Detecting Object Usage Anomalies

Andrzej Wasylkowski
Saarland University
Saarbrücken, Germany
wasylkowski@cs.uni-sb.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-sb.de

Christian Lindig
Saarland University
Saarbrücken, Germany
lindig@cs.uni-sb.de

ABSTRACT

Interacting with objects often requires following a protocol—for instance, a specific sequence of method calls. These protocols are not always documented, and violations can lead to subtle problems. Our approach takes code examples to automatically infer legal sequences of method calls. The resulting patterns can then be used to detect anomalies such as "Before calling `next()`, one normally calls `hasNext()`". To our knowledge, this is the first fully automatic defect detection approach that learns and checks method call sequences. Our JADET prototype has detected yet undiscovered defects and code smells in five popular open-source programs, including two new defects in ASPECTJ.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Class invariants, Statistical Methods; F.3.2 [**Semantics of Programming Languages**]: Program analysis; F.3.3 [**Studies of Program Constructs**]: Control primitives, Object-oriented constructs

1. INTRODUCTION

When interacting with an object, the program has to follow the rules mandated by its interface. For individual methods, the compiler checks whether the caller is allowed to invoke the method, and whether its arguments are correctly typed. The *interplay* of multiple methods, though—in particular, whether a specific sequence of method calls is allowed or not—is neither specified nor checked at compile time. Only if the program fails at run-time may the programmer discover that she would have been required to, say, call `Stack.push()` before invoking `Stack.elements()` or, to check an iterator's value using `Iterator.hasNext()` before incrementing it with `Iterator.next()`. Consequently, illegal call sequences may still loom in the code even though all tests pass.

In this paper, we propose mining *object usage models* from code examples—representations of typical object usage as possible sequences of method calls. These patterns can be used to automatically find locations in programs that deviate from normal object usage—that is, defect candidates.

In Section 2, we describe how to extract object usage models from JAVA code—finite state automata with anonymous states and transitions labeled with feasible method calls (Figure 1(a–b)). For instance, the `iter` model tells us the typical interaction with a JAVA iterator object: Typically, `hasNext()` is called first (in a loop condition), and then either the interaction ends (the loop exits), or `next()` is called and the loop starts again. Our approach focuses on modeling objects from the point of view of single methods, using intraprocedural analysis only. This allows us to get comprehensive results guaranteed by static analysis and yet remain scalable.

In Section 3, we extract *patterns* such as "`next()` can precede `hasNext()`" from models (Figure 1(c)). These patterns are collected over the entire code body and then fed into a classifier which identifies *locations that violate these patterns* (Figure 1(d–e))—that is, likely defect locations. Section 4 summarizes our experiences with five popular open-source programs. Our JADET prototype detects a large number of code smells in these programs, including a previously undiscovered ASPECTJ defect where `next()` *never* precedes `hasNext()`—the first loop iteration always exits.

This is not the first work to extract and check temporal object behavior. However, it is the first to automatically do both for individual objects, and therefore can find errors undetected by others. Our main advantage is being able to find program-specific temporal patterns without the need of any external input apart from the program itself. In Section 5, we discuss the related work in mining temporal object behavior and detecting defects automatically, and highlight the contributions of the present approach. Section 6 closes with a conclusion and consequences for future work.

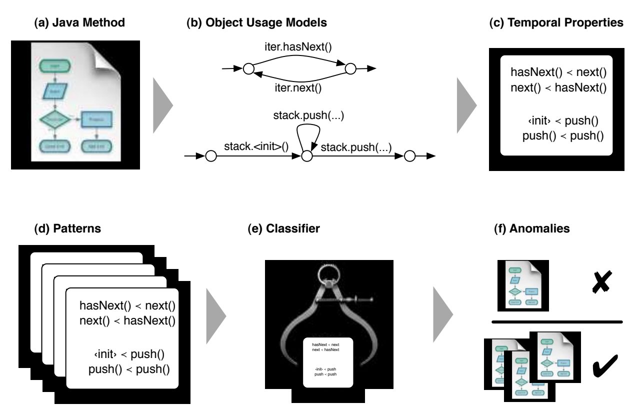

Figure 1: How JADET works. JADET extracts object usage models from JAVA methods. The resulting method call patterns are fed into a classifier which detects abnormal usage.

```
public Stack createStack () {
    Random random = new Random ();
    int size = random.nextInt ();
    Stack stack = new Stack ();
    for (int i = 0; i < size; i++)
        stack.push (random.nextInt ());
    stack.push (-1);
    return stack;
}
public void addElements (Vector dest, Vector src)
throws TooLargeException, TooSmallException {
    if (src.size () > dest.size ())
        throw new TooLargeException ();
    if (src.size () < dest.size ())
        throw new TooSmallException ();
    for (int i = 0; i < dest.size (); i++)
        dest.set (i, (Integer) dest.get (i) +
            (Integer) src.get (i));
    if (dest instanceof Stack)
        ((Stack) dest).push (-1);
}
public void test ()
throws TooLargeException, TooSmallException {
    Stack s1 = createStack ();
    Stack s2 = createStack ();
    try {
        addElements (s2, s1);
    } catch (TooLargeException e) {
        s1.setSize (s2.size ());
        addElements (s2, s1);
    }
}
```

Figure 2: Code exercising vector-manipulating methods.

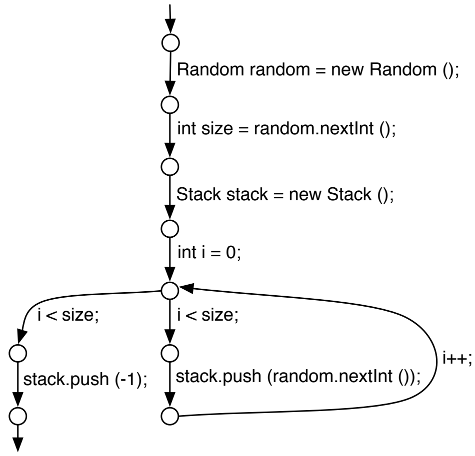

Figure 3: Method model of the `createStack()` method.

2. MINING USAGE MODELS

In order to show, how we mine object usage models from code, we will consider the sample source code shown in Figure 2. We want to create models of objects from the point of view of the methods they occur in. So for instance we create a model of the `random` variable in `createStack()`, or of the `dest` parameter of `addElements()`, etc. More generally, we are going to create models for the following objects:

- Every object that is created (via `new`) in a method—for instance, the variable `random` in `createStack()`.
- Method parameters—such as the `dest` and `src` parameters of `addElements()`.
- Return values—such as the return value of the call to `createStack()` in the `test()` method.
- Exceptions—such as `e` from the catch clause in `test()`.

We will call such objects *abstract objects*.

Our approach takes a set of JAVA classes as input and produces models for all abstract objects existing in methods defined in those classes. This is a two step process. In the first step, we create a model for each method. This model represents the behavior of the method and is similar to a control flow graph. In the second step, we use this method model to construct the set of object usage models of abstract objects occurring in the analyzed method.

2.1 Step 1: Creating a method model

In the first step of the mining process, we create a *method model*. The states of this model are based on *locations in the code,* whereas transitions are labeled with instructions. More precisely, each state corresponds to exactly one instruction and represents the place in the code just before this instruction. For every predecessor of an instruction, there is a transition between the state corresponding to the predecessor and the state corresponding to the instruction.

Additionally, there is single exit state to which all states corresponding to `return` instructions are connected to it with an epsilon transition. The structure of such a model mirrors the code structure and the model itself is closely related to the control flow graph of the analyzed method—with the difference that instructions are labels of transitions instead of states.

A method model is mined on the bytecode level (i.e. with states based on locations in the bytecode and transitions based on bytecode instructions). For the purposes of the presentation, though, we will assume this happens on the source code level. Figure 3 shows the method model constructed by applying the mining algorithm described above to the `createStack()` method code from Figure 2. By convention, we mark the transition to the exit state by a transition without destination state.

In the method model, *exceptions* that may be thrown by the code being analyzed are treated specially. For every instruction in the code, we infer the set of all exceptions that may be thrown as a result of executing this instruction. For example, in case of a method call, we would create a set consisting of all exceptions declared as being thrown by the callee. For each such exception, there are two principal possibilities: either the exception is handled inside the method being analyzed or it is propagated to the caller.

If the exception is handled, the situation is straightforward: exception-handling code is itself represented in the method model, so we just need to add an appropriate transition. In order to recognize a transition as representing an exception being thrown, we annotate its label with the exception's class name.

If the exception is propagated to the caller, we add a new state to the model. In order to distinguish this exceptional state from all others, it is labeled with the exception's class name and represents the exit from the method with the specified exception being thrown. Then we add an appropriate transition to this state, just as in the case of an exception that is handled.

The `test()` method in Figure 2 gives an example of how to treat exceptions. The call inside the `try` block may either return normally or throw one of two exceptions: `TooLargeException` or `TooSmallException`. In the resulting method model, there are exactly three transitions outgoing from the state corresponding to this call. The first one has as its destination the exit state and is labeled just with the method call. The second has as its destination the

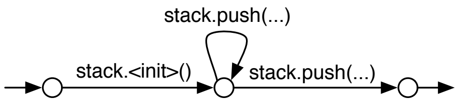

Figure 4: Object usage model of the `stack` variable defined in the `createStack()` method.

state corresponding to the first instruction inside the `catch` block and is labeled with the method call annotated with the name of the exception's class (in this case `TooLargeException`). The third transition has as its destination the state marked with the name of the unhandled exception's class: `TooSmallException`.

2.2 Step 2: Creating an object usage model

In the second step of the mining process, the method model is projected onto the abstract object whose model we want to create. For instance, in case of `createStack()`, we have a method model; now we want to break this down to a separate model for each abstract object *obj* within `createStack()`. This is done by replacing all transitions by *epsilon transitions* that do not call a method on *obj* nor use *obj* as a parameter to some method call.

In order to take local aliasing into account, we apply a data-flow analysis to discover all places in the code where the object might be a target or a parameter of a method call. Labels of the non-epsilon transitions are transformed by replacing each method call with the full name and signature of the method being called. Additionally, if the object of interest was used as a parameter of a method call, the label of the corresponding transition is annotated with the position(s) of this object in the parameter list of the method call. Lastly, in order to reduce the number of states and thus improve comprehension, the epsilon transitions are removed. Such a model, with states based on locations in the code and transitions based on instructions that use the modeled object as an argument or a target of a method call, is called an *object usage model*.

Special attention must be paid to objects that are *cast* in the program. A common pattern is to take an object from a collection or an object resulting from a method call and cast it to several different types depending on the outcome of the `instanceof` operation. What this amounts to is essentially having a set of multiple concrete objects with different types, all of which, however, are represented by the *same abstract object.* Our analysis treats casts accordingly: each cast induces creation of a new object usage model for the abstract object being cast. Each of those models describes usage of the concrete object it represents, including the usage that happens before the object is cast. We found this usage to be actually quite common and it was the main reason we do not treat casts in themselves as inducing abstract objects.

As an example of casts, consider the method `addElements()` from Figure 2. The `dest` variable is an instance of the `Vector` class, but inside the code there is one place where it is cast to the `Stack` class (which is a subclass of the `Vector` class). This means that there are two object usage models created for this variable. The first one models the `dest` variable as a `Vector` only and the second models the `dest` variable as a `Stack`.

To create an usage model for the `stack` variable defined in the `createStack()` method, we transform the method model from the Figure 3 to the model shown in Figure 4. For the purposes of the presentation, the names of the labels have been simplified by removing method signatures and just indicating the number of their parameters using ellipsis instead. In all subsequent models, epsilon transitions will be depicted using dashed lines (with the only exception being the "entry" and "exit" transitions).

Table 1: The pattern $\{$`start()` $\prec$ `stop()`, `lock()` $\prec$ `unlock()`$\}$ is common to three methods, but violated by `get()`

	Temporal property		
Method	`start()` $\prec$ `stop()`	`lock()` $\prec$ `unlock()`	`eof()` $\prec$ `close()`
`get()`	×		×
`open()`	×	×	
`hello()`	×	×	×
`parse()`	×	×	

In practice, all abstract objects in a method share one method model (because they all occur in the same method); the data-flow analysis is conducted only once for each method, simultaneously creating the models for all abstract objects.

3. FINDING ANOMALIES

In this paper, the main application for object usage models is identifying *programming patterns.* We define a programming pattern (or simply *pattern*) as a set of temporal properties over method calls.

As an example of a pattern, consider the set $P = \{$`hasNext()` $\prec$ `next()`, `next()` $\prec$ `hasNext()`$\}$, where $m \prec n$ means that there is a possibility of calling m before calling n (not necessarily directly). This pattern P is exhibited by `iter` object usage model in Figure 1(b).

3.1 Mining programming patterns

To discover patterns such as the one above, we first need to mine *temporal properties* over method calls from the program being analyzed. Initially, we do this on a per-object usage model basis, so that for each object usage model O, we create a *control flow relation* $R(O) = \{(m, n) \mid n \text{ can be called after } m\}$. This relation is by definition transitive. (Note that m and n can be methods of different classes, because object usage models also contain methods called with the modeled object as a parameter.)

Intuitively, control flow relation $R(O)$ for an object usage model O contains all such pairs of method calls (m, n) for which there exists a path through the model O on which m occurs before n (not necessarily directly, i.e. the edges corresponding to m and n do not have to be adjacent).

In the next step, we produce for every method M a set of temporal properties $T(M)$ satisfied by M, defined as $T(M) = \{m \prec n \mid (m, n) \in R(O)$ for some O created by analyzing $M\}$.

After we have created sets of temporal properties over method calls for all methods, we need to mine *common patterns* from those sets. To do this, we use an approach called *frequent itemset mining* [15]. Frequent itemset mining takes as an input a set $S = \{S_1, \ldots, S_k\}$, where each $S_i = \{s_{i_1}, \ldots, s_{i_{m_i}}\}$ is a set of properties, as well as a support threshold *min_support*. It produces as an output a set of patterns $P = \{P_1, \ldots, P_n\}$ that occur in at least *min_support* sets. Formally, if we define $support(T) = |\{S_i \in S \mid T \subseteq S_i\}|$, which is the number of sets of properties that include all properties from pattern T, frequent itemset mining guarantees that $T \in P$ iff $support(T) \geq min\_support$.

Intuitively, support of a pattern is the number of methods that respect that pattern (i.e. exhibit all temporal properties constituting the pattern). Applying frequent itemset mining to the set S containing all $T(M)$ sets will discover a set of all patterns that are exhibited by at least *min_support* methods.

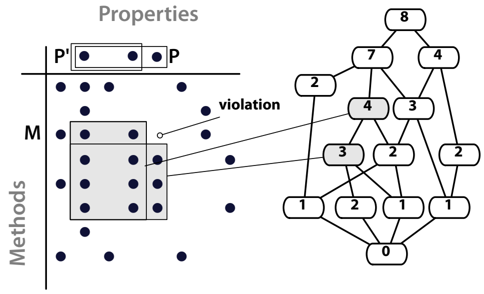

Figure 5: Each pattern corresponds to a block in the cross table over methods and properties. Blocks in turn form a lattice hierarchy. Violations always correspond to neighboring blocks in the hierarchy. The numbers in the lattice denote the support of the corresponding pattern.

However, according to our definition, if we have two patterns P_1 and P_2, where P_2 is a proper subset of P_1 (i.e. each temporal property present in P_2 is also present in P_1, but not vice versa) and yet they both have the same support, both will end up in the final set of patterns. P_2 does not contribute anything to our knowledge, because it occurs only in the same methods in which P_1 does, and so we do not want it to be included in the results. To achieve this, we will restrict ourselves to such patterns, for which every proper superset has less support. Such patterns are called *closed*.

As an example of frequent itemset mining, consider Table 1. In this example, $S = \{T(\textit{get()}), T(\textit{open()}), T(\textit{hello()}), T(\textit{parse()})\}$, where the sets of temporal properties are $T(\textit{get()}) = \{$`start()` $\prec$ `stop()`, `eof()` $\prec$ `close()`$\}$, $T(\textit{hello()}) = \{$`start()` $\prec$ `stop()`, `lock()` $\prec$ `unlock()`, `eof()` $\prec$ `close()`$\}$, and so on. If we assume a support threshold *min_support* $= 2$, frequent itemset mining will produce as an output the set $P = \{P_1, P_2\}$ consisting of two patterns: $P_1 = \{$`start()` $\prec$ `stop()`, `eof()` $\prec$ `close()`$\}$ and $P_2 = \{$`start()` $\prec$ `stop()`, `lock()` $\prec$ `unlock()`$\}$. P_1 has a support of 2 (because only two methods exhibit all its temporal properties: `get()` and `hello()`) and P_2 has a support of 3 (it is exhibited by methods `open()`, `hello()` and `parse()`).

3.2 Detecting anomalies

Frequent patterns minded from code express *normal* object usage. A pattern that is respected by many methods but violated by few represents an *anomaly* and is the focus of our interest.

How do we detect pattern violations? A methods M *violates* a closed pattern P if another (closed) pattern $P' \subset P$ exists such that M respects all properties of P' but not all of P. In Table 1 method `get()` violates pattern $P = \{$`start()` $\prec$ `stop()`, `lock()` $\prec$ `unlock()`$\}$, because $P' = \{$`start()` $\prec$ `stop()`$\}$ is a closed sub pattern of P, for example.

However, not all violation are equally likely to point to real defects. We consider the ratio between the number of methods that respect a pattern P and those that violates it: patterns that are respected by many and violated by few methods are likely to be genuine defects. This ratio is the *confidence* for a violation, defined as $s/(s+v)$ where s is the support of a violated pattern, and v the number of violations. The confidence for `get()` in Table 1 violating pattern P is 3/4 because P has support 3, and P' has support 4, leading to one violation. The support is a value between zero and one, and expressed as a percentage. For practical purposes, only violations exceeding a minimum confidence (parameter *min_confidence*) are considered.

Detecting all violations exceeding a minimum confidence would require to look at all closed patterns in a table like Table 1. Our key insight is that each pattern corresponds a block in the cross table over methods and properties. And a pattern violation corresponds to an imperfect block, like shown on the left of Figure 5. Indeed, such an imperfect block is really a composition of two blocks (or patterns): a slim and tall block, and a wider and short block. They correspond to P' and P in our definition of violations above.

Computing all blocks in a cross table efficiently is provided by Formal Concept Analysis. Blocks form a hierarchy, as shown on the right in Figure 5. Another insight is that imperfect blocks are always formed by two *neighboring blocks* in the hierarchy. Our mining implementation COLIBRI thus computes the concept lattice and inspects all neighboring blocks in order to find all violations [20].

A cross table may have exponentially many patterns. Mining can be still made efficient because support for patterns decreases monotonically in the lattice from the top down. COLIBRI computes only the topmost patterns (or blocks) that both exceed a minimum support (parameter *min_support*) and from these violations that exceed the minimum confidence.

3.3 Ranking anomalies

Anomalies discovered by the process described above are interesting, but there are typically several of them and of course not all of them are true defects. Investigating every single anomaly is time-consuming and would defeat one of the main advantages of our approach: full automatization.

To solve this problem, we developed a method to *rank* anomalies. We wanted our ranking method to favor anomalies that are violations of a *frequently occurring pattern* by a relatively *small number of methods.* We also wanted it to favor anomalies that violate "rare" patterns, i.e. those that are dissimilar from others, because we have noticed that some classes induce many meaningless patterns with many anomalies. This is the case, for instance, for the `StringBuffer` class. A common usage pattern of this class is that we call `append(String)` and then at some later point `toString()` on that same object. This leads to marking the code that calls `append(int)`, but not `append(String)` as anomalous, whereas it is clear that this is definitely not a defect.

To achieve the goals stated above, we assign a number, called *defect indicator*, to each anomaly and then rank them in the descending order of their indicators. The *defect indicator of an anomaly* is defined as $ind = u \cdot s/v$, where s is the support of the pattern being violated, v is the number of methods in which the violation occurs, and u is the so-called *uniqueness factor*. In order to calculate the uniqueness factor, we first have to do the following steps:

- We assign to each pattern P the set of classes $C(P)$, methods called on which are part of the pattern. For instance, in case of a pattern $P = \{X.x() \prec Y.x(), Y.x() \prec Y.y()\}$ we would have $C(P) = \{X, Y\}$.
- We assign to each JAVA class C the number $na(C)$ of anomalies, for which the pattern they violate contains a call to one of the methods of the class C. Formally, $na(C) = |\{A \mid A$ is an anomaly violating a pattern P, where $C \in C(P)\}|$. This tells us the number of anomalies that are related to C.
- We assign to each pattern P the number $np(P)$ being the maximum over $na(C)$'s for all classes $C \in C(P)$. Intuitively, if we consider the class in $C(P)$ that most often occurs in other anomalies (i.e. these anomalies violate patterns containing calls to methods defined by this class), then $np(P)$ tells us the number of those occurrences.

The *uniqueness factor* of an anomaly A is defined as $1/np(P)$, where P is a pattern violated by the anomaly A. This number is equal to 1 if each class from $C(P)$ occurs only in P (indicating uniqueness of P and thus also A). The more anomalies violating patterns having something in common with P, the smaller the uniqueness factor of the anomaly.[1]

4. EXPERIMENTS

We have implemented a prototype called Jadet (for "Java Anomaly Detector") that implements mining of object usage models. To evaluate the effectiveness of Jadet, we have applied it to several complex Java projects (see Table 2): Act-Rbot, which is a cognitive agent based social simulation toolkit/production system[2]; AspectJ, an aspect-oriented extension to the Java programming language[3]; Azureus, a bittorrent client[4]; Columba, an email client[5] and MusiComp, a melody generator[6]. All the experiments were performed under the following conditions:

- We analyzed all methods whose libraries were available. Surprisingly, this was not the case for all methods. In the case of Act-Rbot, the libraries included in the project "jar" file had conflicting versions which prevented us from analyzing thirty methods in one of such libraries. The MusiComp application code contained a call to a method in one of Weka classes that was not defined by that class. In Azureus, we missed the Apple Cocoa-Java classes. In AspectJ, the package `org.aspectj.bea.jvm` was missing.
- A violation is characterized by three numbers: the support of the pattern being violated, the confidence of the violation, and the number of properties missing[7]. We considered a violation as an anomaly if the following three conditions where met: (1) its support exceeded 20, (1) its confidence exceeded 90%, and (3) it did not miss more than two properties. These constants were obtained by empirically testing a few alternatives in order to balance the number of false positives and true negatives, but we did not systematically investigate if these are the optimal values.

4.1 Case study: AspectJ

The largest program in our set is AspectJ, a compiler for the AspectJ language. AspectJ is an extension to the Java programming language making it possible to express cross-cutting concerns that can be later compiled into the bytecode. This is a sufficiently complex, big and mature project to put our technique to a good test.

4.1.1 *Object usage models*

Jadet analyzed 36,044 out of 36,045 methods defined in AspectJ and created 253,084 models in less than 14 minutes on a 1.83GHz Intel Core Duo.

Figure 6 shows a model of a `java.util.Stack` object mined from one of the methods in AspectJ. From this model, we can see that the stack was first created, then elements have been pushed to it in a loop and finally an enumeration of all the elements was

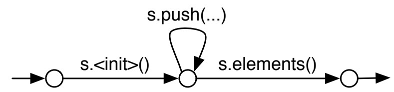

Figure 6: `Stack` model mined from AspectJ.

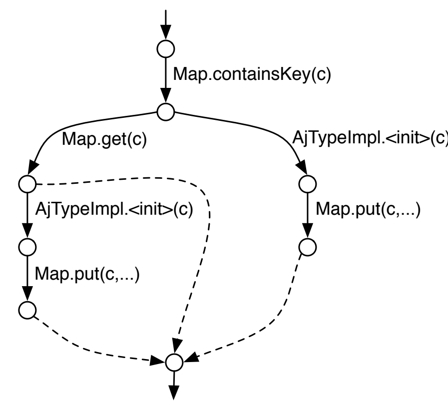

Figure 7: `Class` model mined from AspectJ.

requested. This model illustrates well the difference between approaches that base model states on object states and our approach that bases model states on locations in the code.

Although the first call to `push()` is bound to change the state of the object (from empty to non-empty), it does not result in state-changing transition in our model. This is because there is no difference between the first and the subsequent calls to `push` from the user's perspective. They are all part of a code that puts elements into the stack. If this code were split in, say, two loops, this would be represented in the model by two loops, as well. Similarly, the call to `elements()` that does not change the state of the stack is depicted as a state-changing transition in our model, because the way the stack is used has changed. We no longer put elements into it, but rather we extract them.

Another model is shown in Figure 7. This model does not contain a single call with the object being modeled used as a target. Because of this, limiting model creation to include only such method calls (and not including "external" method calls) would result in an empty model. The model produced by Jadet is quite interesting as it encompasses operations on several different classes in one entity. It shows how AspectJ's internal mapping between Java classes (`java.lang.Class`) and AspectJ type implementations (`AjTypeImpl`) works.

This model has been mined from the method called `getAjType`, which gets an `AjTypeImpl` object corresponding to a given `Class` object (subsequently called `c`). As we can see, the method uses a `Map` object to represent the mapping. It first checks whether the class `c` is already present in the map as a key. If it is not, new type implementation is created and the map is updated (the rightmost path in the model). If the class `c` is present in the map, it is checked whether the corresponding type implementation is up-to-date. If it is not, new type implementation is created and the map is updated (the leftmost path in the model). If the type implementation is up-to-date, nothing else remains to be done (the middle path).

Overall, the mining process resulted in more than 250,000 models being created, and thus we found it impossible to investigate them all. Instead, we looked at a small sample of models mined and selected a number that we found interesting and that were small

[1] A somewhat similar idea of clustering execution profiles based on their similarity to find the faulty ones has been evaluated by Dickinson et al. [10] and found very effective.

[2] http://sourceforge.net/projects/act-rbot/

[3] http://www.eclipse.org/aspectj/

[4] http://azureus.sourceforge.net/

[5] http://columbamail.org/drupal/

[6] http://sourceforge.net/projects/musicomp/

[7] this can be thought of as the width of the hole in the block

Table 2: **Details of the JADET case study subjects**

Program	# Classes	# Methods		# Models	# Patterns	# Anomalies	Mining time	
		Analyzed	Total				Models	Anomalies
ACT-RBOT 0.8.2	2,492	26,501	26,531	187,137	436	192	9:03	0:53
ASPECTJ 1.5.3	2,957	36,044	36,045	253,084	666	276	13:19	3:47
AZUREUS 2.5.0.0	3,585	22,359	22,367	157,313	1244	365	7:28	0:50
COLUMBA 1.2	1,165	6,894	6,894	54,371	97	64	2:06	0:11
MUSICOMP 1.0 beta	347	2,078	2,079	17,321	3	3	1:02	0:04

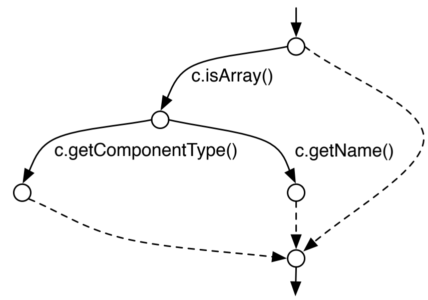

Figure 8: `Class` **model mined from ASPECTJ.**

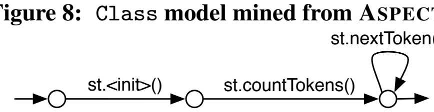

Figure 9: `StringTokenizer` **model mined from ASPECTJ.**

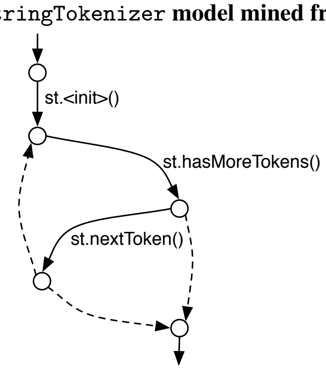

Figure 10: `StringTokenizer` **model mined from ASPECTJ.**

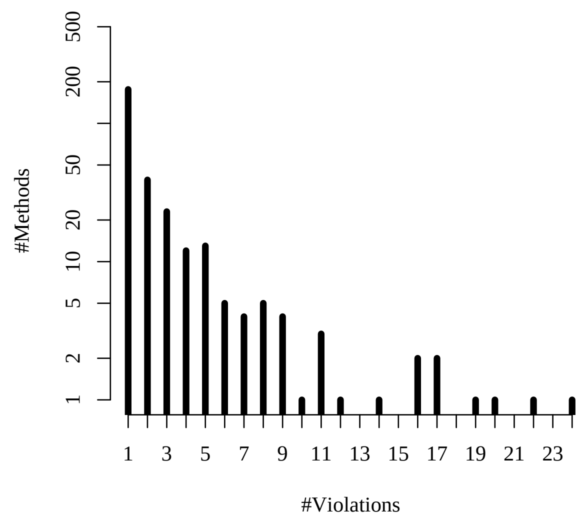

Figure 11: Frequency distribution of a number of violations exhibited by a single method in ASPECTJ.

enough to present (see also Figures 8, 9 and 10). Due to the fact that our small sample already contained a large fraction of interesting models, we are confident that the remaining set contains many more models that could be suitable for program understanding.

4.1.2 Programming patterns and anomalies

Mining programming patterns, finding violations, filtering and ranking anomalies for ASPECTJ took less than 4 minutes and reported 276 anomalies. Those anomalies encompassed 790 violations, occurring in 295 methods (recall that one anomaly may consist of several violations, each in a different method; also, one method may contain several violations). Figure 11 shows the frequency distribution of a number of violations exhibited by a single method.

We inspected these 790 violations manually. Two of them were defects, five were code smells (meaning any program property that indicates something may go wrong [14]) and 84 pointed to code that could be improved with respect to readability and/or maintainability (we will call such results "hints"). Five violations have not been classified, because the source code of the corresponding methods is not part of the ASPECTJ, even though the methods themselves are. The remaining 694 violations were false positives.

Our ranking method proved successful. The Top 10 anomalies (see Table 3) encompassed 15 violations, out of which two were defects (ranked 1st and 7th), three were code smells, one was a hint and nine were false positives (we actually examined eleven anomalies as the Top 10, because anomalies in the 10th and 11th position had identical defect indicators). It is important to notice that both defects and three out of five code smells were ranked in the Top 10. This reduces the impact of the large number of false positives. In practice, a programmer could spend time only on highly ranked violations and still hope to find the most interesting ones.

Let us now take a closer look at the defects that we have found. Figure 12 shows part of the method that contained the first defect (#165631 in the ASPECTJ bug database). The loop in this code processes *only the first element* returned by the iterator, even though it should process all of them. The pattern that is violated in this case is $P = \{\texttt{hasNext()} \prec \texttt{hasNext()}, \texttt{hasNext()} \prec \texttt{next()}\}$; and it is the property $\texttt{hasNext()} \prec \texttt{hasNext()}$ which is missing in the faulty method.

Table 3: **Top 10 anomalies for ASPECTJ.**

Rank	Defect ind.	Support	Uniq. factor	Anomaly type
1	174.25	697	0.250	**defect**
2	174.00	696	0.250	code smell
3	25.00	50	0.500	hint
4	20.00	20	1.000	false positive
5	19.67	59	1.000	3 × false positive
6	19.00	38	0.500	false positive
7	14.87	119	0.125	**defect**
8	13.00	26	1.000	2 × code smell
9	12.00	24	1.000	2 × false positive
10	11.50	23	0.500	false positive
10	11.50	23	0.500	false positive

```
private boolean verifyNIAP (...) {
    ...
    Iterator iter = ...;
    while (iter.hasNext ()) {
        ... = iter.next ();
        ...
        return verifyNIAP (...);
    }
    return true;
}
```

Figure 12: A defect in ASPECTJ. The loop body is executed at most once.

```
public void visitCALOAD (CALOAD o) {
    Type arrayref = stack().peek(1);
    Type index = stack().peek(0);
    indexOfInt(o, index);
    arrayrefOfArrayType(o, arrayref);
}
```

Figure 13: Another defect in ASPECTJ. The method should check the array type, and call `constraintViolated()` if the check fails.

The second defect (#41069 in the BCEL bug database[8]) is much more subtle. It occurs in the method `visitCALOAD()`, shown in Figure 13. Here, $P = \{$`stack()` $\prec$ `constraintViolated()`, `stack()` $\prec$ `stack()`$\}$ is the pattern that is violated (all of the methods were defined in the `InstConstraintVisitor` class). The task of `visitCALOAD()` is to check whether a given `CALOAD` bytecode instruction is legal. This instruction is part of a family of array loading instructions (`AALOAD`, `BALOAD`, `CALOAD`, etc.), each of which takes an array and an index from the stack and pushes back the value contained in the array under the given index. These instructions all differ by the expected type of the elements in the array; for `CALOAD`, for instance, this is `char`.

In order for the instruction to be legal, two conditions must be satisfied: The first element on the stack must be an array of appropriate type and the second element on the stack must be an integer. If these conditions are not satisfied, the method is required to call the `constraintViolated()` method.

The faulty method checks the second condition and the first half of the first condition, i.e. it checks whether the first element on the stack is an array, but not if the array is of an appropriate type. It does this using two helper methods which perform the actual check and call `constraintViolated()` themselves if the check fails. However, neither helper checks whether the array is of an appropriate type. Thus, `visitCALOAD` method itself would have to do the check and, in case it fails, call `constraintViolated()`. And this is this call that JADET discovered to be missing.

[8]BCEL is used as part of ASPECTJ and the defect is located in a method defined by one of BCEL's classes

```
public String getRetentionPolicy () {
    ...
    for (Iterator it = ...; it.hasNext();) {
        ... = it.next();
        ...
        return retentionPolicy;
    }
    ...
}
```

Figure 14: A code smell in ASPECTJ. The loop body is executed at most once—but this is not a defect, since the object iterated upon has at most one element, too.

One should note that the first defect could have been found by tools that check for code violating a fixed set of rules, because the pattern here is very general, and constitutes an anomaly in almost every JAVA program. Finding the second defect, though, requires *domain knowledge* that cannot be coded in a universal rule. Thus, to find the defect, it is necessary to infer domain-specific patterns first, as exemplified by our approach.

Let us now take a look at the code smell ranked in the 2nd position. It is of interest to us, because it has a very high defect indicator (nearly as high as the defect on the 1st position) and yet is not a defect itself. The code of the anomalous method is shown in Figure 14. It is very similar to the code of the defect shown in Figure 12 and, indeed, the pattern that is violated here is also very similar: $P = \{$`hasNext()` $\prec$ `hasNext()`, `hasNext()` $\prec$ `next()`, `next()` $\prec$ `hasNext()`, `next()` $\prec$ `next()`$\}$. The last two properties are missing in the anomalous method, i.e. it is not possible to call `next` twice on the same iterator object.

This time, however, this is not a defect. The collection through which the code iterates is a list of so-called *retention policies* associated with an annotation. Each annotation may *have at most one* retention policy associated with it, and so the collection holds at most one element. This is not documented in the code; it is clear that a simple `if` instead of a `for` loop should have been used.[9]

4.2 More experiments

Table 2 lists all programs we have tried JADET on. In each case, we looked at the Top 10 anomalies and classified them manually.[10] The summary of these experiments can be found in Table 4. Note that each anomaly may encompass more than one violation (as there

[9]The acute reader may have noticed that the patterns in this example and in the first defect are different, when in fact they should not be. The reason for this is impreciseness of the dataflow analysis. The `for` loop in the code smell we have just examined is enclosed in another loop and the iterator "reaches" the `hasNext` instruction of the next iteration of the outer loop just before it is "killed". We plan to solve this problem as part of the future work on increasing the precision of the analysis.

[10]The ACT-RBOT program contained libraries it used included in its own "jar" file instead of them being packaged in different "jar" files. As a result, only five of Top 10 anomalies were anomalies in the program itself and these were the only ones we examined.

Table 4: A classification of the top 10 anomalies for the case study subjects. In all investigated programs, JADET makes suggestions that improve code quality.

Program	# Defects	# Code smells	# Hints	# False positives
ACT-RBOT 0.8.2	2	0	13	4
ASPECTJ 1.5.3	2	3	1	9
AZUREUS 2.5.0.0	1	2	6	4
COLUMBA 1.2	0	0	3	17
MUSICOMP 1.0 beta	0	0	4	6
	5	**5**	**27**	**40**

may be many methods that violate the same pattern in the same way) and thus Top-10 anomalies may in fact mean "15 violations", as in the case of ASPECTJ. As can be seen, JADET is most effective on large programs, resulting in a large number of models. This is expected, because in order to find patterns (and thus be able to find violations), there must be enough data to support them. Also, our approach benefits from stability and consistency of the source code that makes the defects "stick out".

Our results show that JADET can find new defects in software, even in mature code like ASPECTJ. The second defect in ASPECTJ could not have possibly been found by any other existing tool; as illustrated in the examples, the main strength of our approach lies in it being able to construct *project-specific* patterns and violations thereof.

Additionally, our approach is fully automatic; the programmer is only required to take a look at the found anomalies and to decide whether they are real defects or not (admittedly, she has to be competent enough and the time needed to classify anomalies is highly dependent on the level of competencee; this problem, however, is shared by all unsound approaches to defect detection). In particular, she does not have to specify patterns nor in any way configure the tool to fit to the particular project that is to be analyzed. JADET is also very practical, having small execution times and a ranking system in place, thus freeing the user from the burden of examining all anomalies to select the most interesting ones.

4.3 Threats to validity

Out of five defects we have found in the test programs, four have been confirmed by the tool developers. The only defect that has not been confirmed is the second defect in ASPECTJ. The reason for this is that even though we found this defect by analyzing ASPECTJ, this is a defect in BCEL and we submitted the bug report to the BCEL bug database. The project, however, is in a stagnation phase right now and bug reports are not being processed. Nevertheless, we created a small, incorrect `.class` file that passes verification by BCEL because of the issue we have discovered, so we know this is a real defect, even though we do not have any confirmation from the developers.

The main threat to the validity of our results are errors in the code that creates object usage models that are used as a basis for generating patterns. We have spent a long time on checking for correctness of the models that have been created for various programs. We have also created programs for which we constructed models by hand and then checked whether those created by JADET were identical. In the process we have found defects in the model-creating code and those defects have been subsequently removed. We assume that any remaining defects (and imprecisions induced by not having points-to analysis) affect only a very small number of models.

Another threat is that we have investigated only five programs and it is possible that our results do not generalize. We have tried to mitigate this risk by choosing programs that have very different purposes and including two quite big, commonly used projects as subjects: ASPECTJ and AZUREUS.

Other potential problems include defects in frequent itemset mining or in the code that looks for violations and ranks anomalies. We think such errors are very improbable: the code that does frequent itemset mining and violations extraction is mature and has already been used several different projects and none of these projects discovered any defects in it. The code that ranks violations, on the other hand, is very simple and well tested, so that we expect it to be correct as well.

5. RELATED WORK

5.1 Mining temporal program behavior

Mining object usage models was inspired by the work of Eisenbarth et al. on object process graphs [11]. Object usage models are similar to their object process graphs, but we focus on modeling objects from the point of view of single methods, without using expensive points-to analysis, and thus our approach scales to large programs.

Dynamic analysis has also been used to produce program models. Dallmeier et al. [7] use so-called inspectors to discover state of the object at runtime. Lorenzoli et al. [23] use anonymous states and transitions described by method calls annotated with parameters. Quante and Koschke [26] create dynamic object process graphs and Xie et al. [34] create abstract-object-state machines with states being described by possible return values of method calls. These approaches capture only true program behavior instead of a conservative approximation, but this also means that they need extensive test cases to produce at least somewhat complete results and automatic defect detection possibilities are limited.

Other researchers used approaches such as grammar learning [3] or based on model checking [2, 16, 21]. These techniques are not fully automatic. Ammons et al. [3] require manual annotations to relate functions to objects, while Alur et al. [2] and Henzinger et al. [16] rely on initial predicates to be able to get the first abstraction of the program. Liu et al. [21] require specifying sets of functions as input and find instantiations of six categories of rules over those functions.

The PERRACOTTA tool of Yang and Evans [36] also mines temporal rules of program behavior. Their approach can only discover behavior that fits into templates (such as alternation) provided by the user. Williams and Hollingsworth [33] mine software repositories to find function usage patterns where one function is directly called after another one (perhaps conditionally), which is more limited than our approach. Some research has also been done in the area of supporting programmers by providing them with examples of usages of a particular API. Tools that address this problem include MAPO by Xie and Pei [35], PROSPECTOR by Mandelin et al. [24] and XSNIPPET by Sahavechaphan and Claypool [29].

Cook and Wolf [6] have written the seminal work on inferencing finite-state machines from event sequences, where they compare Markov methods, neural networks, and grammar inference as means to construct models; their work applies on software *development processes,* though.

All of the approaches that *mine* temporal properties produce representations that can also be used as *specifications.* In addition, there are also approaches that suggest various formalisms for specifications of temporal behavior. These approaches include the *type-*

state concept by Strom and Yemini [30], *protocols* by Yellin and Strom [37], *interface automata* by de Alfaro and Henzinger [1] and the idea of treating objects as regular *processes* presented by Nierstrasz [25].

5.2 Automatic defect detection

Finding patterns was inspired by the PR-MINER tool of Li and Zhou [19] who use frequent itemset mining to find sets of functions, variables and data types that frequently appear together. In contrast, our approach takes *ordering* into account and is thus able to find defects (like the first defect in ASPECTJ) that could not be found using PR-MINER.

In a recent work, Ramanathan et al. [27] describe CHRONICLER, a tool for mining precedence relations among procedure calls. Their approach is interprocedural and can thus find patterns spanning multiple functions, but is restricted to "must precede" relations (in contrast to our "can precede") and is thus suited to finding a different class of defects than our method. CHRONICLER is also object-insensitive.

Our ranking method for anomalies was inspired by clustering algorithm used by Dickinson et al. [10] to find faulty executions based on their profiles. Dallmeier et al. [8] used differences in sequences of method calls between passing and failing runs to detect defects, but they use dynamic analysis (so that they need a test that fails in order to find a defect) and their granularity (finding defective classes) is larger than ours (finding defective methods). Weimer and Necula [31] learn pairs of matching functions (like `open` and `close`) from method-call traces and look for violations of these pairings in error-handling code only.

Livshits and Zimmermann [22] use software repositories to mine coding patterns and look for their violations in the DYNAMINE tool. In order to find a pattern, DYNAMINE needs to have it added to the repository after the first revision and be confirmed using dynamic analysis afterwards. Kim et al. [18] developed BUGMEM, using fixes mined from software repositories to construct patterns of defects and their fixes. Both approaches require extensive history of software revisions in repository to be effective.

Yang and Evans [36] also focused on finding defects, but their approach is limited to finding behavior that fits into templates provided by the user. Engler et al. [12] also used such a template-based approach. Hovemeyer and Pugh [17] created FINDBUGS, which statically looks for a priori specified bug patterns.

Fink et al. [13] look for violations of a specification given as a typestate, so they can mark a sequence of method calls as faulty only if it was specified as erroneous by the typestate. Reiss [28] developed the CHET system that allows programmers to specify the way a component should be used and checks these specifications. The SLAM tool by Ball and Rajamani [5] uses a model checker to validate temporal safety properties. DeLine and Fähndrich [9] created FUGUE, a tol that allows for specifying typestates for objects and checking the code for conformance to those typestates. Antoy and Hamlet [4] instrument code defining abstract data types to check operations for conformance with an algebraic specification. In each of those cases, the specification has to be entered manually.

6. CONCLUSIONS AND CONSEQUENCES

Our experiments with JADET have successfully demonstrated that our approach can effectively detect new defects in software, even in mature code like ASPECTJ. The approach is fully automatic, requires few resources, and scales up to very complex systems. As it leverages existing code to find anomalies, it automatically adapts to the project conventions at hand. JADET is thus complementary to existing approaches that focus on single methods or check universal programming rules; as a side effect, it promotes consistency in object usage across the product code.

JADET is light-weight and yet effective, given the large percentage of true defects and code smells among the top ranked anomalies. We expect an even higher detection rate in earlier software production stages.

Despite these early successes, we still see much room for improvement. Our future work will focus on the following topics:

Improved code analysis. JADET has a number of limitations in its current implementation when it comes to analyzing code. It does not analyze the whole JAVA language: multithreading and reflection are not supported and instructions that belong to either of these categories are silently ignored during analysis. Fields are always treated as holding an unknown value. Furthermore, JADET does not implement points-to analysis. Addressing these issues would improve accuracy of usage models and can easily be overcome by augmenting the tool. However, this may hurt our goal of creating a tool that is practical and scalable to large programs; and it is not certain whether defect detection would improve as well.

Improved patterns. The patterns we are currently using reflect *only a small part of the knowledge* present in models. In particular, we do not make a distinction between normal and exceptional returns; we also abstract away most of the structural information present in the models. For instance, our patterns can not distinguish between some set of methods being called a fixed number of times and the same set being called in a loop, even though this information is present in the object usage models.

Negative examples. Another possible enhancement is extending the definition of the pattern to include not only information about things that are expected to happen, but also information about things that are expected *not* to happen. As a simple motivational example, consider a class that expects its clients to call one of many "initializer" methods before performing any other operation. This typically means that clients are allowed to call *exactly one* of these methods. Expressing this fact in patterns would allow us to discover cases where more than one "initializer" method is called.

Portable models. Right now, identifying patterns and models requires an existing code base that is "mostly correct". In practice, it could therefore be helpful to reuse patterns from existing projects. While this is straight-forward for projects that use the *same* classes directly, we are investigating measures that would account for usage of *similar* classes, as well as *indirect* class usage.

In general, we find that existing software encodes lots of knowledge that can and should be leveraged for improving quality. We hope that object usage models can make a small, but distinct contribution for leveraging this knowledge.

For future and related work regarding object usage models, see

`http://www.st.cs.uni-sb.de/models/`

Acknowledgments. Martin Burger, Valentin Dallmeier, Stephan Neuhaus, Rahul Premraj, David Schuler, and Thomas Zimmermann, as well as the anonymous reviewers, provided helpful and constructive comments on earlier revisions of this paper. Andrzej Wasylkowski is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

7. REFERENCES

[1] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proc. of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[2] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05: Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109. ACM Press, 2005.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02: Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.

[4] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.

[5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proc. of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[6] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[7] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA '06: Proc. of the Fourth International Workshop on Dynamic Analysis*, pages 17–24. ACM Press, 2006.

[8] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In A. Black, editor, *ECOOP '05: European Conference on Object-Oriented Programming*, 2005.

[9] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP '04: European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*. Springer, 2004.

[10] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE '01: Proc. of the 23rd International Conference on Software Engineering*, pages 339–348. IEEE Computer Society, 2001.

[11] T. Eisenbarth, R. Koschke, and G. Vogel. Static object trace extraction for programs with pointers. *Journal of Systems and Software*, 77(3):263–284, 2005.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, 2001.

[13] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proc. of the International Symposium on Software Testing and Analysis*, pages 133–144. ACM Press, 2006.

[14] M. Fowler. *Refactoring. Improving the design of existing code*. Addison-Wesley, 1999.

[15] J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition, 2006.

[16] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In Wermelinger and Gall [32], pages 31–40.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[18] S. Kim, K. Pan, and J. E. E. James Whitehead. Memories of bug fixes. In *FSE-14: Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–45. ACM Press, 2006.

[19] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In Wermelinger and Gall [32], pages 306–315.

[20] C. Lindig. Mining patterns and violations using concept analysis. Technical report, Saarland University, Software Engineering Chair, Germany, June 2007. Avaliable from `http://www.st.cs.uni-sb.de/publications/`; the software is available from `http://code.google.com/p/colibri-ml/`.

[21] C. Liu, E. Ye, and D. J. Richardson. LtRules: an automated software library usage rule extraction tool. In *ICSE '06: Proc. of the 28th International Conference on Software Engineering (tool demonstrations)*, pages 823–826. ACM Press, 2006.

[22] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In Wermelinger and Gall [32], pages 296–305.

[23] D. Lorenzoli, L. Mariani, and M. Pezzè. Inferring state-based behavior models. In *WODA '06: Proc. of the Fourth International Workshop on Dynamic Analysis*, pages 25–32. ACM Press, 2006.

[24] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 48–61. ACM Press, 2005.

[25] O. Nierstrasz. Regular types for active objects. In *OOPSLA '93: Proc. of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15. ACM Press, 1993.

[26] J. Quante and R. Koschke. Dynamic object process graphs. In *CSMR '06: Proc. of the 10th European Conference on Software Maintenance and Reengineering*, pages 81–90. IEEE Computer Society, 2006.

[27] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE '07: Proc. of the 29th International Conference on Software Engineering*, pages 240–250. IEEE Computer Society, 2007.

[28] S. P. Reiss. Specifying and checking component usage. In *AADEBUG '05: Proc. of the Sixth International Symposium on Automated and Analysis-Driven Debugging*, pages 13–22. ACM Press, 2005.

[29] N. Sahavechaphan and K. Claypool. XSnippet: mining for sample code. In *OOPSLA '06: Proc. of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 413–430. ACM Press, 2006.

[30] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[31] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In N. Halbwachs and L. D. Zuck, editors, *TACAS '05: Proc. of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.

[32] M. Wermelinger and H. Gall, editors. *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. ACM Press, 2005.

[33] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *MSR '05: Proc. of the 2005 International Workshop on Mining Software Repositories*, pages 1–5. ACM Press, 2005.

[34] T. Xie, E. Martin, and H. Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE '06: Proc. of the 28th International Conference on Software Engineering*, pages 835–838. ACM Press, 2006.

[35] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *MSR '06: Proc. of the 2006 International Workshop on Mining Software Repositories*, pages 54–57. ACM Press, 2006.

[36] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proc. the 28th International Conference on Software Engineering*, pages 282–291. ACM Press, 2006.

[37] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.