on programmer productivity (see the user study in Sec. 6.4).

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.
- Generalize this code to remove arbitrariness specific to individual implementations.

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

1.1 Finding Critical Chains with DeLight

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 hard to achieve by static analysis given the scale of the frameworks and their aggressive use of dynamic dispatch and reflection. Rather than analyzing the program statically, we collect its execution traces and organize them off-line for efficient processing of queries about the exercised program behavior.

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*.

The first step in this process is to find events in each concrete execution trace 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,

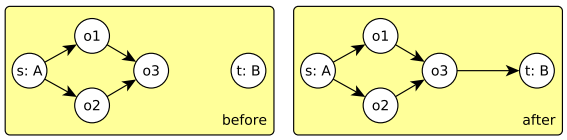

Figure 3. Event $o3.f \leftarrow t$ establishes the critical link between s and t and creates two critical chains.

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 Fig. 3.

Conceptually, we can find a critical link by performing a depth first search from the source object after every memory write operation. Unfortunately, this naïve strategy is as inefficient as it sounds; the cost of running search over the entire heap after every memory update is prohibitive for all but the shortest of traces. Instead, our data management engine DeLight uses *abstraction* to organize and index execution data in a way that makes critical chain computations efficient.

The first insight exploited by DeLight is that it is possible to represent the sequence of heaps generated by the program after each memory update with a single graph we call a *heap series graph*, where nodes correspond to objects, and every edge corresponds to a reference from one object to the other, labeled with the time interval when this reference existed. Given a path in this graph, we can find the set of time steps where this path was present by taking the intersection of the time intervals for every edge in the path.

The heap series graph provides a concise representation of the overall evolution of the heap, but it is too big to explore efficiently. This leads to the second insight exploited by DeLight: the heap series graph can be abstracted into a coarser graph where each node or edge represents a set of objects or references. The abstraction is conservative; any path in the original heap series graph has a corresponding path in the abstract graph, but the abstract graph also contains spurious paths. Thus, if a critical chain is found in the abstraction, then it needs to be checked against the complete graph to ensure that this critical chain is real. Sec. 4 describes this algorithm in full detail. Thanks to the use of abstraction, our algorithm answers the critical chain query in under five seconds for 100 GB of trace data (see Sec. 6.2).

1.2 Synthesizing Code from Critical Chains

The critical chain computation produces a set of events that creates a chain of references from the source to the target object, but these events may belong to the framework code and cannot be invoked directly by the user. What the synthesizer is looking for is the set of actions — including API calls, field updates, class instantiations, *etc.* — that the user code needs to effect in order for those events inside the framework to take place.

Our algorithm is described in Sec. 5.1 and is based on a dynamic form of thin slicing [15]. The slice is the set of events relevant to the creation of the critical chain. By separating events that took place in the user code from those that took place inside the framework, the algorithm identifies the subset of the events in the slice at the boundary between them: relevant framework events from the point of view of the user, and methods in the user code that are called by the framework.

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 projecting the slices to remove extraneous details about the structure of the different user implementations. This exposes a large degree of similarity between them, and allows us to identify distinct patterns of interaction.

We explain the formal model of the program trace data in Sec. 2 and its implementation in Sec. 3. The critical chain and synthesis algorithms are described in Sec. 4 and Sec. 5. We conclude the paper with an experimental evaluation in Sec. 6.

2. DeLight Data Model

DeLight relies on three complementary views of execution data. The first is the *call tree presentation*, which directly models the sequence of instructions executed by each thread and the nesting of method calls (Sec. 2.1). While it provides detailed information focused around any particular point in the execution, it makes it difficult to answer global queries without looking at the entire trace.

To support heap connectivity queries, DeLight provides a complementary graph-based presentation that provides a global view of the evolution of the heap. We call this presentation a *heap series* (Sec. 2.2). The two presentations are connected via time stamps that are assigned to every program instruction.

In order to make queries on the heap series graph more tractable, we introduce *heap abstractions* that capture the essential domain information and approximate the heap series to reduce its size (Sec. 2.3).

2.1 Call Tree Presentation

This 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.*) $\mathsf{type}(a)$ 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 as used, for example, in computing slices.

2.2 Heap Series Presentation

The call tree presentation is useful for analyzing call patterns, but when reasoning about the evolution of heap connectivity, a graph-based view is more desirable. A heap $\mathcal{H}$ in our model is a directed multi-graph on the set of values $\mathcal{V}$ and edges labeled by fields $\mathcal{F}$. An edge $a \xrightarrow{f} b$ denotes the fact that the value of non-static field f of an object instance a is b. We use set algebra on the set of edges to describe updates to a multi-graph.

The effect of an event on the heap is the addition and removal of edges. Starting from the empty heap $\mathcal{H}_0$, we build a sequence of heaps $\{\mathcal{H}_t\}$ by applying t-th event to $\mathcal{H}_{t-1}$. The rule for the field write event is:

$$[\![b.f \leftarrow a]\!]\mathcal{H} \;=\; \left(\mathcal{H} \setminus (b \xrightarrow{f} \mathcal{V})\right) \cup (b \xrightarrow{f} a)$$

(all f edges coming out from b are removed and an f edge from b to a is added.) We do not store static field writes in the heap model, and the remaining events have no heap side-effects.

The sequence of heaps $\{\mathcal{H}_t\}$ is compacted into a *heap series* presentation $\widehat{\mathcal{H}}$, which is a directed multi-graph on the set of values $\mathcal{V}$ and edges labelled by pairs of fields $\mathcal{F}$ and non-empty *time intervals*: $\mathcal{F} \times 2^{\mathbb{Z}}$. Here an edge records all indexes t for which $\mathcal{H}_t$ has an edge labelled by f between the nodes:

$$a \xrightarrow{(f,T)} b \in \widehat{\mathcal{H}}$$

whenever there was an edge in one of the heaps:

$$T \;=\; \{t \mid a \xrightarrow{f} b \in \mathcal{H}_t\} \wedge |T| > 0$$

The heap series model and the call tree model are connected by the times of the events. This allows us to quickly jump from a time on some edge in $\widehat{\mathcal{H}}$ to the call stack for the corresponding event and vice versa. Time intervals are represented either as decision trees or as unions of disjoint segments. The motivation for using time intervals instead of multiple heaps and the real gain in compacting heap series come from a simple observation: most fields are not updated frequently.

Abstract fields Containers are pervasive in Java code, and their simple interfaces encapsulate complex heap representations. DeLight approximates the internal behavior of container objects via *abstract fields*. For example, we model lists with the binary relation $\mathsf{content} \in \mathcal{F}$, where $a \xrightarrow{\mathsf{content}} b$ if a is an instance of java.util.ArrayList and b is an element of the list a. Similarly, the abstract field $\mathsf{values} \in \mathcal{F}$ matches a map to its values: $a \xrightarrow{\mathsf{values}} b$ if a is an instance of java.util.HashMap and b is a value for some key in the map a. Finally, the abstract field array relates an array object to its elements.

Figure 4 describes how DeLight computes heap series $\widehat{\mathcal{H}}$ by observing only the method enter and exit events for collection classes. For example, List.remove takes a position as an input and returns the removed list element at that position, allowing us to deduce the effect on the heap without knowing the list content ahead of time. Note that unlike concrete fields, abstract fields may have multiple edges between two instances (*e.g.* if a list contains duplicates) or many outgoing edges from an instance (since containers usually have many elements.) Using only method events does not let us build an absolutely precise model; DeLight does not, for example, handle element position in a list, removal via an iterator, or **null** value in a map.

2.3 Heap Abstractions

The heap series graph $\widehat{\mathcal{H}}$ is quite large: the number of nodes and edges easily reaches millions. To support efficient computation on $\widehat{\mathcal{H}}$, DeLight provides several *heap abstractions* aimed to further reduce the size of the graph using semantic domain knowledge while preserving properties of interest. We use heap abstractions to derive approximate answers to queries which we then refine by selectively querying $\widehat{\mathcal{H}}$. The critical property is the

$$[\![\text{call List.add}(l,o)]\!]\mathcal{H} \;=\; \mathcal{H} \cup (l \xrightarrow{\text{content}} o)$$
$$[\![\text{call List.add}(l,i,o)]\!]\mathcal{H} \;=\; \mathcal{H} \cup (l \xrightarrow{\text{content}} o) \text{ where } i \in \mathbb{Z}$$
$$[\![\text{call List.clear}(l)]\!]\mathcal{H} \;=\; \mathcal{H} \setminus (l \xrightarrow{\text{content}} \mathcal{V})$$
$$[\![\text{call List.remove}(l,o)]\!]\mathcal{H} \;=\; \mathcal{H} \setminus (l \xrightarrow{\text{content}} o)$$
$$[\![\text{call List.remove}(l,i); \text{return } o]\!]\mathcal{H} \;=$$
$$= \mathcal{H} \setminus (l \xrightarrow{\text{content}} o) \text{ where } i \in \mathbb{Z}$$

$$[\![\text{call Map.put}(m,k,v); \text{return } o]\!]\mathcal{H} \;=$$
$$= \left(\mathcal{H} \setminus (m \xrightarrow{\text{values}} o)\right) \cup (m \xrightarrow{\text{values}} v)$$
$$[\![\text{call Map.clear}(m)]\!]\mathcal{H} \;=\; \mathcal{H} \setminus (m \xrightarrow{\text{values}} \mathcal{V})$$
$$[\![\text{call Map.remove}(m,k); \text{return } o]\!]\mathcal{H} \;=\; \mathcal{H} \setminus (m \xrightarrow{\text{values}} o)$$

Figure 4. A selection of the heap update rules for abstract fields: here l is an instance of java.util.ArrayList; k, v, and o are arbitrary values; m is an instance of java.util.HashMap.

heap connectivity: if two values are connected by a path in $\mathcal{H}_t$ for some t, then they are connected in the heap abstraction $\mathcal{A}(\widehat{\mathcal{H}})$ of $\widehat{\mathcal{H}}$.

In the heap series $\widehat{\mathcal{H}}$, we call a simple path

$$a_0 \xrightarrow{(f_1,T_1)} \ldots \xrightarrow{(f_k,T_k)} a_k$$

viable if the edge intervals share a common time. The *lifetime* of the path is the intersection $\cap_i T_i$ of these time intervals. Viable paths have non-empty lifetimes.

Our abstraction techniques are all based on the idea of summarizing groups of objects and groups of field connections akin to type-based field-aware static analysis [2]. The research question is what makes a good group of objects and how to merge edges in $\widehat{\mathcal{H}}$. We attempted to answer this question by empirically analyzing heap series graphs while keeping the end-to-end perspective of how DeLight is used by MatchMaker. Our requirement for the abstraction is somewhat different from static analysis since our goal is to optimize queries rather than prove properties. We use terminology of *graph homomorphisms* to describe our technique.

Graph homomorphisms are natural graph transformations that preserve connectedness, and they are characterized by two functions: cluster that maps nodes of $\widehat{\mathcal{H}}$ to nodes of $\mathcal{A}(\widehat{\mathcal{H}})$ and relabel that maps edge labels of $\widehat{\mathcal{H}}$ to edge labels of $\mathcal{A}(\widehat{\mathcal{H}})$. The soundness of these abstractions rests on the following property: for any viable path a_i in $\widehat{\mathcal{H}}$ there exists a viable path b_i in $\mathcal{A}(\widehat{\mathcal{H}})$ such that both paths have the same length and $b_i = \mathsf{cluster}(a_i)$. The reverse is not always true: there are false candidate abstract paths that have no concrete counterparts. We call this process of taking a candidate viable path in the abstraction and attempting to find a concrete path *concretization*.

Since viability of paths is determined by the time intervals on the edges, relabel must respect these intervals. A simple way to ensure that is to only allow over approximation where a time interval is mapped to a larger set of times. If a group of edges is mapped to the same edge in the abstraction, then the abstract edge time interval must at the very least include the union of the concrete time intervals.

Our graph homomorphisms are composed in the order they are described. The composition respects the property as long as each individual transformation does. The quality of abstraction is the measure of how many false candidate paths are introduced. We found the following techniques effective in reducing the size of the heap abstraction, our main goal, but having high quality of abstraction.

Type-based abstraction Type-based clustering is appealing since a typical heap connectivity query is concerned with the types of the end-points rather than concrete instances. Intuitively, objects of the same type play the same role in the way they interact with the rest of the heap.

This abstraction introduces many false paths for containers, for which the type carries very little information. For example, if all maps are deemed equivalent, any object pointing to a map reaches any value of any map. Therefore, we parametrize our abstraction by *exception types*, for which instances are not merged. Collection types are the only exception types in our implementation.

To summarize, type-based abstraction is defined as:

$$\mathsf{cluster}(o) = \begin{cases} o & \text{for exception types} \\ \mathsf{type}(o) & \text{otherwise} \end{cases}$$

Field abstraction Edge labels in the heap series are pairs of a time interval and a field. Field abstraction strips the field from the pair, leaving only the time interval:

$$\mathsf{relabel}((f,T)) = T$$

The effect is that viable paths remain viable but they lack information about fields between every two consecutive clusters. The job of the concretizer is to select fields that would connect objects belonging to the clusters of the candidate path. Multiple such fields are possible if the fields are declared in the same class and have the same value type.

Time abstraction The edges in the resulting abstraction are labelled with just time intervals. Conceptually, this abstraction graph is now simple since the time intervals can be merged together with the set union operation. Time abstraction amounts to expanding the merged time intervals to larger sets: $\mathsf{relabel}(T) = T'$, where $T \subseteq T'$.

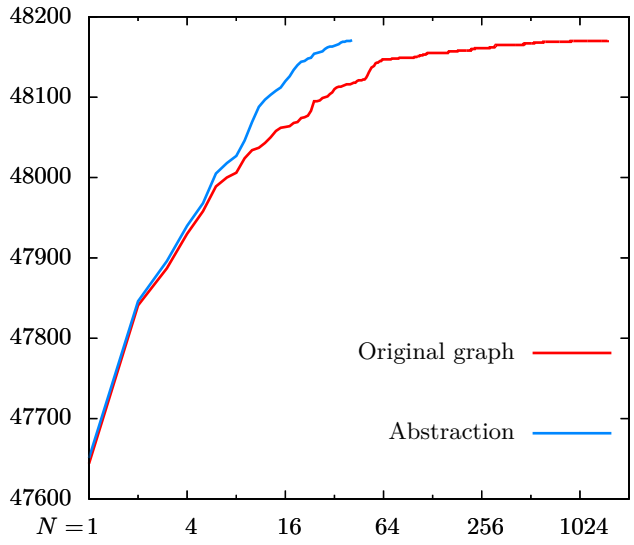

Figure 5. Effect of the time abstraction: Y-axis is the number of edges with N or less contiguous ranges in their time intervals.

This operation increases lifetimes of all paths, and so it preserves viable paths.

The gain comes from the smaller representation of the time intervals. We represent them with a list of disjoint half-open ranges like $[l, h)$. A frequently updated field has an irregular time interval with many such ranges. Determining viability of a path requires taking set intersection of edge time intervals along the path. When a complex time interval occurs on the path, it adversely affects the computation time. Moreover, since we are dealing with large data structures, memory representation becomes important. Complex time intervals cannot be simply represented in memory and so require more space. Time abstraction applies a simple expansion to every range $[l, h)$ in the interval: $\left[\left\lfloor \frac{l}{r} \right\rfloor \cdot r, \left\lceil \frac{h}{r} \right\rceil \cdot r\right)$ (where r is a tunable parameter). When the distance between two consecutive ranges is smaller than r, they collapse into a single range.

Figure 5 (red) shows the number of such ranges in time intervals in a real heap series built from trace e_3_25 (see Sec. 6.1). After expanding edges with more than 10 ranges with $r = 128 * 1024$, the edges with irregular time intervals disappear from the abstraction (blue).

3. DeLight Collection and Storage

The basic architecture of DeLight system is shown in Fig. 6. The components of our system are: the trace collector that extracts traces from executions of programs, the query engine that organizes traces and exposes the high-level interface, and the lower-level

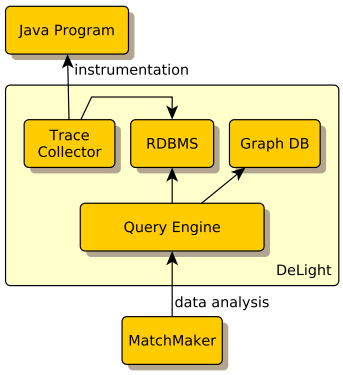

Figure 6. System architecture

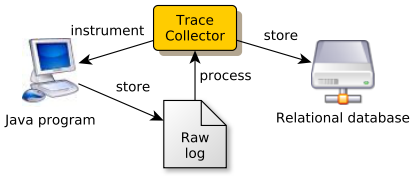

Figure 7. Collection agent

storage databases. MatchMaker is the target client of DeLight.

The work flow of building a database of program behaviors consists of four steps:

1. Executing the subject Java program with a dynamically instrumenting Java agent (Sec. 3.1).
2. Interacting with the program while the agent records the raw log.
3. Off-line processing of the raw log and storing the data in databases (Sec. 3.2).
4. Registering the new trace with the query engine under a single query interface (Sec. 3.3).

3.1 Instrumentation

Instrumentation is performed dynamically using ASM bytecode instrumentation framework [1]. Every application class is modified upon loading to record its events in a raw log file. Once all the desired functionality of the program is exercised, the collection agent processes the log file and bulk inserts the data into a relational database (see Fig. 7).

Filtering events Significant portion of events are safely ignored without degrading the quality of the synthesized code. Statements are instrumented selectively emphasizing object interactions in the application code as opposed to internals of the language library.

First, the trace collector does not distinguish between primitive values and treats them all as a single abstract

value. In a high-level language like Java, much of the program logic is encoded in the interaction between objects rather than in the manipulation of primitive values.

Second, the recording framework ignores most of the events related to classes in the Java standard library. It does not record any field reads or writes to instances of java.* and javax.* classes. Among JDK classes, it only records public method boundary events for commonly used collection classes such as java.util.ArrayList and java.util.HashMap. The rest of the program (*e.g.* application packages such as org.eclipse.*) is fully recorded. Since the collector instruments the methods themselves, callbacks from the standard library to the application are still recorded.

Concurrency DeLight handles concurrent programs by serializing events from multiple threads with a lock to the single raw log. Thus, MatchMaker produces code that may potentially be executed concurrently since it uses events from DeLight database. Currently, DeLight does not log synchronization statements. We have not yet found it to be problematic since frameworks like Eclipse manage thread coordination themselves, relieving user code from manual concurrency control. Without thread synchronization information in the database, MatchMaker cannot generate correct synchronization statements in the cases where user thread coordination is required.

3.2 Data Representation

The two data model presentations call for different kinds of representations. Call tree view closely matches the relational table form. All events are stored in a single nested-set table that has columns for the timestamp, thread, stack depth, successor (matching method exit event), and event-specific details. Parameters to method calls and object to type mapping are stored in separate tables. The underlying RDBMS engine is MySQL 5.1. The tables are augmented with specialized database indexes to optimize slicing queries.

For graph-based heap series, we use the graph database Neo4J 1.2 [12]. This database is optimized for graph traversals which is the main use-case in DeLight. The heap series is built by replaying the entire event sequence. Heap abstractions are represented using in-memory graph data structures and are built from the heap series.

3.3 Benefits of the Data-Driven Approach

The advantage of a data-driven engine is that it scales well by partitioning the trace data and running the queries on each partition in parallel. In addition to trace-level parallelism, each individual trace can be split into several smaller traces in order to balance the lengths of traces across partitions. The more traces are put into DeLight, the more queries it can potentially answer since more of the program behavior is exercised.

4. Critical Chain Algorithm

MatchMaker algorithm performs three different kinds of analysis on the data in order to generate code. First, it finds critical chains connecting objects of the two given types as we describe in this section. Then it computes a slice based on the events in each critical chain, and finally, it synthesizes the code from the slices (Sec. 5).

A chain is a simple path in the heap connecting objects via directed edges labeled with concrete or abstract fields. As the heap evolves over time, chains form and disappear, but with the entire sequence of heaps at hand, we are able to determine the earliest moment when the two given objects get connected by a chain.

We call the event corresponding to this moment the *critical event*, and any chain between the two objects created by the critical event is called a *critical* chain (recall from Fig. 3 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.

We formulate the critical event query 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).

Thus, we have developed an algorithm for computing critical chains that makes use of heap abstraction $\mathcal{A}(\widehat{\mathcal{H}})$ to drive an exhaustive graph search on heap series $\widehat{\mathcal{H}}$. By framing the problem as graph search, we can also do more than just finding the very first critical chain. We give the user the ability to search for the subsequent chains, providing an iterator-like interface to

Input: chain $\{b_0, \ldots, b_k\}$ in $\mathcal{A}(\widehat{\mathcal{H}})$, graph view G of $\widehat{\mathcal{H}}$
  for $i = 0$ **to** k **do**
    choose a_i such that $\mathsf{cluster}(a_i) = b_i$ and
      $\{a_0, \ldots, a_i\}$ is a simple, viable chain in G
  return $\{a_i\}$ or the shortest unsatisfiable prefix of $\{b_i\}$

Figure 8. Algorithm CONCRETIZE.

Input: clusters $s, t \in \mathcal{A}(\widehat{\mathcal{H}})$, graph view G of $\widehat{\mathcal{H}}$
  $\mathcal{A}_G \leftarrow$ restriction of $\mathcal{A}(\widehat{\mathcal{H}})$ to duration time interval of G and
    subgraph of nodes reachable from s and to t
  $Q \leftarrow \langle\{s\}\rangle$
  while Q is non-empty **do**
    $\mathbf{b} = \{b_0, \ldots, b_k\} \leftarrow Q$
    if $b_k = t$ **then**
      match CONCRETIZE($\mathbf{b}, G$):
        case a in $G \Rightarrow$ **return a**
        case prefix $\mathbf{c} \sqsubseteq \mathbf{b} \Rightarrow$ remove all $\mathbf{p} \in Q$ such that $\mathbf{c} \sqsubseteq \mathbf{p}$
    else
      for all $b_k \to o \in \mathcal{A}_G$ **do**
        if b' $= \{b_0, \ldots, b_k, o\}$ is simple and viable **then**
          $Q \leftarrow Q \cup$ **b'**

Figure 9. Algorithm SEARCH

query chains between two objects. The algorithm consists of two parts: SEARCH that performs a traversal of $\mathcal{A}(\widehat{\mathcal{H}})$ to find candidate paths and CONCRETIZE that checks whether the candidate paths have concrete counterparts in $\widehat{\mathcal{H}}$.

4.1 Concretization

Algorithm CONCRETIZE (Fig. 8) is the necessary validation routine that takes a candidate chain in $\mathcal{A}(\widehat{\mathcal{H}})$ and attempts to find a corresponding chain in $\widehat{\mathcal{H}}$. In addition to a candidate chain, it takes as input a *graph view* of $\widehat{\mathcal{H}}$ that (1) excludes certain set of edges, (2) restricts edge time intervals to a certain duration time interval. The algorithm iteratively expands nodes in $\widehat{\mathcal{H}}$ that are in the view and match the candidate chain in $\mathcal{A}(\widehat{\mathcal{H}})$. If it fails to find the full concrete chain, it reports the shortest unsatisfiable prefix of the candidate chain.

4.2 Graph Search

Algorithm SEARCH (Fig. 9) is an all-paths breadth-first search algorithm on $\mathcal{A}(\widehat{\mathcal{H}})$. It takes source and target clusters in the abstraction graph, and the graph view of $\widehat{\mathcal{H}}$. It builds a queue of candidate chains in $\mathcal{A}(\widehat{\mathcal{H}})$ in a breadth-first fashion, calling CONCRETIZE when the end node of the chain matches the target cluster. If concretization succeeds, the algorithm stops; otherwise, it eliminates all chains in the queue that start from the same unsatisfiable prefix.

4.3 Query Interface

Combination of graph search and concretization allows us to define the following arsenal of high-level queries for enumerating meaningful chains in program traces.

Find a chain between two types The first time the user tries to search for a critical chain, he/she may only have the types of the end points in mind. Type-based clustering in the heap abstraction is particularly suitable to this kind of query. To answer such query, we simply execute SEARCH on $\widehat{\mathcal{H}}$ with the two input types.

Find a critical chain between two objects Once we have one chain between two objects, we can minimize the critical event time by iterative execution of SEARCH on a graph view of $\widehat{\mathcal{H}}$ with the time duration $[0, t)$, where t is the critical time of the previous chain. The view also restricts the source and end objects of the chain to the ones in the previous chain. The algorithm converges when no new chain is found.

Find the next chain Sometimes, the first chain might not be the one that is desirable. Therefore, one may need to find a subsequent chain by specifying an edge to drop from the current chain in $\widehat{\mathcal{H}}$. This is done by passing a graph view to SEARCH that skips certain edges in $\widehat{\mathcal{H}}$ and restricts time intervals to $[t, \infty)$, where t is the critical event time of the first chain.

4.4 Benefits of the Heap Abstraction

Apart from significantly reducing the size of the graphs, heap abstractions improve the search algorithm. First, the queue of abstract paths is a more compact representation than a queue of concrete paths: they share prefixes, have simpler time intervals, and do not have fields. Second, concretization limits expansion to only a subset of concrete nodes improving performance of the graph database (since heap series can only be partially loaded into memory, it must reside on disk). Finally, non-viable abstract paths are eagerly eliminated during search. For example, the query used in the user study (Sec. 6.4) eliminates around 10% of abstract paths each accounting for all concrete paths having it as a prefix.

5. MatchMaker Synthesis Algorithm

The goal of the synthesis algorithm is to produce *user code* that when called by the framework causes a critical chain to form. Its starting point are the critical chains computed in Sec. 4. Each critical chain identifies the set of events that created each of its links, but these events are usually deep inside the framework and invisible to the user. In order to derive user code from these events, the algorithm follows a two-step process: (1) first, it identifies for each critical chain the actions of the user code that led to its creation, and (2) then it generalizes from those specific instances to produce the essential code needed to create the chain. The first step is achieved through a form of slicing, while the second step is achieved through a new projection algorithm that eliminates trace-specific details from the slice. The sections that follow elaborate on each of these steps.

5.1 Dynamic Thin Slicing

The synthesis algorithm relies on a dynamic form of *thin slicing*, an approach to slicing pioneered by Sridharan *et al.* that has been shown to be very effective when computing slices for the purpose of program understanding [15, 19]. The basic observation behind thin slicing is that traditional slices contain too much information; for example, a slice for an operation that retrieves an element from a data-structure will contain not only the event that inserted the element into the data-structure but also many other events that modified the data-structure, including many that added and removed other unrelated elements.

Thin slicing avoids many of these irrelevant events by ignoring value flows to base pointers of heap accesses; for example, for a field write $b.f \leftarrow a$, thin slicing only follows value flows to a, whereas a traditional slicing algorithm would also follow value flows to b. Thin slicing also does not follow *control dependencies*. 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 (they call these *expansions* of thin slicing). In our case, the algorithm cannot rely on the programmer to decide when it might be useful to follow value flows to base pointers, so instead it applies a set of simple heuristics to do *automatic expansion* based on the observation that the synthesis algorithm cares primarily about user code and not so much about what happens in framework code:

- When slicing on value a the assignments $b.f \leftarrow a$ or $a \leftarrow b.f$ causes the algorithm to follow the base pointer b if and only if: (a) the statement is within a framework method, but the value of b is of a user-defined class, or (b) the statement is within a user method, and the producer statement and the consumer statement of the value are not in the same method.
- When examining a statement e in user code, walk through the call stack of e and find the nearest call $o'.f()$ where f is defined in user code and the caller is part of the framework. Add the dependency between the call event and e through object o' to the slice. The rationale for including this dependency in the slice is that object o' must be correctly set in order for the framework to call the right method f, so e has a control dependency on o' to the call event.

We use the symbol $\mathcal{S}$ to refer to the dependency relation computed by the slicing algorithm. A tuple $e \xrightarrow{o} e'$ belongs to $\mathcal{S}$ whenever the event e is a producer of the object o which is then directly consumed by the event e'.[1] Figure 11 shows a fragment of the slice for the statement WidgetViewer.addView(x) in Fig. 10. In this simple example, we assume that MyWidget and MyView are user classes while WidgetViewer, Widget, and View are framework classes.

```
class WidgetViewer { // Framework class
  void main() {
    init();        // e1
    initViews(); // e6
  }
  void init(...) {
    Widget u = ... // e2
    u.init();      // e3
  }
  void initViews(...) {
    Widget u = ... // e7
    u.regViews();  // e8
  }
  static void addView(View x) { ... }
}
class MyWidget extends Widget { // User class
  private MyView f;
  @Override void init() {
    MyView x = new MyView(); // e4
    this.f = x;              // e5
  }
  @Override void regViews() {
    MyView x = getView();    // e9
    WidgetViewer.addView(x); // e12
  }
  private MyView getView() {
    MyView x = this.f; // e10
    return x;          // e11
  }
}
```

Figure 10. Example code.

The approach defined above raises an important question: how does the system know where to draw the boundary between user and framework code? MatchMaker draws the boundary by relying on the Java package mechanism. In Eclipse, each plug-in corresponds to a *bundle* of packages. For example, the Eclipse Java Development Tools plug-in corresponds to a bundle containing all packages with the name org.eclipse.jdt.∗, and the TeXlipse plug-in corresponds to a bundle containing all packages with the name org.texlipse.∗. For each plug-in in the DeLight program behavior database, we manually specify its corresponding bundle. The classes in the bundles are potential user classes, and classes not in any bundle are considered framework classes. When MatchMaker finds a critical chain from object A to object B, if the class of A is in some bundle X, then MatchMaker treats all classes in X as user classes.

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

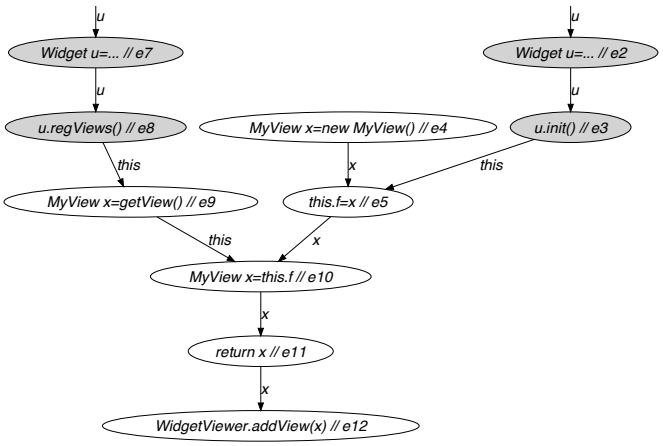

Figure 11. Thin slice for the event WidgetViewer.addView (x) in Fig. 10. Shaded events are framework events.

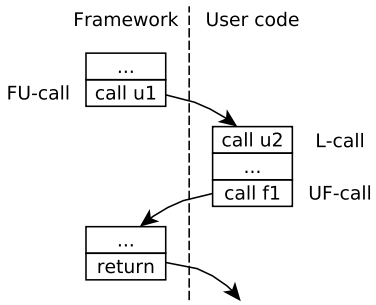

Figure 12. User code and framework interaction.

MatchMaker does the same thing to B, and all other classes are treated as framework class. By this means, MatchMaker gets the framework-user separation and uses it to guide slicing and projection.

5.2 Projection to User Code

The slice created by MatchMaker contains all the statements that were necessary to create the critical chain in a particular heap. However, a lot of the code in the slice 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 user code. MatchMaker addresses this problem by computing a *projected slice* that eliminates framework code as well as superfluous details from the original slice. The projection process also has the effect of normalizing the slices, making it easier to compare the results from many different traces. Specifically, projection causes many slices to become identical, and those that remain different usually correspond to different ways of using the framework.

The essence of the interaction between user and framework code is captured by calls that cross the framework-user boundary. Calls from the framework to the user are called FU-calls, and calls from the user code to the framework are called UF-calls. In the example from Fig. 10, e3 and e8 are FU-calls, while e12 is a UF-call; the rest of the calls, like e9 are termed L-calls, because they stay local to either the framework or the user code (see Fig. 12).

FU-calls and UF-calls describe the interaction between framework and user: FU-calls tell us which classes to extend and which method to override, while UF-calls tell us which APIs to call. The projected slice must preserve information about FU-calls and UF-calls as well as relevant user code, but simplify away the complexity and arbitrariness of the specific sample code from which the database was built. For the running example, the projected slice will indicate that init() must write to f and regViews() must pass the value of f to addView(), but the fact that this involves a call to getView() is only a detail of this particular example and not relevant in the projected slice.

We first introduce several definitions to talk about the dynamic structure of calls. For any user event e, the *cover* of e is the latest enclosing FU-call of e. In other words, all calls on the stack from $\mathsf{cover}(e)$ to e are user L-calls except for $\mathsf{cover}(e)$ itself which must reside in the framework code but invoke a user method. For example, in the sample program $\mathsf{e3} = \mathsf{cover}(\mathsf{e5})$ and $\mathsf{e8} = \mathsf{cover}(\mathsf{e10})$. Dually, we define the cover of a framework event to be its latest enclosing UF-call. We say that an event e is a *framework event* (written as $\mathcal{F}(e)$) if it is generated by a statement in the framework code; otherwise, the event is a *user event* (or $\mathcal{U}(e)$). The *local producer* of a user event e and an object o, consumed by e, is the earliest event that produced object o and has the same cover as e. The local producer is always a user event. For example, the local producer of the object x in e12 is e10 since they have the same cover e8.

The projection algorithm takes the set of events E establishing the critical chain as a starting point. It uses the slicing relation $\mathcal{S}$ and the function cover, both provided by DeLight, to compute the projected slice $\hat{E}$ consisting only of user events. The algorithm consists of two steps: the base step computes the initial set $\hat{E}$ from E and the iteration step expands $\hat{E}$ to reach the least fixed point.

Base step First, every user event from E is included: $\{e \mid e \in E \wedge \mathcal{U}(e)\}$. Next, from the set $E^* = \mathcal{S}^*(E)$ of all events in the thin slice, the algorithm adds the following groups of user events:

- covers of the framework events:

$$\{\mathsf{cover}(e) \mid e \in E^* \wedge \mathcal{F}(e)\}$$

- return events of the covers of the user events:

$$\{\text{return event of the call } \mathsf{cover}(e) \mid e \in E^* \wedge \mathcal{U}(e)\}$$

- user events that produce data consumed by the framework events:

$$\{e' \mid e \in E^* \wedge \mathcal{F}(e) \wedge \mathcal{U}(e') \wedge \left(\exists o : e' \xrightarrow{o} e \in \mathcal{S}\right)\}$$

Iteration At every step, the algorithm picks an event $e \in \hat{E}$ and an object o that it consumes. If there is no local producer, the producer e' (*i.e.* $e' \xrightarrow{o} e \in \mathcal{S}$) must lie outside of the call $\mathsf{cover}(e)$. In that case, the algorithm adds e' to $\hat{E}$ only if e' is a user event. If there is a local producer for e and o, then the algorithm adds it only if that local producer got the value from the heap or from a constructor. In the example, this is relevant when the algorithm picks event `e10` and object **this**; in that case, the local producer of **this** is event `e9`, but `e9` gets the value from the parameter list of the call `e8`, so `e9` is not added to $\hat{E}$.

Using the above rules, MatchMaker computes the least fixed point of $\hat{E}$. It is easy to synthesize code from the projected slice $\hat{E}$: the covers tell us which methods to override and the events in $\hat{E}$ tell us what instructions to put in these overridden method, while the dependency relation $\mathcal{S}$ glues the instructions together by data dependency and gives a partial order on the instructions. In this sense, the projected slice is as expressive as user source code, but it elides low-level details such as reordering of independent instructions.

Just before synthesizing the code, MatchMaker havs two more things to do. First, MatchMaker finds for each user class `A` the most generic framework class to extend from. For each FU-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 UF-call `g(x0, x1, ..., xk)` where `x0` is the receiver and the rest `xi`'s are actual parameters, if the original type of the j-th formal parameter (including 0-th, the receiver) is `A_j` and `xj` is an instance of `A`, then `A` must be a subclass of `A_j`. MatchMaker extracts from FU-calls and UF-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`.

Finally, MatchMaker gives each user class `A` a pretty name: if `A` is determined to extend from `C`, then `A` is called `MyC`. MatchMaker names each variable of type `X` by making the first letter of `X` lower case and appending a unique number suffix. MatchMaker also renames each field `X.f` of type `Y` to `X.fY` (again add numbers to avoid conflict field names). Unresolved variables are named `??`, the *holes* in the code returned to the user.

6. Evaluation

We have implemented MatchMaker and performed a user study (see Sec. 6.4) that demonstrates the positive effect of our tool on developer productivity on a challenge problem inspired by our motivating example. To assess the extent of validity of MatchMaker assumption, we have also performed a generality case study (see Sec. 6.3). In the case study, we analyzed the quality of the code synthesized by the projection algorithm for a number of pairs of classes from Eclipse.

We open this section with a description of the experimental setup: the amount of trace data we collected and DeLight collection performance. Sec. 6.2 focuses on the critical chain algorithm performance and demonstrates that DeLight finds chains in just a few seconds on 100 GB of trace data, implying that DeLight can be a foundation for an interactive tool like MatchMaker.

6.1 Experimental Setup

We have collected around 100 GB of data from nine executions of Eclipse 3.6.1 with several plug-ins installed (see Fig. 13). All executions except one took less than 5 minutes. We performed the following actions in Eclipse:

1. clicking on items in the toolbar and menu;
2. editing Java and `.properties` files;
3. editing BibTeX and TeX files using TeXlipse;
4. editing a Python file using PyDev;
5. editing a grammar using ANTLR IDE;
6. invoking a shortcut (*e.g.* to save a document);
7. navigating using outline viewer;
8. using a spell checker to correct a word;
9. invoking auto completion while typing in an editor;
10. browsing a CVS repository and its commit history;
11. running a sample RCP application;
12. running a sample Eclipse editor plug-in application.

These actions were selected as a representative sample of what a user *may* do in Eclipse, not as a sample of MatchMaker queries we expected to run. We did strive for completeness, however, in that we tried to exercise similar functionality (such as auto-complete) in all editors.

The effect of instrumentation on the subject application is noticeable in the early phase of instrumentation when a large number of classes are loaded, instrumented, and verified by the class loader. Once dynamic instrumentation completes, the application runs slower but still at an interactive speed, allowing us to use features like

Database	# of events	Execution time	Processing time	SQL DB size	Heap series graph size	Heap abstraction size	Build time	Graph DB size
e_2_28	68 M	<5 min	61 min	6.9 GB	0.887 M / 1.675 M	50 K / 75 K	199 s	334 MB
e_3_3	573 M	60 min	1076 min	57.3 GB	4.143 M / 9.098 M	225 K / 292 K	35 m	1.7 GB
e_3_25	37 M	<5 min	28 min	3.6 GB	0.495 M / 1.206 M	30 K / 48 K	159 s	223 MB
e_3_26	81 M	<5 min	76 min	7.9 GB	1.333 M / 2.747 M	67 K / 93 K	6 m	531 MB
e_3_27	52 M	<5 min	47 min	5.1 GB	0.599 M / 1.403 M	36 K / 57 K	200 s	262 MB
c_3_28	24 M	<5 min	19 min	2.3 GB	0.365 M / 0.840 M	22 K / 35 K	98 s	158 MB
e_3_31	35 M	<5 min	30 min	3.6 GB	0.686 M / 1.272 M	39 K / 56 K	181 s	255 MB
t_4_3	29 M	<5 min	24 min	2.9 GB	0.416 M / 1.004 M	26 K / 41 K	121 s	186 MB
h_4_4	3 M	<5 min	3 min	395 MB	0.100 M / 0.230 M	4 K / 7 K	12 s	44 MB

Figure 13. Collected execution data. For graph sizes, n/e means a graph with n nodes and e edges. Execution time is the running time of the subject application. Processing time is the time to process and store a trace in MySQL. Heap build time is the aggregate time to construct heap series and abstraction and store graphs Neo4J.

auto-completion without causing internal Eclipse time-outs. The numbers in Fig. 13 suggest that 1 minute of execution produces roughly 1 GB of data and requires 10 to 20 minutes for off-line processing.

We used a quad-core machine with 7 GB RAM for DeLight collection and MatchMaker queries. Both tools are implemented in Scala. We have allocated 5 GB to Java HotSpot 64-bit JVM 1.6.0. We ran Neo4J 1.2 graph database in the same JVM instance and MySQL 5.1 on the same machine.

6.2 DeLight Query Performance

To evaluate performance of DeLight critical chain queries, we took a sample of pairs of types (see Fig. 14). For every such pair, we asked DeLight to find a chain for every pair of subtypes on every database (nine of them) in succession. We report the total number of chains found and the average time per chain. If the chain was not found, it could be because of two reasons: Search ran out of the imposed execution time bound, or the chain was genuinely missing from the collected data. We report the total number of time-outs (and also as a percentage of all failed queries). For the remaining missing chains, we indicate the average time. We manually inspected the pairs that time-out and concluded that the chains, indeed, are unlikely to be present (often, since the two end types belong to two distinct plug-ins.)

For every chain, we executed the critical chain query. We show the number of queries that time-out in Search and the average time for the remaining successful queries. Note that a timed-out critical chain query does not necessarily result in a wrong answer. It simply indicates that Search was unable to prove non-existence of a path in a graph view. Even if the answer is not exact, the current candidate chain returned by the critical chain algorithm is likely to be good enough for MatchMaker.

6.3 MatchMaker Generality Case Study

We have performed the following experiment to evaluate the generality of MatchMaker in the context of Eclipse. During the process of developing MatchMaker we have learned four class pairs for which the tool generates adequate source code: Editor to Scanner, Editor to ICompletionProposal (an auto-completion choice in Eclipse), Menu to Action, and Toolbar to Action. The synthesized code is close to the code that would be written by a human expert, except for two kinds of imperfections:

- Some method call parameters may not resolve and are shown as ?? (holes). The human needs to consult the documentation and tutorials to find out the exact values for these holes.
- Some statements cannot be captured by the critical chain, such as the call to reconciler.setDamager() in the Editor–Scanner example (see Fig. 2). The critical chain chosen by MatchMaker passes through the link from the reconciler to the repairer, but not through the damager, so MatchMaker is unable to find the call to setDamager. The human needs to manually add the missing statements.

We think that the additional job is relatively easy for the human, because she now has the knowledge of all the necessary classes and majority of the 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 15, we show for each class pair the extra changes the human needs to make.

In addition to the above mentioned four class pairs, we tried a dozen more pairs from Eclipse. We selected these pairs as follows: we took eclipse.jdt.internal.ui plug-in, treated everything in eclipse.jdt.∗ as the user code, and extracted all framework classes that are extended and used inside this plug-in. There are more than two hundred of these classes. We examined these classes manually and picked pairs of classes that seemed relevant to each other judging from their names. We were able

Source type (# of subtypes)	Target type (# of subtypes)	Find a chain				Find a critical chain	
		found		not found		t.o.	avg time
		#	avg (max) time	t.o. (%)	avg time		
AbstractTextEditor (17)	RuleBasedScanner (40)	46	1.3 s (4.6 s)	14 (0.2 %)	1 ms	10	1.6 s
AbstractTextEditor (17)	ICompletionProposal (44)	37	2.8 s (11.6 s)	20 (0.3 %)	1 ms	20	0.9 s
WorkbenchWindow (1)	PartEventAction (14)	74	1.4 s (20.6 s)	6 (11.5 %)	1 ms	66	147 ms

Figure 14. Critical chain computation performance. SEARCH execution was bounded by 30 seconds ("t.o." indicates the number of queries in which the algorithm ran out of time.)

Source type	Target type	# holes	# miss
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
IContentAssist-Processor	ContentAssist-InvocationContext	1	0
ITextEditor	IAutoEditStrategy	0	0
ITextEditor	ContextBased-FormattingStrategy	1	2
ITextEditor	TextFile-DocumentProvider	0	0

Figure 15. Generality evaluation. First column shows the names of the class pairs. "# holes" and "# miss" list the number of holes and missing statements in the generated code.

to pick 16 pairs of classes that we thought related with each other. Then we ran MATCHMAKER on each of these pairs, and examined the generated code to see whether it is acceptable glue code to establish 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 generates reasonable code; and for the remaining 4 pairs MATCHMAKER generates empty code, but we have yet to find whether it is because of incompleteness of our program behavior database or because of limitations of MATCHMAKER hypothesis.

From Figure 15 we can see that for the pairs for which MATCHMAKER generates solutions, the resulting code is quite close to the correct answer. This suggests that the MATCHMAKER approach can be generally applied to a wider range and is not restricted to the elaborated example.

6.4 User Study

We have conducted a user study to measure the effect of MATCHMAKER on programmer productivity. The results of the user study show that for the programming task we tested, MATCHMAKER improved programmer productivity by 49 percent on average, and the improvement is statistically significant.

Task Description For our user study, subjects were asked to implement *Syntax Highlighting* in an editor for a new language—the SKETCH language developed by our group. Specifically, the subjects were asked to implement syntax highlighting for two keywords in the language: **implements** and the operator ??. As a starting point, we provided them with an incomplete class that extends and overrides RuleBasedScanner. The class implemented highlighting only for the keyword **implements**, but they had to complete it and connect it to the editor by writing glue code like the one in Fig. 2.

As can be seen from Sec. 6.3, the code generated by MATCHMAKER might not be perfect: it may contain holes or miss statements. To observe how users deal with imperfect synthesized code, we intentionally chose a task for which MATCHMAKER generates code with a hole and a missing statement, so the MATCHMAKER user had 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, and might perform even better, but that would not give us a comprehensive example of using MATCHMAKER.

Methodology We recruited participants through mass advertising around the campuses of MIT and Harvard with the promise of two free movie tickets for any participant who attends the study. When participants arrived, they were randomly assigned to one of two groups: those in the *control* group were simply given a description of the task and were told they could use any information available on the Internet to help them complete the study; those in the *experimental* group were given an additional 10 minutes to review a tutorial on MATCHMAKER. The tutorial showed how to use MATCHMAKER on an unrelated matching problem. The subjects in the experimental group were 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 did not know they were in the control group, or even of the existence of MatchMaker; they were led to believe that the purpose of the study is simply to analyze programmer's use of the Eclipse framework.

The work environment was a virtual machine created with VirtualBox. The virtual machine was set up with Eclipse IDE, Google Chrome browser, and other frequently used applications to allow users to read various documentation formats. The virtual machine was set up to do screen-captures at 1 frame per second; this, together with the "local history" feature of Eclipse IDE and the subject's browsing history gave us a very complete picture of the programmer's actions during the user study.

Subjects A total of nine subjects completed the user study. Four of them were randomly assigned to the experimental group (we will call them MatchMaker users M1, M2, M3, and M4), four others (we will call them C1, C2, C3, and C4) were assigned to the control group, and the other one was an expert (E0) in the Eclipse framework.

All subjects in both experimental and control groups had competent Java programming experience, but none of them had ever written any plug-in for Eclipse or similar object-oriented frameworks. On the other hand, E0 had five years of experience writing Eclipse plug-ins – it is safe to say that he knows significantly more about the Eclipse framework than we do. He was firstly assigned to the experimental group and given the MatchMaker tool, but he never used it because he was extremely comfortable writing Eclipse code without the aid of the tool. After he finished the task he told us in the questionnaire that he was an expert. Therefore we put him into a separate category.

Aside from the nine subjects who finished the task, there were another three subjects (X1, X2, X3) who did only part of the task and were excluded from the result: X1 was assigned to the control group. He spent around 40 minutes but still did not finish the glue code, and then he was distracted by a phone call and had to quit the study. X2 was assigned to the experimental group. He spent about 20 minutes and successfully established partial connection (see Phase 2 of Sec. 6.4.2) between editor and scanner, and then he also quit because of a phone call. X3 was assigned to the control group. He spent around half an hour but did not finish the glue code, and complained to us that he was totally clueless and felt that he "would never finish the task", and then he quit.

6.4.1 Result

Figure 16 shows time spent on the task by each subject; the total time consumption is split into four work items:
Glue Coding is the time spent writing and debugging *glue code* to match the Editor class with the Scanner class. By glue code we mean the source code that pieces components in the framework together, such as the code shown in Fig. 2.
Web Browsing for Glue Code is the time spent browsing the web (including searching and reading the documentation, tutorials, *etc.*) to find out how to write glue code.
Task Coding is the time spent writing *task code* to complete the Scanner. Task code is the source code in the subclass of RuleBasedScanner that identifies the two keywords.
Web Browsing for Task Code is the time spent 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: 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 for Glue Code, and any web browsing immediately before writing task code is considered Web Browsing for Task Code.

Figure 16 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. Web Browsing for Glue Code and Glue Coding together form the time spent on the matching problem (connecting two classes, Editor and Scanner, together). From the figure we can see that MatchMaker improves productivity by reducing the time spent on the matching problem. On average, control subjects spent 82.25 minutes writing the glue code to match Editor with Scanner and 98 minutes to finish the whole task, while MatchMaker reduces these times to 35.75 minutes (56% improvement) and 50.25 minutes (49% improvement), respectively.

The Wilcoxon Rank-Sum test shows that the difference between the times spent on the whole task by the control group and the experimental group is statistically significant (p-value=0.03), and the difference between the times spent on the matching problem by the control group and the experimental group is also statistically significant (p-value=0.03). Even if we add 10 minutes on reviewing the MatchMaker tutorial to the total time spent by each MatchMaker user, the experimental group still performs faster than the control group, and the p-value of the Wilcoxon Rank-Sum test remains 0.03,

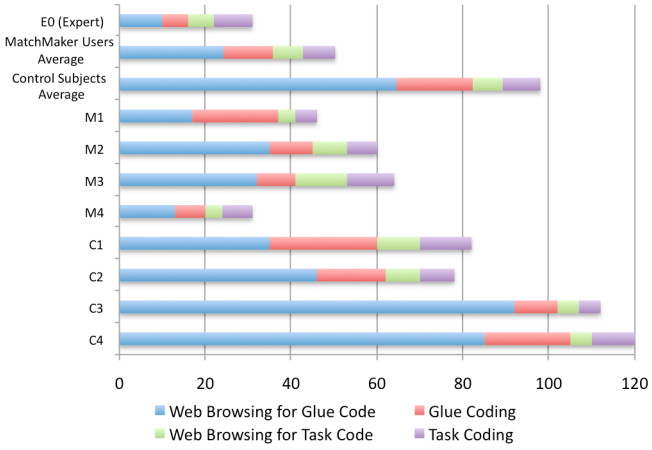

Figure 16. Time spent on the task (in minutes), split to four work items.

indicating a statistically significant difference between the two groups.

6.4.2 Observations

We pick a representative MatchMaker user M3 to describe in detail the process of using MatchMaker to solve the task. The process is naturally split into four phases:

Phase 1 M3 started by entering a query "TextEditor – RuleBasedScanner" into MatchMaker, and got the synthesized code after less than one minute. Then he copied and pasted the code into his project, and split it into several class files. At this point he had to fill the hole in the statement reconciler.setRepairer(dr, ??), and he did not know that there is a missing statement reconciler.setDamager.

Phase 2 M3 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 make the program run. Because setDamager was missing, this was 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 M3 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 of the missing setDamager, he fixed it immediately and got the correct glue code.

Phase 4 M3 spent another 23 minutes developing the task code, including browsing the web and writing and debugging the code.

This is a very typical usage of MatchMaker. The other three users all showed a similar pattern.

The control subjects, on the other hand, struggled with glue coding. For example, C1 and C2 both browsed the web for more than 25 minutes before writing the first line of glue code, and both spent a total of one hour to produce correct glue code. The other two control subjects C3 and C4 spent more than 80 minutes browsing the web to find the answer to the matching problem, and more than 100 minutes to solve it.

It is worth pointing out that there are a number of tutorials on the web that describe exactly how to implement syntax highlighting, several of which were visited by all four control subjects. However, these tutorials are either poorly written or contain too much information, so it took them a significant amount of time to extract the relevant facts from these tutorials. As an example, Subject C3 came to a tutorial with all the details of implementing syntax highlighting within his first three searches, but this tutorial is very verbose and contains information about many components related to text editors in Eclipse and long descriptions explaining the internal relationship among those components, so C3 spent about 7 minutes reading the tutorial without extracting anything essential. Instead, he turned to the official API reference of Eclipse, a JavaDoc style document with little information about interactions among classes. After spending a lot of time on the API reference, C3 discovered another tutorial on the official Eclipse web site, which is the most authoritative tutorial on this topic. Although this tutorial is more concise than the previous one, it is still verbose and hard to read, so C3 quickly closed it and turned to the API reference again. During Web Browsing for Glue Code, C3 visited the tutorial on the Eclipse web site a total of four times, but for the first three times only read it for less than 2 minutes. At last he realized that it had the information he needed, read it carefully for 5 minutes, and got the answer, but he had already spent more than an hour on web browsing. The other control subjects faced similar difficulties in deciding which documents were valuable and in extracting useful information from the long tutorials.

The MatchMaker users also browsed the web for help, but they did it in a different way. They had specific goals when browsing: to fill the holes in the statements generated by MatchMaker, or to find out the missing statement. In both cases they used the class names or the method names as keywords to quickly skim the tutorial or even search around the tutorial, so they were more efficient in deciding which tutorials were valuable and locating the essential code snippet. MatchMaker helped them by improving their effectiveness in using the online resources. This is

supported by the data: MatchMaker users spent 24.25 minutes on Web Browsing for Glue Code, compared to 64.5 minutes on average of control subjects. The Wilcoxon test shows that the difference is statistically significant (with p-value=0.04).

The framework expert was different from subjects in both control and experimental groups: he knew exactly what he was looking for on the web and found it very quickly, so he could finish glue code in 16 minutes.

Overall, our observations from the subjects allow us to draw 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 turn, 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 statistically 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.

7. Related work

The idea of using large corpus of data for program understanding has seen many incarnations in the past few years. Prospector [9], XSnippet [13], MAPO [20], PARSEWeb [17], and Strathcona [7] mine source code repositories and assist programmers in common tasks: finding call sequences to derive an object of one type from an object of another type, complex initialization patterns, and 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.

Whyline [8] combines source code analysis with dynamic analysis to assist debugging by tracing input/output events together with the program, and suggesting questions that relate external observations to internal method calls. DeLight could potentially serve as a common framework for tools like Whyline and MatchMaker that need to query program executions. 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.

PQL [10] proposes a query language for analyzing program execution statically or dynamically. It is aimed at finding design defects in the code and as such requires detailed knowledge of the code. 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. FUDA [6] is closely related in its goal of producing program templates from example traces. Like our system, this tool also leverages the distinction between user and framework code to project slices. However, the API trace slicing used in FUDA only uses shared objects in argument lists of calls to detect dependencies in the heap. FUDA does not keep track of the heap updates. All of the above tools rely on light-weight dynamic analysis and as such, require manual effort in formulating queries in a specialized language or instrumenting programs specifically for these queries. MatchMaker attempts to reuse databases for answering all 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.

Program synthesis systems such as SKETCH [14] 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.

The projection algorithm in Sec. 5.2 is an instance of dynamic amorphous program slicing [5]. Like traditional program slicing, amorphous slicing simplifies a program while preserving a projection of its semantics. Unlike traditional program slicing that only allows picking a subset of the program, amorphous slicing may use any simplifying transformation. The projection algorithm used by MatchMaker is amorphous because it links consumer events directly to their local producers.

8. Conclusion

We have 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.

Acknowledgement We would like to give special thanks to Professor Greg Morrisett for helping us conduct part of the user study at Harvard University. We would also like to thank the participants in our user study. This research was supported by the National Science Foundation grant CCF-1049406 and by MIT's Computer Science and Artificial Intelligence Lab (CSAIL).

References

[1] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.

[2] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. PLDI '98, pages 106–117, New York, NY, USA, 1998. ACM.

[3] Eclipse. Helios release notes, 2010.

[4] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[5] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *J. Syst. Softw.*, 68:45–64, October 2003.

[6] A. Heydarnoori, K. Czarnecki, and T. T. 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.

[7] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[8] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[9] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[10] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[11] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Neo4J. Home page, 2011.

[13] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. OOPSLA '06, pages 413–430, New York, NY, USA, 2006. ACM.

[14] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. ASPLOS-XII, pages 404–415, New York, NY, USA, 2006. ACM.

[15] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[16] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

[17] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[18] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. PLDI '08, pages 125–135, New York, NY, USA, 2008. ACM.

[19] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. PLDI '10, pages 174–186, New York, NY, USA, 2010. ACM.

[20] 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.