CREATING APPLICATION PROGRAMMING INTERFACE CODE TEMPLATES FROM USAGE PATTERNS

by

Tristan Joseph Ratchford

School of Computer Science
McGill University, Montreal, Quebec

October 2011

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Application programing interfaces promote reuse by facilitating interaction between software components and/or software libraries. API code templates are parameterized API scenarios that can be quickly instantiated by copy-and-pasting or through support from integrated development environments. They provide the skeletal structure of an API coding scenario and let developers simply “fill in the blanks” with the details of their coding task. Unfortunately, creating relevant API code templates requires time and experience with the API. To address these problems we present a technique that mines API usage patterns and transforms them into API code templates. Our intuition is that API usage patterns are a solid basis for code templates because they are grounded by actual API usage. We evaluate our approach performing retroactive study on the Mammoth, ArgoUML, and Eclipse projects to see if API code templates created from earlier versions could have been helpful to developers in later versions. Our results show that, on average, each API code template our technique mined could have helped developers with creating six, nine, and twelve *new* methods in Mammoth, ArgoUML, and Eclipse, respectively. In our evaluation, we mined many API code templates from the three test projects that provide evidence that our technique could have helped developers learn and use an API faster in many opportunities.

Résumé

Les interfaces de programmation (API) encouragent la réutilisation de code en facilitant l'interaction entre les composantes du logiciel et ses librairies. Les «templates» d'API sont des scénarios d'utilisation de l'API paramétrées pour êtres rapidement instanciés par copier-coller ou par le soutien intégré des environnements de développement. Ils fournissent le squelette d'un scénario d'utilisation de l'API et laissent au développeurs la simple tâche de «remplir les espace». Malheureusement, afin de créer des «templates» pertinents, du temps et de l'expérience avec l'API sont nécessaires. Pour résoudre ces problèmes, nous présentons une technique par laquelle les «templates» d'API sont découverts en analysant des scénarios d'utilisation existants. Notre intuition est que de tels scénarios sont une base valide pour la découverte de «templates» car ils sont une représentation existante de l'API en action. Nous évaluons notre méthode en effectuant une étude rétroactive sur les projets Mammoth, ArgoUML, et Eclipse pour voir si les modèles créés à partir de versions antérieures auraient été utiles aux développeurs dans les versions ultérieures. Nos résultats illustrent que, en moyenne, chaque «template» d'API créé par notre technique aurait permis aux développeurs de créer six, neuf et douze nouvelles méthodes dans les projets Mammoth, ArgoUML, et Eclipse, respectivement. De plus, dans notre évaluation, de nombreux modèles de code API ont été créé à partir des ces trois projets, ainsi prouvant que notre technique pourrait avoir aidée les développeurs à apprendre et utiliser une API plus rapidement dans de nombreuses occasions.

Acknowledgements

I would like to personally thank my supervisor Martin Robillard for giving me my chance and shaping me into the researcher and I am today. Thank you to my colleague and friend David Kawrykow. Thank you Barthélémy Dagenais, Annie Ying, and Ekwa Duala-Ekoko, who I also consider as friends, for all the support and guidance over the years. Further thanks to Bart and Annie for their comments and edits. Thank you Laurie Hendren for helping me create APUX and Philippe Fournier-Viger for the SPMF framework. Thank you Yannick Thiel for the translation services. Special thanks to my family for their caring and support. And most importantly, I would like to dedicate my thesis to Rachel Round for all the support and love during my highest and lowest points.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

An application programming interface (API) serves as an interface between different software components and facilitates interaction. When a project or method uses an API, it is subscribing to the services the API provides. Another way to refer to this relationship is to say the project is a *client* of the API.

API code templates are parameterized API scenarios that outline a protocol of necessary object instantiations and method calls needed to implement functionality with an API in a client method or project. They provide the skeletal structure and let developers customize them with details related to their own task. To illustrate the concept of API code templates we present Figure 1.1; a code template documenting how to create a URL hyperlink using the `org.eclipse.swt` API. All method calls and types in boldface are the invariant points—the rest is up to the developer to customize, for example by changing the URL or tooltip text. Figure 1.1 is an example of a real-world code template. An almost identical code snippet was posted on the StackOverflow forums in response to the question "How can I make a hyperlink in a jFace Dialog [which also uses the `org.eclipse.swt` API] that when clicked opens the link in the default web browser".[1]

The benefits API code templates provide are two-fold: They free developers from remembering boilerplate code and also allow them to reuse blocks of code quickly into their program by copy-and-pasting. Some integrated development environments (IDEs), such as Eclipse or Netbeans, even have code completion support for code templates which allow a

[1]http://stackoverflow.com/questions/3968620/how-can-i-add-a-hyperlink-to-a-swt-jface-dialog

```
//Parent is an SWT composite
Link myLink = new Link(parent, SWT.NONE);
myLink.setText(‘‘http://www.cs.mcgill.ca/~tratch’’);
myLink.addSelectionListener(new SelectionAdapter()
{//Anonymous class implementation});
mylink.setToolTipText(‘‘Click here to visit my page’’)
GridData lCenter = new GridData(SWT.Center);
myLink.setLayoutData(lCenter);
```

Figure 1.1: Example API Code Template: Creating a Link in SWT

user to type the first few letters of a template (e.g. typing *while*) and activate a command (e.g. by pressing `ctrl` and `space`) to automatically reuse the template into their code. If the user is using Eclipse, they can also press `tab` to cycle through the variation points and customize the template for their own purposes.

API code templates also make learning and using an API easier for developers. For example, Figure 1.1 includes some non-obvious API method calls which may be overlooked by developers who only study the API's reference documentations (such as JavaDocs). For instance, to centre the text a developer must create a `GridData` object and pass it as a parameter to `setLayoutData()` to align the text of the link. In addition, it may not be clear to novice programmers that an `EventListener` needs to be added to the `Link` object to intercept `SelectionEvents`. Therefore, it is common practise for developers of APIs to include code templates along with their documentation to help. Apache Lucene, a text search engine API, includes a functional "hello world" web application template, albeit large, which allows users to immediately customize the web content and tweak their configuration as needed.[2] It is also not uncommon for users of APIs to supplement templates included by API producers by creating their own. For example, `http://code.activestate.com/` is a website that contains over 3700 code "recipes" created by developers for Python APIs. Some of the API recipes include converting a JPEG to a PDF, downloading a YouTube video, and using RSA encryption. The underlying theme is that

[2]`http://lucene.apache.org/java/2_4_0/demo3.html`

documenting API scenarios as code templates is a useful endeavour for API producers and consumers alike.

Unfortunately, creating API code templates requires a time investment and experience with the API—sometimes, the API documenters are not even the ones who coded it, as is the case with some open-source project documentation teams [4]. Luckily for the developer who posed the question to StackOverflow, another developer was able to share his or her knowledge of the `org.eclipse.swt` API. However, many forum posts go unanswered, and documenting such API scenarios can be lengthy. In addition to the time burden, it can be hard to gauge whether an API code template will be relevant to other developers.

Recently, there have been a number of approaches that look for *API usage patterns* (common ways in which developers use APIs, as demonstrated by client code). Our intuition is that API usage patterns represent practical API usage scenarios because they are inferred from actual programs that use APIs. Based on this intuition, inferring API code templates from API usage patterns will likely be useful to developers because they will be relevant, practical, and correct. Thus, we present a technique for mining API usage patterns and transforming them into code templates.

In fact, the code template in Figure 1.1 was actually found in eight different methods in the Eclipse IDE (version 3.1) and inferred using our technique. The fact that Figure 1.1 is derived from eight different methods and is almost identical to the StackOverflow solution is one piece of evidence to show that developers can indeed use APIs similarly. It also that API code templates inferred from API usage patterns practical, relevant and useful to developers learning an API because Figure 1.1 reveals non-obvious method call sequences.

We implemented our technique in a tool called Maui. Maui works by analyzing one or more projects that use an API and transforming individual instances of API usage into regular expressions. We then use machine learning to discover patterns among the regular expressions and transform the patterns into API code templates. We evaluate Maui by performing a retrospective study to see if API code templates created from earlier versions of three large Java projects could have helped developers in later versions. Our results yield some interesting API code templates and show evidence that our technique could have been useful for developers in these three projects. On average, each API code template which Maui mined could have helped developers create six, nine, and twelve *new* methods in three different Java projects (Mammoth, ArgoUML, and Eclipse, respectively). Considering that we mined dozens of templates in ArgoUML and hundreds in Eclipse, there were a multitude of opportunities in which Maui could have helped developers save time using or learning an API. The contributions of our approach are: A novel technique for mining API usage patterns from multiple projects that include control flow, multiplicity of API method calls, and interaction between multiple APIs; and a technique for transforming these API usage patterns into API code templates in the Eclipse IDE.

We present our approach in Chapter 2 by first introducing our main unit of analysis, the *regular expression*, in Section 2.2. In the remaining sections of Chapter 2 we present the components of our approach that find API usage patterns and transform them into API code templates: The API Usage Extraction Phase (APUX) (Section 2.3), the Pattern Mining Phase (Section 2.4), and the Code Template Creation Phase (Section 2.5). In Chapter 3, we describe the methodology of our evaluation (Section 3.1) and present our results in Section 3.2. Finally, we present related works in Chapter 4 and our conclusions in Chapter 5.

Chapter 2
Creating API Code Templates from API Usage Patterns

2.1 Overview

Our approach consists of three primary steps: First, abstract how a project uses an API, second, search for patterns, and third, transform the patterns into API code templates. The data passed between each of these steps are *regular expressions*. A regular expression, or Regex for short, is an expression that describes a set of strings. For example, `([0-9][0-9][0-9]-[0-9][0-9][0-9][0-9])`, where `[0-9]` could be any digit between zero and nine inclusively, is a regular expression describing the set of all possible telephone numbers (ignoring area codes). With our technique, we create a Regex for each client method in a given client project by mapping each API call the client method makes to a unique symbol. The result is a string of symbols (API calls) that describe how a particular client method uses an API or APIs. In other words, our technique represents instances of API usage in a client project as Regexes.

Our technique then looks for patterns across all Regexes using a pattern mining algorithm. Patterns supported by enough Regexes are transformed into code templates. The templates our technique yields may be mined from one or more projects and include method calls from one or more APIs. Furthermore, the order and multiplicity of the API method calls are preserved along with any control flow structures (i.e. `if-else` blocks or loops).

We implement our technique as an Eclipse plug-in, called Maui for the Java programming language. We explain our approach in the remainder of the chapter by explaining Maui's Regex data structure in Section 2.2, and then detail each phase Maui performs to create API code templates.

2.2 The Regex Data Structure

Maui abstracts how a client method uses an API or multiple APIs by creating a Regex that describes all possible execution sequences of API calls which the client method could make. Sometimes only a single sequence of API calls describes a method. For example, in Figure 2.1 the client method `getSin()`'s usage of the `java.lang.Math` API can only be described by the execution sequence $(Math.sin(), Math.round())$ because if `getSin()` is invoked, both methods are guaranteed to be called exactly once and according to the order of execution.

```
double getSin(double pAngle)
{
     double x = Math.sin(pAngle);
     x = Math.round(x);
     return x;
}
```

$$Math.sin()Math.round()$$

Source Code Regex Form

Figure 2.1: Example of individual API calls

Maui considers the following program locations as API method calls: A super constructor call when the super class is provided by a third-party library or framework; a method call when the declared class of the method is provided by a third-party library or framework; and a constructor call when the declared class is provided by a third-party library or framework. A method in a client project that does not make any API calls is not a client method. For example, a setter that assigns a field does not contain any API calls. Strictly speaking, non-client methods are represented by the empty set of execution sequences, but our approach ignores this case altogether.

If an API call is repeated consecutively, for example in Figure 2.2, a plus is used to indicate that it is called one or more times. To be succinct in our diagrams, we represent method calls as individual letters. Folding consecutive identical calls into a single letter and indicating this case with a plus makes pattern matching much easier because we observed during our early experimentation with the approach that consecutive identical calls were rarely called with the same multiplicity. Maui allows users to turn folding consecutive identical calls on or off, at their discretion, but they cannot specify different folding sizes.

```
clientMethod()
{
     a();
     b();
     b();
     b();
     c();
     b();
}
```

$$ab^+cb$$

Source Code Regex Form

Figure 2.2: Example of consecutive identical calls representation

Client methods with control flow (i.e. `if-else` blocks and loops) can result in multiple and/or mutually exclusive possible sequences of API calls; Maui uses special operators and braces to denote these cases. The motivation for representing control flow in Regexes is that we want to be able to capture API coding scenarios that are more complex than straight sequences of API calls.

Representing `if-else` blocks

With `if-else` constructs, the execution of an `if` block is mutually exclusive from the execution of an `else` block. Maui represents this behaviour by wrapping the entire `if-else` block in curly braces and separating the `if` block from the `else` block using the || symbol. The || symbol indicates that either the `if` block or the `else` block will be executed exactly once. Figure 2.3 demonstrates the `if-else` notation by showing that the API call `a()` is called before either `b()` or `c()`, before `d()` is finally called.

```
clientMethod(int x)
{
     a();
     if(x == 2)
     {
       b();
     }
     else
     {
       c();
     }
     d();
}
```

$a\{b\|c\}d$

Source Code

Regex Form

Figure 2.3: Example of a Condition Letter without a body

Maui does not analyze the boolean condition of an `if-else` block, therefore Maui will always model `if` and `else` blocks as mutually exclusive even if the condition causes one to always be executed. Maui also supports nested `if-else` blocks and handles `switch` statements by converting them into chains of nested `if-else` blocks, as they would appear in bytecode.

Maui represents `if-else` constructs that do not contain API calls as an empty string. However, `if-else` blocks that contain only a single API call in either the `if` block or the `else` block, but not both are still represented. For example, if Figure 2.3 omitted the call to `c()`, the resulting representation would be $a\{b\|\}d$.

Representing Loops

Maui represents loops in Regexes by wrapping the loop body between two square brackets and annotating it with a * symbol to indicate that the body will be executed 0 or more times.[1] Figure 2.4 is an example how Maui represents a loop in a Regex. In Figure 2.4,

[1]The * symbol is also known as a Kleene closure.

`a()` and `d()` are called exactly once, whereas `b()` and `c()` may be called zero or more times. To reiterate, since Maui does not inspect boolean conditions, it is possible that Maui could misrepresent a loop body as executing zero or more times, even though it could unconditionally execute zero times or unconditionally execute more than zero times. Maui handles `do-while` loops by unrolling the first iteration of the loop body, showing that it will be executed exactly once. For example, if Figure 2.4 was instead a `do-while` loop, the resulting Regex would then be $abc[bc]^*d$.[2]

```
clientMethod(int x)
{
     a();
     while(x < 10)
     {
       b();
       c();
     }
     d();
}
```

$a[bc]^*d$

Source Code

Regex Form

Figure 2.4: Example of a Loop representation

Representing Control Flow Conditions

In addition to `if-else` blocks and loops, Maui also captures API calls made in the conditions of any control flow constructs. Maui represents them by wrapping the API calls in parentheses, separating them by commas, and prepending them to the control flow construct (e.g. $(hasNext())[next()]^*$). Maui does not retain the order of API calls or any boolean logic. Therefore, Maui considers $(hasNext() \&\& isFoo())[next()]^*$ and $(isFoo() || hasNext())[next()]^*)$ as identical. Also, control flow constructs that only contain API calls in their condition are represented with an empty body (e.g. $(hasMoreTokens())[]^*$).

The rationale for enclosing API calls found in condition statements with parentheses is to signify that they are different from regular API calls. For example, the Regex

[2]We could also represent $abc[bc]^*d$ as $a[bc]^+d$, but Maui currently does not have the functionality to represent such cases.

$(isStreamOpen())[foo()]^*$ could also be represented as $isStreamOpen()[foo()]^*$. However, since $isStreamOpen()$ would almost always be called in a conditional statement it seems unnatural to treat it as a regular API call. Further, in the case of multiple API calls in a condition statement, the execution of all calls is not guaranteed. For example, with the condition $(hasNext()\&\&isFoo())$, $isFoo()$ would not be evaluated if $hasNext()$ evaluates to false. Thus, representing them as regular API calls may cause inaccuracies in the Regex.

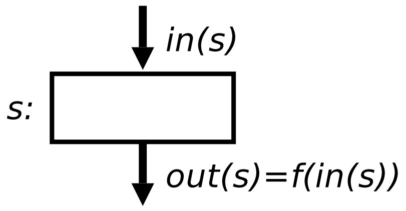
2.3 API Usage Extraction (APUX)

Maui creates a Regex for each client method in a client project in the API Usage Extraction phase, or APUX for short. APUX creates a Regex for a client method by traversing its control flow graph in order of execution and recording any API method calls found along the way in an ordered list. When APUX reaches the end of the client method (i.e. a return statement), the ordered list of API calls becomes the Regex for that client method. APUX is implemented as a custom forward flow static analysis using the Soot Java optimization framework [29].

A control flow graph (CFG) is a graph representation of all possible paths of execution through a program, or part of a program (e.g. a method). A node in a control flow graph is called a basic block and represents a code expression. Nodes are connected by directed edges called jump targets, which indicate the flow of execution. There are also two special nodes to denote the entry and exit of a control flow graph (e.g. the start and end of a method).

CFGs can be built from various representations such as Java source code to Java bytecode. Maui specifically traverses a 3-address intermediate representation of Java called Jimple. An intermediate representation is an abstract form of a programming language designed to aid the analysis of programs. A 3-address intermediate representation means that the program is broken down into the general form $result = operand_1, operator, operand_2$. For Maui, this means that every basic block will contain at most one method call. Further, since each basic block can only contain a single method call, nested expressions, such as nested method calls, are broken down into individual basic blocks and are connected based on the order of their execution. For example, `new Point(getX(), getY())` would be represented as $getX() \rightarrow getY() \rightarrow newPoint()$.[3] Each CFG Maui traverses represents the control flow of an individual client method.

APUX builds a Regex for a client method by sequentially visiting each basic block, in order of execution, and passing an ordered list of API calls between basic blocks. Figure 2.5 depicts how APUX flows through a statement s. A basic block s receives the ordered list of API calls as input from its predecessor q, which is denoted by the function $in(s)$. If s contains an API call l it appends the API call to the end of the list and passes it to its successor ($out(s)$). When APUX reaches the end of the CFG via a `return` statement the list of recorded API calls becomes the Regex for that client method.



$$in(s) = out(q) \quad | \quad q \text{ is the direct predecessor of } s$$

$$f(out(s)) = \begin{cases} in(s) & \text{if s does not contain an API call} \\ in(s) + l & \text{if s contains the API call } l. \end{cases}$$

Figure 2.5: Flow through a statement s

APUX is conservative in that it makes no initial assumptions about the API calls in a client method or in a given statement. Figure 2.6 states the Maui's starting approximations for a given statement s. The first line states that the Regex at the beginning of a client method is empty and the second states that APUX makes no assumptions about the $out(s)$ until we visit s. The $\perp$ symbol indicates that the initial output for any given statement s is unknown, until it is visited.

Figure 2.7 demonstrates how APUX collects API calls to build a Regex in a client method that does not contain any control flow. We show the Regex representation before and after each basic block Figure 2.7 in parentheses. The CFG for Figure 2.7 is a sequence of three method calls, two of which are API calls, a and c, and the other, b, is not. APUX

[3]This example is a simplification. In reality, there are a number of intermediate operations between each of the method calls `new Point(getX(), getY())`—for example, storing the values of `getX()` and `getY()`.

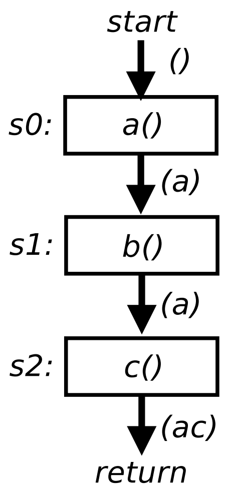
$$\begin{aligned} out(Entry) &= \top = () \\ out(s) &= \bot = \text{Unknown} \end{aligned}$$

Figure 2.6: Starting approximations

visits each basic block sequentially from the beginning, with the initial Regex representation being empty (i.e. $()$). Since $s0$ contains an API call, APUX appends a method call to the end of the Regex and passes it as input to $s1$. The basic block $s1$ however, does not contain an API call, so it will simply forward the Regex as is to $s2$, where APUX will append c. The resulting Regex for the client method in 2.7 is ac.

```
clientMethod()
{
   a(); //s0
   b(); //s1 (non-API)
   c(); //s2
}
```

Source Code



Control Flow Graph

Figure 2.7: An example of how APUX generates a Regex for a client method that has no control flow

2.3.1 Handling Control Flow

APUX takes special consideration for control flow constructs because they cause the CFG to diverge into multiple branches and rejoin later at some other basic block. The basic block where the paths rejoin is called a *merge point*. Since APUX builds Regexes by passing an ordered list of API calls between basic blocks, APUX needs to keep track of all branches and merge them appropriately to generate a single Regex representation for a client method. Henceforth, we will refer to `if` and `else` blocks as *branches*. Loops technically also contain branches, but we distinguish them from `if-else` blocks we continue to refer to them as loops.

The core of APUX's strategy for merging branches and loops (denoted by the $\bowtie$ operator) is based on the fact that up until a branch or loop head, where the CFG diverges, the Regex representation is the same. In other words, regardless of what happens when the CFG diverges, the Regex will still have the same *prefix*, up until that point. Based upon this fact, the general APUX strategy for handling control flow constructs is to store the Regex built so far (the prefix-Regex), and build new *interim-Regexes* for each branch or loop body. When the interim-Regexes meet at a merge point, APUX merges them and appends them to the stored prefix-Regex. Our strategy is recursive and allows APUX to generate Regexes in the face of nested control flow constructs.

There are two major challenges with this strategy: Keeping track of which prefix-Regexes belong to which branch or loop, and knowing which interim-Regexes to merge and when. To address this challenge, APUX uses the branch and loop heads as its main point of reference when keeping track of different paths. When APUX reaches a branch or loop head for the first time, it stores the Regex built so far (which we call the prefix-Regex) in a map called the Prefix-Map, using the loop head or branch head as the key. Also, when APUX creates the interim-Regexes, it tags them with the branch or loop head that they were created from, to ensure that it can correctly merge the branches. To better explain the merging process, we walk through how APUX handles branches and loops separately.

Branch Merging

An `if` statement in a client method will have two targets flowing from it in the CFG: One target will point to the first basic block in the `if` block and the other will point to the first basic block in the `else`. The two paths will converge at the first basic block following the `if-else` blocks. If the `if` statement does not have a corresponding `else` the second branch will also point to the first following basic block.

When APUX reaches a branch head it stores the Regex created up until that point (prefix-Regex) in the Prefix-Map. APUX then creates two empty interim-Regexes and passes them as input to the two starting basic blocks within the `if` and `else` blocks. Separately, both interim-Regexes will append any API calls in their respective branches to their interim-Regexes. When the two branches eventually meet, APUX merges the Regexes of each individual branch (the interim-Regexes) by enclosing them with braces and inserting a || symbol between them to create a single representation. APUX then appends the single representation to the Regex stored in the Prefix-Map and continues to traverse the CFG.

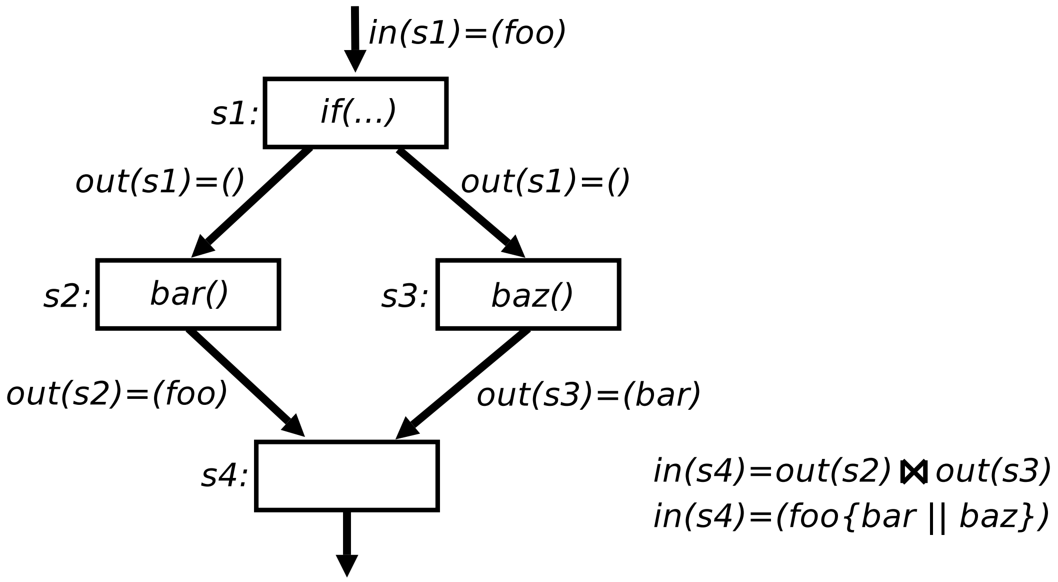


Figure 2.8: Typical branch merge scenario

Figure 2.8 demonstrates a typical branch merging situation of an `if-else` block. In this example, the branch head is $s1$ and the merge point is $s4$. Assume that method calls foo, bar, and baz are all Target API calls. Initially, the branch head $s1$ receives the Regex (foo) from its predecessor. APUX then stores the Regex (foo) in the Prefix-Map and uses the the branch head $s4$ as the key. Then APUX creates two empty Regexes, tags them with their origin $s4$, and passes them as input to the starting basic blocks in the `if` and `else` blocks. Finally, when the merge point $s4$ is reached, APUX recognizes that the branches come from the same head because they have been tagged with their origin $s4$. Since they both share the same origin, APUX merges them to produce the Regex $(foo\{bar\|baz\})$, which is fed into $s4$. In the code, $s4$ would be the first statement that directly follows the `if-else` block.

Loop Merging

With branches, the CFG diverges at a branch point and eventually rejoins at a later point. However, with loops the diverging and converging point is the same: the loop head. The difficulty with merging loops is knowing which control flow is outside the loop and which is inside. APUX differentiates control flows as internal or external to a loop by performing a domination analysis on each basic block. In brief, a unit that dominates itself is a loop statement. The domination analysis can also tell us which branch heads are loop heads by looking for the first statement to dominate itself.

Another difficulty is knowing *when* to merge branches because APUX visits the loop head at least twice; the first time being when it is first encountered in the CFG and the second time is from its loopback statement. APUX relies on the starting approximations, stated in Figure 2.6, to figure out when to merge. Recall that the initial Regex for any given basic block s is unknown (i.e. $out(s) = \perp$). APUX will merge paths converging on a loop heads when all converging paths contain a value (i.e. no paths output a $\perp$).

Figure 2.9 depicts the two times APUX encounters the loop head and the result of the merge operation.

The analysis first encounters $s1$ after processing $s4$. At this point, the $out(s4)$ is known, but $out(s3)$ is not known because APUX has not visited any of the loop body statements (indicated by the $\perp$ symbol). APUX then pushes an empty Regex down the loop body path to collect API calls. When APUX encounters the loop head for the second time, it now has information about both branches and can merge them to produce $in(s5)$. Loops are merged by creating a loop body Regex f and appending it to the prefix-Regex before the loop. In Figure 2.9, APUX appends $out(s3)$ with $out(s4)$ to get the resulting Regex $(foo[bar]^{*})$.

Handling `do-while` is simple because the CFG already unrolls the loop body once, whereas the rest of the iterations still appear as a regular loop.

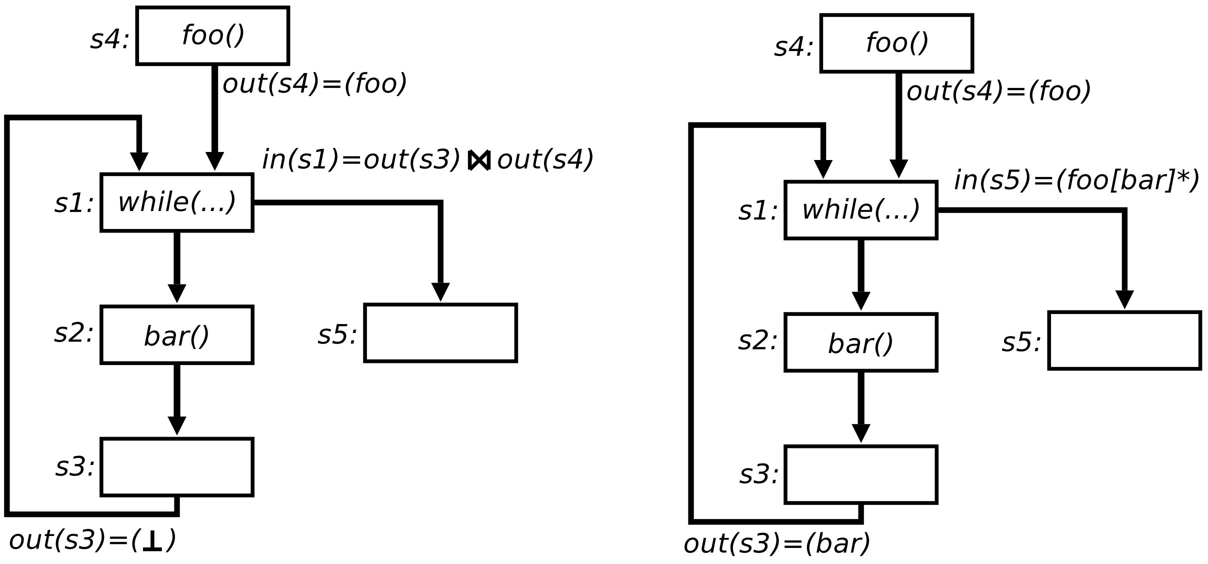


Figure 2.9: Typical loop merge scenario when encountering a loop head for the first (left) and second time (right)

Merging More than Two paths and Hybrid Merging

Nested `if-else` blocks and/or loops can result in more than two paths converging at a single point. Figure 2.10 shows an example of a CFG that results from four paths converging. The difficulty with merging paths created from nesting is knowing which paths to merge and getting the nesting right. For example, the $in(s)$ is four interim-Regexes. If APUX merges the wrong two branches, the result will be mismatching `if` and `else` blocks. APUX overcomes this mismatching problem by tagging each interim-Regex with the branch or loop head it originated from. So when more than two paths meet at a common point, APUX can pair them off by comparing their tags to merge them. Tagging also helps APUX nest `if-else` blocks and loops correctly because a higher level control flow path cannot be completed before a lower level path. To illustrate this point and demonstrate how APUX merges more than two paths we walk through the process occurring in Figure 2.10.

A CFG Figure 2.10 results from `if-else` blocks that both nest their own `if-else` blocks. When the APUX reaches $s1$, the CFG branches into two paths representing the `if` and `else` logic. APUX stores the Regex created thus far in the Prefix-Map, with $s1$ used as

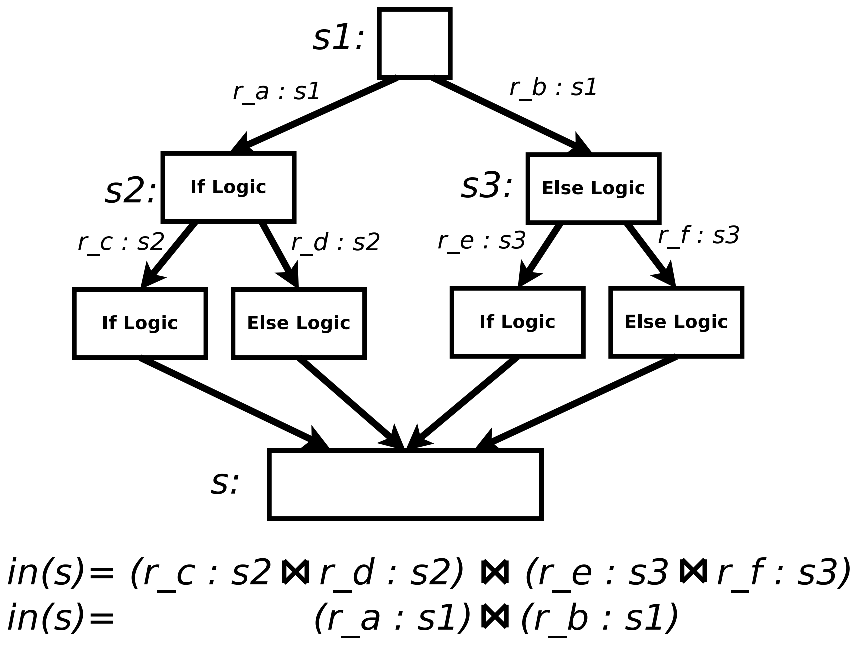


Figure 2.10: Mutli-branch merge scenario

the key, and creates two interim-Regexes labelled `r_a:s1` and `r_b:s1`. The labels `r_a` and `r_b` are unique identifiers to, and the common suffix `:s1` is the tag indicating these interim-Regexes originated from $s1$. The `r_a:s1` and `r_b:s1` interim-Regexes travel down their respective paths picking up API calls along the way. When `r_a:s1` reaches the nested branch head $s2$, the same process will occur with $s1$: `r_a:s1` will be stored in the Prefix-Map with $s2$ used as the key and two new interim-Regexes will be created with $s2$ tags, `r_c:s2` and `r_d:s2`. Similarly, when `r_b:s1` reaches $s3$, two new interim-Regexes are created, `r_e:s3` and `r_f:s3`. When all four interim-Regexes reach s APUX can finally merge them all. APUX takes any given interim-Regex and finds its mate by comparing their origin tags. Suppose APUX initially merges `r_e:s3` with `r_f:s3` to create $p = \{r\_e : s3 \| r\_f : s3\}$. This new branch letter p encapsulates all `if` and `else` logic related to the branch head $s3$; in other words, we have captured and modelled all API calls within one of the nested `if-else` blocks. Since all paths related to $s3$ have been merged, we can retrieve the prefix-Regex stored under the key $s3$ and append p. Interestingly, the prefix-Regex mapped to $s3$ is actually `r_b:s1`, the interim-Regex originating from $s1$ that represents the highest level `else` block.

Now APUX has three interim-Regexes to merge : `r_c:s2`,`r_d:s2` and `r_b:s1`—this is where the origin tags sort out the nesting. APUX cannot merge `r_b:s1` with any other interim-Regexes because the recursive nature of APUX's strategy prohibits merging higher level nested statements with lower level ones. Thus, APUX will merge `r_c:s2` with `r_d:s2` and append the result to `r_a:s1` before merging `r_a:s1` with `r_b:s1`.

Hybrid merging occurs when more than two paths originating from both loop and branch heads converge. For example, hybrid merging may arise if a loop directly follows an `if-else` block. For the most part, the origin tags and Prefix-Map sort everything out. However, when the merge point is a loop head, APUX performs an additional dominating analysis to identify interim-Regexes internal to the loop head separately from those external to the loop. APUX then merges all interim-Regexes from inside the loop and all interim-Regexes from outside the loop separately, before merging their results.

Handling Multiple `return` Statements and Other Control Flow

APUX creates a Regex for a client method by passing an ordered list of API calls between the basic blocks of a client method's CFG. At the end of the CFG (e.g. a `return` statement) the ordered list of API calls becomes the Regex for the client method. APUX can capture all API calls for client methods that have a single `return` statement because all paths have the same originating and terminating points (i.e. the beginning of the method and the single `return` statement, respectively). However, client methods that have multiple `return` statements can create situations where diverging paths never rejoin, making it impossible to represent `if-else` block or loops with APUX's merging strategy. For example, Figure 2.11 is an example of a CFG where diverging paths never rejoin because of a `return` statement.

Similarly, `break` and `continue` statements can also create situations where APUX's merging strategy fails because they cause the control flow to suddenly break out of an `if-else` block or loop. To handle these special control flow situations APUX replaces all `break` and `continue`, and `return` statements with placeholder static method calls to a dummy library (e.g. `Dummy.RETURN`) before running its Regex analysis. When APUX eventually runs its APUX analysis, it will be on CFGs that will appear to have one `return`

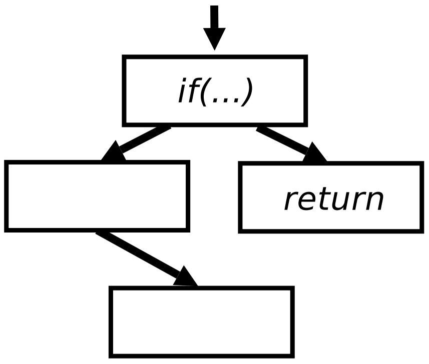


Figure 2.11: Example of how return statements leave dangling control flow

statement and no early exits from `if-else` blocks or loops. APUX uses static method calls as placeholders for `return` statements and the other control flow statements mentioned as a way of keeping the position of the statements for possible future analysis.

Despite changing the control flow of the client methods, replacing the control flow statements mentioned with placeholder static method calls, does not change the resulting Regex representation in any way. For example, the Regex generated for Figure 2.11 would be $\{\ldots \| \ldots\} \ldots$ regardless of having a `return` statement or a dummy statement.

2.3.2 Client Code with Exceptions

Similar to how `break`, `continue`, and `return` statements create CFGs which APUX cannot handle, APUX also cannot handle CFGs that contain checked exceptional logic (e.g. CFGs that contain `try-catch` blocks or `throw` statements). APUX cannot handle CFGs with `throw` statements because they allow a CFG to have multiple exit points, similar to CFGs with multiple `return` statements. To overcome CFGs with `throw` statements, APUX also replaces all `throw` statements with static calls to a dummy library (e.g. `Dummy.THROW`). Replacing `throws` statements with static calls to a dummy library does not change the resulting Regex representation of the client method.

However, APUX cannot remedy client methods that contain checked exceptions (e.g. `try-catch` blocks or `throws` statements in the signature). Unlike CFGs that contain `throws` statements or multiple `return` statements, APUX cannot perform a simple refactoring —like replacing the statements with a static call to a dummy library—to transform the CFG into something it can handle. As a result, all client methods containing `try-catch` blocks or containing a `throws` statement in its signature are skipped by APUX and are not represented by a Regex. As future work, we hope to implement logic in the core APUX algorithm to handle CFGs with exceptional logic in them.

2.3.3 Word Transforms

After APUX generates a Regex for each client method, it runs a number of transforms on the Regex. These word transforms further generalize some Regexes to facilitate pattern mining, while still retaining the original meaning.

Folding Multiple Identical Sequential Calls

APUX folds multiple identical sequential API calls into a single API call, with a plus indicating that the client method is invoking an API method one or more times, as demonstrated in Figure 2.2.

Flattening Redundant Branches

Some `if-else` blocks contain a single branch (e.g. $\{a\}$) indicating that the execution of the nested API calls is conditional, meaning that they will be executed *zero or one times*. However, `if-else` blocks that nest a single branch containing an additional single-branch, for example $\{\{a\}\}$, do so redundantly because a will still only be executed zero or one times. To remove the redundancy APUX flattens such branches. For example, the word $\{\{a\}\}$ would become $\{a\}$ since its execution is already conditional.

Merging Identical Branches

The Regex `if-else` representation denotes mutually exclusive API calls. However, when API calls in both `if` and `else` blocks are identical (e.g. $\{a\|a\}$) their executions are no longer mutually exclusive, but actually guaranteed. Thus, APUX transforms `if-else` blocks containing identical branches into unconditional call. For example, APUX would transform the Regex $a\{b||b\}c$ into abc. Since APUX merges identical branches recursively, $a\{bbb||bb\{b||b\}\}c$ would become $a\{bbb\}c$. APUX also merges nested branches that both have identical conditions or no conditions at all. For example, APUX would merge $a\{(x)b||(x)b\}c$ into $a\{(x)b\}c$, but would not merge $a\{(x)b||b\}c$.

2.3.4 Method Inlining

Different developers will have different programming styles. Some developers may prefer to use helper methods to encapsulate certain logic and others may prefer to keep it in the method body. Thus, matching API usage habits of different developers becomes more challenging if API methods are placed in helper methods on one client, but not another. To avoid this scenario, APUX inlines helper methods that make API calls. To avoid in-lining a large call chain—and possibly the entire project—APUX only inlines helper methods one level away that contain API calls and no calls to other client methods. APUX does not generate separate Regexes for inlined methods and does not consider the visibility (e.g. `public` or `private`) of methods when inlining methods. Inlining is also an optional feature that can be toggled on or off.

2.4 Pattern Mining

The goal of the pattern mining phase is to find patterns among the set of Regexes APUX extracted. In particular, the pattern mining phase looks for *frequent subsequences* among the Regexes. In essence, a Regex is basically an ordered sequence of symbols that represent API calls and operators containing information about the multiplicity of API calls in a client method. Therefore, finding frequent subsequences among Regexes is equivalent to finding common sequences of API method calls that different developers have made. The resulting patterns are the API usage patterns that become API code templates.

Finding API usage patterns among Regexes is a two-phased process : First, Maui clusters Regexes based on concept (similar groups of API calls) and then feeds each cluster to a sequential pattern miner. The sequential pattern miner looks for frequent subsequences of API calls with a high support count. The support count of a subsequence is the number of Regexes where the subsequence appears. Maui declares frequent subsequences with a support count higher than a user-defined threshold as API usage patterns and transforms them into API code templates. To increase the quality of the API code templates, Maui removes uninteresting Regexes which only have a single API call before pattern mining.

2.4.1 Clustering

Maui groups Regexes based on API calls to improve the sequential pattern miner's effectiveness. If we give the pattern miner a diverse set of Regexes covering a wide variety of API calls, the odds of finding a high-supported pattern are low. From our prototyping experience, only patterns containing API utility calls (e.g. `print()`) exist across a large corpus of Regexes. While this result is still interesting, these API utility call patterns do not capture the API usage scenarios that developers would probably want to encode into templates. If the user-defined support is set too low, the patterns are too weak to be useful.

The intuition is that certain API calls are related to certain API usage scenarios. Suppose we are looking for API usage patterns in a project that pertains to a GUI API. If we feed the pattern miner a set Regexes representing every method within the project, we will likely get patterns pertaining to the GUI, but also patterns that could pertain to logging, databases, etc. In addition to having unwanted patterns, the patterns may not include the desired GUI patterns because their support values may be lower than the support values of the utility patterns. Thus, clustering Regexes based on similar API calls is a way of loosely partitioning them based on a higher-level concept (e.g. GUI related API calls vs. DB related API calls). In contrast, if we run the pattern miner on the entire corpus of Regexes, the different API usage scenarios may interfere with each other and impact the mining process negatively. This phenomenon was observed by Zhong et al. [33], who originally proposed clustering as a solution to the dilution of API calls across a corpus of client methods.

Maui clusters Regexes using a hierarchical unsupervised clustering algorithm with method overlap as our distance measure. The distance between two Regexes, R_x and R_y, is calculated using the Jaccard index [11] as follows :

$$Distance(R_x, R_y) = \frac{|APIcalls \in R_x \bigcap APIcalls \in R_y|}{|APIcalls \in R_x \bigcup APIcalls \in R_y|}$$

A user defined distance threshold determines an instance's membership into a cluster. Through experimentation we found 0.35 to be the optimal threshold. In our experience, higher thresholds generated templates consisting of ubiquitous API method calls like getters and setters, whereas lower thresholds generated templates with too little support. We chose hierarchical clustering over other clustering methods because the number of resulting clusters does not need to be specified a priori. Instead, hierarchical clustering starts with every Regex as its own (trivial) cluster and iteratively joins clusters until there remains only one cluster containing all Regexes. There are several strategies for joining clusters (for example joining two clusters with the smallest average distance). Through prototyping we found Ward's method [31] to join clusters to work the best. The Ward strategy considers the union of every possible pair of clusters and combines the two clusters whose combination results in the smallest increase in error sum of squares.

The result of a hierarchical clustering analysis is a dendrogram of clusters. At the bottom of dendrogram, each cluster contains a single Regex (the trivial clusters), and at the top of the dendrogram is a single cluster containing all clusters. The dendrogram gives a range of sizes and relative distance between instances. Smaller clusters tend to be tighter in terms of relative distance, whereas larger clusters tend to be looser. Maui culls all clusters with sizes between five and thirty to make the results of the sequential pattern mining more meaningful (explained in Section 2.4.2).

Maui performs its hierarchical clustering using a modified version of Weka's [4] implementation.

2.4.2 Sequential Pattern Mining

Maui then looks for API usage patterns among the clusters using *sequential pattern mining*, a technique commonly used in market research to discover frequent sequential patterns among buying data with temporal or other ordering information. For example, a temporal

[4] http://www.cs.waikato.ac.nz/ml/weka/

pattern may be that customers frequently purchase a computer and a monitor together and purchase a printer at a later point.

Sequential pattern mining works by looking for common subsequences of items from a group of sequences. Sequences are composed of ordered itemsets and represent transactions in a database. In market research the items represent products, itemsets represent purchases, and the ordering represents the temporal ordering of those purchases. In our approach items and itemsets represent API calls and control flow, and their ordering represents their temporal ordering in a control flow graph.

Subsequences of items are ranked by their *support* value. The support of a subsequence is equal to the number of sequences that contain the subsequence. For example, from the sequences abc,bc, and bcd, the subsequence bc has a support value of two, whereas the subsequences b and c have support values of three. A subsequence is declared a *pattern* if its support count is greater than or equal to a user defined threshold. This threshold is usually defined as a percentage of the number of sequences given. For example, consider the sequences abc, bc, dc, and x. A threshold of 75% would mean that a subsequence would have to be supported by at least three sequences. Thus, the subsequence c is considered a pattern because it is supported by three sequences, but bc would not because it is only supported by two.

Before Maui can mine for patterns by first transforming each Regex into a transaction and inserts it into a transaction database. Maui models Regexes as transactions by representing individual API calls as items contained in individual itemsets. Maui sequences the itemsets according to the ordering found in the Regex (which represents their ordering found in the CFG of the method they were taken from). For example, a method `foo()` that makes calls to API methods `a()`, then `b()`, and then finally `c()` would be converted into a transaction $T_{foo} = (\{a\}, \{b\}, \{c\})$.

For Regexes that contain control flow, special item symbols are inserted to represent the beginning and end of control flow blocks. Symbols IF, END_IF, $ELSE$, and END_ELSE are used to denote the beginning and ending of `if-else` blocks and $LOOP$ and END_LOOP symbols are used for loop blocks. For example, the the Regex $a\{b||c\}d$ would become $(\{a\}, \{IF\}, \{b\}, \{END\_IF\}, \{ELSE\}, \{c\}, \{END\_ELSE\}$, $\{d\})$. Control flow blocks that have API calls within their condition are inserted into the same itemset as the corresponding entry symbol. For example, the condition `if(buffer .isOpen() && hasMoreTokens())` becomes the item set $(\{IF, isOpen(), hasMore Tokens()\})$. It is important to note that since the conditional API calls are placed in an itemset the ordering information of these calls is lost, but since the Regex model does not retain the logical operators, losing the ordering information does take anything further away form the approach.

API calls denoted with a plus (i.e. called one or more time) are treated as single calls by the pattern miner. For example, the pattern miner considers abc and ab^+c as identical. However, if a pattern is discovered with an API call denoted with a plus, the resulting pattern will also contain a plus. For example, considering the Regexes abc, b^+c, and bcd, b^+c could result as a possible pattern. The pattern b^+c shows that b may be called one or more times, while still accurately describing the Regexes it was derived from.

Maui uses the BIDE [30] sequential pattern mining algorithm offered by the SPMF pattern mining framework [6]. The advantage of using this particular algorithm is that it mines *closed* sequential patterns, meaning that subsequences that have the same support as their super sequences are ignored. Ignoring subsequences contained in larger sequences greatly reduces the number of results produced, helping users find more pertinent templates faster. Mining for closed sequences also ensures that patterns that contain control flow do not have mismatching control flow symbols because each entry symbol (e.g. IF) is matched with an exit symbol (e.g. END_IF) with the same support.

2.5 Code Template Creation

Following the pattern mining phase, the user is presented with a list of patterns, derived from each cluster, that can be made into code templates. The user manually inspects and selects which pattern they wish to use as the basis for their code template. Currently, Maui presents the user with the raw output from the pattern mining phase. For example, $(\{a\}, \{IF\}, \{b\}, \{END\_IF\}, \{c\})$ could be a pattern presented to the user, where a, b, and c are API calls. In the future, we would like to present to the user a preview of what the pattern would look like as a template. Once the user has selected a pattern, Maui then creates a template and presents it to the user.

Given a selected pattern, Maui creates a code template by placing each API call or control flow statement on a separate line. For each API method call in a chosen pattern we create a `String` of the fully qualified name of the method call. Control flow items are automatically converted into Java source code. For example, $IF, \ldots, END\_IF$ is transformed into `if(){...}`. After Maui creates an API code template, it adds the template to the user's Eclipse `TemplateStore` (a collection of templates a user may invoke in Eclipse)) allowing the user to further customize the template by adding or removing code, adding comments, or restructuring the template as they see fit.

Currently, Maui does not perform a data dependency analysis and cannot automatically create code to pass data between API calls in the template. Creating variables and passing output from API calls as input to other API calls is left to the user. However, in our experience this process has been straight-forward. We observed that, frequently, API calls that have parameters receive their data from the API call directly before it. All code templates created by Maui and presented in this paper have been created manually with this process with little effort. In the future, we would use data dependency analyses similar to those used in refactoring tools to automatically create API code templates that contain the data relationships between objects and methods in the template.

Chapter 3
Evaluation

The two main use cases for Maui are from the perspective of the API producer and the API consumer. The API producer would use Maui to create code templates or fill an API code cookbook from multiple client projects; the API consumer would do these things by mining their own project to teach or help fellow developers. For our evaluation, we recreated the API consumer scenario because it is a greater technical challenge. Due to the mining nature of Maui, drawing API code templates from a single project is more likely to yield fewer templates than drawing from a corpus of projects. To recreate the scenario and to evaluate the quality and relevance of the templates Maui creates, we conducted a retrospective study to see if templates created from an earlier version of a client project could have been useful to the developers who worked on a later version. We consider a template identified by Maui in an earlier version to be "useful" if it is supported by more client methods in in later versions of the project. Recall, the support value of a code template is equal to the number of client methods that contain the same sequence of API calls as the code template. The intuition behind our "usefulness" metric is that an API code template with a higher support value in a later version of a project represents instances where developers could have used the code template, created from the earlier version of the project, to implement some functionality of the API. In addition to discovering if the API code templates Maui creates are useful we also seek to answer more research questions (RQ) :

- **[RQ1]** Can Maui generate useful and relevant API code templates for developers?

- **[RQ2]** Will API code templates created from one project be useful to developers in another project? In other words, can API code templates be project idependent?
- **[RQ3]** Are API code templates created from API usage patterns stable? In other words, if we create an API code template today, will it be still be relevant in the future? This measures if developers change how they use an API overtime (assuming the API code stays the same), due to new requirements in the client project or changing programming styles.
- **[RQ4]** In what situations do Maui-generated code templates help developers the most?
- **[RQ5]** How applicable is Maui for different sized projects?

3.1 Methodology

For each client project we chose an API and two snapshots from the client's revision history. We carefully studied the evolution of each project (by analyzing the revision history commit logs and comments, and reading any history documentation on the project websites) to appropriately select snapshots that would recreate our envisioned API consumer scenario. The ideal initial snapshot, which will henceforth be referred to as the *init* version, is one when the API has already been adopted by the team and used in the code. We selected init versions by considering how much time elapsed since the start of the project, the number of commits, and details that suggest stability in the documentation. This is an ideal criterion because the developers have already gone past the learning curve, so creating templates at this time will be representative of how the API is actually used in the project. Choosing the point in time where the API is first adopted may result in templates where the developers were learning or experimenting. The later snapshot, henceforth referred to as the *final* version, was chosen by taking the latest possible version that used the API to allow for a long enough time span for the developers to hypothetically use the templates.

After selecting init and final versions for each project we created API code templates from both versions and kept those that had two or more API calls and a minimum support of

35% in the cluster that they were derived from. Recall from Section 2.4.1 that we generate patterns from clusters of size five or larger, which means that each template created in the will be supported by a minimum of two client methods ($0.35 \times 5 =\sim 2$).

Then, we take each code template created from the init version and look for a matching template in the final version using the matching heuristics defined in Section 3.1.3. For each init template we compare the support count and supporting methods between it and the set of matching templates found in the final version of a project. We measure the quality of the init template using a number of quality metrics discussed in Section 3.1.4.

3.1.1 Dataset

Our dataset consisted of three Java projects: the McGill Mammoth Project, ArgoUML, and the Eclipse IDE. We chose these three client projects based on their relative size to represent a small (Mammoth is ∼80 KLOC (Rev 3033)), medium (ArgoUML version 0.32.2 is ∼560 KLOC) and large client project (Eclipse version 3.7 is ∼4.3 Million LOC)[1]. By looking at different sized projects we can evaluate how applicable Maui is based on the size of its input (**RQ5**).

For each project we selected a single API to derive templates from the init and final versions. For all three projects we selected APIs that were graphics/GUI related. We selected graphics/GUI APIs because often developers compose/link GUI widgets and elements; even a trivial task involves a few API method calls. Therefore, API code templates will be a practical for GUI APIs.

We introduce each project in our evaluation and explain the rationale behind how we selected init and final versions to create templates from:

McGill Mammoth Project (jME API)

The Mammoth[2] project is a massively multi-player game research framework whose goal is to provide an environment for experimentation in areas such as distributed systems,

[1]The LOC statistics for ArgoUML and Eclipse do not include blank lines or comments and were taken from `http://www.ohloh.com`. Mammoth's statistics came from the project's lead Alexandre Denault

[2]`http://mammoth.cs.mcgill.ca/`

fault tolerance, databases, modelling and simulation, artificial intelligence and aspect-orientation. We chose to create templates based on its usage of the jMonkeyEngine API. The jMonkeyEngine (jME)[3] is a game engine made especially for 3D development. The entire project is based on Java and graphics are supported through OpenGL[4] via LWJGL[5]. The Mammoth project is six years in development and it has been using the jME API for roughly five years. Table 3.1 shows important related commits in its SVN revision history. Mammoth began using jME in a separate branch of the project until November 2007 when the branch was merged with the trunk (commit 1259). We decided not to make commit 1259 the init version because we estimated that the rest of the team had not enough time to learn the jME API. Instead we chose commit 1460 as the init version because the commit comment indicates that the team had prepared for a 3D demo, and thus we can assume that the jME API usage is stable.

Commit #	Date	KLOC	Note
1460	2008-02-28	$\sim$62	3D Demo Branch Merged
3033	2011-06-1	$\sim$80	Latest Version

Figure 3.1: Mammoth Init and Final Versions

ArgoUML (org.tigris.gef API)

ArgoUML is the leading open source UML modelling tool and includes support for all standard UML 1.4 diagrams.[6] We chose to create templates of ArgoUML's use of the org.tigris.gef API. [7] The org.tigris.gef API is a Java library to help develop new applications that involve editing diagrams and connected graphs. The org.tigris.gef API was originally part of the ArgoUML project, but became an independent project in 2000. We chose version 0.16 as the init version of ArgoUML because the release notes indicate that both the project

[3] `http://jmonkeyengine.com/`
[4] `http://www.opengl.org/`
[5] `http://lwjgl.org/`
[6] `http://argouml.tigris.org/`
[7] Not to be confused with the org.eclipse.gef API

and API stabilized as 615 issues were resolved. For the final version we took the latest version available.

Version #	Date	KLOC	Note
0.16.0	2004-07-19	∼125	New version of GEF - fixing numerous bugs
0.32.2	2011-04-04	∼560	Latest Version

Figure 3.2: ArgoUML Init and Final Versions

Eclipse IDE (org.eclipse.swt API)

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It can be used to develop applications in Java and, by means of various plug-ins, other programming languages.[8] We created templates from Eclipse's use of the Standard Widget Toolkit (SWT). SWT is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented.[9] The SWT is an Eclipse community project and was developed tightly with Eclipse, allowing us to choose any public version of Eclipse and have its use of the SWT be stable. To be safe we chose version 3.1 to ensure that the API usage had stabilized because there were significant changes between versions 2.1 and 3.0 when Eclipse made the move to the OSGI framework.[10] For the final version we took the latest release.

Version #	Date	KLOC	Note
3.1	2005-06-28	∼3500	Second major version
3.7	2011-06-22	∼4300	Latest Version (Juno)

Figure 3.3: Eclipse Init and Final Versions

[8] Adapted from `http://www.eclipse.org/`
[9] Adapted from `http://www.eclipse.org/swt/`
[10] `http://www.eclipse.org/equinox/documents/transition.html`

3.1.2 Exceptions In Client Code

One of Maui's limitations (discussed in Section 2.3.2) is that it cannot handle client code that contains checked exceptions (i.e. `try-catch` blocks and `throws` statements in the client method signature). Currently, Maui skips all client methods containing checked exception logic. To show that our evaluation is not skipping a significant amount of client methods we present Table 3.4, which reports the total number of client methods in each project along with the number of client methods skipped due to exception logic.

Project	# Skipped	# Total	(%)
Mammoth (Init)	3	92	3
Mammoth (Final)	6	100	6
ArgoUML (Init)	39	922	4
ArgoUML (Final)	46	1045	4
Eclipse (Init)	816	9566	9
Eclipse (Final)	1205	14386	8

Figure 3.4: Number of client methods skipped due to checked exceptions

3.1.3 How We Match Templates Between Init and Final Versions

We match templates generated from the init version with templates generated in the final version by calculating a similarity measure between a template $t_i \in T_{init}$ with $t_j \in T_{final}$, that takes into account the API calls and sequencing made by both t_i and t_j. Since we only compare the API calls and their sequencing between two template we do not need to generate complete templates, which would require some manual inspection (as described in Section 2.5). For our evaluation, we consider all patterns—detected with our pattern miner with a minimum support value of 35%—as templates. Using the patterns output from our pattern miner as opposed to complete code templates does not provide any additional benefits or drawbacks to our evaluation.

Instead of looking for identical templates between init and final versions, our similarity measure looks at common method overlap, based on the Jaccard index [11], to allow templates to slightly change between init and final versions. A template can change from init to final version by introducing or omitting API calls. Observing how much an API code template evolves also gives insight into how stable an API code template is (**RQ3**). Thus, for each $t_i \in T_{init}$ we categorize each $t_j \in T_{final}$ into one of three sets based on similarity: perfect, near, and half. A perfect match means both t_i and t_j have exactly the same API calls and sequencing—they are identical. A near match between t_i and t_j means t_j has 75% of the same API method calls and sequencing as t_i A half match, means that t_j has exactly half the amount of API calls that t_i has, and the same sequencing. We consider two templates having the same sequencing if the the intersection of API calls between the two templates have the same order. We do not consider control flow when comparing the order of two templates. The exact steps of our matching scheme are as follows:

- For each project, generate two sets of templates T_{init} and T_{final} from the init and final versions of a given target project and API.
- Filter out templates from T_{init} and T_{final} that have less than two API calls.
- For each template in $t_i \in T_{init}$, calculate the similarity between t_i and $t_j \in T_{final}$ and place t_j into one of the following sets per t_i :
  - **[Perfect Matches]** where $sim(t_i, t_j) = 100$
  - **[Near Matches]** where $75 < sim(t_i, t_j) \leq 100$
  - **[Half Matches]** where $50 < sim(t_i, t_j) \leq 75$
  - **[Discarded]** where $50 < sim(t_i, t_j)$

  We calculate the similarity between two templates $t_i \in T_{init}$ and $t_j \in T_{final}$ as follows :

$$sim(t_i, t_j) = \begin{cases} 0 & \text{if } t_j \text{ does not have the same sequencing as } t_i \\ \frac{|APIcalls \in t_i \bigcap APIcalls \in t_j|}{|APIcalls \in t_i \bigcup APIcalls \in t_j|} & \text{if } t_j \text{ does have the same sequencing as } t_i. \end{cases}$$

To make the results more meaningful we filter out all templates of size one because we do not consider these to be interesting. We still consider templates with two API calls because we did not want to rule out templates of the form $(a)b^*$ (i.e. a condition check a before successive calls to b). We speculate that these templates are good candidates for code completion supported templates.

3.1.4 Template Quality Metrics

We measure an API code template's usefulness (**RQ1**) by observing its relative support between init version and final versions of a client project. A template's support value is the number of client methods that contain the same sequence of API method calls as the template. For example, given the sequences $S_1 = (a, b, c)$, $S_2 = (a, b, c, b)$, and $S_3 = (a, d, d, d, c)$, the template $T_x = (a, b)$ will have a support count of two because it is supported by S_1 and S_2, whereas the template $T_y = (a, c)$ will have a support count of three because it is supported by all three sequences.

An increase in relative support value for an API code template between init and final versions means that there are more client methods in the final version that exhibit the same API usage pattern as the API code template created in the init version. Therefore, if we created API code templates in the init version, a developer could have used them to create the *new* supporting client methods in the final version. We consider this scenario a positive result. For example, suppose an API template T has a support value of five in the init version and ten in the final version. A developer could have used T to create those additional five client methods.

A decrease in relative support indicates that creating API code templates using Maui may not be worth the effort because any additional API usage would not have been based on the templates. As a reminder, we ensured that both init and final versions used the same target API, in order to control for API migration. We consider static support values as neither positive nor negative; on one side they do document how a client projects uses an API (a positive result), but on the other side it still takes time to use Maui and its not clear how useful they would be (a negative result).

We measure an API code template's relative support value to observe if there are *new* client methods in the final version where a developer could use an API code template to implement some functionality of the API. However, solely observing a template's relative support value may overlook situations where client methods are refactored or restructured, or when client methods are removed and newer ones were added, but not necessarily in the same context. For example, suppose a template (`new Button()`, `addEventListenter()`) has a support of seven in both init and final versions, but six of the seven supporting methods exist only in the final version and not the init version. Despite having the same support value (seven) in both versions, it is possible that one feature that used a `Button` was phased out and another completely different feature that also uses `Button` was introduced in six *new* methods. We take this into account in addition to the relative support value as a measure of usefulness.

In more concrete terms, we also observe the percentage of initial supporting methods of an API code template that continue to support the same template in the final version. For example, suppose a template T, is supported by client methods `foo()`, `bar()`, and `baz()` in the init version of a project. Then suppose that in the final version of the project T is now supported by `foo()`, `bar()`, and `moo()`. Since two of the three initial supporting methods continue to support T, namely `foo()`, `bar()`, the percentage of inital client methods that continue to support T is ∼66% (2/3). We say that these initial supporting client methods *persisted* in the final version. We use this persistence metric in conjunction with the relative support. A drop in persistence combined with a stable or increasing support count indicate new client methods were created where a developer could have used an API code template. A drop in relative support between init and final versions will always result in a drop in persistence, so we continue to interpret this as a negative result. We consider stable persistence of support methods combined with a stable relative support count as neither positive or negative. We do not track refactorings, so if a client method was renamed we consider deleted and added again.

Another interesting feature about measuring the persistence of supporting methods is that it may indicate whether the API code templates Maui creates are project-independent (**RQ2**). For example, a template having low persistence and the same support value between init and final versions indicates that context has changed, but the functionality it implements is still the same. This may indicate that the functionality is componentized in the API code template. We postulate that if an API code template can be useful in different situations within one project, that it may be useful in other projects as well.

3.2 Results

We present the results of our evaluation as follows: First, we present the aggregate results in Section 3.2.1 of how we matched templates created in the init version with templates created in the final version. Here we present the total number of init and final templates generated, as well as the number of perfect, near, and half matches between init and final versions. The aggregate results tell us how many templates created from the init version withstood the test of time and appeared in the final version. To reiterate, a template may exist across versions, but its supporting methods may not. Then, in Section 3.2.2 we present the quality of individual code templates by examining at how many times an individual API code template could have helped a developer, on average.

3.2.1 Aggregate Matching Results

Table 3.1 aggregates the results of the template matching between the init version of each project to the final version of each project. Table 3.1 also reports the number of templates that contained control flow in the perfect and near matching categories, denoted by #CF.

Table 3.1: Aggregated Template Matching Results

Project (API)	Init Templates	Final Templates	Perfect Matches	(#) with CF	Near Matches	(#) with CF	Half Matches
Mammoth (jME)	11	9	2	1	0	0	3
ArgoUML (org.tigris.gef)	182	150	95	49	76	7	461
Eclipse (org.eclipse.swt)	3065	3335	2591	451	14421	1959	227527

Comparing the number of templates derived from the init version to the final version for each project gives the sense of how many templates Maui yields from differently sized projects based on KLOC. An interesting result is that the raw number of templates derived from the init version was higher than the number of templates derived from the final version in the Mammoth and ArgoUML projects (from 11 to 9 in Mammoth and 182 to 150 in ArgoUML). It is interesting because these two projects expanded in terms of KLOC, but the decrease in code templates created may indicate that the expansion did not add any new coding scenarios.

Along with the total number of init and final templates are the aggregated totals of each matching categories. In the Mammoth project, only two of the original eleven ($\sim$18%) templates persisted between init and final versions. It turns out that these two templates are derived from utility methods in Mammoth. The first template involved positioning a camera in 3D space (`cam.setLocation(new Vector3f(1.0,2.0,-5.0))`), and the second was related to conditionally attaching or detaching spatial nodes and then updating a geometric state (`{node.attachChild() || node.detachChild()} scene.updateGeometricState()`). The support values for both of these templates are ten and seven, respectively. Both of these templates are prime candidates for code completion templates because they are utility-related and already highly supported; other developers could benefit from quickly instantiating these templates. Thus, these two code templates provide some evidence that Maui is at least capable of creating code completion type templates (related to **RQ4**). After investigating the three half matches in the Mammoth project derived from the final version, we report that they were not actually related to their matched init templates.

In ArgoUML, 95 of the original 182 templates persisted (approximately 52%), which we feel is a considerable amount, given the seven-year span between the init and final versions. Upon inspection of the templates, we report that 17 (approximately 18%) of them are related to initialization of API objects and changing the state of API objects, both of which are useful in code cookbooks and code completion. For instance, Figure 3.5 is a template that initializes and sets the layer of a diagram. Figure 3.5 is an interesting initialization template because it contains two non-obvious API calls, `setGraphNodeRenderer()` and `setGraphEdgeRenderer()`, which allow for custom node and edge icons. Considering that this template is supported by six client methods, it would seem like useful information for a novice programmer to have the first time they create a `LayerPerspectiveMutable` object rather than compiling and running their code to only find out the look and feel of thier diagram is different from the rest of their team's.

```
LayerPerspectiveMutable layer = new LayerPerspectiveMutable(...);
layer.setGraphNodeRenderer(...);
layer.setGraphEdgeRenderer(...);
diagram.setLayer(layer);
```

Figure 3.5: Initialization Template from ArgoUML

In contrast with the Mammoth project, ArgoUML had many more templates fall into the near matches and half matches category, 76 and 461 respectively. We were curious to see if the templates in the near matches category were longer or shorter than their init counterpart, and further analysis showed that 39% had expanded and 61% had shrunk. With a few exceptions, it appeared that most of the length change was due to the inclusion or omission of minor API method calls like getters and setters. Added condition checks were some of the more interesting API code template expansions because it showed that adding control flow context to our APUX phase was relevant. For instance one template includes an API check `isVisible()` before subsequent API calls. Similar to the Mammoth project, but in greater numbers, the half matches were tenuously related to their init counterparts.

The Eclipse project yielded a very high amount of templates in both init and final version (3065 and 3335 respectively), as well as 2591 (84%) perfect matches between versions. In contrast with the Mammoth and ArgoUML project, we found that the near matches that expanded usually added meaningful API calls to the init template, such as a `addModifyListener()` or `addSelectionListener()`. Upon further analysis we found the near matches did not show a particular trend for growing ($\sim$56) and shrinking ($\sim$44), which agrees with the results found with ArgoUML. However, we feel that 74 half matches per init sequence was too many, and similar to Mammoth and ArgoUML, these matches were tenuous.

To summarize our matching observations, in Table 3.1, we report that our approach does not seem to be as applicable for smaller scale projects (less than 100KLOC), as opposed to larger projects like ArgoUML and Eclipse (**RQ5**). On the other hand, we consider this result to be a positive point for Maui because it means that developers of medium and large projects can use Maui to create a substantial number of different code templates using only their own code base. It also implies that API producers may only need a few projects (as opposed to dozens) to generate API code templates for their documentation.

We also report that a substantial number of API code templates created in the init versions of the ArgoUML and Eclipse projects continued to exist in the final versions as well and did not undergo any changes (Perfect Matches category). In the ArgoUML project 52% (95/182) of the templates created from the init version remained in the final version and in the Eclipse project 84% (2591/3065) persisted across versions. These results provide evidence that a substantial number of API code templates remained stable (**RQ3**) over years of developement (seven and six years between init and final versions for ArgoUML and Eclipse respectively). This result provides evidence that API code templates created from API usage patterns can still remain applicable to developers years after they were created.

Table 3.1 also reports the number of perfect and near matching templates that contained control flow constructs. We chose not to report the number of half matching templates because they are usually tenuously related to their corresponding init templates (discussed above). Reporting the number of templates that contain control flow constructs is useful because these results indicate how important it is to capture control flow when mining for API code templates. In the Mammoth project one of the two (50%) perfect matching code templates contained control flow and zero of the near matching category (since there were zero near matches). In the ArgoUML project, 51% (49/95) of the perfect matching templates contained control flow and 9% (7/76) of the near matching templates contained control flow. In the Eclipse project, 17% (451/2591) of the perfecting matching templates contained control flow and 14% (1959/14421) of the near matches contained control flow. Considering that over 50% of the perfect matches in ArgoUML and nearly 20% of the perfect matches contained control flow, these results provide evidence that capturing control flow is important because a substantial number of templates exhibit that developers use control flow constructs in conjunction with their API calls.

3.2.2 Individual Template Support Results

To get more insight into whether API code templates would be useful for developers (**RQ1**), we present the results of our evaluation that report the average number of times an API code template could have helped a developer implement some functionality offered by the API. Table 3.2 reports the average support count of all API code templates generated from each project. The second column (Init Templates) presents the average support count of each code template found in the init version of each project. For example, the entry for the Mammoth project is 4.6, meaning that on average, each template generated from the init version of the Mammoth project was supported by roughly five client methods. The third and fifth column, present the average support values for the perfect and near matching templates found in the final versions of each project. From our observations in Table 3.2.1 and further inspection, we decided to omit the results related to the Half Matches category in Table 3.2 because the matches were too tenuous to be useful.

In Table 3.2 we also report the average percentage of initial client methods that continued to support templates in the final version. Using the terminology we defined in Section 3.1.4, we say that these supporting client methods *persisted* across versions. The fourth and sixth columns display the average percentage of initial supporting client methods that continued to support the same templates in the final version of each project for the templates in the perfect and near matches categories, respectively. For example, the average percentage of initial supporting client methods that continue to support the same template in the final version of the Mammoth project is 57%. This means that, on average, each template generated in the final version of the Mammoth project, and placed in the perfect matching category, is supported by 57% of the original client methods.

In all three projects, the average support value for each template created in the final version was higher than the support values for templates created in the init version. This result means that on average, each code template in the perfect matching category had more client methods exhibiting the same API functionality than templates created from the init versions. For instance, templates created from the init version of the Eclipse project were on average supported by approximately ten client methods. Meaning that on average, each

Table 3.2: Results : Average Template Support and Persistence Ratio

Project (API)	Init Templates	Perfect Matches	(%) Persist	Near Matches	(%) Persist
Mammoth (JMonkey)	4.6	8.5	57	N/A	N/A
ArgoUML (org.tigris.gef)	6.90	10.27	9	9.42	1
Eclipse (org.eclipse.swt)	10.27	16.20	35	14.33	29

template created from the init version of Eclipse had approximately ten client methods displaying the same functionality. In contrast, the templates generated from the final version of Eclipse, were on average supported by approximately 16 and 14 client methods for the perfect and near matching categories, respectively.

In the perfect matches category (column 2 of Table 3.2), we report an increase of approximately four, three, and six supporting methods for each code template on average, for the Mammoth, ArgoUML, and Eclipse projects, respectively. These increases mean that *each* API code template created in the init versions of the Mammoth, ArgoUML, and Eclipse projects could have helped developers implement the same API functionality in an additional four, three, and six instances, respectively. In the near matches category (column 4 of Table 3.2) we report similar a similar increase for ArgoUML and Eclipse projects (roughly three and four methods, respectively).

We consider these numbers to be substantial when the total number of perfect and near matches are considered. For instance, there were total of two perfect matches between init and final versions of Mammoth. This result translates into eight instances (4×2) where an API code template could have helped a developer, which is a small amount. However, considering that ArgoUML had 95 perfect matches and Eclipse had 2591, this shows that there could have been a multitude of opportunities for Maui to help developers implement API functionality.

When we consider the low average percentage of initial supporting methods that persisted, we see an even greater increase in the number of opportunities where Maui could have helped developers. In the perfect matches category for ArgoUML, on average only nine percent (approximately one client method (6.9×0.09)) of the initial supporting methods continued to support the same templates in the final version. Which means, on average each perfect matching template generated from the final version of ArgoUML retained only one of its original supporting client methods. Considering that each template in the perfect matches category was supported by roughly ten client methods, this means that on average, nine of those ten client methods did not exist in the init version of ArgoUML. In other words, these nine client methods were *new* methods added to later versions of ArgoUML. This translates to an average of nine opportunities per template where the developers of ArgoUML could have had code templates at their disposal. In the perfect matches category of Eclipse 35% of the initial supporting methods continued to support the same templates in the final version, which translates to roughly three or four client methods (10.27×0.35). This result means that on average there are approximately twelve opportunities per template where the developers of Eclipse could have benefited.

The near matches also show this trend, with an average of eight new methods in ArgoUML (assuming an average of one client method persisting) and eleven in Eclipse (assuming an average of three client methods persisting). To summarize the results found in Table 3.2: the reasonable increase in support for perfect and near matching templates gives us reason to believe that the templates created by Maui may have been useful to developers between init and final versions of their respective projects.

The increase in relative support of API code templates paired with the low persistence of supporting client methods also provides evidence that API code templates are project independent (**RQ2**) because the functionality found in the templates stayed the same, but the context did not. This implies that API code templates created from one project are likely to be useful for developers in another project. API producers especially benefit from this result because they can create relevant API code templates using Maui for documentation purposes.

```
Button myButton = new Button(parent, SWT.NONE);
myButton.setText(‘‘Cancel’’);
```

Figure 3.6: Example of a Good Code Completion template

3.2.3 Examples of Mined API Code Templates

The results found in Table 3.2 quantitatively showed evidence that creating API code templates with Maui can be useful to developers. To qualitatively show how and why they are useful to developers we present a few mined templates. For instance, consider the template in Fig 3.6 mined from Eclipse. Albeit short, the template in Fig 3.6 is a good candidate for a code completion template because it is highly repeatable. In the final version of Eclipse, this template was supported by 140 new instances, on top of the 105 persisting instances from the init version.

Another interesting template, Figure 3.7, shows the ability of APUX to detect templates with control flow. Figure 3.7 demonstrates how to create a new SWT table, populate it with columns, and set the text of the columns.

```
Table lTable = new Table(...);
myTable.setLayoutData(new GridData(...));
myTable.setLinesVisible(...);
while(...)
{
    TableColumn lCol = new TableColumn(lTable,...);
    lCol.setText(...);
}
```

Figure 3.7: Example of a Template with Control Flow from ArgoUML

Figure 3.7 is interesting because its uses are two-fold; it could be useful as a code completion template because in our own experiences we rarely create a table without immediately populating it with multiple entries; and it could be useful as a code cookbook entry because most novice programmers would expect to add the columns directly to the

`Table` object via an `addColumn()` method, rather than passing the parent `Composite` object to the constructor of the `TableColumn`.

Smaller templates are better suited for code completion templates, whereas larger ones are better for cookbook entries. To show that Maui can mine both sets (**RQ4**), we present the distribution of templates lengths, which do not include control flow statements, in Table 3.3. Our initial thoughts were that the distribution would be dominated by 2-letter templates, but Table 3.3 shows a reasonable spread of template lengths.

Table 3.3: Template Length Distribution

Project (API)	Version	2-API calls	3-API calls	4-API calls	≥5-API calls
Mammoth	Init	4	1	0	6
(JMonkey)	Final	3	2	2	2
ArgoUML	Init	69	45	34	34
(org.tigris.gef)	Final	75	35	19	21
Eclipse	Init	527	401	463	1674
(org.eclipse.swt)	Final	697	537	507	1594

3.2.4 Further Qualitative Findings

Discovering Useful Infrequent Calls

From investigating the results of we observed that Maui can also spot invariants of an API usage pattern. Some of the patterns we observed were almost identical, but varied by either one or two API calls. Also, were surprised to find that some API calls and objects were not frequent enough to be included in a template. For instance, in the Eclipse project we came across many templates that added an `EventListener` (i.e.`addSelectionListener()`), but only on a few occasions did a corresponding concrete `EventListener` appear in the template as well. Upon further inspection we found out that each supporting client method would add their own custom `EventListener`, which lowers the probability of a common `EventListener`. Creating an accompanying `EventListener` is crucial information for developers, even if we can only tell them to create an object of the interface type `EventListener`. Therefore, some worthwhile future work could involve searching for further commonalities between supporting methods by looking at the type hierarchies between some of their elements.

Common Template Types

From browsing the results, we noticed that a substantial number of templates are related to creating a new API object and setting it up (e.g. `setFont()`, `addToLayer()`, and `addListener()`). We believe the cause is the nature of graphics and GUI APIs because they routinely involve creating and setting up widget-like objects. We conclude that if our evaluation was performed with, say, XML editors we would see many patterns of the form $openFile(), [createElement(), setAttribute()]^*, closeFile()$.

3.3 Threats to Validity

The biggest threat to the validity is that the results of our evaluation do not translate to other types of APIs because graphics/GUI type APIs are highly repeatable. Since ours is a data mining approach, we strongly depend on the frequency of common API usage scenarios. Other, less repeatable APIs would require many client projects to discover the same number and quality of API usage patterns from which to create code templates. Our results also do not translate to framework-type APIs where common usage involves implementing interfaces and extending classes. Currently, our approach cannot detect classes that extend an API class or implement an API interface, but we can detect super calls.

We also discovered in our evaluation that Maui may not be as applicable to smaller-scale projects like Mammoth (less than 100KLOC). However, small scale projects could possibly increase the yield of templates by including another client project of the API.

Another threat to validity is that we did not control for API evolution when mining for patterns from the init and final versions of each project. Instead, we simply assumed that the package names would persist between API versions. However, any changes between versions of the API could only be detrimental to our results.

We also did not track any refactorings between the init and final versions of the client projects. Not tracking refactorings could only affect the results related to the persistence of initially supporting client methods of templates. If a client method was renamed, we consider this change to be a deletion of the old method and the introduction of the new method, instead of a considering the two the same.

Chapter 4
Related Work

Over the past ten years there have been numerous approaches that leverage API usage information to accomplish a myriad of goals. These goals include: bug detection, example (snippet) retrieval, program navigation, inferring API specifications, and documentation. To the best of our knowledge, our approach is first to leverage API usage patterns to create API code templates.

4.1 Frequent Itemset Mining Approaches

The first group of approaches use frequent itemset mining to discover common patterns of API usage and generate association rules. An association rule describes the conditional probability between variables in a transactional database. Association rules were originally developed for market research to discover regularities between products in a transaction database [15]. For example, given a customer buys a computer and monitor they are then also likely to buy a printer with some probability.

CodeWeb [20], was one of the earliest approaches to use association rules as a means of finding API reuse patterns. CodeWeb specifically mined the reuse patterns between API methods and classes being reused by a specific program. For example, one reuse pattern CodeWeb found was that 100% of clients that override `doIt()`, also override `undoIt()`.

FrUiT [3] is a framework understanding tool that refines CodeWeb's techniques, such as filtering spurious or uninteresting rules, to recommend relevant API elements given their current coding context. For example, if a user instantiates an `IWizardPage`, FrUiT would recommend making calls to elements like `addPage()`. In another paper [2], Bruch et al. devise three code completion systems based on different measures of API usage patterns; one system was based on usage frequency of API elements, another used association rules between API elements, and the third was a novel clustering technique. The author's findings showed that their k-NN-inspired algorithm outperformed the other proposed code completion systems, based on a cross-validation study of `org.eclipse.swt`, but only showed marginal improvements to the association rule technique.

PR-Miner [16] is another tool to mine API usage association rules, but which treats these rules as implicit programming rules and violations to these rules as possible bug locations. For example, an implicit programming rule could be to always follow a call to `lock()` with a call to `unlock()` and a violation of this rule would be the omission of `unlock()`. Indeed, Li and Zhou were able to identify 16 bugs in Linux, 6 in PostgreSQL, and 1 in the Apache HTTP server by looking at the top 60 violations to patterns detected by their tool. However, the authors report that even with pruning, a large number of the association rules were false positives. DynaMine [18], like PR-Miner, looks for API association rules and for violations, but differs in that it mines rules from software revision histories, rather than project snapshots. DynaMine also differs in that it dynamically instruments the patterns to look for violations. DynaMine's evaluation also reported many false positives.

In a comparison study [14] between frequent itemset mining and sequential pattern mining (the technique used by Maui see Section 2.4.2), Kagdi et al. comment that false positives are caused by the lack of ordering information. For example, out-of-order rules are hard to fix because there is no information on where to insert the missing call. Generally, frequent itemset mining is much faster than sequential pattern mining, but without ordering useful information like the multiplicity of elements, the order of the elements, and the context information of elements is lost. Maui has an advantage over these approaches because it retains the ordering of elements and multiplicity. Further, Maui can mine arbitrary regular patterns, unlike CodeWeb [20, 21] or DynaMine [18].

Some approaches remedy the out-of-order problem by encoding ordering information directly into their itemsets. For example, with the JADET tool [32] Wasylkowski et al. mine sets of temporal properties showing how a client method uses methods of an API.

For example, a set of temporal properties could $P = \{hasNext() \prec next(), next() \prec hasNext()\}$, where $m \prec n$ means that there is a possibility of calling m before calling n. JADET only considers a single project to mine association rules, but this approach was extended by Gruska et al. [10] to mine from 6000 Linux projects. Gruska et al's intuition was that violations may not be detected in a single project if all sequences of calls to the API are incorrect, but if compared against 6000 projects the violations would surface. Alattin [27], by Thummalapenta and Xie, tries to look for frequent condition check patterns before or after a given API method. For example, "boolean check on return for f_a before f_1" or "constant check on return for f_b after f_1. Even though these rules are unordered, the definition of the condition checks includes information about the correct position. Combining Alattin's approach with APUX may bolster Maui's support of API related conditions.

4.2 Sequential Pattern Mining Approaches

The closest cluster of approaches to Maui are those that also use sequential pattern mining to infer API usage patterns. Like frequent itemset mining, these approaches look for frequent patterns in a transactional database, but differ by encoding information about the temporal ordering and multiplicity into the itemsets.

MAPO [33], presented by Zhong et al., is a tool that leverages API usage patterns found with sequential pattern mining for code snippet retrieval. The API usage patterns are used as an index to match queries with code snippets. MAPO works by looking at the sequences of API calls a client method makes and takes a subset of sequences that cover all API calls. The client methods are then clustered according to the API methods they invoke and by the natural language terms found in their method name and enclosing class. Each cluster is fed into a sequential pattern miner (BIDE [30]) that looks for frequent subsequences of API method calls. The resulting patterns become the index for the parent cluster and are used to recommend snippets by comparing the user's query to the index. Using the representative sets along with sequential pattern mining, Zhong et al.'s approach can mine patterns that are of any length and preserve the temporal ordering of API calls. However, unlike Maui this scheme does not include control flow constructs or any indication of the multiplicity of API calls, which could be useful to users creating code templates.

Some API usage approaches use sequential pattern mining to generate sequential association rules, $X \Rightarrow Y$, where all items in the sequence X must come before items in the sequence Y.

For example, an approach by Theummalapenta et al. [28] looks for sequential association rules where the antecedent contains regular API call sequences and the consequent contains calls that occur in exceptional cases. The authors generate these rules by populating two sequence databases, one with normal execution sequences and one with exceptional sequences, and annotate each sequence depending on which DB they are in. These annotations are used to build rules of the form $FC_c^1 \ldots FC_c^n \wedge FC_a \Rightarrow FC_e^1 \ldots FC_e^m$, which translates to: an API call FC_a should be followed by the function-call-sequence $FC_e^1 \ldots FC_e^m$ in exception paths, when preceded by the function-call sequence $FC_c^1 \ldots FC_c^n$.

Kadgi et al. present an approach [13] to include syntactic context (e.g. if an API call is in a for loop) when mining for sequential association rules. During pattern mining, their approach inserts tags around API calls indicating that they are within a control flow structure. For example, an API call `a()` inside an `if` statement guarded by a call to `b()`, would be represented as $< if\_cond = "b" > a < /if\_cond = "b" >$. Maui's Regex model uses a similar strategy (discussed in detail in Section 2.4.2), by including tags to represent control flow, but only includes tags at the beginning and end of an API sequence within the control flow instead of tagging each element. The two strategies are more or less equivalent, but Maui's strategy results in better performance because it inserts fewer tags into the sequences, making it easier for the pattern miner because of the reduction in complexity.

4.3 Graph-Based Approaches

GrouMiner, presented by Nguyen et al., is another tool for extracting object usage patterns, but uses graph-based algorithms instead of frequent itemset mining [23] to discover common usage patterns. Essentially, Grouminer creates a (directed acyclic graph) DAG called a *groum* (graph-based object usage model) to represent an object's usage in code; nodes represent object actions (i.e method calls or field accesses) and edges represent the order these actions are called. GrouMiner then mines for usage patterns by looking for isomorphisms among groums. Nguyen et al. encode Groums into Exas vectors to facilitate finding isomorphisms. GrouMiner also detects violations by looking for subgraphs of Groums that have a lower frequency than a user-defined threshold.

PARSEWEB and Prospector are two graph-based API usage approaches which try to answer call chain queries, where a user specifies a *source* type and a desired *destination* type and wishes to know the sequence of intermediate method calls to go from *source* to *destination*. PARSEWEB [26], a tool by Thummalapenta and Xie, answers call chain queries by mining for frequent method invocation sequences (MISs) from code snippets found on Google Code search. After clustering similar MISs, PARSEWEB creates a DAG from the source type to the destination type and returns the shortest path between the two, since there could be multiple possible answers. Prospector [19], developed by Mandelin et al., uses an API's class declarations, field declarations, and method signatures to build a *signature graph* to answer call-chain queries, instead of building graphs from ASTs. In a signature graph, nodes are class types declared by the API and edges represent the operations to go from one type to another. Like PARSEWEB, Prospector answers queries by finding the shortest path between one API type to another. Additional information about legal downcasts can be added to the signature graph by analyzing client code.

4.4 Regex-Like Approaches

Gabel and Su propose a Binary Decision Diagram (BDD) based approach to mine two letter micro-patterns (for example ab^+) for API specifications. [8]. The authors later extend this approach with a tool called Javert [7], to chain the micropatterns in to larger patterns. In a different approach [9], the authors mine temporal properties from dynamic traces online using a viewing sequences of API method calls in a sliding window and monitoring the historical statistics of how many times a candidate pattern is followed or violated. Liu et al.'s approach [17] also mines two-letter patterns that follow three particular scenarios, initialization (i.e. an `init` method followed by some other call), push-pop, and finalization (i.e. an API call followed by a call to a `close()` call). They generate a list of API usage rule candidates by replacing the symbolic function names in one of the three existing rule templates with concrete API function names. Then they feed the templates into a model checker to see which are valid.

4.5 Usage Statistic Approaches

API usage statistic approaches study API usage by looking at the frequency with which each API element is used by clients as an indicator for importance. Usage statistic approaches are usually the basis for API recommendation systems to recommend interesting or crucial parts of an API to a developer. PopCon [12], by Holmes and Walker, is a tool that simply counts the number of times an API element is referenced based on four structural relationships: method calls, field references, inheritance, and method overriding. Jadeite [25] is another tool, by Stylos et al., that aims to improve API JavaDocs by helping direct developers' attention to useful API elements. For example, Jadeite resizes the font of API elements in the class and package view proportionally to the number of Google hits they get. Aktari [22], by Mileva et al, looks at the popularity of different versions of an API to help developers select the best version. Aktari works by looking at the referenced APIs for a corpus of projects along with the version number and keeps track of the most used and most reverted to APIs to recommend to developers.

4.6 Typestate

So far, the approaches mentioned and model API usage by looking at client code that uses the API. These approaches can be considered as *positive* because they describe *what is*, rather than *what should be*, which instead would be normative. The tools that study API usage to find bugs or defects work by finding an API's protocol by observing popular usage patterns and flagging anomalies as bugs. However, these bugs and defects are found after the fact and would be cheaper to fix if they were found earlier in the development stage.

Thus, there exist normative approaches where API designers define their protocol explicitly and violations can be detected statically, much like how type checking is done. These approaches build on top of a programming language concept called *Typestate*. Typestate tracking is a compile-time program analysis technique that enhances program reliability by detecting type-correct applications of operations which are "nonsensical" in their current context [24]. The idea is that a typestate captures the notion of an object being in an appropriate (or inappropriate) state for the application of a particular operation. For example, invoking a method on a null object reference is legal as far as types are concerned, but will cause an error because the object is not in an "initialized" state. An object's type state may change when a legal operation is invoked. So in our example, when the object's constructor is called, the typestate of the object now becomes "initialized" and operations that were previously illegal can now be invoked. The typestate concept was extended to the object

Deline and Fahndrich extend the typestate concept by proposing a statically checkable typestate system, Fugue, to declare and verify state transitions and invariants in imperative object-oriented programs [5]. An effect of this work is that Pre and Post annotations can restrict the order of methods clients can invoke on an object based on the typestate it is in. For example, suppose calling `closeStream()` causes a stream object to move into a "closed" typestate; this restricts calls to methods that have an "open" typestate as a precondition, such as `getData()`. Thus, API builders can explicitly define and enforce a protocol at compile time. Deline and Fahndrich's work has been extended by Bierhoff et al. by adding access permissions on top of object typestates [1]. Access permissions describe the ways in which an object can be aliased. The possible access permissions are exclusive, exclusive modifying, read-only, immutable, and shared access.

Chapter 5
Conclusion

5.1 Summary

Maui is a tool that automatically creates API code templates based on API usage patterns from client projects. The contributions of our approach are a novel technique for discovering API usage patterns that include information about the ordering of API calls and multiplicity, as well as information about control flow. In a retrospective analysis with the Mammoth, ArgoUML, and Eclipse projects, we generated API code templates from earlier versions of the project, using Maui to see if they could have been useful to developers in later versions. In our evaluation we found that, on average, each code template created from the earlier version of each project could have helped developers in the later version in at least dozens of situations. We also observed that a substantial number of API code templates created from the earlier versions of our dataset remained identical in the final version, which may imply that templates created with Maui could be useful to developers years after they are created. As for the individual API code templates, we found that Maui is capable of creating a range of differently sized API code templates that could be useful for quick code injection or more complex scenarios. Finally we also observed that Maui is most applicable for medium and large sized projects (>100KLOC). While this result implies that Maui is not applicable for smaller projects, it also means that API producers will not need a large corpus of projects to create API documentation using Maui. From the perspective of the API user and the API developer, Maui is a useful tool for generating API code templates.

5.2 Limitations and Future Work

Maui is currently a proof of concept and could benefit from a few improvements. The largest limitation of Maui is the manual step which the user needs to perform to create a functional API code template. Maui can find the invariant API method calls and sequence them, but does not give much support to the user regarding how to construct the template. In our experience, this manual step is relatively short and straight-forward, but users would benefit more if we automated this step. As future work we plan to automate this step by incorporating a data dependency analysis to chain the API calls together and pass necessary data between calls. Since Maui is built on top of a static analysis framework, we could easily extend Maui to support these additions. Additionally, we could extend Maui to recommend objects, parameters, or boolean conditions based on information found in the the support methods used to derive the template.

Another limitation is that Maui currently cannot generate API code templates from client methods that contain checked exceptions because it does not have the logic to properly traverse those kinds of CFGs. This limited us from studying the JBoss project's use of the JDom API. For future work, we would extend APUX to handle such CFGs. Another limitation is that Soot does not always give accurate source line and offset information about Jimple statements, causing Maui to occasionally fail on certain methods. We have fixed some of the issues related to this, and submitted patches, but there is still work left to be done.

The last and most important piece of future work that we will suggest is to devise a filtering or recommendation system to help documenters zero in on the most pertinent templates (especially given that Eclipse produced over 3000 candidate templates). Possible suggestions could be based on the coder's current programming context or perhaps an initial seed of API method calls. It could also be based on the browsing behaviour of developers on the API's JavaDoc.

Bibliography

[1] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd AITO European Conference on Object-Oriented Programming*, pages 195–219, 2009.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 213–222, 2009.

[3] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE support for framework understanding. In *Proceedings of the 21st ACM SIGPLAN OOPSLA Eclipse Technology eXchange*, pages 55–59, 2006.

[4] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proceedings of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 127–136, November 2010.

[5] R. Deline and M. Fahndrich. Typestates for objects. In *Proceedings of the 18th AITO European Conference on Object-Oriented Programming*, pages 465–490, 2004.

[6] P. Fournier-Viger. Spmf : A sequential pattern mining framework. `http://www.philippe-fournier-viger.com/spmf/`, 2011.

[7] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 339–349, 2008.

[8] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, pages 51–60, 2008.

[9] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 15–24, 2010.

[10] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th ACM International Symposium on Software Testing and Analysis*, pages 119–130, 2010.

[11] L. Hamers, Y. Hemeryck, G. Herweyers, M. Janssen, H. Keters, R. Rousseau, and A. Vanhoutte. Similarity measures in scientometric research: The Jaccard index versus Salton's cosine formula. *Information Processing and Management*, 25:315–318, May 1989.

[12] R. Holmes and R. J. Walker. Informing eclipse API production and consumption. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse technology eXchange*, page 70–74, 2007.

[13] H. Kagdi, M. L. Collard, and J. I. Maletic. An approach to mining call-usage patterns with syntactic context. In *Proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering*, pages 457–460, 2007.

[14] H. Kagdi, M. L. Collard, and J. I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *Proceedings of the 4th ACM/IEEE International Workshop on Mining Software Repositories*, pages 20–, 2007.

[15] W. Klösgen and J. M. Zytkow, editors. *Handbook of data mining and knowledge discovery*. Oxford University Press, Inc., New York, NY, USA, 2002.

[16] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software engineering*, pages 306–315, 2005.

[17] C. Liu, E. Ye, and D. J. Richardson. LtRules: An automated software library usage rule extraction tool. In *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, page 823–826, 2006.

[18] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305, 2005.

[19] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 48–61, 2005.

[20] A. Michail. Data mining library reuse patterns in user-selected applications. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 24–33, 1999.

[21] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd ACM/IEEE International Conference on Software Engineering*, pages 167–176, 2000.

[22] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proceedings of the Joint International and annual ERCIM Workshops on Principles of Software Evolutionand Software Evolution Workshops*, pages 57–62, 2009.

[23] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 383–392, 2009.

[24] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.

[25] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers. Improving API documentation using API usage information. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, page 119–126, 2009.

[26] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering*, page 204–213, 2007.

[27] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 24th ACM/IEEE International Conference on Automated Software Engineering*, page 283–294, 2009.

[28] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*, pages 496–506, 2009.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 13–, 1999.

[30] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of the 20th IEEE International Conference on Data Engineering*, pages 79–, 2004.

[31] J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[32] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 35–44, 2007.

[33] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the 23rd AITO European Conference on Object-Oriented Programming*, pages 318–343, 2009.