StarCharter: Static Concept Mining from Code

Static Analysis for Software Engineering: Project Report

Steven She
shshe@uwaterloo.ca

December 10, 2008

Framework usage is a difficult and often frustrating process requiring in depth knowledge and understanding of the framework API. To ease the learning process, programmers often use example code or recipes as guides when using frameworks. Unfortunately, these resources tend not to be readily available.

We present StarCharter, a tool for synthesizing concept recipes by mining example code. Recipes describe a procedure for implementing framework concepts. Using StarCharter, users begin constructing a recipe by *starring statements* which they think are essential to the concept's implementation. Next, StarCharter infers code queries from these statements and searches for related examples. A concept slicer is then applied to each example to find related statements and types. Finally, the recipe miner reverse-engineers a recipe by finding associations and patterns in the sliced usages.

1 Introduction

Object-oriented frameworks provide reusable abstractions for building software systems. Unfortunately, understanding framework usage is often a difficult and time consuming process. Developers frequently use code snippets and existing code as guiding examples. However, these sample usages are typically not readily available to the user. For example, finding relevant implementation code may require a web search to locate the correct tutorial article followed by searching within the page for the code snippet. Existing concept usages in other applications and contexts provide a bountiful repository of examples, however, current techniques require a user to manually scour multiple source files to locate the relevant lines of code. Even then, the concept's usage tend to be scattered over multiple classes or methods and tangled with other unrelated concepts. In this report, we present StarCharter[1], a tool for aiding developers in locating relevant examples and constructing a recipe describing framework usage.

[1] The purpose of the tool is analogous to the process of identifying constellations in the night sky, in my opinion.

A user begins with StarCharter by specifying statements that he knows are essential to a concept's implementation. This is accomplished by *starring* code statements in the IDE editor. Afterwards, StarCharter derives related statements through *program slicing*, a process for retrieving statements that affect or are affected by the starred statements. The resulting slice constitutes a single example usage.

Examining a single example usage provides only one particular implementation of the concept. The user will not know with certainty that the chosen usage is correct and most likely the code will be incomplete. In addition, the analyzed implementation may contain superfluous code that is unrelated to the chosen concept. Luckily, there are often a considerable amount of example code accessible using a code search engine. As a result, StarCharter derives a code query from the starred statements and executes the query on a search engine to find related usages.

The final step in StarCharter involves the creation of an concept recipe [HBC07] using data mining. A concept recipe has the advantage of consolidating the patterns from multiple example usages into a single model. Superfluous and erroneous statements are filtered since these statements are typically infrequent. In addition, StarCharter derives probabilistic expression which can be used for providing guidance in future tool support.

StarCharter makes the following contributions:

- Heuristics for inferring code search queries from statements.
- Concept slicing, a specialized program slicing geared for analyzing framework usage.
- Synthesizing a model of framework usage in the form of a recipe.
- Providing an Eclipse plug-in for program slicing.

The remainder of this report is organized as follows. In Section 2, each component of StarCharter will be discussed in depth. Section 3 describes the current implementation prototype, Section 4 discusses related work, Section 5 discusses future work and Section 6 presents conclusions.

2 StarCharter

StarCharter aims to be a lightweight and interactive tool for understanding framework usage.

- **Lightweight**. The user should expend minimal effort to construct an initial recipe. The approach should be built into an IDE providing easy access for the user to invoke the tool and view the output.
- **Interactive**. While creation of the initial recipe should require minimal effort, the tool should also allow the user to iteratively modify the input and progressively see improved results.

An overview of StarCharter is shown in Figure 1. The workflow begins with the user selecting a set of starred statements and an optional framework boundary. There are four components in StarCharter:

1. **Statement Starring**. StarCharter's workflow begins with the user starring a set of statements. These starred statements represent what the user thinks is essential to the concept's implementation.

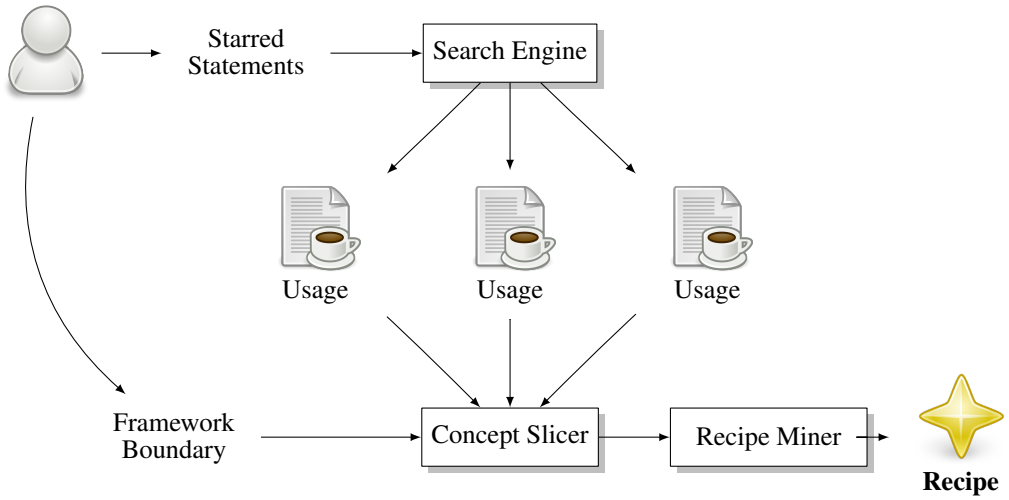


Figure 1: StarCharter overview

2. **Search Engine**. The search engine is responsible for finding related concept usages given the starred statements. It first derives a usage query given the set of starred statements then uses the query to find example usages.
3. **Concept Slicer**. After the search engine has found a set of example usages, the concept slicer finds related statements. The user can optionally input a framework boundary to restrict the slicer to only events pertaining to a framework. The output after processing by the concept slicer is set of program slices.
4. **Recipe Miner**. The program slices are combined into a single recipe using the recipe miner. The recipe miner mines for a feature model by discovering associations and patterns. The resulting recipe describes the aggregate behaviour exhibited in all the examined slices.

2.1 Starring Statements

StarCharter begins with the user selecting several statements in a sample program. These selected statements form a set of *starred statements*, which define the statements that the user believes to be essential to the concept's implementation. The starred statements have two purposes: (1) the statements are used as the starting points for slicing, and (2) they are used to construct a code query for finding related usages with the search engine. StarCharter currently supports two types of statements: *method invocations* and *variable assignments*.

2.2 Finding Concept Usages

The *search engine* in StarCharter is responsible for finding related examples given a set of starred statements. The starred statements are first generalized into a code query using several heuristics. Next, the search engine finds related usages that satisfy the inferred code query.

```
1  JLabel label = new JLabel("Hello World!");
★ 2  Container cp = getContentPane();
☆ 3  cp.add(label);
```

(a) LabelUsage.java

```
1  JButton button = new JButton("Click Here!");
❶ 2  JPanel panel = new JPanel();
② 3  getContentPane().add(button, 2);
```

(b) ButtonUsage.java

Figure 2: Finding related usages

The StarCharter search engine locates code for statements satisfying a given *code query* [ABC07]. A code query is an expression that matches a pattern found in framework usage code. Code queries are like meta-expressions that describes properties of the code itself, similar to aspect pointcuts [KLM+97]. A *matched statement* is one that satisfies a given code query. Code queries are *inferred* from the starred statements using the following heuristics:

- **Method Calls**. Given a starred method call, StarCharter will construct a code query that searches for all method invocations with the same name in the same type. For example, a starred call to foo.bar(x,y,z) will construct a query for all methods contained in foo's type with the name bar. Thus, parameter types and values are ignored in the inferred code query.
- **Variable Assignments**. A starred assignment will generate a code query for all assignments to variables with the same type. For example, the starred assignment Foo foo = getFoo(bar) will create a query that finds statements where a variable is assigned a instance of type Foo. The right-hand side of the assignment can either be a method or constructor invocation.

Multiple queries can be disjunctively composed such that the search results contain a union of the results from the smaller queries. Implementing conjunctive queries such that each search result satisfies all the conjoined queries is left for future work.

In Figure 2a, the user has starred two statements in LabelUsage.java. The first statement is an assignment to a Container variable shown by the ★. This starred statement is generalized into a code query that matches all variable assignments to Container variables. As a result the assignment to the panel variable ❶ is matched by the search engine in a separate source file, ButtonUsage.java. The second starred statement is a method invocation to Container.add(Component), located at ☆. This statement is generalized to a code query that matches all invocations to methods with the signature Container.add(...). This query matches the statement marked by ② in ButtonUsage.java.

In addition to the code query, the search engine also requires a *search scope*. The search scope defines the types and methods that may be returned by the search engine. StarCharter currently restricts the search scope to the current working Eclipse project and all associated libraries.

Sample search results for a code query are shown in Figure 3. Each matched statement is represented as a row in the table. The compilation unit, method and location of each usage is

	Compilation Unit	Method	Line
1	HelloWorld1.java	constructor	10
2	HelloWorld2.java	constructor	7
3	HelloWorld3.java	init	14
4	HelloWorld3.java	paint	20

Figure 3: Sample search results for a query

```
void foo() {
  A a = new A("Arr!");
  B c = bar(a);
}
```

```
B bar(A a) {
  new C();
  return new B(a);
}
```

Figure 4: A Java program with two methods

recorded by the search engine. It is possible for multiple usages from the same source file to match the same query such as the two results in HelloWorld3.java.

Each of the matched statements represent a usage that exhibits behaviour similar to the starred statements. However, the matched statements may not contain all relevant statements. As a result, StarCharter uses a technique called slicing to construct a more complete example usage.

2.3 Program Slicing

Program slicing is a technique introduced by Weiser for reducing a program to only the relevant statements needed to implement a given *slicing criterion* at a specific point of interest [Wei81]. The slicing criterion consists of a set of variables whose behaviour is to be maintained in the resulting slice. Thus providing a single variable x as the slicing criterion, program slicing will remove statements that do not affect or are not affected by variable x. Given all of a program's variables as the slicing criterion, the entire program will be return as a slice.

Slicing uses a combination of data-flow and control-flow analysis to find dependent statements. The results of the analyses are stored in a structure called a program dependence graph. A *program dependence graph* (PDG) is a structure where the nodes represent program statements and edges represent data or control dependence. A statement x is control dependent on y if the execution of y affects the execution of x. Similarly, a statement x is data dependent on y if the execution of y affects the value of x.

There are two common forms of slicing: forward and backward slicing. A *forward slice* contains all the statements which may have some effect on the slicing criterion. On the other hand, a *backward slice* contains all the statements which are affected by the slicing criterion. Combining the statements from both forward and backward slices forms a *complete slice*.

Slicing on PDGs alone enables only intraprocedural analysis. The system dependence graph (SDG) extends the PDG to enable interprocedural analysis [HRB90]. In a SDG, each method is

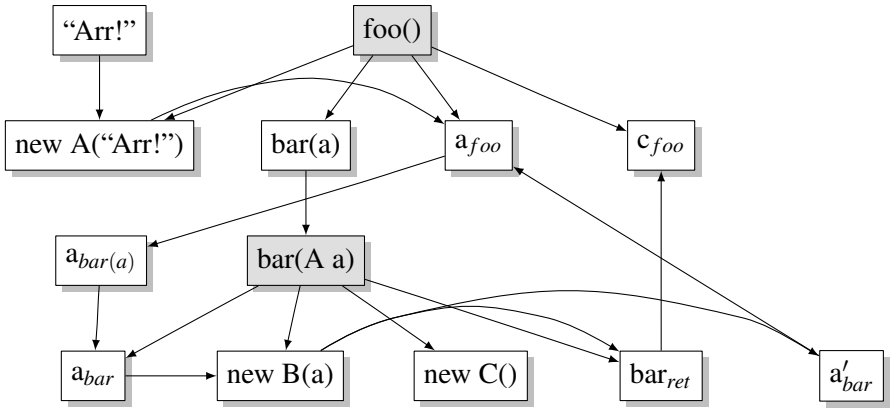


Figure 5: System dependence graph showing data dependence

represented by its own PDG. Method parameters are represented as *formal-in* and *formal-out* nodes and call arguments are represented as *actual-out* nodes. Method invocations are represented by edges connecting the arguments at the call site, termed the *actual-out* nodes, to the input parameters of the target method, called the *formal-in* nodes.

To demonstrate SDGs, we first present a small example Java program in Figure 4. This program has two methods, foo and bar and involves two types, A and B. Figure 5 presents the SDG constructed from this simple program. Only data dependences are present in this SDG. The gray nodes represent method declarations and white nodes represent statements. Variable have a subscript denoting their context. For example, a_{foo} refers to variable *a* in the foo method while $a_{bar(a)}$ refers to variable *a* at the time of the bar(a) method invocation. Variables suffixed with a prime such as a'_{bar} represent the variable value at the end of the method execution. A special variable suffixed with *ret* represents the return value of a method. Edges represent data dependences between two nodes such that the source *affects* the target node. For example, the literal "Arr!" affects the constructor invocation new A("Arr!"). The return value of method bar, denoted by bar_{ret} affects the value of variable *c* in foo, denoted by c_{foo}.

Forward and backward slicing is accomplished using graph reachability algorithms on the SDG. Dependencies that are indirectly related to the slicing criterion are found due to the transitive nature of the reachability algorithms. Calculating reachability on the SDG in Figure 5 will result in a forward slice. Reversing the edges and calculating reachability will result in a backward slice.

As a simple example of slicing, the return value of bar represented by bar_{ret} reaches only the variable c_{foo}. As a result, the forward slice contains only two elements: bar_{ret} and c_{foo}. A forward slice on the variable a_{foo} would reveal the following statements: a_{foo}, $a_{bar(a)}$, a_{bar}, new B(a), a'_{bar}. In both these cases, unrelated statements are removed by slicing.

StarCharter uses a custom-built, interprocedural, flow-insensitive program slicer. The slicer implements the SDG, thus enabling interprocedural analysis. Dependence graphs for methods are constructed only when a method invocation exists in the same method as the input usage. Control dependence has not been implemented in the current version.

```
▲ 1  ImageIcon icon = new ImageIcon("middle.gif", "...");
  2  JLabel l1 = new JLabel("...", icon, JLabel.CENTER);
  3  JLabel l2 = new JLabel("Text−Only␣Label");
★ 4  JLabel l3 = new JLabel(icon);
▼ 5  l3.setVerticalTextPosition(JLabel.BOTTOM);
▼ 6  l3.setHorizontalTextPosition(JLabel.CENTER);
  7  add(l1);
  8  add(l2);
▼ 9  add(l3);
```

Figure 6: Forward and backward slice

2.4 Concept Slicing

StarCharter uses *concept slicing* to locate related framework usage statements. Concept slicing is a specialized version of program slicing that is geared towards isolating the framework usage code of a concept. A *framework boundary* specification is provided to restrict the slicer to only statements relating to the framework of interest.

Program slicing finds related statements given a slicing criterion of variables. However, input into the slicer in StarCharter is simply a matched statement that can either be a method invocation or a variable assignment. Consequently, StarCharter translates these statements into slicing criterions.

A matched method invocation translates into a slicing criterion of the target and all parameter variables. For example, the slicing criterion for a call foo.bar(x,y,z) will consist of the variables: foo, x, y and z. A matched variable assignment directly translates to a slicing criterion consisting solely on the assigned variable. Since StarCharter is currently flow-insensitive, there is no need to specify a point of interest (ie. line number) at which slicing should begin.

Figure 6 illustrates the slicing procedure on a code snippet. Assume that line 4 is a matched statement given to the concept slicer. Since this statement is an assignment, StarCharter uses its assigned variable, l3 as the slicing criterion. The statements present in the forward slice are shown with the ▼ symbol. The statements in the backward slice are shown with the ▲ symbol. The union of forward and backward slice constitutes the complete slice. Note that statements related to the l3 variable are part of the slice while statements involving solely l1 and l2 are excluded.

With program slicing, a slice captures all statements that affect the variables in the slicing criterion. However, it is often the case that the slice is too general and contains too many statements to be useful for understanding framework usage. Take the example in Figure 7. This example adds a List concept in lines 5–6. Applying forward and backward slicing on the label variable at line 2 would find that all statements excluding line 5 are related (shown by the ▲, ▼ and ▼ symbols).

StarCharter uses a *framework boundary* specification to restrict slicing to only statements related to a framework of interest. The framework boundary consists of a set of Java type and package names. Take Figure 7 once again. This time around, assume that the user is interested in the usage of the Java Swing toolkit. Thus the user sets the framework boundary to javax.swing.*. Slicing on

```
▲ 1  ImageIcon icon = new ImageIcon("middle.gif", "...");
★ 2  JLabel label = new JLabel(icon);
▼ 3  label.setVerticalTextPosition(JLabel.BOTTOM);
▼ 4  label.setHorizontalTextPosition(JLabel.CENTER);
  5  List<JLabel> items = new ArrayList<JLabel>();
▼ 6  items.add(label);
▼ 7  add(label);
```

Framework boundary: javax.swing.*

Figure 7: Filtering statements using a framework boundary

this example would filter the statement at line 6 since the type of the items variable belong in the java.util package.

A feature of StarCharter is that it does not expand the slicing criterion as it encounters new variables. As a result, the resulting slice may not be executable. This behaviour avoids the case where slicing causes an explosion of dependent statements where the entire program is added to the slice. In Figure 7, StarCharter did not slice on the usage of the items variable even though it sliced the related items.add(label) statement. Consequently, the items variable was left uninitialized. Variables that are not initialized either through a method or constructor invocation are considered *free variables*. A free variable is considered a separate concept and a second StarCharter invocation may be used to mine their usage. Whether this behaviour is too restrictive for constructing useful recipes remains an open question.

The experience so far with StarCharter has been that the slices have been relatively contained. This may be due to the fact that the slicing criterion is not expanded during the search or to the single starred statement restriction in the current prototype. It is also unclear whether the framework boundary specification is sufficient for limiting cases where program slicing does yields too large a slice.

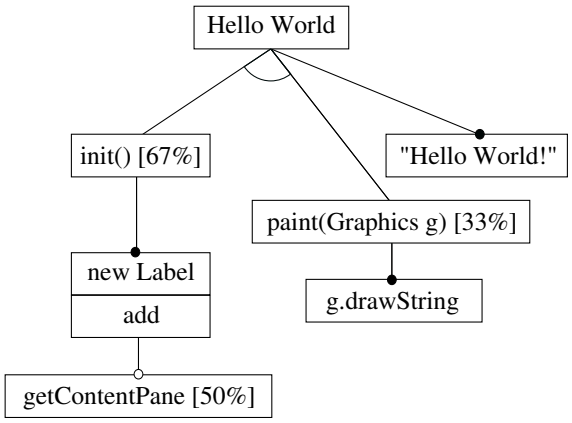
2.5 Synthesizing the Recipe

The final component of StarCharter is the recipe miner. The *recipe miner* is responsible for synthesizing a single recipe from a set of usage slices. Each individual slice provides a narrow view of the concept's implementation by describing only a single example usage. Each slice may contain superfluous or erroneous statements. StarCharter applies data mining to distinguish between common and uncommon statements. Common statements generally reflect statements that are required by the concept's implementation. Uncommon statements are typically application-specific and are optional or unrelated statements. The recipe miner extracts information by comparing the usage in all slices and presents their patterns and associations in a model called a concept recipe.

StarCharter uses *feature model mining* to synthesize a recipe in the form of a *probabilistic feature model* [She08]. A probabilistic feature model (PFM) consists of a *feature hierarchy* and an optional set of *hard and soft constraints* [CSW08]. A feature model can be instantiated as a set of selected features called a *configuration*. The feature hierarchy is a tree of features where each feature may

Slice	Location	Statement
S_1	paint(Graphics)	drawString(...)
S_1	paint(Graphics)	"Hello World!"
S_2	init()	add(Component)
S_2	init()	**new** JLabel
S_2	init()	getContentPane()
S_2	init()	"Hello World!"
S_3	init()	**new** Label
S_3	init()	add(Component)
S_3	init()	"Hello World!"

(a) Sample set of slices



(b) Mined recipe

Figure 8: Mining a recipe from a sample set of slices

have one or more child features called subfeatures. Features may either be solitary or grouped. A solitary feature is either optional, meaning that the feature may or may not be present or mandatory, meaning that it must be present. A grouped feature is a subfeature of either an OR-group (inclusive-or) or an XOR-group (exclusive-or).

A PFM may also have an additional set of hard and soft constraints. A hard constraint is one that must be satisfied in all legal configurations. On the contrary, a soft constraint describes a suggestion that may or may not be followed.

Feature model mining uses association rule mining to derive constraints. *Association rules* are implication-like expressions with an associated *support* and *confidence* [AIS93]. Support is a measure of statistical significance and confidence is a measure of rule strength. We mainly ignore support since our sample sets tend to be small. Confidence can be interpreted as a conditional probability of the consequent (right-hand side) given the antecedent (left-hand side) is present.

Feature model mining is demonstrated in Figure 8. First, a sample set of slices is shown in Figure 8a. Each row represents a statement in a slice. Applying the feature model mining algorithm on this sample set would synthesize the feature model shown in Figure 8b. The root feature, Hello World represents the user's concept of interest. Edges denote subfeature relations. Solid circles denote mandatory features. Empty circles denote optional features. The empty arc between init and paint(Graphics g) denotes an XOR-group. The attached features (add and new Label) represent an AND-group such that both features must be selected or neither. The percentages beside certain statements represent soft constraints. These are equivalent to their conditional probability of being selected given that their parent statement is selected. For example, getContentPane is selected 50% percent of the time, given that add and new Label are present.

The mined association rules and soft constraints represent the patterns that were discovered in the sample set of slices. It may be that all the slices were incomplete and a statement was missing from the set. It may also be the case that a superfluous statement was present in all slices and thus appears as a mandatory statement. As a result, the mined concept recipe should be considered a rough guide or be subsequently verified by a framework expert.

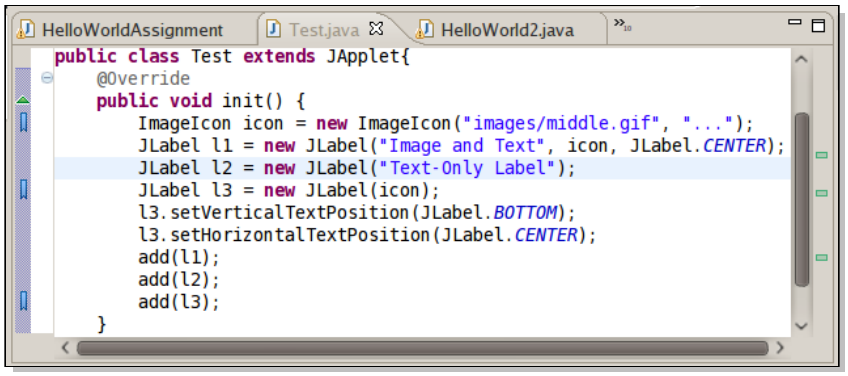


Figure 9: StarCharter editor markers

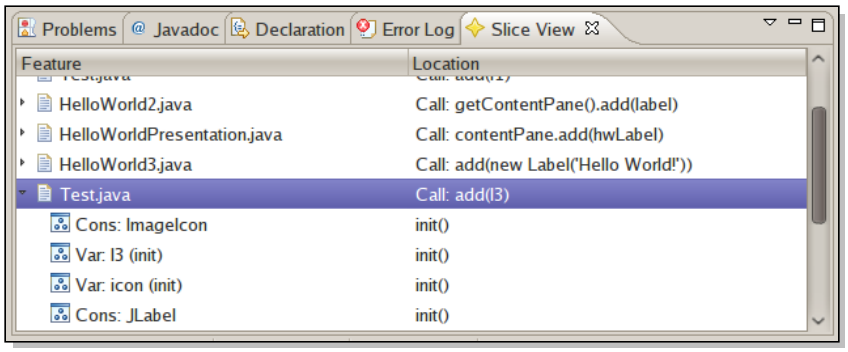


Figure 10: StarCharter Slice View

3 Implementation

StarCharter is implemented as an Eclipse plug-in. StarCharter currently contributes a Slice View and marker generation to the Eclipse workbench.

The current StarCharter prototype supports only selecting a single starred statement as the query. StarCharter is activated in the text editor either by: (a) highlighting a statement and right-clicking, (b) right-clicking with no selection, or by (c) right-clicking on the left ruler. In the first case, highlighting a statement will *star* only the statement covered by the highlight. When there is no selection, StarCharter assumes the user is interested in the entire line and star any statement in that line. Right-clicking on the left ruler behaves in the same manner as right-clicking without a selection. In future versions, the left ruler would be used to place stars on the statements, enabling the user to star multiple statements as described earlier.

A screenshot of the Eclipse editor with StarCharter is shown in Figure 9. Here, StarCharter was executed on the starred statement `add(l3)`. The left ruler shows markers identifying the location of related statements found by the concept slicer in the currently open source file.

Figure 10 shows a screenshot of the Slice View. The Slice View shows a list slices found using the StarCharter search engine sorted by source file and matched statement. Within each slice is a list of related statements found in the forward and backward slice. Clicking on a slice will update the

markers in the text editor to reflect the selected slice. For example, the slice that is currently selected shows the markers in Figure 9. Clicking on another slice in Test.java will update the markers to show the statements in the selected slice.

Double-clicking on a slice will open the selected source file and update the editor markers to reflect the selected slice. Double-clicking a statement will highlight the selected statement in the editor window. Note that there is currently a bug preventing users from opening types and fields. A screencast illustrating the current StarCharter prototype is available online[2].

The search engine in StarCharter is implemented using the Eclipse JDT Search Engine (JDTSE). JDTSE provides the search capability that is invoked when using the semantic Java Search in Eclipse. The heuristics for inferring queries described in Section 2.2 is implemented by analyzing method signatures using JDT and the dependence graph constructed by the concept slicer.

The concept slicer component uses the Eclipse Java Development Tools (JDT) [Ecl08a] to parse and construct an abstract syntax tree (AST) from a Java source file. A custom-built dependence graph generator is used in StarCharter. StarCharter output a set of GraphViz files for each generated program dependence graph under the `/output` directory of the Eclipse project that StarCharter was executed in. The slicer generates a system dependence graph for each slice.

The concept slicer is currently interprocedural and flow-insensitive. Loops are treated as if they are executed once and method invocations are handled using the system dependence graph. Recursion is handled by processing only method invocations up to a maximum specified depth. Exceptions and break statements are also not handled in the current implementation.

The integrated between the mining algorithm and StarCharter has been disabled in the prototype due to some critical bugs. The feature model mining algorithm as described in Section 2.5 has been fully implemented using the LCM algorithm by Uno et al. [UKA04] and the Eclipse Modeling Framework [Ecl08b] as its data model.

4 Related Work

Prospector. Prospector [MXBK05] is a tool for finding *jungloids*, which are sequences of field or method and constructor invocations that derive a specified *output type* from an *input type*. Prospector gathers the majority of its jungloids from the framework API. StarCharter is fundamentally different from Prospector in that it analyzes example usages to derive the slices. However, StarCharter may be augmented with framework API information in the future. Prospector does employ data mining to retrieve examples of downcasts. StarCharter does not support these statements at the moment. One disadvantage of jungloids are that they are restricted to a single variable input. This restriction is not present when using SDGs since multiple paths can converge on the same node.

PARSEWeb. PARSEWeb [TX07] is a tool for finding method invocation sequences by searching over code found through an internet code search engine. PARSEWeb and StarCharter share a similarity in the search engine component. However, StarCharter does not use *input type* $\rightarrow$ *output type* queries like PARSEWeb. Regarding program analysis, StarCharter uses well established program slicing techniques for finding related statements. In addition, StarCharter outputs a single concept recipe while PARSEWeb presents each individual method sequences.

[2] `http://gsd.uwaterloo.ca/~shshe/starcharter.html  user: static  password: analysis`

5 Future Work

StarCharter is still a work in progress. An evaluation remains to be done to determine the example usages that StarCharter is capable of mining. Another issue relates to the use of feature models for representing concept recipes. Whether this is appropriate for the future requires further evaluation. Below are some ideas for short term extensions for StarCharter:

Downcasts. StarCharter currently does not allow typecast statements to be starred. Prospector [MXBK05] mined for downcasts by mining on a corpus of examples. This approach may be integrated into StarCharter in the future.

White-box frameworks. Using a white-box framework involves extending framework classes and overriding certain callback methods. This form of framework usages requires knowledge of the inner workings of the base class. The techniques in StarCharter can be very useful for mining white-box framework usage.

More precise slicing. StarCharter uses a custom-built slicer based on early program slicing research. Since then, there has been substantial research in slicing. A technique for slicing object-oriented systems was presented by Liang and Harrold [LH98]. Hammer and Snelting developed a Java-specific slicer [HS04].

Improved search heuristics. The current heuristics for inferring queries from starred statements can be improved. For example, StarCharter currently search for all method calls contained in the same class as the starred statement for method invocations. This can be generalized such that superclasses are included in the search as well.

6 Conclusions

This report has introduced StarCharter, an light-weight and interactive tool for synthesizing concept recipes from a set of starred statements. Using StarCharter, a user would begin constructing a recipe by starring a set of statements. A search engine locates other usages and a concept slicer is used to find related statements. Finally, a recipe miner is applied to synthesize a single recipe describing the concept's framework usage. The tool has been implemented as an Eclipse plug-in. StarCharter is tool that is useful for framework understanding, framework modeling and as an interactive development aide.

References

[ABC07] M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki, "Automatic extraction of framework-specific models from framework-based application code," in *ASE*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 214–223.

[AIS93] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD '93*. New York, NY, USA: ACM, 1993, pp. 207–216.

[CSW08] K. Czarnecki, S. She, and A. Wąsowski, "Sample spaces and feature models: There and back again," in *SPLC '08*. IEEE, Sep. 2008, to appear.

[Ecl08a] "Eclipse java development tools (JDT) subproject," Eclipse Foundation, 2008. [Online]. Available: http://www.eclipse.org/jdt/

[Ecl08b] "Eclipse Modeling Framework," Eclipse Foundation, 2008. [Online]. Available: http://www.eclipse.org/modeling/emf/

[HBC07] A. Heydarnoori, T. T. Bartolomei, and K. Czarnecki, "Comprehending Object-Oriented Software Frameworks API Through Dynamic Analysis," School of Computer Science, University of Waterloo, Technical Report CS-2007-18, October 2007.

[HRB90] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[HS04] C. Hammer and G. Snelting, "An improved slicer for java," in *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2004, pp. 17–22.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin, "Aspect-oriented programming." Springer-Verlag, 1997, pp. 220–242.

[LH98] D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs," in *ICSM '98: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1998, p. 358.

[MXBK05] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," *SIGPLAN Not.*, vol. 40, no. 6, pp. 48–61, 2005.

[She08] S. She, "Feature model mining," Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2008.

[TX07] S. Thummalapenta and T. Xie, "PARSEWeb: a programmer assistant for reusing open source code on the web," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 204–213.

[UKA04] T. Uno, M. Kiyomi, and H. Arimura, "LCM ver. 2: Efficient mining algorithms for frequent/-closed/maximal itemsets," in *FIMI*, 2004.

[Wei81] M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.