Mining Operational Preconditions

Andrzej Wasylkowski
Dept. of Computer Science
Saarland University, Saarbrücken, Germany
wasylkowski@cs.uni-sb.de

Andreas Zeller
Dept. of Computer Science
Saarland University, Saarbrücken, Germany
zeller@cs.uni-sb.de

Abstract

A procedure's client must satisfy its precondition*—that is, reach a state in which the procedure may be called. Preconditions describe the state that needs to be reached, but not how to reach it. We use static analysis to infer the sequence of operations a variable goes through before being used as a parameter: "In* `parseProperties(String xml)`*, the parameter* `xml` *normally stems from* `getProperties()`*." Such* operational preconditions *can be learned from code examples and checked to detect anomalies. Applied to* AspectJ*, our* OP-Miner *prototype found 288 violations of operational preconditions, uncovering 9 unique defects and 48 unique code smells.*

1. Introduction

When using a function, a client must ensure that the function's *precondition* is satisfied—the condition that has to be met before its execution. Even simple functions can have surprisingly complex preconditions. The AspectJ method `reapPropertyList(List list)`, for instance, takes a list, as indicated by the parameter type. In order to function properly, though, `list` must be nonempty, its first element must be a class, and subsequent elements must be objects whose node class is equal to the class being the first element of the list. Using JML specification syntax, this precondition reads as:

```
static List ASTNode.reapPropertyList(List list)
@requires list.size() >= 1
@requires list.get(0) instanceof Class
@requires \forall int i; 0 < i && i < list.size();
  list.get(i) instanceof StructuralPropertyDescriptor &&
  ((StructuralPropertyDescriptor)list.get(i)).
     getNodeClass() == list.get(0)
```

This is an example of an *axiomatic* precondition, which is the base of several verification and validation approaches. While it describes the precise state of the program and the method's parameters, it does not tell *how to achieve this state*: Where does `list` come from? How do we construct it such that the precondition is satisfied?

To answer such questions, programmers usually refer to *usage examples* in existing code. In the case of `reapPropertyList()`, we can examine one of its callers, say, `getPropertyList()`, to learn that the `list` in question would be constructed from a set of properties. Without explicitly stating the `reapPropertyList()` precondition, the code in Figure 1 shows *how* to meet it.

```
public List getPropertyList (Set properties) {
    List list = new ArrayList ();
    createPropertyList (this.cl, list);
    Iterator iter = properties.iterator ();
    while (iter.hasNext ()) {
        Property p = (Property) iter.next ();
        addProperty (p, list);
    }
    reapPropertyList (list);
    if (list.size () == 0)
        Debug.log ("Empty property list");
    return list;
}
```

Figure 1. Sample usage of `reapPropertyList`.

Looking for such examples and extracting what *needs* to be done for the precondition to be satisfied is difficult and error-prone. We therefore introduce the concept of *operational preconditions* specifying *how* to satisfy a function's requirements. Operational preconditions come in the form of sets of *sequential constraints,* expressing control and data flow through function calls. Figure 2 shows the operational precondition for the `list` parameter of `reapPropertyList()`, as extracted from the calling function `getPropertyList()`. After the constructor and the initialization via `createPropertyList()`, further properties are added with `addProperty()` before the actual call. The goal of this research is to mine operational preconditions from code in order to detect their *violations.*

We have implemented a tool called OP-Miner that learns operational preconditions and checks them against violations. In seven open source programs OP-Miner detected

```
list.<init>() < createPropertyList(...,list)
list.<init>() < addProperty(...,list)
list.<init>() < reapPropertyList(list)
createPropertyList(...,list) < addProperty(...,list)
createPropertyList(...,list) < reapPropertyList(list)
addProperty(...,list) < addProperty(...,list)
addProperty(...,list) < reapPropertyList(list)
```

Figure 2. Operational precondition for the `list` parameter of `reapPropertyList()`, where $e_1 \prec e_2$ means "e_1 precedes e_2"

115 violations that were defects or code smells. These violations include seven previously unknown AspectJ defects, two of which lead to crashes.

This paper is organized as follows. We first give a high-level summary of our technique, followed by detailed descriptions of all steps involved (Section 2). Our experiments with OP-Miner are reported in Section 3. After discussing the related work (Section 4), Section 5 closes with conclusions and consequences.

2. Mining Operational Preconditions

We have implemented the OP-Miner tool that learns and checks operational preconditions of methods by static program analysis. The basic idea is as follows: *If a method is called sufficiently often, we can inspect all its callers and deduce from them the operational preconditions of the method.* More specifically, the process of learning operational preconditions and detecting their violations consists of the following steps:

- OP-Miner takes a program as its input. For each statically identifiable object in that program, OP-Miner creates a set of *sequential constraints* that characterize the way this particular object is being used. (See Section 2.1.)
- For each method called in the program OP-Miner looks for sets of sequential constraints that are *common to many objects* passed as actual parameters to that method. Each such set forms an *operational precondition*; Figure 2 shows an example. (See Section 2.3.)
- OP-Miner looks through all the objects used as actual parameters and identifies those that *violate* the operational preconditions found. These are reported to the user for investigation. (See Section 2.4.)

2.1. Creating Sequential Constraints

The first step in learning operational preconditions and detecting their violations is creating sequential constraints for all statically identifiable objects in the program. Our analysis is intraprocedural, so each method is analyzed separately and each has its own set of statically identifiable objects it uses. By a statically identifiable object we mean the following: *formal parameters* of methods (including the implicit `this` parameter), *objects created* via `new`, *return values* of method calls (as in `x = map.items()`), values read from *fields* (including static fields, as in `x = System.out`), and explicit *constants* (such as `null` and `"OK"`).

Every statically identifiable object participates in some operations performed by the method that uses it. For example, if `x` is a formal parameter of `foo` (and therefore a statically identifiable object associated with `foo`) and `foo` calls `bar(x)`, then calling `bar` is an *event* associated with `x`, because `x` is being passed as an actual parameter to `bar`. An event associated with an object is one of the following:

- a method call (including constructor calls) with the object being used as the *target* or a *parameter* (possibly in multiple positions), e.g., `x.bar(y, z)` is an event associated with `x`, `y`, and `z`.
- a method call with the object being the value that was *returned,* e.g., `x = map.items()` is an event associated with `x`.
- field access with the object being the value that was *read*, e.g., `x = System.out` is an event associated with `x`.
- a comparison of a return value of a method with a boolean or integer *constant*, e.g., `list.size() == 0` is an event associated with `list`. (Actually, two events associated with `list` stem from this expression: the call to `size()` and the comparison of the return value of that call with 0.)

Consider the code shown in Figure 1. There are six events associated with the `list` object:

- e_1 — the `ArrayList` constructor call
- e_2 — the call to `createPropertyList()`
- e_3 — the call to `addProperty()`
- e_4 — the call to `reapPropertyList()`
- e_5 — the call to `ArrayList.size()`
- e_6 — comparing `ArrayList.size()` with 0

A *sequential constraint* associated with an object is a pair of events associated with this object, in which the first event precedes the second one (not necessarily directly, i.e., there may be other events that happen in between). The precedence relation is based on control flow, i.e., if there is a flow of control from an event e_1 to e_2, we say that e_1 *precedes* e_2, denoted as $e_1 \prec e_2$. Using the above-enumerated events associated with the `list` object, we can construct many sequential constraints from them. One example is

$e_2 \prec e_4$, another is $e_3 \prec e_3$ (because it is possible to call `addProperty()` multiple times). In general, sequential constraints for an object represent an *ordering of events* that are associated with this object.

The component of OP-MINER that extracts sequential constraints was adapted from the JADET tool [33]. One of the most important innovations we have introduced for the purpose of mining operational preconditions is that a comparison of the value returned from a call with a constant is treated as a separate event. This allows us to not only represent the fact that, for example, `Iterator.next()` is called after `Iterator.hasNext()`, but also that for this to happen `hasNext()` must have returned `true`. Another important OP-MINER feature is that it keeps track of the position of the parameter being used when passing an object to a method. While JADET could only report that "An object that was used in a call to `createPropertyList()` was later used in a call to `addProperty()`", OP-MINER extends this to "An object that was used *as the first parameter* in a call to `createPropertyList()` was later used *as the second parameter* in a call to `addProperty()`". This allows OP-MINER to distinguish between objects being passed to the same methods, but as different parameters.

2.2. Fine Points of the Analysis

Before we further describe how OP-MINER learns operational preconditions from sequential constraints, we would like to make a slight digression and present some of the challenges involved in mining OPs. We will also explicitly list trade-offs we made to get a scalable and practical implementation.

Usage across method boundaries. One of the potential problems that we have to solve when trying to specify an operational precondition of a method is that the OP may cross methods' boundaries. Consider again the example shown in Figure 1. Just by looking at `getPropertyList()`, we were able to discover the operational precondition of `reapPropertyList()`. But what would happen if `getPropertyList()` was split in two parts? To discover the association between them, we would need interprocedural analysis. Our analysis is, however, *intra*procedural. The drawback is of course that we lose information. On the other hand, though, our approach scales very well (see Section 3) and we avoid potential problems with too many aliases causing imprecision and resulting in too vague OPs.

Choosing the right granularity. `addProperty()` and `createPropertyList()` are part of an operational precondition of `reapPropertyList()`. But which other methods are being called by `createPropertyList()`? Should its callees be part of the operational precondition? In fact, deciding on the right *granularity*—considering a method as an atomic operation or as the union of its callees—can be difficult. OP-MINER's granularity is fixed: The OPs of a method contain only operations that are performed directly by the caller of the method. This is on purpose, as we find that very often methods operate on a specific abstraction level and we want our OPs to express operations on that level, too.

Aggregating usage alternatives. As there are *multiple ways a precondition can be satisfied*, there can be multiple operational preconditions. One example is the `JPanel` class, which is a part of the JAVA SWINGGUI framework. Each panel is associated with a layout manager, which is by default a flow layout manager. If the user wants the panel to use another manager, this can be either given when constructing the `JPanel` object or set later using the `setLayout()` method (see Figure 3). Both are equivalent and each constitutes a separate operational precondition for methods that require the panel to have a non-default layout manager. OP-MINER is fully capable of creating multiple OPs for a single formal parameter of a single method.

```
public void createGUI () {
    ...
    JPanel panel1 = new JPanel (new BorderLayout ());
    panel1.add (checkbox, BorderLayout.NORTH);
    ...
    JPanel panel2 = new JPanel ();
    panel2.setLayout (new BorderLayout ());
    panel2.add (textarea, BorderLayout.WEST);
    ...
}
```

Figure 3. Setting a layout manager.

No primitives. We focus *on objects only*, ignoring primitive values. We have experimented with operations on primitive values being part of OPs. For instance, if a primitive value was an actual parameter of a method, we would include operations on it as part of the method's OPs. However this caused a too large drop of precision and resulted in many sequential constraints of the form "first add two numbers and then multiply them", which were not helpful.

2.3. Learning Operational Preconditions

After sequential constraints for all objects in the program have been created, OP-MINER can start learning operational preconditions for all methods called in that program. For a given method M OP-MINER looks for sets of sequential constraints that are *common to many objects* passed as actual parameters to M. Each such set forms an operational precondition. Thus, each OP not only represents *what* needs

to be done, but also *in what order*, because sequential constraints represent ordering among events. However, since operational preconditions are supposed to describe what needs to be done *before* passing an object to M, sequential constraints where one of the events happens *after* M has been called can be filtered out as irrelevant. This is because if an event happens after the call, it cannot possibly be a factor influencing the correctness of the call.

After this preliminary filtering comes the time for actual learning. OP-Miner needs to decide whether remaining sequential constraints associated with the actual parameters of M are really relevant or just *noise*—that is, infrequent usages that are particular to specific callers. Our solution to this problem is as follows: If a particular set of sequential constraints occurs frequently (i.e., at several call sites to M), then all elements in this set are relevant and, in fact, constitute an OP of M. For example, if there are many calls to `reapPropertyList(list)`[1], but before only a few of them `adjustPropertyList(list)` is called, the sequential constraint `adjustPropertyList(list)` $\prec$ `reapPropertyList(list)` will occur rarely and thus will be discarded as not being part of true OPs. This learning by eliminating noise is done by *formal concept analysis.*

Concept analysis is, broadly speaking, a technique for finding *patterns* [16]. It takes as its input a set of conceptual objects, a set of conceptual properties, and a cross table associating objects with properties. Figure 4 shows a sample cross table with rows being conceptual objects and columns being conceptual properties. Concept analysis produces all concepts found in the cross table, where a concept is a set of objects A and a set of properties B such that every object in A is associated with all properties from B and sets A and B are *maximal,* i.e., it is not possible to add elements to one set without influencing the second one. Intuitively, a concept is a rectangle (not necessarily contiguous) in the cross table: Figure 4 shows two such rectangles.

We use the Colibri tool [23] for formal concept analysis, with conceptual objects being statically identifiable objects; and conceptual properties being sequential constraints. For each method called in the program we create a separate cross table such that we can focus only on objects passed to that method. We also introduce a parameter called *minimum support*, which puts a lower boundary on the number of conceptual objects in a concept. This makes sure that Colibri reports sets of sequential constraints that are common to at least *minimum support* many statically identifiable objects passed to the method; it thus eliminates noise caused by infrequent usage. The resulting sets are operational preconditions of that method. Figure 2 shows the operational precondition for the `reapPropertyList()` method used by the code in Figure 1.

[1]Of course the names of the objects being passed are irrelevant, but we give them for clarity.

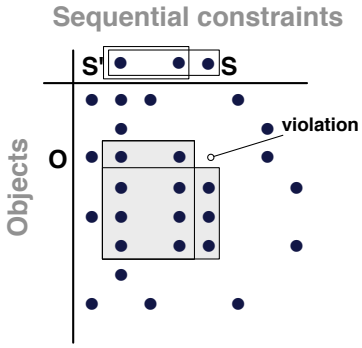

Figure 4. Detecting violations via concept analysis. Each rectangle corresponds to an operational precondition; gaps indicate potential violations [23].

2.4. Finding Violations

As its last step, OP-Miner looks through all the objects used as actual parameters and identifies those that violate the operational preconditions found. Consider the code shown in Figure 1 and the OP of the `reapPropertyList()` method as shown in Figure 2. It is clear that the `list` object satisfies the OP. But if, say, a call to `createPropertyList()` was missing, we would say that `list` violates the OP. The same if this call happened for example after all the calls to `addProperty`. Generally, we can say that *if there are sequential constraints that are present in the OP, but missing in the object passed to the method, the object violates the OP*.

However, we need to weaken the condition given above a bit. After all, if the OP is violated by many objects, it is more probable that the OP is too restrictive than that all those objects are in an incorrect state when being passed to the method. We deal with this by introducing a parameter called *minimum confidence*, which is a number between zero and one. We calculate the confidence of a potential violation using the formula $c = s_{OP}/(s_{OP} + s_{PV})$ where c is the confidence, s_{OP} is the support of the OP (i.e., the number of objects that adhere to this OP) and s_{PV} is the support of the potential violation (i.e., the number of objects that violate the OP in the same, particular way). Just as we used minimum support to ensure that only frequently occurring sets of sequential constraints get reported as OPs, we use minimum confidence to ensure that only infrequently occurring divergences from OPs get reported as real violations.

To detect violations in the way described above, we again use Colibri [23]. Detecting violations is equivalent to detecting "gaps" in the concept analysis cross table. In Figure 4, we can see that the object O violates the OP represented by the set of sequential properties S, because it contains only the subset S' of those properties. We can also say that the confidence of this violation is 0.75, because the

support of the OP S is 3 (the set S is common to three objects, so $s_{OP} = 3$) and there is only one violation of the OP that consists of elements in S' only (so $s_{PV} = 1$). Thus, $c = s_{OP}/(s_{OP} + s_{PV}) = 3/(3 + 1) = 0.75$. Detecting violations in this way was introduced by Lindig [23] and used in our previous work on detecting object usage anomalies [33]. In the context of this paper, the important thing is that we can find both OPs and their violations in a way that is both effective and efficient.

3. Evaluation

To evaluate the effectiveness of OP-Miner, we have applied it to several complex Java projects (see Table 1): Act-Rbot, a cognitive agent based social simulation toolkit, Apache Tomcat, a servlet container, ArgoUML, a UML design tool with cognitive support, AspectJ, an aspect-oriented extension to the Java programming language, Azureus, a bittorrent client, Columba, an email client, and jEdit, a programmer's text editor. All the experiments were performed under the following conditions:

- We analyzed all classes that truly belong to the project[2] and all methods whose libraries were available.[3]
- A potential violation is characterized by two numbers: the *support* of the operational precondition being violated and the *confidence* of the violation. We considered a potential violation as a real one and thus worth reporting only if the following two conditions were met: (1) its support was at least 20 and (2) its confidence level was at least 0.9. 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.

Table 1 contains a list of our case study subjects, including their size (number of classes, number of methods) and a short summary of our results (number of methods for which operational preconditions have been deduced, number of violations found and time needed to perform the analysis).

3.1. 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 define crosscutting concerns that can be later compiled into the bytecode. AspectJ is a sufficiently complex, big and mature project to put our technique to a good test.

OP-Miner reported 288 violations of operational preconditions in AspectJ. Each reported violation contained the following information: the operational precondition being violated, the object that violates the precondition and the set of sequential constraints that are missing (i.e., needed to satisfy the operational precondition). It is important to notice that OP-Miner not only reports violations, but also shows what is missing; if the violation is indeed a defect, it shows how to fix it. We inspected those 288 violations manually and classified them into four categories:

Defects. This category is self-explanatory, but there is one important point we want to make here. It sometimes happens that there is a method that violates the contract of its base class, but the application itself does not fail because of this. However, if the method is public, we still mark it as defective, because someone may cause it to fail by following the contract to the letter.

Code smells. This category contains all violations that are not defects, but the violating methods have properties indicating that something may go wrong [15] or they might be improved in a way that improves readability, maintainability or performance of the program. An example might be a method that uses a `for` loop to iterate through a collection and breaks unconditionally out of the first iteration. If the collection can have at most one element, this code will work, but it cannot be treated as fully correct.

False positives. This category contains all violations that are neither defects nor code smells.

Out of the 288 violations reported for AspectJ, we categorized 9 as defects, 48 as code smells and the rest as false positives. This means that 57 or 20% of the violations are worth investigating. Also, all violations are unique, i.e., each object violating an operational precondition appears only once. We achieved this by automatically removing all duplicates and retaining only one violation for each object. We frequently observed that all violations by the same object are more or less equivalent and thus fall into the same category, i.e., all are either defects or code smells or false positives. As a consequence, we have found 9 *unique* defects and 48 *unique* code smells in AspectJ; we applied the same duplicate removal for our other subjects.

Let us present some of the most interesting violations found. Out of nine defects found by OP-Miner in AspectJ, two are severe enough to cause a compiler crash (reported as bugs #218167 and #218171 in the AspectJ bug database and corrected after our report). Both are simple typos occurring in two different methods; both violate the operational pre-

[2] For example, we ignored third-party libraries in the Act-Rbot jar file.

[3] Surprisingly, this was not the case for all methods. In Apache Tomcat, there is a reference to a non-existent method from JavaMail library. In Azureus, the Apple Cocoa-Java classes were not available for analysis. In the AspectJ compiler, the package `org.aspectj.bea.jvm` was missing.

Table 1. Details of the OP-MINER case study subjects

Program	Origin	# Classes	# Methods		# Methods with OPs	Total time
			Total	Analyzed		
ACT-RBOT 0.8.2	www.acis.nl/researchdocs	344	3 401	3 401	81	3:24
APACHE TOMCAT 6.0.16	tomcat.apache.org	1 295	15 178	15 177	137	4:02
ARGOUML 0.24	argouml.tigris.org	1 653	12 123	12 123	174	5:17
ASPECTJ 1.5.3	www.eclipse.org/aspectj	2 957	36 045	36 044	347	9:21
AZUREUS 2.5.0.0	azureus.sourceforge.net	3 585	22 367	22 359	160	5:22
COLUMBA 1.2	www.columbamail.org/drupal	1 165	6 894	6 894	73	1:43
JEDIT 4.2	www.jedit.org	641	4 327	4 327	48	1:11

```
for (Iterator it = c1.iterator(); it.hasNext();) {
    E e1 = (E) it.next();
    ...
    for (Iterator it2 = c2.iterator(); it.hasNext();) {
        E e2 = (E) it2.next();
        ...
    }
    ...
}
```

Figure 5. In ASPECTJ, an inner loop checks the iterator of the outer loop.

```
protected void checkcast (int baseId) {
    this.countLabels = 0;
    if (classFileOffset + 2 >= bCodeStream.length) {
        resizeByteArray();
    }
    ...
    writeUnsignedShort (...);
    ...
}
```

Figure 6. A code smell in ASPECTJ: The `resizeByteArray()` callee is implemented twice with the same code in the same class hierarchy.

conditions of the `Iterator.next()` method. The skeleton of the defective code is shown in Figure 5.

Four of the defects are located in methods that violate the contract of the methods they override. All of them take a progress monitor instance as one of the parameters; in all cases, the overridden method says this instance may be `null`. One of the operational preconditions of `IProgressMonitor.done()` states that the monitor should be the return value from the factory method `Policy.monitorFor(IProgressMonitor)`. It turns out that this very method handles `null` by returning an instance of the class `NullProgressMonitor`. This solves the problem of the monitor being `null`. Since the defective methods do not have this call and do not check explicitly for `null` (which would be an acceptable, but inferior alternative), they throw a `NullPointerException` exception if they are called with `null` as the progress monitor, in spite of the correctness of such a call in itself.

Figure 6 shows an example of a violation marked as a *code smell.* The `checkcast()` method is correct, but one of its callees might cause maintenance problems in the future. OP-MINER reported this method because it violates the operational precondition of the `writeUnsignedShort()` method, which states that `resizeByteArray()` should be called first. At the first sight this looks like a defect in OP-MINER, but a closer look reveals that there are two identical implementations of `resizeByteArray()` in two classes, where one class inherits from another. Because the method in the base class is private, it cannot be called by `checkcast()`, so there is a second implementation provided. Moreover, both copies of `resizeByteArray()` can do their work correctly, because fields they access are public! So the implementation hides the method in order to prevent it being abused, but allows access to the fields to everyone. This code does not fail, but its design is dubious.

Analyzing ASPECTJ from creating sequential constraints to discovering violations took less than 10 minutes on a 1.83 GHz Intel Core Duo machine.[4] We have investigated all 288 violations manually, looking at each of them and deciding whether the violation is a defect, a code smell or a false positive. The categorization process took us altogether about twelve hours, which is about two and a half minutes per violation on average, including, if needed, the creation of test cases or code changes that would trigger the defect. We were able to dismiss many violations as false positives quickly thanks to OP-MINER giving us rough information on what the fix should be (the missing sequential constraints). On the other hand, some violations were difficult enough to require half an hour or more of investigation. When we were unsure if the code was correct and were unable to trigger incorrect behavior, we classified it as correct and the violation as a false positive to make sure that our reported precision rate is not an overestimation of what an expert in the project would achieve.

[4] The component we adapted from JADET was optimized, which is the reason why the times are much better than reported for JADET alone [33].

Table 2. Summary of the results for the experiment subjects. (See Section 3.2 for a discussion.)

Program	# Violations		# Defects	# Code smells	# False positives	Efficiency
	Total	Investigated				
ACT-RBOT 0.8.2	42	42	4	15	23	**45%**
APACHE TOMCAT 6.0.16	42	42	0	5	37	**12%**
ARGOUML 0.24	185	26	1	6	19	**27%**
ASPECTJ 1.5.3	288	288	9	48	231	**20%**
AZUREUS 2.5.0.0	189	107	1	13	93	**13%**
COLUMBA 1.2	34	34	2	9	23	**32%**
JEDIT 4.2	8	8	0	2	6	**25%**
	788	**547**	**17**	**98**	**432**	**21%**

```
public JStatusBar () {
    ...
    JPanel rightPanel = new JPanel ();
    rightPanel.setOpaque (false);
    rightPanel.add (resizeIconLabel,
        BorderLayout.SOUTH);
    ...
}
```

Figure 7. A defect in COLUMBA. Using border layout constants for this particular `JPanel` instance is wrong. The code does not crash just because `JPanel` ignores its misuse.

3.2. Other Case Study Subjects

In addition to ASPECTJ, we have applied OP-MINER to several other projects (see Table 1). For those other projects, we only investigated a limited number of top-ranked violations[5]. A summary of the results of those investigation can be found in Table 2. For each of the projects, we present the total number of violations, the number of violations that we actually investigated, the number of defects, code smells and false positives found by investigating them and the efficiency, i.e. the percentage of violations that were defects or code smells.[6] Overall, 21% of all violations we have investigated were defects or code smells.

Let us take a closer look at some of the defects we found in programs other than ASPECTJ. Figure 7 shows a constructor in the COLUMBA code. This constructor is defective because it uses border layout constants when adding components to the panel, which uses flow layout. The code works, however, because flow layout ignores constants it receives,

```
public String getPreferredEmail () {
    Iterator it = getEmailIterator ();
    IEmailModel = (IEmailModel) it.next ();
    ...
}
```

Figure 8. This COLUMBA code misses a call to `hasNext`, which can cause this method to throw an exception.

as it does not expect any. How did OP-MINER detect this defect? It turns out that one of the operational precondition of the two-parameter version of the `add()` method is that the `JPanel` is created using a default constructor and that the `setLayout()` method is called afterwards.[7] Since there is a call to the two-parameter version of the `add()` method and `JPanel` is created using its default constructor, what is missing is the call to `setLayout()`. Two other defects in COLUMBA and AZUREUS are shown in Figure 8 and in Figure 9.

3.3. Limitations and Threats to Validity

The most important limitation of our approach is that it needs substantial code bases to learn from. While this limitation can be partially circumvented (e.g. if one wants to use some library and wants OP-MINER to check if one is not making any mistakes, one can use someone else's program to learn from), it is an unavoidable price for the ability to tap into developers' knowledge and experience that is contained in those code bases. Also, OP-MINER is only useful for single-threaded programs, but it can handle the whole JAVA language, including exception handling. Another lim-

[5] We have used a slightly changed version of the ranking system used by JADET [33]

[6] In case of ARGOUML, we also found 15 violations in generated code. We decided to not classify these and to ignore them when calculating the efficiency.

[7] Another, equally good, operational precondition is that the layout is specified as a parameter to the `JPanel`'s constructor; this one, however, is not applicable here, because the default constructor has been used instead; note how one precondition—having a non-default layout—is equivalent to several operational preconditions.

```
protected void loadPluginList () {
    ...
    List bits = new ArrayList ();
    while (...) {
        ...
        if (...) {
            bits.add (...);
            break;
        }
        else {
            bits.add (...);
            ...
        }
    }
    String version = (String) bits.get (0);
    String cvs_version = (String) bits.get (1);
    String name = (String) bits.get (2);
    ...
}
```

Figure 9. This AZUREUS code does not check the size of the `bits` list before accessing its elements. This defect is now fixed.

itation is the abstraction we use, namely sequential constraints: these are not only context-insensitive (so cannot express the fact that for example `pop()` should not be called more times than `push()` was) but also weaker than a regular language, for example they cannot distinguish between a method being called two times in a row and multiple times in a loop.

We have identified the following threats to validity:

- We have investigated seven programs with different application domains, sizes and maturity and our results seem fairly consistent across those programs. However, it is possible that they do not generalize to arbitrary projects; proprietary, closed-source programs may have very different properties.
- The tools we have used (JADET and COLIBRI) could be defective. We think this is very improbable, especially for COLIBRI, whose implementation is publicly available [23]. As for JADET as well as the OP-MINER code, we have used and thoroughly validated it, so we believe that any defects left affect only a small number of violations and thus do not spoil the results overall.
- The results of the categorization process performed on violations might depend on the expertise of the human applying the approach. However, if anything, this would make our results *better* than reported—because we have marked violations as defects only if we were completely sure that they are indeed defects (e.g. by crashing the program, making sure the contract was violated, seeing the code changed in the way suggested by OP-MINER, etc.). An experienced developer may spot potential problems where we see false positives.

4. Related Work

To the best of our knowledge, the present work is *the first to take an operational view at preconditions*—learning and checking what needs to be done to call a function. However, there are many other approaches that learn from existing code or that detect defects.

4.1. Learning from Code

Ernst et al. [13] have written the seminal work on inferring invariants dynamically using DAIKON. All invariants are of course axiomatic in our sense and thus incomparable with operational preconditions. Ramanathan et al. [28] produce axiomatic preconditions, unordered usage information ("this value was also used as a parameter of the following functions: ..."), origin information and constraints on method calls of the form "a call to `g` is always preceded by a call to `f`". However, these constraints are "must" as opposed to ours "may" and are created separately from the static information mentioned earlier. The upshot of this is that the interplay between methods that can be represented is more limited than what OPs can represent. They used their approach to find defects, too, but unfortunately did not report on the rate of false positives.

There are many approaches to modelling behavior, including the seminal work by Cook and Wolf [7], which however applies to software development process and not the software itself. Some are based on static analysis [11, 30, 32], other on dynamic analysis [8, 17, 18, 27, 36], grammar learning [4] or model checking [2, 3, 19, 24]. They allow one to understand how the program, class or objects of a particular class behave or should be used, but their focus is on *entities* instead of *operations*. They can thus learn which sequences of method calls are correct and which are not, but not what to do to call a particular method.

The PERRACOTTA tool by Yang and Evans [38] mines temporal rules of program behavior. Their approach can only discover behavior that fits into templates (such as alternation) provided by the user. These templates are limited to occurrences of method calls and are thus weaker than our sequential constraints that contain precise information about the flow of objects between parameters of different methods. Williams and Hollingsworth [35] mine software repositories to find function usage patterns where one function is directly called after another one (perhaps conditionally), which is again 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 [26, 31, 37]. These have different focus than OP-MINER: MAPO by Xie and Pei [37] provides sample sequences of method calls, but does not relate them to objects. This provides information about the order in

which the methods are typically called, but not how objects flow through them. PROSPECTOR by Mandelin et al. [26] and XSNIPPET by Sahavechaphan and Claypool [31] can only provide information on how to create an object of a given type. This means that if a method does not return any value or returns a value of a frequently occurring type (like an integer or `Object`), they cannot learn anything about how it is typically used.

4.2. Automatic Defect Detection

There is an abundance of work on automatic defect detection and we can just present several examples. Some approaches are equipped with fixed lists of "defect patterns" and check the code looking for places where these are violated [12, 20]. These can be quite effective and can be tweaked to include only those patterns, violations of which are with high probability defects. This source of strength is also their main limitation: the patterns are project-independent and even though theoretically it is possible to create project-dependent patterns, this would mean a lot of work that cannot be reused in other projects.

A lot of work has been done on detecting code that violates a specification given a priori. Testing [5], model checking [6, 29], and static analysis [10, 14] have been used for that purpose. These approaches are typically very effective and precise when looking for violations, but the specification has to be provided by the user and this is the main weakness. Creating specifications for project-specific classes is a lot of work that cannot be reused in other projects. Also, these approaches can find code that is incorrect, but not one that is unnatural and can cause maintenance problems.

Another active research area, which is also the most related to our work, is on approaches that learn rules from code [22,33], traces [9,34] and software repositories [21,25] and then check the program for conformance with these rules. PR-MINER [22] uses frequent itemset mining to find sets of functions, variables and data types that frequently appear together. OP-MINER is stronger than PR-MINER in that it learns proper sequencing of calls, not just their occurrence and this allows us to detect a much broader range of defects, e.g, the two defects that crash ASPECTJ could not be detected by PR-MINER, because calls to both `hasNext()` and `next()` are present; it is just that they do not occur when they should.

Learning from dynamic traces [9,34] and software repositories [21, 25] can be pretty effective, but requires more than just code, i.e., good tests that exercise the program as much as possible or an extensive history of software revisions in the repository. The latter is typically not available for new projects and the former guarantees accuracy, but at the cost of not covering the whole code base and being highly dependent on the quality of the test suite.

In own earlier work, we have created JADET, a tool for detecting object usage anomalies [33]. JADET focuses on *whole usage patterns* instead of preconditions and focuses only on methods, in calls to which the object participated, without taking the information about it being a parameter, etc. into account. In contrast, OP-MINER learns and checks as *which specific parameter the object was used.* Thus, OP-MINER can distinguish between objects passed to the same methods, but as different parameters, and provide a much more fine-grained picture. Furthermore, JADET handles *conditional* preconditions dependent on the return value of earlier methods (such as `next()` being called only if `hasNext()` returns `true`).

There are also other differences: OP-MINER checks each object passed to the callee for being a violation and JADET looks only at callers as a whole. This means that if some caller contains more than one call to a specific callee, and at least one of them is correct and one incorrect, JADET would not be able to find a defect, because the correct usage would overshadow the incorrect one. The two defects that crash ASPECTJ are in methods that contain two usages of an iterator, with one being correct and another one not; neither of these could be detected by JADET.

5. Conclusions and Consequences

In modern object-oriented programs, most complexity stems not from within the methods but from method compositions. OP-MINER both addresses and leverages this complexity. It learns from actual method usage and detects deviant method compositions, thus automatically discovering a surprisingly high number of both subtle and blatant defects in well-tested production code.

As OP-MINER learns from code, it automatically adapts to the project conventions at hand, making it orthogonal to approaches that check for fixed patterns. The approach is lightweight and easily scales to large bodies of code.

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

Procedural languages. Parameters are a part of every programming language, and therefore our approach easily extends to other languages. C and C++ programmers, for instance, could benefit from operational preconditions such as "For `close(int fd)`, the parameter `fd` normally stems from a call to `open()` and is used in calls to `read()` and `write()` before the call to `close()`." The challenge, of course, is to develop and apply appropriate static analysis tools.

Interprocedural analysis. Right now, our analysis detects anomalies only within procedures—it is an intraprocedural analysis. One may argue that going interproce-

dural would give further advantages, such as considering what happens inside a function called. Unfortunately, breaking this abstraction barrier brings risks in our context. As an example, consider the code

```
Thing t = ThingFactory.makeThing();
doSomething(t);
```

where `makeThing()`, among others, eventually invokes the `Thing` constructor—which we can find out via interprocedural analysis. Thus, we could make the `Thing` constructor an operational precondition for `doSomething()`. This would be an error, though, because the point of `ThingFactory` is to use only `ThingFactory` for creating `Thing` objects. Breaking the abstraction barrier via interprocedural analysis brings opportunities, but also risks, and our future research will investigate when and how to use it.

Usage abstractions. Our current model of operational preconditions requires specific sequences of events, including method calls. However, anomalies in parameter usage can only be detected if there are sufficiently many "normal" instances to learn from—and there are many ways to achieve the state required by a function. We want to search for *abstractions* that callers may have in common, even if the parameters provided have different sequential constraints associated with them. Possible abstractions include common "deep" callees, as inferred from interprocedural analysis, or common type transformations.

Ranking violations. Most methods only have a small number of callers to learn from. The resulting low support is not so much a challenge for *detecting* OP violations, but becomes a challenge when *ranking* violations—in other words, telling programmers which violations to focus upon first, and thus reducing the number of false positives. We expect usage abstractions, as discussed above, to provide higher support and thus assist in ranking violations effectively.

Early programmer support. Once mined, operational preconditions can be easily integrated into the programming environment, making recommendations on how to compose method invocations—and how to avoid errors. Likewise, operational preconditions can become part of the documentation, pointing programmers to functions that may be relevant for their task.

All in all, existing code examples form a resource that is still seldom tapped for program analysis. As long as there are more good examples than bad examples to learn from, we can leverage the "good" code to identify the most blatant "bad" code. The fact that such learning and detecting can be done automatically, as realized in the OP-Miner tool, should bring some promise and relief.

For future and related work regarding OP-Miner and mining object usage, see

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

Acknowledgments. Irina Brudaru, Martin Burger, Valentin Dallmeier, Michael D. Ernst, Yana Mileva and Venkatesh Prasad Ranganath provided helpful and constructive comments on earlier revisions of this paper. Patrick Cousot and Michael D. Ernst sparked inspiring discussions. Andrzej Wasylkowski is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

References

[1] *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*, 2005.

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07*, pages 25–34, 2007.

[3] 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, 2005.

[4] 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, 2002.

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

[6] 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, 2001.

[7] 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.

[8] 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, 2006.

[9] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP '05*, 2005.

[10] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP '04*, volume 3086 of *Lecture Notes in Computer Science*, 2004.

[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, 2001.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[14] 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, 2006.

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

[16] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin – Heidelberg – New York, 1999.

[17] C. Ghezzi, A. Mocci, and M. Monga. Efficient recovery of algebraic specifications for stateful components. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 98–105, 2007.

[18] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP '03*, volume 2743 of *Lecture Notes in Computer Science*, pages 431–456, 2003.

[19] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE-13* [1], pages 31–40.

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

[21] 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, 2006.

[22] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13* [1], pages 306–315.

[23] 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/.

[24] 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, 2006.

[25] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13* [1], pages 296–305.

[26] 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, 2005.

[27] J. Quante and R. Koschke. Dynamic protocol recovery. In *WCRE '07: Proc. of the 14th Working Conference on Reverse Engineering*, pages 219–228, 2007.

[28] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *PLDI '07: Proc. of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 123–134, 2007.

[29] 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, 2005.

[30] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of uml sequence diagrams. In *PASTE '05: Proc. of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 96–102, 2005.

[31] 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, 2006.

[32] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proc. of the 2007 international symposium on Software testing and analysis*, pages 174–184, 2007.

[33] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07*, pages 35–44, 2007.

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

[35] 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, 2005.

[36] 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, 2006.

[37] 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, 2006.

[38] 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.