# Object Usage: Patterns and Anomalies

Andrzej Wasylkowski

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten der
Universität des Saarlandes

Saarbrücken, 2010

Printed 14 September 2010

| | |
|---|---|
| *Day of Defense* | 13 September 2010 |
| *Dean* | Prof. Dr. Holger Hermanns |
| *Head of the Examination Board* | Prof. Dr. Reinhard Wilhelm |
| *Members of the Examination Board* | Prof. Dr. Andreas Zeller |
| | Prof. Dr. Sebastian Hack |
| | Dr. Gordon Fraser |

*To Łucja, with love*

# Contents

# List of Figures

# List of Tables

# Abstract

Using an API often requires following a protocol—methods must be called in a specific order, parameters must be appropriately prepared, etc. These requirements are not always documented, and not satisfying them almost always leads to introducing a defect into the program. We propose three new approaches to help cope with this problem:

- We introduce the concept of so-called *object usage models*, which model how objects are being used. We show how to efficiently *mine* object usage models from a program.
- We show how to use object usage model to find *patterns* of object usage and *anomalous* object usages. We have implemented the technique in a tool called "JADET" and evaluated it on six open-source projects. JADET was able to find insightful patterns, and had found defects and code smells in all six projects. In total, JADET found 5 defects and 31 code smells.
- We introduce the concept of *operational preconditions*. Traditional preconditions show the *state* an object must be in before being used as a parameter. Operational preconditions show *how* to achieve that state. We have created a tool called "Tikanga" that mines operational preconditions as temporal logic ($CTL^F$) formulas. We have applied Tikanga to six open-source projects, and found 12 defects and 36 code smells. This is the first time that specifications in the form of temporal logic formulas have been fully automatically mined from a program.

# Zusammenfassung

In vielen Fällen erfordert die Verwendung einer Programmbibliothek das Einhalten eines Protokolls: Methoden dürfen nur in einer bestimmten Reihenfolge aufgerufen werden und Parameter müssen im richtigen Zustand übergeben werden. Derartige Anforderungen sind nur selten dokumentiert, obwohl eine Nichtbeachtung häufig einen Fehler im Programm verursacht. In dieser Arbeit stellen wir drei neuartige Ansätze zur Lösung solcher Probleme vor:

- Wir präsentieren *Objektverwendungsmodelle*, eine neue Art, die Verwendung eines Objektes in einem Programm zu charakterisieren und zeigen, wie solche Modelle effizient aus Programmen gelernt werden können.
- Wir zeigen, wie man Objektverwendungsmodelle einsetzen kann, um Verwendungsmuster zu lernen und Stellen zu finden, an denen Objekte auf ungewöhnliche Art verwendet werden. In einer Evaluation mit sechs quelloffenen Programmen war unser Prototyp JADET in der Lage, 5 bisher unbekannte Fehler und 31 Stellen schlechten Programmierstils in allen sechs Programmen zu finden.
- Wir führen eine neue Art von Vorbedingungen für den Aufruf von Methoden ein. Herkömmliche Vorbedingungen zeigen, in welchem Zustand ein Objekt sein muss, um als Parameter für einen Methodenaufruf verwendet zu werden. Im Gegensatz dazu zeigen die hier vorgestellten *operationalen Vorbedingungen*, wie der benötigte Zustand erreicht wird. Unser Prototyp "Tikanga" lernt operationale Vorbedingungen und repräsentiert sie als temporallogische ($CTL^F$) Formeln. Wir haben Tikanga auf sechs quelloffene Programme angewendet, und dabei 12 Fehler und 36 Stellen schlechten Programmierstils identifiziert. Unser Ansatz ist der Erste, der vollautomatisch Spezifikationen in der Form von temporallogischen Formeln aus einem Programm lernt.

# Acknowledgments

First and foremost, I thank my adviser Andreas Zeller for supporting me for the last five years and for teaching me many of the things I learned while working on my PhD. A big thank you also goes to Sebastian Hack for being my second examiner and for sharing with me with some insights into research while in Dagstuhl in 2010. Part of my doctoral studies was financially supported by a research fellowship of the DFG Research Training Group "Performance Guarantees for Computer Systems"; I thank all the people that made it possible.

While working on my PhD I had the pleasure of working with excellent people. I thank Christian Lindig for answering my many questions during the first months of my work, and for a lot of help in moulding some of the ideas I had into the research that ultimately became this dissertation. I had a lot of helpful discussions with Valentin Dallmeier. Valentin also proofread this dissertation and translated its abstract into German. Thank you! I thank Michael Ernst for the discussions we had while Michael was in Saarbrücken on a sabbatical. I thank Kim Herzig, Sebastian Hafner, Christian Holler, and Sascha Just for maintaining the infrastructure at our chair. Without them I would never be able to do some of the research I did. A special thank you goes to Yana Mileva, my office mate through most of my PhD time.

My family provided a lot of help and support, without which I would not be able to accomplish what I did. I thank my parents, Grażyna and Stefan, for always believing in me and for all the help and support that actually made this PhD possible. My son, Julian, made a day without a smile impossible. Last, but not least, I thank my wife, Łucja, for her love and support, and for believing in me even in the most difficult times. I dedicate this work to her.

# Chapter 1

# Introduction

If a programmer wants to use the API, she has to use it correctly—and this typically means making sure that the API is used the way it was intended to be used. Since a typical API consists of a number of functions (or classes/methods in object-oriented languages), this boils down to knowing how to combine these functions to accomplish the task that the programmer is interested in. If the API is not documented in any way, this is a very difficult task. However, even if documentation is available, the task can still be difficult, as the documentation can be outdated, incomplete, difficult to understand, or—as is often the case when natural language is being used—ambiguous. The best solution would be having an up-to-date, complete and easy-to-understand documentation combined with formal specification of the API (to resolve any ambiguity issues), but this is a standard that will not be achieved for a long time, if ever, especially considering how difficult it is to write a formal specification even for simple functions. Programmers typically try to cope with this problem by consulting code examples, where the API they are interested in is actually used. However, there is no guarantee that these examples are actually correct, and the programmer—trying to get to know how to use the API—is not in a position to decide if the code uses the API correctly or not. As a result, it is possible that the code written by the programmer will turn out to be defective, too, and the program will fail during testing (which is the optimistic scenario) or at a client's site. In any case, the program will have to be fixed, and the costs of fixing the program are larger the later in the development cycle the defect is found (Dunn 1984).

In this dissertation we will present a set of approaches that are designed to help the programmer use the API correctly, by providing her with the following information:

```
public List trimPropertyList (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 () == 1)
        Debug.log ("Empty property list");
    return list;
}
```

Figure 1.1: Sample source code using iterator and list-operating API.

**Models of API usage.** We introduce the concept of *object usage models* that show how objects are being used in a program. Object usage models are finite state automata (FSAs) that show which events (typically method calls) can follow which other events, and how the object being modeled participates in the events. We show how object usage models can be fully automatically mined from a given program. (See Chapter 2). As an example, consider the method shown in Figure 1.1. One of the objects used in this method is the iterator `iter`. Its object usage model is shown in Figure 1.2.

**API usage patterns.** We introduce the concept of *sequential constraints* and patterns consisting thereof. Sequential constraints represent sequencing of events related to one object, and patterns are frequently occurring sets of sequential constraints that represent sequencing of events related to one or multiple object. We show how we can abstract object usage models into sets of sequential constraints, and how we can find sets that occur frequently and thus form patterns. (See Chapter 3). For example, the object usage model shown in Figure 1.2 can be abstracted into a set of sequential constraints such as:

```
RETVAL: Set.iterator ≺ Iterator.hasNext @ (0)
Iterator.hasNext @ (0) ≺ Iterator.next @ (0)
...
```

If we analyze a program where many iterators are being used, we will come up with the following frequently occurring pattern:

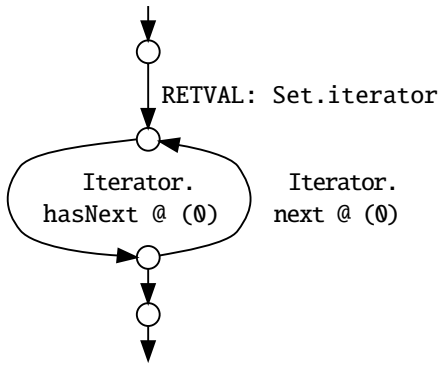



Figure 1.2: Object usage model of the `iter` object from Figure 1.1.

```
Iterator.hasNext @ (0) ≺ Iterator.hasNext @ (0)
Iterator.hasNext @ (0) ≺ Iterator.next @ (0)
Iterator.next @ (0) ≺ Iterator.hasNext @ (0)
Iterator.next @ (0) ≺ Iterator.next @ (0)
```

This pattern states that iterators are used by calling `hasNext()` before calling `next()` and vice versa, and this is indeed how iterators should be used.

**Operational preconditions.** We introduce the concept of *operational preconditions*. They are akin to traditional preconditions, but instead of saying *what* the state of a parameter needs to be for a function call to be correct, operational preconditions say *how* to achieve that state, thus helping the programmer to actually write code that correctly uses a function. Operational preconditions are expressed as temporal logic formulas, and this is the first time temporal logic specifications are fully automatically inferred from programs. (See Chapter 4). As an example, the operational precondition of the first parameter (the list) of the `reapPropertyList()` method used by the code in Figure 1.1 contains formulas such as:

$$\mathrm{AG}\,\texttt{ASTNode.createPropertyList @ (2)}$$
$$\mathrm{AG}\,(\texttt{ASTNode.createPropertyList @ (2)} \Rightarrow \mathrm{EX}\,\mathrm{EF}\,\texttt{ASTNode.addProperty @ (2)})$$

The first of those formulas states that the list must always be passed as the second parameter to `createPropertyList()`, and the second that

after passing the list as the second parameter to `createPropertyList()`, there should exist a path where the list is passed as the second parameter to `addProperty()`.

**Potential defects in code.** We show how programs can be checked for conformance with API usage patterns and operational preconditions found earlier. Our approach is fully automatic and results in a user being given a ranked list of methods that violate a pattern or an operational precondition, respectively. We show that it is effective in finding previously unknown defects in existing programs. (See respective sections in Chapters 3 and 4).

Code that uses an iterator and does not call `hasNext()` before calling `next()` (i.e., violates the iterator pattern shown above) will get reported to the user as potentially defective. Likewise code that calls `reapPropertyList()`, but only calls `createPropertyList()` conditionally (i.e., violates the operational precondition shown above).

## 1.1 Publications

This dissertation builds on the following papers (in the chronological order):

- **Wasylkowski, Andrzej**. 2007. Mining object usage models. In *ICSE COMPANION 2007: Companion to the proceedings of the 29th International Conference on Software Engineering*, 93–94. Los Alamitos, CA: IEEE Computer Society. Presented at the Doctoral Symposium.
- **Wasylkowski, Andrzej**, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *ESEC-FSE 2007: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 35–44. New York, NY: ACM.
- **Wasylkowski, Andrzej**, and Andreas Zeller. 2009. Mining temporal specifications from object usage. In *ASE 2009: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 295–306. Los Alamitos, CA: IEEE Computer Society.
- Gruska, Natalie, **Andrzej Wasylkowski**, and Andreas Zeller. 2010. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *ISSTA 2010: Proceedings of the nineteenth international symposium on Software testing and analysis*. (At the time of writing this has not been published yet).

# Chapter 2

# Mining Object Usage Models

## 2.1 Motivation

Programs can be in general treated as processes operating on objects. This is especially clear in the context of object-oriented languages, but is also true for all programming languages: any entity that can be stored and manipulated by a program is essentially an object of some kind. Each such object has its type, such as an integer, a character, a database connection, etc. The type of an object puts a restriction on what the program can do with that object. For example, integers can be added, characters can be concatenated, database connections can be initialized, and so on. Without losing generality we can say that each type corresponds to a class in an object-oriented language, and each object of that type corresponds to an instance of that class.[1] This allows us to represent the set of operations that can be performed on a certain object of a certain type by the set of methods that are applicable to objects of that type; accordingly, the type—and thus the set of methods—puts a restriction on what can be done with objects of that type.

However, specifying the set of allowed operations is not enough to guarantee that objects will always be properly used. Consider a file object that can be opened, read from, and closed. These operations are not always applicable—for example, the file must first be opened before it can be read from, and reading from it is not allowed after it has been closed. We can say that each object apart from its type is characterized by its state, and some

[1] This is not true for so-called primitive types, but these can be wrapped into classes.

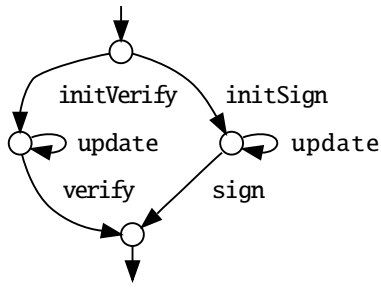



Figure 2.1: Typestate for the `java.security.Signature` class.

operations are applicable only in certain states. This idea has been known by the name of *a typestate* (Strom and Yemini 1986) and is based on describing possible sequences of operations that can be performed on objects of a certain type using a finite state automaton (FSA). Figure 2.1 shows a typestate for the `java.security.Signature` class. Even without knowing what is the purpose of this class, we can immediately see that there are two principal usage modes: verifying and signing, and they should not be mixed (e.g., calling `initVerify()` and then `sign()` is incorrect).

Typestates contain some very useful information, but one of their drawbacks is that they are limited to operations that can be performed *on* an object. This has a very important disadvantage: the abstraction level is constant for each type. First, this makes it impossible to model how methods using a higher abstraction level (e.g., treating a list as a specific list containing specific data, and not a generic data structure) can be combined to perform meaningful operations. One good example is if a Java programmer has written a method that fills a given vector with a given number of random integers, and another method that expects such a vector as one of its parameters. These two methods are related in that the second one needs the result of the first one to work correctly, but this relationship cannot be represented using typestates. Another problem introduced by the constant abstraction level is that it is impossible to model the behavior of values of primitive types, such as integers. In an ideal world each distinct concept would be represented in a program by a distinct type, and this problem would be irrelevant, but in reality this is not the case—file handles in C are a perfect example of what reality looks like.

To avoid the problems associated with the constant abstraction level this dissertation introduces *object usage models (OUMs)*. An object usage model is a nondeterministic finite automaton (NFA) that describes sequences of op-

erations that a particular object goes through in a particular method. The idea is to use the abstraction level of the method using the object. This of course means that an OUM not only must represent calls, where the object of interest is the call target, but also where it is a parameter. Another important consideration is that not all objects used inside a method are created by that method. Some come as parameters, others are read from fields, etc. Hence, an OUM must represent the origin of the object being modeled, which is not needed in case of a typestate, because its starting state is always the point where the lifecycle of the object starts. Before we introduce the formal definition of an object usage model, let us first introduce the concept of an *event* associated with an object.

**Definition 2.1** (Event)**.** *An event* associated with an object is one of the following:

- A method call with the object being used as the target or a parameter (possibly in multiple positions). Here we differentiate between *normal* and *abnormal* (i.e., because of an exception being thrown) return from the call.
- A method call with the object being the value that was returned.
- A field access with the object being the value that was read.
- A cast of the object to a different type (as in the Java expression `(A)b`).

Events are represented using strings as follows:

- A method call with the object being used as the target or a parameter is denoted by *Class.method signature* `@` *parameters*, where:
  - *Class* is the fully qualified (i.e., including full package path to the class) name of the class defining the method being called (e.g., `java.lang.Object`)
  - *method* is the name of the method being called (e.g., `hashCode`)
  - *signature* is the type signature of the method being called (e.g., `(ZC)V`; see the Java virtual machine specification (Lindholm and Yellin 1999) for a description of symbols used in signatures). This part is only used to differentiate between overloaded methods' names and is otherwise irrelevant.
  - *parameters* is a list of numbers indicating positions in the parameter list, where the object being modeled was used as a parameter; thus, if the call happened with the object being used as the third

parameter only, this would be `(3)`; if the object was used as the third and fifth parameter, this would be `(3, 5)`; if the object was used as the target of the call, this would be `(0)`; and so on.

Thus, for example, a call to `hashCode` with the object as the target of the call would be denoted by `java.lang.Object.hashCode()Z @ (0)`.

If the return is abnormal as a result of an exception being thrown, we denote it by `EXC(`*exception*`):` *Class.method signature* `@` *parameters*, where the additional *exception* is the fully qualified (i.e., including full package path to the class) class name of the exception that was thrown.

- A method call with the object being the value that was returned is denoted by `RETVAL:` *Class.method signature*, where *Class*, *method*, and *signature* are as defined above; thus, for example, the fact that the object is the return value of the `clone` method call would be denoted by `RETVAL: java.lang.Object.clone()Ljava/lang/Object;`.

- A field access with the object being the value that was read is denoted by `FIELDVAL:` *Class.field*, where *Class* is as defined above and *field* is the field's name. `FIELDVAL: java.lang.System.in` is one example.

- A cast with the object being cast to a different type is denoted by `CAST:` *Class*, where *Class* is a fully qualified name of the class, to which the object is cast, as in `CAST: java.lang.String`.

**Definition 2.2** (Object usage model)**.** Let *Classes* be the set of all valid fully qualified Java classes' names. *An object usage model* is a tuple $oum = (Q, \Sigma, T, q_0, F, Exc)$, where $Q$ is a finite set of states, $\Sigma$ is a finite set of *events* associated with the object being modeled, $T : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is a transition function, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, and $Exc : F \to Classes \cup \{\perp\}$ is a function that assigns exceptions' classes' names to final states.

An object usage model is essentially a nondeterministic finite automaton with epsilon transitions. Its final states are designated by the type of the method exit they represent. There is always only one final state $f_0 \in F$ representing normal exit from the method, for which $Exc(f_0) = \perp$. There can also be any number of final states $f_1, \ldots, f_n$ representing abnormal exits from the method (i.e., because of an exception being thrown). In this case $Exc(f_i)$ is always a fully qualified name of the class of the exception being thrown.

Before we show how OUMs are being created, let us take a look at a sample OUM. Consider the source code shown in Figure 2.2. One of the

```
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);
    return list;
}
```

Figure 2.2: Sample source code using list-operating API.

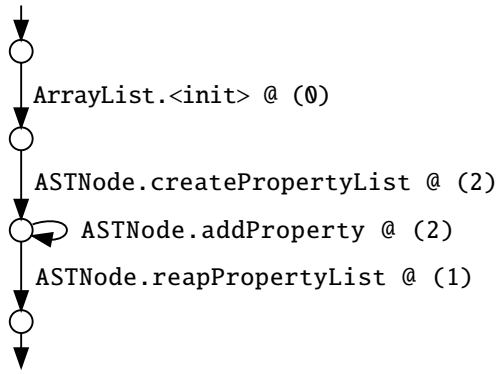



Figure 2.3: OUM for the list created by the method from Figure 2.2.

objects being used in this code is the list created in the very first line of the method `getPropertyList`. The method performs some operations on the list, and these operations can be represented using the OUM shown in Figure 2.3. There is one important point to be made here. Object creation is not represented as a separate event. This is strictly unnecessary, because each created object has to have its constructor called, and it is always the first method call on such an object; thus having the constructor call as the first event in the OUM is a sign for the object being created by the method as opposed to being received as a parameter, a return value of a method call, or being read from a field. We will also always follow the same convention when presenting OUMs:

- We will remove the packages' names and methods' signatures from the events' labels to improve readability. We will do this whenever it does not introduce any ambiguity.
- *Epsilon transitions* will be denoted by unlabeled dashed edges.
- The *initial state* $q_0$ is always denoted by a state with an ingoing arrow that has no source state associated with it.
- The one final state representing *normal exit* is always denoted by an anonymous state with an outgoing arrow that has no destination state associated with it.
- All final states representing *abnormal exits* (if present) are always denoted by states labeled with the name of the class of the exception, and they always have an outgoing arrow that has no destination state associated with it.

## 2.2 How Object Usage Models Are Created

Object usage models for a particular method can be created by performing forward data flow analysis on that method. Before we get to the detailed description of the approach, let us provide you, the reader, with a high-level overview of the technique.

Our goal when creating OUMs was not only to allow them to be flexible when it comes to the abstraction level being used (as indicated earlier), but also to mirror the programmer's idea about how the object being modeled is supposed to be used. This idea is present in the source code using the object, and the structure of that code is a very important part of it. Consider as an example a method that fills a collection with elements in a loop just to add one more element immediately after the loop. From the collection's point of view, this last addition does not differ from all the previous additions. Can we then just represent the way the collection is being used by having only a loop in the OUM with addition as the operation being performed? Is this what the programmer intended? While the answer to the first question may not be obvious, the answer to the second question must of course be a resounding "no." This is not what the programmer intended.

If we are to mirror programmer's intentions, we have to include relevant parts of the program's structure in the OUMs. To achieve this goal, we have decided to base states of the OUMs on the locations in the code. This idea is not new, as it has been already used in the work of Eisenbarth, Koschke, and

Vogel (2002) on static trace extraction.[2] Its main advantage is that it allows the OUMs to mirror the program's structure (e.g., if an operation performed on an object is in a body of a loop, it will be part of a respective loop in the OUM; if an operation is conditional, it will be conditional in the OUM, and so on). The main drawback of this technique is that it can overfit at times. One example is if a method operating on a collection uses an iterator to iterate through one element of the collection only. This is not a common iterator usage, and yet the OUM will mirror it. In Section 3.4 we will show how can we turn this drawback into an advantage.

Object usage models are created by performing intraprocedural data flow analysis on the program's methods. The analysis keeps track of events that happen to objects in a method, and locations where they happen. Our analysis works on Java bytecode. The reason for this is that the bytecode's syntax is quite low-level, and thus particularly suitable for analysis purposes. Let us now give some preliminary definitions that will be of use later on.

**Definition 2.3** (Control flow graph)**.** Let $m$ be a method and let *Classes* be the set of all valid fully qualified Java classes' names. The control flow graph of $m$ is a tuple $G = \langle V, E, v_0, v_1 \rangle$, where:

- $V$ is the set of nodes in the control flow graph. Each bytecode instruction in $m$ is represented by a separate node[3], and there are also two additional nodes: $v_0$ and $v_1$.
- $E \subseteq V \times V \times (\mathit{Classes} \cup \{\bot\})$ is the set of edges in the control flow graph. Given an edge $e = (v_a, v_b, c)$, $v_a$ and $v_b$ are the source and target nodes, respectively, and $c$ is either $\bot$ if the edge denotes exception-free execution of the instruction in $v_a$, or the name of the exception thrown when executing the instruction in $v_a$ if the edge denotes an exception-causing execution of that instruction.
- $v_0$ is the single artificial entry node, and given $m$'s actual entry node $v_{entry}$ there exists $e \in E$ such that $e = (v_0, v_{entry}, \bot)$.
- $v_1$ is the single artificial exit node, and each node corresponding to a bytecode instruction resulting in an exit from $m$ (this can be either a normal return instruction, or any instruction that causes an abnormal exit from $m$) has an edge leading to $v_1$.

[2] We will defer the comparison of their work with that of ours until Section 2.5.

[3] Java bytecode permits so-called subroutines in the bytecode. These can lead to a very imprecise analysis results if handled straightforwardly, so we have decided to inline them everywhere they are being called. This means that instructions that occur in those subroutines can have more than one node assigned to them.

**Definition 2.4** (Abstract objects)**.** Let $m$ be a method. Obj($m$) is the set of abstract objects used by $m$. Formally, $x \in \text{Obj}(m)$ iff $x$ is an object (i.e., not a primitive value) and one of the following holds:

- $x$ is the *this* pointer.
- $x$ is one of the formal parameters of $m$.
- $x$ is a result of a read of field value instruction present in $m$ (as in `x = System.out`).
- $x$ is a return value of a method call occurring in $m$ (as in `x = map.items()`).
- $x$ is a constant appearing in $m$ (like `"OK"` or `null`).
- $x$ is an object created in $m$ (as in `x = new Calendar()`).

Apart from *this* and formal parameters of $m$, each abstract object is uniquely identified by the location of the bytecode instruction that creates it.[4]

**Definition 2.5** (Variables)**.** Let $m$ be a method. Var($m$) is the set of variables (in the Java bytecode sense) used by $m$. Formally, $\text{Var}(m) = \text{StackVar}(m) \cup \text{LocalVar}(m)$, where:

- $\text{StackVar}(m) = (sv_0, \ldots, sv_{\text{max\_stack}(m)-1})$, where max_stack($m$) is the maximum depth the operand stack of $m$ can have at any point during the execution of $m$.
- $\text{LocalVar}(m) = (lv_0, \ldots, lv_{\text{num\_locals}(m)-1})$, where num_locals($m$) is the size of the array holding local variables of $m$.

The Java virtual machine allocates an operand stack and a local variable array for each method. The stack has a limited maximum depth, which is fixed for each method. The same is true for the array of local variables, which also contains entries for *this* (for a nonstatic method), and all formal parameters of $m$. Java also distinguishes between values that are byte-sized and word-sized, and each slot in the stack and in the array is byte-sized. This means that the Java virtual machine allocates two slots for each word-sized value. Our data flow analysis framework will mirror this behavior, but the second slot will always be treated as empty (i.e., as a slot with no object assigned to it).

[4] Constants are in this respect no exception, as there is a dedicated bytecode instruction that creates them and pushes them onto the stack.

**Definition 2.6** (Internal model)**.** Let $m$ be a method, $G = \langle V, E, v_0, v_1 \rangle$ be $m$'s control flow graph with a designated entry node $v_0$ and a designated exit node $v_1$, and *Classes* be the set of all valid fully qualified Java classes' names. An internal model based on $m$ is a tuple $im = (Q, \Sigma, T, q_0)$ where $Q = V \cup Classes \cup \{\bot\}$ is a finite set of states, $\Sigma$ is a finite set of all possible events stemming from $m$, $T \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation, and $q_0 = v_0 \in Q$ is the initial state corresponding to the entry node of $m$'s control flow graph.

The set of states of an internal model contains states corresponding to control flow graph nodes (elements of $V$), one state corresponding to the normal exit from $m$ ($\bot$), and states corresponding to abnormal exits from $m$ (elements of *Classes*). The transition relation is essentially a list of all connected states with the information about the type of a transition connecting them (either some event or an epsilon transition).

**Definition 2.7** (Merging internal models)**.** Let $m$ be a method, and $im = (Q, \Sigma, T, q_0)$ and $im' = (Q, \Sigma, T', q_0)$ be internal models based on $m$. $im'' = \text{im\_merge}(im, im')$ is a *merge* of $im$ and $im'$, and is defined as follows: $im'' = (Q, \Sigma, T'', q_0)$ where $T'' = T \cup T'$.

**Lemma 2.8.** im_merge is idempotent (i.e., $\text{im\_merge}(im, im) = im$).

*Proof.* Follows directly from the definition of im_merge and the fact that the set union operation is idempotent. □

**Lemma 2.9.** im_merge is associative (i.e., $\text{im\_merge}(im_1, \text{im\_merge}(im_2, im_3)) = \text{im\_merge}(\text{im\_merge}(im_1, im_2), im_3)$).

*Proof.* Follows directly from the definition of im_merge and the fact that the set union operation is associative. □

**Lemma 2.10.** im_merge is commutative (i.e., $\text{im\_merge}(im_1, im_2) = \text{im\_merge}(im_2, im_1)$).

*Proof.* Follows directly from the definition of im_merge and the fact that the set union operation is commutative. □

**Definition 2.11** (Data flow facts)**.** Let $m$ be a method. $A = \mathcal{P}(\text{Obj}(m))^{\text{Var}(m)} \times \mathbb{N} \times ((IM \times Q) \cup \{\bot, \top\})^{\text{Obj}(m)}$ is the set of potential data flow facts about $m$, where:

- $IM$ is the set of internal models based on $m$ (see definition 2.6).
- $Q$ is the set of states in all internal models (recall that all internal models for a fixed $m$ have the same set of states, see definition 2.6).

Data flow facts in our setting represent associations between variables and objects, and keep track of the events each object can go through when the method is executed. The second goal is achieved by constructing for each object its internal model on the fly while performing the data flow analysis. After the analysis is completed, internal models will be transformed into object usage models. (We will describe this later in more detail.) The natural number that is a part of each data flow fact is the current depth of the operand stack (see definition 2.5).

**Definition 2.12** (Join semilattice)**.** Let $m$ be a method. $L = \langle A, \underline{0}, \underline{\geq}, \vee \rangle$ is a join semilattice of the OUM-constructing data flow analysis framework, where:

- $A$ is the set of potential data flow facts about $m$ (see the definition 2.11).
- $\underline{0}$ is an element of $A$ defined as follows: $\underline{0} = (var2obj_{\underline{0}}, 0, obj2im_{\underline{0}})$ where $var2obj_{\underline{0}}(v) = \emptyset$ for all $v \in \mathrm{Var}(m)$ and $obj2im_{\underline{0}}(o) = \bot$ for all $o \in \mathrm{Obj}(m)$.
- $\underline{\geq}$ is a partial order defined as follows: $a \underline{\geq} b$ iff $a = a \vee b$.
- $\vee$ is a join operation defined as follows: Let $a = (var2obj_a, depth_a, obj2im_a)$ and $b = (var2obj_b, depth_b, obj2im_b)$ be elements of $A$. $a \vee b = c = (var2obj_c, depth_c, obj2im_c)$ is defined as follows:
  1. $var2obj_c(v) = var2obj_a(v) \cup var2obj_b(v)$ for all $v \in \mathrm{Var}(m)$
  2. $depth_c = \max(depth_a, depth_b)$
  3. $$obj2im_c(o) = \begin{cases} \top & \text{iff } obj2im_a(o) = \top \text{ or } obj2im_b(o) = \top, \\ \bot & \text{iff } obj2im_a(o) = \bot \text{ and } obj2im_b(o) = \bot, \\ (im_a, q_a) & \text{iff } obj2im_a(o) = (im_a, q_a) \text{ and } obj2im_b(o) = \bot, \\ (im_b, q_b) & \text{iff } obj2im_a(o) = \bot \text{ and } obj2im_b(o) = (im_b, q_b), \\ (im, q) & \text{iff } obj2im_a(o) = (im_a, q) \text{ and } obj2im_b(o) = (im_b, q) \text{ and } im = \mathrm{im\_merge}(im_a, im_b), \\ \top & \text{iff } obj2im_a(o) = (im_a, q_a) \text{ and } obj2im_b(o) = (im_b, q_b) \text{ and } q_a \neq q_b. \end{cases}$$

**Theorem 2.13.** $\underline{0}$ is a bottom element of $L$ (i.e., $a \vee \underline{0} = a$ for all $a \in A$).

*Proof.* Follows directly from definitions of $\underline{0}$ and $\vee$. □

**Lemma 2.14.** $\vee$ is idempotent (i.e., $a \vee a = a$ for all $a \in A$).

*Proof.* Set union, *max*, and im_merge operations are idempotent (see lemma 2.8), and from this and the definition of $\vee$ given above it follows that $\vee$ is idempotent as well. □

**Lemma 2.15.** $\vee$ is associative (i.e., $a \vee (b \vee c) = (a \vee b) \vee c$ for all $a, b, c, \in A$).

*Proof.* Trivial from the definition of $\vee$ by considering all possible cases for the *obj2im* part of all elements and the fact that the set union, *max*, and im_merge operations are associative (see lemma 2.9). □

**Lemma 2.16.** $\vee$ is commutative (i.e., $a \vee b = b \vee a$ for all $a, b \in A$).

*Proof.* Follows from the definition of $\vee$ and the fact that the set union, *max*, and im_merge operations are commutative (see lemma 2.10). □

**Lemma 2.17.** $\underline{\geq}$ is reflexive (i.e., $a \underline{\geq} a$ for all $a \in A$).

*Proof.* From the definition of $\underline{\geq}$ it follows that $a \underline{\geq} a$ iff $a = a \vee a$. On the other hand lemma 2.14 shows the right part of the equivalence to be true, and this proves that the left part of the equivalence is true as well. □

**Definition 2.18** (Data flow analysis framework)**.** Let $m$ be a method, $Classes$ be the set of all valid fully qualified Java classes' names, and $live : V \rightarrow \mathcal{P}(\mathrm{Var}(m))$ be the mapping assigning to each node in a control flow graph the set of variables alive directly before that point in the program execution. The data flow analysis framework (Marlowe and Ryder 1990, 124–129) we use to create internal models from $m$ is defined as $D = \langle G, L, F, M \rangle$ where:

- $G = \langle V, E, v_0, v_1 \rangle$ is $m$'s control flow graph.
- $L$ is a join semilattice from definition 2.12.
- $F \subseteq L^L$ is a class of transfer functions, that is, the smallest class of functions satisfying the following conditions:
  - $F$ has an identity function $I$, such that $I(a) = a$ for all $a \in L$.
  - Each control flow graph edge $e \in E$ has a corresponding function $f_e \in F$ defined as follows: Let $a = (var2obj, depth, obj2im)$ be an element of $L$ and let $e = (v_x, v_y, c)$ be the edge. We define $f_e(a)$ as a function that represents the effect of the transition information in $v_x$, $v_y$, $c$ and the liveness status of variables on the *var2obj* and *obj2im* mappings, and on the *depth* value. (See below for a detailed discussion of these functions.)
  - For any two functions $f, g \in F$ the function $h = f \circ g$ is in $F$.

  - For any two functions $f, g \in F$ the function $h(a) = f(a) \vee g(a)$ is in $F$.

- $M : E \to F$ is an edge transition function that assigns to each edge from a control flow graph a function from $F$. For a given $e \in E$, $M(e) = f_e \in F$.

Functions in $F$ created from the edges of the control flow graph add transitions to internal models of objects according to the events these objects participate in. Generally, the idea is to build the internal models during the analysis. Upon analysing each control flow graph edge, transitions are added to internal models of objects. If a particular object participates in the event denoted by the edge, the transition represents that event. If it does not, the transition is an epsilon transition. If an object is assigned to dead variables only, its model is not updated, etc. There are around two hundred Java bytecode instructions, so presenting an appropriate function in $F$ for each instruction would be too tedious and we will refrain from doing it. Instead, we will show one example to give the reader a feel for how the functions look like.

Some Java bytecode instructions are responsible for moving elements from the operand stack to the local variables array and back. One of such instructions is `aload`. This instruction pushes a local variable of type reference (i.e., word-sized) stored in the local variables array at a given index *index* onto the operand stack. Let $lv_i$ be the $i$-th element of a local variable array, $sv_i$ be an element of the operand stack at depth $i$ (see definition 2.5), and $e = (v_x, v_y, \bot) \in E$ be a control flow graph edge with $v_x$ containing an `aload` instruction. (We know that the last element of the edge tuple is $\bot$, because the `aload` instruction always succeeds; compare the definition 2.3.) The function $f_e \in F$ is defined as follows: Let $a = (var2obj, depth, obj2im)$ be an element of $L$. Then $f_e(a) = b$ where $b = (var2obj', depth', obj2im')$ where:

- $var2obj'(v) = \begin{cases} \emptyset & \text{iff } v \notin live(v_y), \\ O & \text{iff } v \in live(v_y) \text{ and } v \neq sv_{depth} \text{ and } var2obj(v) = O, \\ O' & \text{iff } v \in live(v_y) \text{ and } v = sv_{depth} \text{ and } var2obj(lv_{index}) = O'. \end{cases}$

- $depth' = depth + 2$

- $obj2im'(o) = \begin{cases} x & \text{iff } o \notin var2obj(v) \text{ for all } v \in \mathrm{Var}(m) \text{ and } obj2im(o) = x, \\ x & \text{iff } o \in var2obj(v) \text{ for some } v \in \mathrm{Var}(m) \text{ and } obj2im(o) = x \in \{\bot, \top\}, \\ (im', q') & \text{iff } o \in var2obj(v) \text{ for some } v \in \mathrm{Var}(m) \text{ and } obj2im(o) = (im, q) \text{ where } im = (Q, \Sigma, T, q_0) \text{ and } im' = (Q, \Sigma, T', q_0) \text{ where } T' = T \cup \{(v_x, \epsilon, v_y)\} \text{ and } q' = v_y. \end{cases}$

**Theorem 2.19.** The data flow analysis framework $D = \langle G, L, F, M \rangle$ from definition 2.18 is distributive (i.e., $f(a \vee b) = f(a) \vee f(b)$ for all $a, b \in L$ and $f \in F$).

Proof of the theorem above would require knowing the definitions of all functions in $F$ and we did not give them. We will instead give a brief sketch of the proof. Functions in $F$ manipulate the depth, the mapping from variables to abstract objects, and the mapping from objects to internal models. All those functions only add new transitions to the internal models, and the join operator takes a union of those; thus, distributivity holds here (it does not matter if we first add elements to two sets and then take their union, or if we do this the other way around). The depth for a given function $f$ is changed in the same way irrespectively of the lattice element involved, and the join operator takes a maximum of two numbers. Again, this makes distributivity hold here (it does not matter if we first increase two numbers by the same increment and then take their maximum, or if we do this the other way around). The last element is the mapping from variables to abstract objects. All functions in $F$ change this mapping only by overwriting existing elements, and the join operator only creates a union of two mappings; thus, distributivity holds here as well (it does not matter if we first overwrite two mappings with the same value and then take their union, or if we do this the other way around).

Based on the properties we have stated in lemmas 2.14, 2.15, 2.16, 2.17, and theorems 2.13 and 2.19 we conclude that the data flow analysis framework from definition 2.18 is solvable (i.e., the iterative algorithm for data flow analysis works [Marlowe and Ryder 1990; Aho, Sethi, and Ullman 1988, 680–94]).

After creating internal models for all abstract objects in a method, we transform them into object usage models by removing unused states.[5] Let $im = (Q, \Sigma, T, q_0)$ be an internal model, and *Classes* be the set of all valid

[5]For a given internal model these are states corresponding to source code locations at which the object being modeled was either not visible or not alive.

fully qualified Java classes' names. An object usage model stemming from $im$ is defined as $oum = (Q', \Sigma', T', q_0', F, Exc)$ where:

- $Q' = \{q \in Q : \exists p, \sigma . (p, \sigma, q) \in T\} \cup \{q_0, \bot\}$ (where $\bot \in Q$; see the definition 2.6)
- $\Sigma' = \Sigma$
- $T'$ is defined as follows: $T'(q, \sigma) = R$ iff $q \in Q'$ and $R = \{r : (q, \sigma, r) \in T\}$
- $q_0' = q_0$
- $F = (Classes \cap Q') \cup \{\bot\} \subseteq Q'$
- $Exc$ is defined as $Exc(c) = c$ for all $c \in F$

In the end, the analysis described above gives us an object usage model for every abstract object in the method being analyzed. By repeating this process for every method in a program we get object usage models for every abstract object in the program.

## 2.3 Minimizing Object Usage Models

Object usage models created by applying the analysis described in the preceding section accurately reflect structures of methods they stem from, but they can be at times too large to be understandable while at the same time not containing a lot of information. This is because an object usage model inherits states from a method, and methods can have hundreds of instructions. At the same time, the object being modeled does not participate in most of these instructions, and thus most transitions in an object usage model are epsilon transitions. Consider the source code shown in Figure 2.2. The object usage model shown in Figure 2.3 is a minimized one; the raw, unminimized object usage model is shown in Figure 2.4. It has over 30 states and a similar number of epsilon transitions.

Minimizing models is necessary, but simply collapsing two states when there is an epsilon transition between them is too invasive. Imagine a case when there is a conditional event in the model (i.e., there are two states connected by two transitions: one of them being an epsilon transition and another being the event transition [see Figure 2.5(a)]). The meaning of such a structure is that the instruction to which the event corresponds is optional and can be in some circumstances omitted; however, if we collapse those two states, the conditional event turns into a loop, and this changes the meaning of the structure as well (see Figure 2.5(b)). Even though this particular case

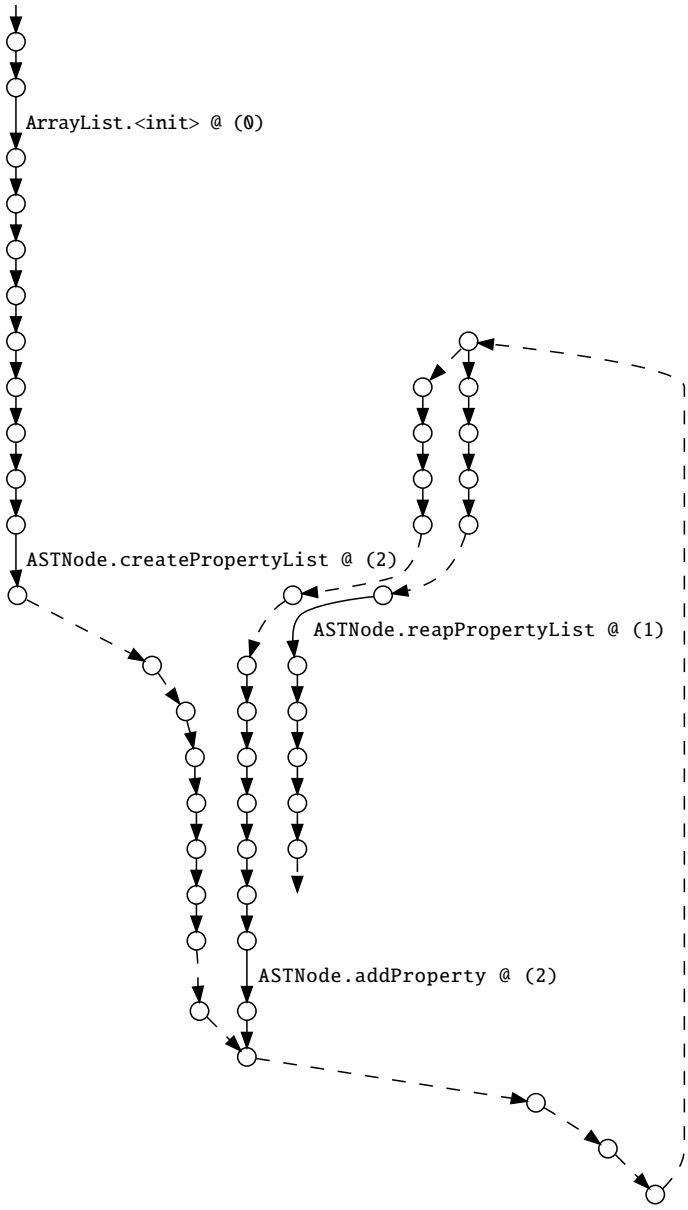
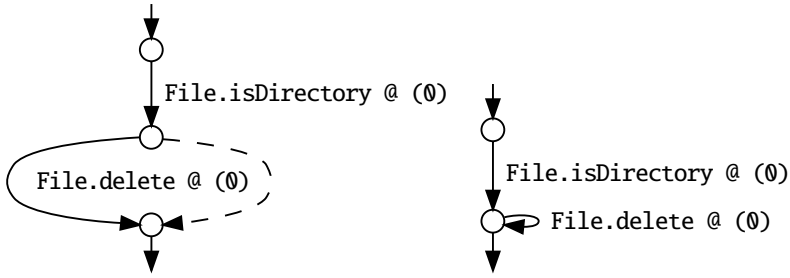
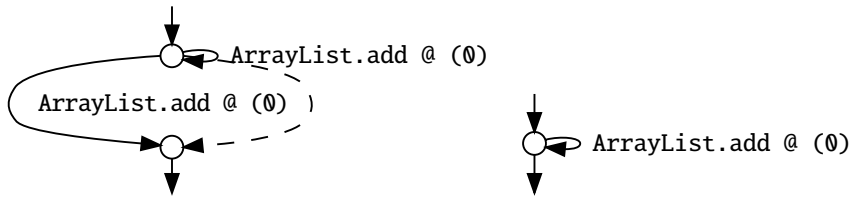



Figure 2.4: Raw, unminimized OUM for the list from method in Figure 2.2.




(a) OUM before minimization (b) OUM after naïve minimization

Figure 2.5: OUM, for which minimization by merging states connected with epsilon transitions leads to undesirable results: the language changes.




(a) OUM before minimization (b) OUM after naïve minimization

Figure 2.6: OUM, for which minimization using standard finite state automata minimization algorithms leads to undesirable results: the structure changes.

can be properly handled using standard finite state automata minimization algorithms, these are unusable for minimizing object usage models in general, because they can preserve the language of the automaton (as they would in the example above) but cannot preserve its structure. Imagine a case when there is a loop in some state, and that the same state has two additional outgoing transitions leading to some other state (the same for both). Let one of those transitions be an epsilon transition, and another be the same transition as the one in the loop (see Figure 2.6(a)). Now, merging the two states preserves the language but destroys the structure (see Figure 2.6(b)). The language in both cases is $e*$ where $e$ is the event, but the structure in the first case is a loop followed by a conditional event, and in the second case it is only a loop. Even if there is only an epsilon transition between two states, collapsing them can change the meaning of the structure; thus, we can clearly see that a different approach is needed.

As stated above, we need to minimize the models while preserving their structure. Our idea behind preserving the structure of a model is to preserve the programmer's intentions, which are inherent in this structure. On the other hand, we do not aim at minimizing each model to the point where it cannot be minimized further without destroying its structure. There is no clear benefit in having the smallest possible model instead of one that is simply small enough. Once again, what we want is to make it easier for a human to understand the model, and to do this we need models that do not consist of mostly epsilon transitions.

We have come up with three simple cases where epsilon transitions can be removed without destroying a model's structure. Minimizing the model to the point where neither of those cases holds anymore has proven to be enough to get rid of most epsilon transitions and thus make models easier to understand (see Section 2.4 for examples of minimized object usage models). When introducing the conditions, under which epsilon transitions can be removed, we have assumed that $oum = (Q, \Sigma, T, q_0, F, Exc)$ is the object usage model given as an input. An epsilon transition can be removed if:

- it is a loop in some state. Formally, if $T(q, \epsilon) = R$ for some $q$ and $q \in R$, we can change the transition function, so that $T(q, \epsilon) = R \setminus \{q\}$.

- it is a transition between two different states $q_1$ and $q_2$, where $q_1$ is not the entry state and it has no other outgoing transitions. Intuitively, the idea here is that once we reach $q_1$ we have no other choice, but to go to $q_2$ without any event happening in-between. Formally, if $T(q_1, \epsilon) = \{q_2\}$ for some $q_1 \neq q_2$ where $q_1 \neq q_0$, and $T(q_1, \sigma) = \emptyset$ for all $\sigma \neq \epsilon$, we can merge $q_1$ and $q_2$ by replacing $Q$ and $T$ with $Q'$ and $T'$, respectively, as follows:

  - $Q' = Q \setminus \{q_1\}$
  - $T' : Q' \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q')$ is defined as follows:

$$T'(q', \sigma) = \begin{cases} R & \text{iff } T(q', \sigma) = R \text{ and } q_1 \notin R, \\ R' & \text{iff } T(q', \sigma) = R \text{ and } q_1 \in R \text{ and} \\ & R' = (R \setminus \{q_1\}) \cup \{q_2\}. \end{cases}$$

- it is a transition between two different states $q_1$ and $q_2$, where $q_2$ is not a final state and it has no other ingoing transitions. Intuitively, the idea here is that there is no other way to reach $q_2$ than through $q_1$, so if we ever get to $q_2$, we must have been in $q_1$ directly before (no event could have happened in-between). Formally, if $q_2 \in T(q_1, \epsilon)$ for some $q_1 \neq q_2$ where $q_2 \notin F$, and $q_2 \notin T(q, \sigma)$ for all $q, \sigma$ such that $q \neq q_1$ or $\sigma \neq \epsilon$, we

Table 2.1: Projects used as case study subjects

| Program | Size (K SLOC) | Classes | Methods |
|---|---|---|---|
| Vuze 3.1.1.0 | 345 | 5,532 | 35,363 |
| AspectJ 1.5.3 | 327 | 2,957 | 36,045 |
| Apache Tomcat 6.0.18 | 254 | 1,462 | 16,347 |
| ArgoUML 0.26 | 187 | 1,897 | 13,824 |
| Columba 1.4 | 100 | 1,488 | 8,590 |
| Act-Rbot 0.8.2 | 47 | 344 | 3,401 |

can merge $q_1$ and $q_2$ by replacing $Q$ and $T$ with $Q'$ and $T'$, respectively, as follows:

- $Q' = Q \setminus \{q_2\}$
- $T' : Q' \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q')$ is defined as follows:

$$T'(q', \sigma) = \begin{cases} R & \text{iff } T(q', \sigma) = R \text{ and } q' \neq q_1, \\ R' & \text{iff } T(q', \sigma) = R \text{ and } q' = q_1 \text{ and} \\ & R' = (R \setminus \{q_2\}) \cup T(q_2, \sigma). \end{cases}$$

Applying the transformations given above until a fixed point is reached, where none of those three cases apply, is enough to make object usage models much smaller. Additionally, these transformations not only preserve the language of a model but also preserve its structure (i.e., loops remain loops and branchings remain branchings), which is exactly what we want. We would like to emphasize once again the fact that we do not claim these transformations produce the smallest model possible. It is possible that more transitions can be removed and more states can be merged without destroying the model's structure; however, the transformations given above do make the models much smaller and thus more understandable for humans.

## 2.4 Examples of Object Usage Models

We have implemented the data flow analysis framework and the minimization algorithm presented in Sections 2.2 and 2.3, and analyzed six open-source projects with it. Table 2.1 shows the summary of the projects. For each project we report on the version analyzed, Java Physical Source Lines of Code (SLOC)[6], and the number of classes and methods comprising the project. We

[6]generated using 'SLOCCount' by David A. Wheeler

Table 2.2: Object usage models created by analyzing case study subjects

| Program | OUMs created | Total time (mm:ss) |
|---|---|---|
| Vuze 3.1.1.0 | 237,569 | 3:08 |
| AspectJ 1.5.3 | 233,731 | 3:09 |
| Apache Tomcat 6.0.18 | 117,042 | 1:10 |
| ArgoUML 0.26 | 95,463 | 2:10 |
| Columba 1.4 | 63,981 | 0:30 |
| Act-Rbot 0.8.2 | 54,079 | 1:08 |

have tried to analyze only those classes that truly belong to a project; for example, we ignored third-party libraries in the Act-Rbot jar file.

Our analysis created object usage models for each of the projects shown in Table 2.1. The summary of the results can be found in Table 2.2. We report on the number of object usage models created and on the time (wall clock time, averaged over ten consecutive runs) that was needed to perform the analysis on a 2.53 GHz Intel Core 2 Duo machine with 4 GB of RAM. We did not do an experimental evaluation of the models' usefulness, but we think that object usage models can be very helpful for programmers not knowing how to use certain classes or having to maintain code they are not very familiar with. To support this statement, we took a closer look at models extracted from one of the projects we analyzed, namely, AspectJ. AspectJ is an aspect-oriented extension to the Java programming language. Investigating all 233,731 object usage models that we extracted is of course impossible because of their sheer number. Instead, we looked at a few randomly selected models. Below we present and explain some of those that we found particularly interesting and small enough to be easily understandable.

Figure 2.7 shows a model of a `java.util.Stack` object mined from one of the methods in AspectJ. The model tells us that the stack was first created, then elements have been pushed onto it in a loop, and finally an enumeration of all the elements was requested.[7] 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 nonempty), it does not result in a state-changing transition in our model. This is because in

[7] An astute reader will notice that the call to `elements()` is depicted as a call to a method from the `Vector` class. The reason for this is that most parts of AspectJ are compiled using Java 1.1, and this results in method calls being represented as happening on an object of a class that actually defines the method. Newer Java versions always represent method calls as happening on an object of a class that was actually declared as the type of the target.

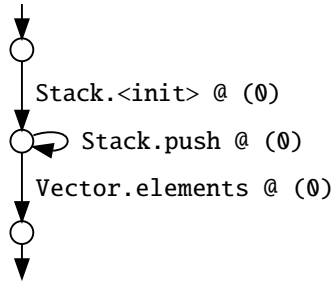



Figure 2.7: OUM for a `Stack` object mined from AspectJ.

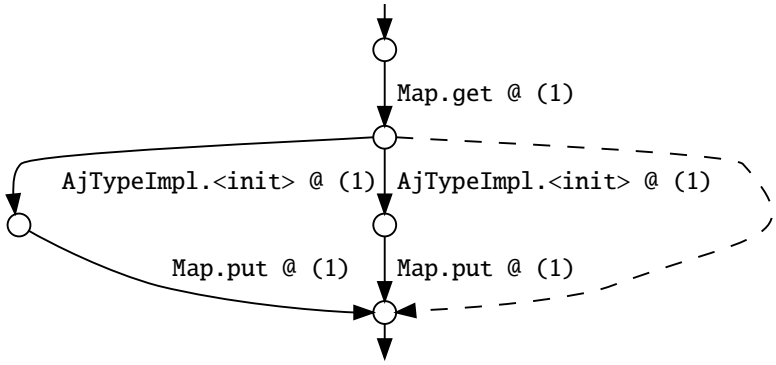



Figure 2.8: OUM for a `Class` object mined from AspectJ.

this particular case there is no difference between the first and the subsequent calls to `push()` from the programmer's perspective. 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, because the way the stack is being used has changed. We no longer put elements into it, but rather we extract them.

Another model is shown in Figure 2.8. This model does not contain a single call with the object being modeled used as a target. What is more important, limiting the model to only such operations (e.g., by not sticking to the method's abstraction level, but by going deeper until either such calls are found, or the object is not passed anywhere else anymore) would result in an empty model. There are two reasons for this. First, `Map` is an interface,

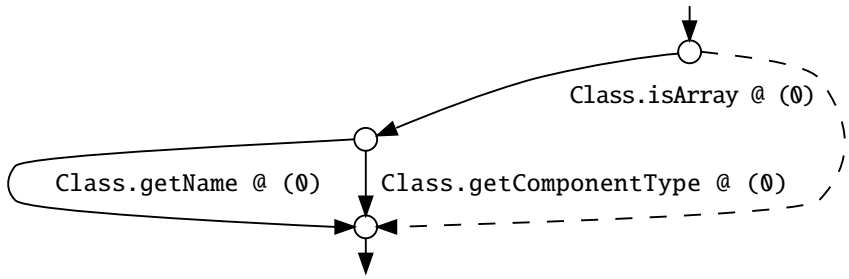



Figure 2.9: OUM for a `Class` object mined from AspectJ.

so all its methods are only declared and the implementation is unknown. Second, `AjTypeImpl` constructor only stores the class reference it gets in a field, without issuing any method calls. Our object usage model is quite interesting, as it encompasses operations on objects of two different classes in one entity; and it has two identical, but distinct paths. It shows how AspectJ's internal mapping between Java classes (objects of `Class` class) and AspectJ type implementations (objects of `AjTypeImpl` class) works. This model has been mined from the method called `getAjType()`, which gets an `AjTypeImpl` object corresponding to a given `Class` (subsequently called `c`). As we can see, the method uses a `Map` object to represent the mapping. It first gets the mapping for the class `c` from the map. If the mapping is not present, a new type implementation is created and the map is updated (the leftmost path in the model). If the class `c` is present in the map, there are two possibilities: either its corresponding `AjTypeImpl` object is returned (the rigthmost path), or a new `AjTypeImpl` object is created, put in the map, and then returned (the middle path). Now, the model reveals an interesting information: the fact that `c` is present in the mapping does not preclude creating new type implementation for it—that is why there are two identical paths through the model. Why is that so? If we take a look at the code, we will see that what is actually being stored are weak references to type implementations, and the code checks for their nullness. Only if a reference is `null`, a new type implementation is created.

A different model of an object of class `Class` is shown in Figure 2.9. We can learn several things from it. First, arrays are represented in Java just like classes. We know this because the first event in the model is a call to a method that checks whether this particular `Class` instance represents a real class or just an array. Second, methods `getComponentType()` and `getName()` behave differently when called on instances that represent real classes than

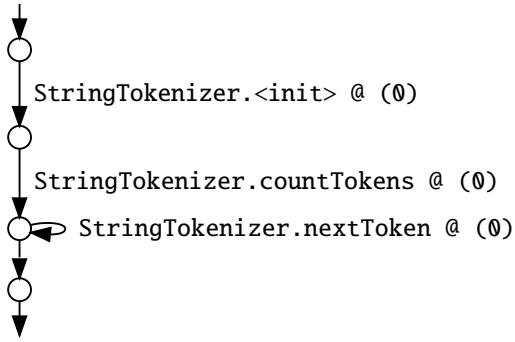



Figure 2.10: OUM for a `StringTokenizer` object mined from AspectJ.

when called on instances that represent arrays. We know this because only one of those methods is called, and the decision which one is dependent on the result of the call to `isArray()`[8]. A quick glance at the API specification for the Java 2 Platform Standard Edition 5.0 at the Sun Microsystems webpage confirms these facts.

There is one important lesson to be learned from the two examples above. The way objects should be used depends not only on their class but also on their purpose in the program. There is a deeper concept behind this, namely, that of so-called *abstract types* (O'Callahan and Jackson 1997; Guo et al. 2006). The idea here is that it often happens that variables having the same type are used for different purposes. For example, this is the case when using integers to represent both age and distance. Variables representing those two concepts will have the same type, but their purpose, their real type, differs. In the first case this is years, and in the second case this is kilometers. This underlying type of a variable is called *an abstract type*. Two variables sharing the same type, but having different abstract types may have the same usage patterns, but may also have completely different usage patterns. Our analysis does not partition variables according to their abstract types but it does allow us to see differences in usage of objects having the same type, as in the two examples given above.

A related, but orthogonal concept is that of a type allowing different usage patterns that boil down to doing the same thing. This redundancy allows the user to choose calls that for some reason fit him more (this can be stylistic consistency, code convention, etc.). Figures 2.10 and 2.11 illustrate

[8]The object usage model does not show this explicitly, but we can see that the choice of calling either `getComponentType()` or `getName()` comes after calling `isArray()`.

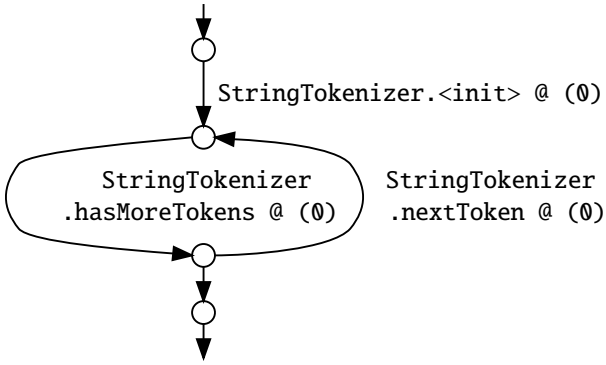



Figure 2.11: OUM for a `StringTokenizer` object mined from AspectJ.

this concept. Both show object usage models of `StringTokenizer` instances used for the same purpose, but in different ways. Figure 2.10 shows iterating through tokens by first retrieving the number of tokens available (the call to `countTokens()`), and then calling `nextToken()` multiple times. Another philosophy is more akin to the way iterators are being used and is shown in Figure 2.11. Here there is a pair of calls to `hasMoreToken()` and `nextToken()` which is being called multiple times. In other words, in this setting we do not a priori know the number of tokens we will have to go through. Instead, at each step we check if there is at least one more token we can extract.

## 2.5 Related Work

Our work (Wasylkowski 2007; Wasylkowski, Zeller, and Lindig 2007) has been preceded by works of other researchers, both in the area of specifying how the models[9] should be built to be useful, and in the area of inferring models from artifacts such as code, execution traces, etc. There are also works that appeared later and some that build on our discoveries presented in this chapter. An important related research direction is finding ways to check programs for conformity with a priori specified models with the assumption that deviations point to defects. There is a lot of work in these areas, and we can show only a fraction of what is available. Additionally, sometimes a work does not fit into one category (e.g., if authors have introduced new ideas for

[9]In this section we will use the name „model" for each formalism having as its goal modeling the way classes or objects are, or should be used. In some case these will be finite state automata, in other axiomatic expressions, etc.

how the models should look like alongside with an inference mechanism for them). In such a case we put the work in the subsection where we think it fits best.

### 2.5.1 Modeling Usage of Classes and Objects

The seminal work concerning modeling the way classes are supposed to be used is the work on *typestates* by Strom and Yemini (1986). Briefly, the idea is that values not only have *type* but also have *state* (e.g., a file can be either open or closed), and we can represent operations available in each such state using specially crafted finite state automata, so-called typestates. See Section 2.1 for a more detailed description of this approach.

Yellin and Strom (1997) introduced so-called *protocols*. The main difference between protocols and typestates is that protocols use *outgoing* as well as *ingoing* method calls as transitions. The authors have described how this distinction can be used to find sets of components that fit together and to build adapters for sets of components that do not. A similar idea with the same distinction was presented by de Alfaro and Henzinger (2001) in their work on so-called *interface automata*. Later, Chakrabarti et al. (2002) introduced so-called *stateless* and *stateful software module interfaces*, with even more expressiveness.

Allan et al. (2005) introduced *tracematches*. A tracematch is a pattern of events combined with code that will be triggered if the execution of the program matches the given pattern. Tracematches can be used as a specification language for expected object behavior, but also—what is impossible when using typestates—for specifying expected behavior of multiple objects when there is interplay between them.

Other authors have introduced their own ideas as well, either as models external to the program (Nierstrasz 1993), or as extensions to programming languages allowing as one of the possibilities the specification of how methods change objects' states (DeLine and Fähndrich 2001; Lee and Xiong 2001; DeLine and Fähndrich 2004; Bierhoff and Aldrich 2005).

### 2.5.2 Inferring Models

Cook and Wolf (1995, 1998) have introduced the technique they called *process discovery*. Their work focused on creating a model for the behavior of a software development process. They create finite state automata based on traces of the events recorded during software development. To this end they have applied and compared approaches based on Markov models, neural networks, and grammar inference. This seminal work had a big influence

on further model inference approaches, even though its goal was to discover software development processes instead of models of program behavior.

Our idea of object usage models was inspired by the idea of *object process graphs* introduced by Eisenbarth, Koschke, and Vogel (2002, 2005) in their work on *static trace extraction*. Object process graphs have nodes labeled with events (most importantly method calls, reads from, and writes to variables) and unlabeled edges. Whenever there is a call to a method defined by the program being analyzed, the callee's object process graph is included in the caller's process graph as an extension of the method call node. This makes the analysis quite costly (over six minutes for the benchmark program with less than 150 methods), and the object process graphs quite large (with the average number of nodes being 209 for the benchmark program). Later the same idea was presented by Quante and Koschke (2007) in a dynamic setting, where completeness of the results was sacrificed for their precision and scalability of the analysis.

Nguyen et al. (2009) presented GrouMiner, which mines so-called *graph-based object usage models* (groums). Groums are directed acyclic graphs representing object usage; they extend the notion of events used in our work to include control-flow structures (`while`-loop, `if`-statement), and can model usage of multiple objects when there is interplay between them[10]

Ammons, Bodík, and Larus (2002) have introduced the idea of *mining specifications* from programs. They use dynamic analysis of C programs to create so-called *specifications automata*, where nodes are function calls, and edges represent the ordering and have weights assigned to them (the more often a particular ordering was observed, the more weight is assigned to the edge). Their automata are at first created from traces by a probabilistic finite state automaton learner. Later, edges with low weights are pruned. This approach is quite precise, but pruning may result in removing legitimate behavior that was not observed because of the weakness of the test suite used to generate the traces. Another problem is that the user is required to provide information relating functions to objects (e.g., to specify that `bind()` uses the return value of `socket()`).

Whaley, Martin, and Lam (2002) presented both static and dynamic analysis techniques for inferring finite state automaton models of classes. Static analysis is server-side (i.e., the component to be modeled is analyzed, not its clients). The inferred FSAs use methods' names as states' labels, and anonymous transitions shown allowed ordering of method calls. One interesting thing in this work is that it proposes using multiple *submodels* per class, each focusing on a certain aspect of the class being modeled. In practice, each submodel focuses on a field or a set of fields of the class being mod-

[10]Section 3.6 contains further details on GrouMiner.

eled. One problem of this work is that even though the models are FSAs, in reality they just represent sets of properties of the type "a call to `b` may follow after a call to `a`" (i.e., the actual language used is weaker than regular).

Alur et al. (2005) presented an approach for synthesizing *interface specifications* for Java classes (really slightly modified typestates) by using server-side analysis. Their approach is based on model checking and requires the user to provide two things as an input: the initial predicate abstraction, and exceptions that, when thrown, indicate illegal usage of the class. One problem is that the user needs to have some knowledge of the class already to provide the inputs. Another is that there is an underlying assumption here that an interaction with a class is correct if it does not cause an exception to be thrown; however, this is too permissive, as it is possible that an incorrect interaction leads to wrong results instead.

A similar idea based on using model checking on the class definition was presented by Henzinger, Jhala, and Majumdar (2005). Their approach creates full interfaces (i.e., ones which are both safe and permissive); however, it suffers from the same problems as the work described above: a user needs to provide the initial predicate abstraction and define what does it mean that a class is in an erroneous state (only in this case this does not have to be an exception). This means that even though the interfaces are guaranteed to be safe with respect to the erroneous state defined, they can in reality be too permissive (i.e., unsafe) if the definition of the erroneous state is incorrect, or is too complicated to be applicable to model checking.

Shoham et al. (2008, 2007) presented an approach for mining specifications (really typestates) from programs. They use interprocedural static analysis to find out how a client program uses a given class, and based on that knowledge construct a specification for that class. Their approach is quite accurate but also time consuming: they have reported running times of less than 30 minutes for their benchmarks, which were all applications with less than 5000 methods[11].

Pradel and Gross (2009) presented a dynamic analysis technique for generating specifications out of traces of program runs. Their approach builds FSAs for sets of related object, with a FSA modeling sequences of method calls. This work suffers from the same problem as Whaley, Martin, and Lam's work: the language actually used is weaker than regular, with its expressiveness restricted to just being able to say that one method call can follow another method call. On the other hand, focusing on a set of related objects instead of one class is a very interesting and potentially very valuable idea.

[11] AspectJ, which we used as our main evaluation subject, has over 36,000 methods

Acharya et al. (2007) presented an approach for mining so-called *usage scenarios and specifications*. Both are sets of partial orders between method calls, like "`XCreateGC()` is typically called before `XFreeGC()`." The difference is that specifications are stronger (i.e., contain partial orders that occur more often compared to those in usage scenarios, and can thus be trusted more). They can be represented as graphs where nodes are method calls and edges represent partial order relationships. The authors use a model checker to create traces related to the set of methods given as an input. One problem with this approach is that a user must provide as a seed a set of methods that are related and he is interested in for the approach to be useful in finding other related methods and finding the relationship between them.

Dallmeier et al. (2006) introduced so-called *object behavior models*, which are finite state automata with states providing the information about the state of an object and transitions being method calls. They use return values of so-called inspectors (pure methods with no parameters) to characterize state of an object. This allows them to represent facts such as "Adding an element to an empty vector (i.e., one for which `isEmpty()` returns `true`) causes it to become a nonempty one (i.e., one for which `isEmpty()` returns `false`)". They use dynamic analysis to produce their models, and this of course makes the quality of the results dependent on the quality of the test suite used as the analysis input.

Xie et al. developed a set of approaches constructing so-called *object state machines*. Object state machines are somewhat related to typestates, but their states and transitions are labeled with additional information. The authors have used several types of information here: return values of so-called observer method calls (Xie and Notkin 2004a), fields' values (Xie and Notkin 2004b), and branch coverage information (Yuan and Xie 2005). Their approach is based on dynamic analysis (i.e., they observe the behavior of the program, and construct the machines based on their observations). This makes object state machines very precise but also most of the time incomplete. A related drawback is that the quality of the final results very heavily depends on the quality of test cases used as analysis subjects.

Other authors have introduced their own ideas and inference methods as well (Lorenzoli, Mariani, and Pezzè 2006, 2008; Reiss and Renieris 2001; Mariani and Pezzè 2005; Gabel and Su 2008), including reverse engineering of sequence diagrams (Rountev, Volgin, and Reddoch 2005; Rountev and Connell 2005; Systä, Koskimies, and Müller 2001), statecharts (Systä, Koskimies, and Müller 2001), live sequence charts (Lo, Maoz, and Khoo 2007), inferring algebraic specifications (Henkel and Diwan 2003; Ghezzi, Mocci, and Monga 2007), probabilistic FSAs (Lo and Khoo 2006) and using software repositories to mine usage rules (Williams and Hollingsworth 2005).

There is also a body of work in the area of automatically preparing (or finding) code examples or suggestions for the user on how to use a certain method, class, or sets thereof. MAPO (Xie and Pei 2006; Zhong et al. 2009) provides sample sequences of method calls given a few methods the user is interested in. Strathcona (Holmes and Murphy 2005) uses structural information from the user's code to find related code examples in an example repository. Prospector (Mandelin et al. 2005), XSnippet (Sahavechaphan and Claypool 2006), and PARSEWeb (Thummalapenta and Xie 2007) provide the user with information on how to create an object of a given type.

### 2.5.3 Validating Programs against Models

Reiss (2005) developed the CHET system, which lets developers use *extended finite state automata over parametrized events* to specify the way components should be used, and checks programs for conformance with these specifications. CHET allows a variety of different events, such as method calls, returns from calls, allocations, method entries, etc. Specifications written for CHET can additionally contain variables, whose values are changed when certain events happen. This gives a lot of flexibility and power, but the user is required to create the automata himself.

Fink et al. (2006, 2008) showed how we can *verify* programs for their conformance with typestates given as input. They have used a novel staging approach, where there is a set of verifiers, from least accurate and fastest to most accurate and slowest, and potential points of failure are verified with the simplest verifier possible first. This makes the whole analysis as precise as the analysis implemented in the best verifier, but keeps performance costs at a reasonable level. Bodden, Lam, and Hendren (2008) presented an approach for verifying programs for their conformance with tracematches. They also use staged analysis. Another verification approach for tracematches was presented by Naeem and Lhoták (2008). Das, Lerner, and Seigle (2002) presented ESP: a tool for checking if a program satisfies a given temporal safety property (provided as a FSA—being essentially a typestate).

Dwyer and Purandare (2007) proposed *combining static and dynamic analysis* to check if programs violate a given typestate. Their idea is to use static analysis first, and then do a so-called *residual dynamic analysis* on those parts of the program, on which the static analysis was not able to give definite results. This reduces the potentially very high cost of having to instrument the whole program with instructions checking, whether a typestate property has been violated or not. Dwyer, Kinneer, and Elbaum (2007) propose so-called *adaptive online analysis*, aiming at checking if programs violate given "protocol FSAs" (very closely related to typestates). Their

idea is to replace traditional "dynamic analysis produces a trace that is later analyzed" approach with an online (analyze the output while the program is running), adaptive (change the dynamic analysis scope depending on the analysis results) approach.

## 2.6 Summary

This chapter makes the following contributions:

- We have introduced the notion of *object usage models*. Compared to other means of modeling classes' and objects' usage, the have the following advantages:
  - They allow the programmer to focus on one object at a time, thus freeing him from distraction caused by long code snippets, where the usage the programmer is interested in is scattered across whole methods.
  - Object usage models use a variable abstraction level—they show how an object is being used from the perspective of a method using it, and thus *from the perspective of the programmer that wrote the method*.
- We have presented an interprocedural static analysis that creates object usage models from program's bytecode. We have shown that our analysis scales to large programs: Analyzing Vuze (the largest program we used in our experiments, 345 K SLOC, 35,363 methods in 5532 classes) takes slightly more than three minutes and results in 237,569 object usage models being produced.

To learn more about our work on model mining and related topics, see:

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

# Chapter 3

# Patterns and Anomalies in Object Usage

## 3.1 Introduction

In the preceding chapter we have shown how we can mine object usage models from a program's code. These models represent the way objects are used in methods and are intended to capture the programmers' intentions as well. Each abstract object has its own model, and thus there are as many models representing usage of a particular type as there are abstract objects of that type. For example, there are 1050 abstract objects of the `java.util.Iterator` type in the AspectJ project, and each of those objects has its own model. With that many models for a single type, each representing a particular usage of that type, we can expect to find patterns recurring in those models. For example, it might be that some method is always called before another method can be called (like `hasNext()` is typically always called before `next()`), or that there are some specific initialization and/or shutdown methods that have to be called if the object is to be kept in a consistent state (like `initVerify()`), and so on. And whenever there is a pattern, there can also be models that do not adhere to it and can therefore be flagged as being anomalous. Anomalous models can be signs of inconsistency or perhaps rarity of the usage they represent, but they can also point us to code that is defective. It is this automatic defect detection possibility that we will focus on in this chapter.

## 3.2 Finding an Appropriate Abstraction

To find patterns in a set of object usage models we have to choose the abstraction in which we will look for those patterns. This is very important, because depending on the abstraction chosen different patterns can be found, and some can be more valuable and containing more insight than others. One obvious but too extreme abstraction is to treat each object usage model as its own abstraction. A pattern in this setting is just a set of identical models. All models that differ are considered to be violating the pattern. This abstraction is simple to understand and easy to implement. However, it assumes that models do not contain noise, and this assumption is overly simplistic. As an example consider a set of models representing the way files are being used. It can happen that most of those models are identical (i.e. the file is first created, then it is read or written from, and later it is closed). But what if someone checks the size of the file as well? Is she violating the pattern in any way? By the definition we have given above, she does; however, it is obvious that checking the file size does not violate the pattern per se.

Unfortunately, there is no single abstraction that we could call "the right one." The reason for this is as follows: the goal of any kind of abstraction should be to make some models indistinguishable (i.e., abstract away from their details); but how do we know if we have not abstracted away an important difference, or kept a difference that is unimportant? The answer to this question depends in particular on the way the API is supposed to be used, and because this is something we want to learn (i.e., do not have the knowledge a priori), we cannot construct a flawless abstraction. There is a very broad spectrum of abstractions, and many of them can be shown to be useful in some way. In this chapter, we will focus on a simple abstraction based on possible sequencing of method calls when using a class[1]; this will allow us to gently introduce the concept of patterns and anomalies. In Chapter 4, we will show how these concepts can be applied to a much more sophisticated and powerful abstraction.

### 3.2.1 Sequential Constraints Abstraction

For the purpose of finding patterns in object usage we can abstract each model into a set of so-called *sequential constraints* and define a pattern as a set of such constraints. We call this abstraction the *sequential constraints abstraction*. The underlying idea is as follows: Object usage models for a certain class represent the way objects of that class are used throughout the program. The emphasis is on method calls and their ordering. Since

[1] Section 3.2.2 contains some ideas on how to refine this abstraction.

comparing models directly to find the required ordering of method calls is too rigid, we can abstract each model into a set of constraints describing the ordering of method calls represented by the model. This abstraction leads of course to information loss but nevertheless it lets us compare *sets* instead of models, and this is what makes it possible to actually find patterns. If a particular ordering is present in sets abstracted from many (or perhaps all) models for a particular class, this ordering can be considered part of a pattern that has to be followed if objects of that class are to be used consistently or even correctly.

Before we introduce the sequential constraints abstraction formally, let us first define what we understand by a *reachability relation* of a given object usage model:

**Definition 3.1** (Reachability relation)**.** Let $oum = (Q, \Sigma, T, q_0, F, Exc)$ be an object usage model (see definition 2.2). $T' \subseteq Q \times Q$ is called a *reachability relation* of $oum$ iff $T'$ is the transitive closure of $\{(q_1, q_2) : q_1 = q_2 \text{ or } \exists \sigma \,.\, q_2 \in T(q_1, \sigma)\}$.

Intuitively, a pair of states in a specific OUM is a member of the reachability relation of that OUM iff there is a path (potentially empty) between those two states. We will use this relation in the definition of sequential constraints abstraction below:

**Definition 3.2** (Sequential constraints abstraction)**.** Let $oum = (Q, \Sigma, T, q_0, F, Exc)$ be an object usage model (see definition 2.2) and $T' \subseteq Q \times Q$ be the reachability relation of $oum$ (see definition 3.1). The sequential constraints abstraction of $oum$ is defined as $\text{sca}(oum) = \{\sigma_1 \prec \sigma_2 : \exists q_1, q_2, q_3, q_4 \,.\, q_2 \in T(q_1, \sigma_1) \text{ and } (q_2, q_3) \in T' \text{ and } q_4 \in T(q_3, \sigma_2) \text{ and } \sigma_1 \neq \epsilon \text{ and } \sigma_2 \neq \epsilon\}$.

The sequential constraints abstraction of an object usage model contains all pairs of events such that there is a path through the model with the first event in a pair preceding the second event in a pair (perhaps indirectly, as expressed by the reachability relation between the states $q_2$ and $q_3$ in the definition above). This precedence relation is denoted using $\prec$, with $\sigma_1 \prec \sigma_2$ expressing the fact that $\sigma_1$ precedes $\sigma_2$.

Let us illustrate the definition of a sequential constraints abstraction using an example. Consider the object usage model of a `Stack` object shown in Figure 3.1. The sequential constraints abstraction of that model consists of the following sequential constraints:

```
Stack.<init> @ (0) ≺ Stack.push @ (0)
Stack.<init> @ (0) ≺ Stack.elements @ (0)
Stack.push @ (0) ≺ Stack.push @ (0)
Stack.push @ (0) ≺ Stack.elements @ (0)
```

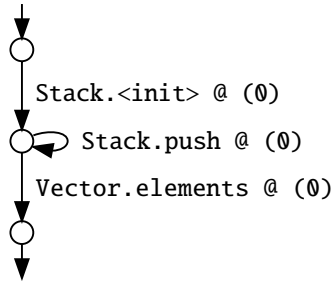



Figure 3.1: OUM for a `Stack` object.

The main benefit of the sequential constraints abstraction is that it can express the sequencing of events that an object goes through using a simple set of constraints. This simplicity, however, is also one of its weaknesses. Consider another object usage model of a `Stack` object, shown in Figure 3.2. It differs from the one in Figure 3.1 in that `push()` is called twice instead of being called in a loop. And yet the sequential constraints abstractions of those two models are identical (i.e., the models are indistinguishable under this abstraction). In general we can say that the sequential constraints abstraction does not differentiate between an event occurring multiple times and once in a loop. Despite this weakness, it allows us to find quite accurate and interesting patterns and their violations, whereas its simplicity makes it easy to understand and facilitates gentle introduction of the concept of patterns and violations.

### 3.2.2 Other Possibilities

Obviously, the sequential constraints abstraction is not the only abstraction possible. There are other possibilities that we did not try out but that remain viable alternatives worth investigating. Generally, there are two directions possible: improving the sequential constraints abstraction, or coming up with an entirely new abstraction. In Chapter 4 we will introduce a new, much more powerful abstraction based on temporal logic formulas. For now, though, let us concentrate on possible ways of improving the sequential constraints abstraction:

**"Must" and "succeeds" constraints.** Constraints in the sequential constraints abstraction are "may" constraints (i.e., $\sigma_1 \prec \sigma_2$ iff $\sigma_1$ *may* precede $\sigma_2$). This means that if there is *any* path through the OUM where $\sigma_1$ pre-

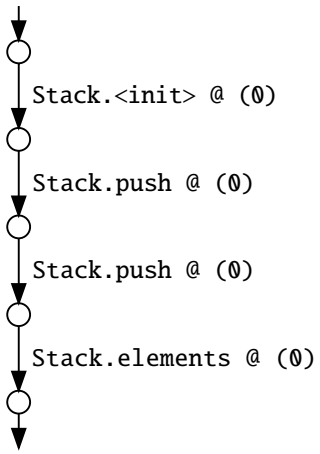



Figure 3.2: Hypothetical OUM for a `Stack` object with the same sequential constraints abstraction as the OUM shown in Figure 3.1.

cedes $\sigma_2$, this constraint will be present in the abstraction; however, it is easy to come up with examples where an event must precede another event on *all* paths for the usage to be correct, e.g, whenever an initialization is required before anything can be done. Adding "must" constraints could deal with this issue, but it introduces a distinction that is currently not present and would have to be dealt with as well: for "may" constraints, stating that $\sigma_1$ may precede $\sigma_2$ and that $\sigma_2$ may succeed $\sigma_1$ is equivalent, but not so for "must" constraints. The solution would mean differentiating between "must precede" and "must succeed" constraints.

**"Precedes all" and "Precedes first" constraints.** If we add the aforementioned "must" constraints, we become able to differentiate between "must precede first" and "must precede all" constraints.[2] The idea here is that some method has to be called just once, and then another method can be call as many times as needed. This is the case, for example, with `List.size()` and `List.get()` methods; once the first gets called, we can call the second as many times as needed, because we have the ability to check if the index is not outside the boundaries. But there is also another possibility, namely, that some method must be called before every call to some other method. A good example here is the

[2] "may precede first" and "may precede all" are do not seem as useful.

pair `Iterator.hasNext()` and `Iterator.next()`. Similarly we could add "must succeed last" and "must succeed all" constraints. Distinguishing between those two possibilities might add useful information to the patterns and thus increase their overall accuracy and usefulness.

**Contextual constraints.** It would be interesting to add contextual information to constraints (e.g., "Typically `Iterator.hasNext()` is called inside a conditional expression of a `while` loop, and `Iterator.next()` in its body"). This might help detect programming constructs that are typically used together with certain methods. This would, however, require the object usage models to be extended as well, because currently contextual information is not represented by them. A more limited extension to the constraints (and one that does not requiring changes to object usage model) is just differentiating between events that happen in a loop and those that happen outside of any loop. The main challenge here would be to develop an approach for dealing with nested loops.

## 3.3 Detecting Whole-Program Patterns

### 3.3.1 General Approach

After abstracting object usage models into sequential constraints we can start looking for patterns amongst those constraints. Finding patterns amongst models of objects of the same class gives us information about how objects of this class are typically used; however, what we would like to learn are patterns that also show how objects of several classes are typically used *together*. To achieve this goal, we do not look for patterns amongst object usage models, but amongst methods those models stem from. In other words, we now abstract each method into a set of sequential constraints that have been created from object usage models that stem from that method. An example shown in Figure 3.3 will make it clear: We start with the `copy` method shown in 3.3(a). There are exactly four abstract objects used by `copy` (see definition 2.4): the two input objects `out` and `in`, the return value of a call to `Vector.set` at the end of the loop, and the return value of a call to `Vector.get` near the end of the loop. Their object usage models and those models' sequential constraints abstractions are shown in Figures 3.3(b)–(d). `copy`'s sequential constraints abstraction is now simply a *union* of those four sequential constraints abstractions, as shown in Figure 3.3(f). Formally, the method's sequential constraints abstraction is defined as follows:

**Definition 3.3** (Method's sequential constraints abstraction)**.** Let $m$ be a method, Obj($m$) be the set of abstract objects used by $m$ (see definition 2.4), and

let $oum(x)$ be defined as an object usage model of an abstract object $x$. Let $oums(m)$ be the set of object usage models stemming from $m$, defined as $oums(m) = \{oum(x) : x \in \text{Obj}(m)\}$. $m$'s *sequential constraints abstraction* is defined as $\text{sca}(m) = \bigcup_{oum \in oums(m)} \text{sca}(oum)$.

Now we can look for sequential constraints that are common to many methods and thus discover patterns amongst those methods. These might turn out to be limited to objects of one class only, but we now also have the ability to discover patterns encompassing several objects of different classes. The general idea we make use of is as follows: if a particular set of sequential constraints is *common* to many methods, it is a *pattern* shared by those methods. Formally, we can define a pattern in the following way:

**Definition 3.4** (Pattern, support, size, closed pattern)**.** Let $m_1, \ldots, m_n$ be methods. $P$ is a *pattern* supported by $s$ methods iff $|\{m_i : P \subseteq \text{sca}(m_i)\}| = s$. $s$ is called the *support* of $P$. $|P|$ is called the *size* of $P$. $P$ is a *closed pattern* iff for all $P' \supset P$ we have $|\{m_i : P' \subseteq \text{sca}(m_i)\}| < s$.

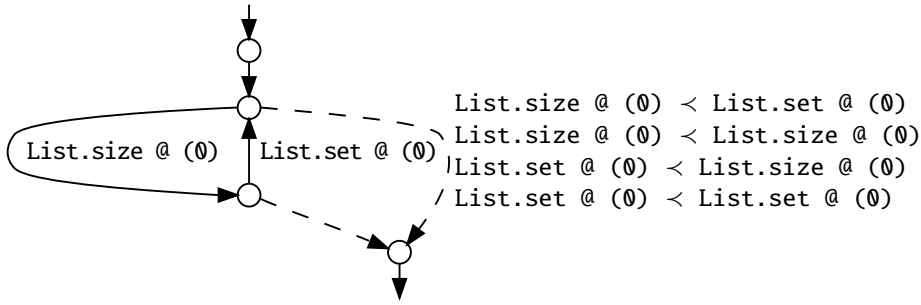
Speaking informally, a *pattern* is a set of sequential constraints that occurs in sequential constraints abstractions of a number of methods, and the support of a pattern is that number. The *size* of a pattern is the number of sequential constraints that constitute it. A pattern is *closed* if adding even a single sequential constraint to it would cause the support of the resulting pattern to drop (i.e., at least one of the methods that support the original pattern does not support the new one). We will normally use the term "pattern" to mean "closed pattern", and all exceptions will be explicitly stated.

To detect patterns as defined above we use *formal concept analysis* (Ganter and Wille 1999). Formal concept analysis takes as an input a set of conceptual objects, a set of conceptual properties, and a cross table representing an association between conceptual objects and conceptual properties that hold for those objects. The idea is to find sets of conceptual properties that are common to a number of conceptual objects. If we take methods to be conceptual objects and sequential constraints to be conceptual properties, this boils down to finding patterns as defined by us above.

Figure 3.4 shows a sample cross table with conceptual objects represented by rows and conceptual properties represented by columns. Generally speaking, discovering all patterns in such a setting is equivalent to discovering all filled rectangles in the cross table. (Not every rectangle needs to be contiguous—it is enough if there is a transposition of columns and rows in the cross table such that the rectangle becomes contiguous). Each filled rectangle corresponds to a pattern (not necessarily closed; this depends on the rectangle itself), where the rectangle's columns are the properties that

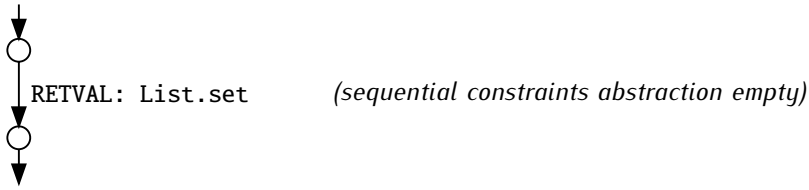
```
static void copy (List out, List in, int from, int to) {
    for (int index = from; index < to; index++) {
        if (index >= out.size ()) {
            return;
        }
        out.set (index, in.get (index));
    }
}
```

(a) Sample input method copy.



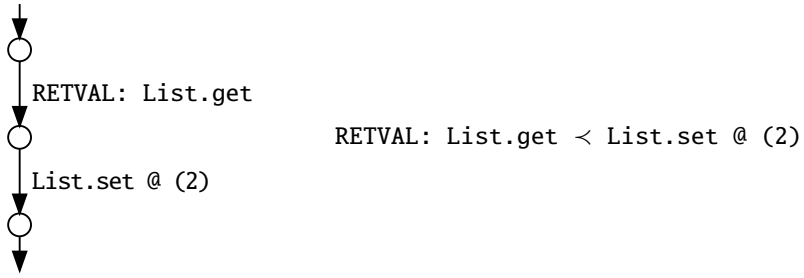

(b) Object usage model for out and its sequential constraints abstraction.


(c) Object usage model for in and its sequential constraints abstraction.




(d) Object usage model for the return value of Vector.set with its (empty) sequential constraints abstraction.

Figure 3.3: Creating method's sequential constraints abstraction. (Continued on next page.)




(e) Object usage model for the return value of `Vector.get` and its sequential constraints abstraction.

```
List.get @ (0) ≺ List.get @ (0)
List.size @ (0) ≺ List.set @ (0)
List.size @ (0) ≺ List.size @ (0)
List.set @ (0) ≺ List.size @ (0)
List.set @ (0) ≺ List.set @ (0)
RETVAL: List.get ≺ List.set @ (2)
```

(f) Sequential constraints abstraction of the `copy` method.

Figure 3.3: Creating method's sequential constraints abstraction. (Continued.)

form the pattern, and its rows are objects that support the pattern (see Figure 3.4). The support of the pattern shown in the Figure 3.4 is 4 (because there are four conceptual objects that exhibit this pattern), and the size of the pattern is 2 (because the pattern consists of two conceptual properties). We will not go into more detail on formal concept analysis here; the reader can find additional information elsewhere (Lindig 2007).

### 3.3.2 Fine-tuning the Approach

The number and size of patterns that will be discovered can be influenced by adjusting two input parameters of formal concept analysis: *minimum size* and *minimum support*. *Minimum size* specifies the minimum number of conceptual properties (in our case—sequential constraints) that a pattern must consist of if it is to be reported to the user. In our case this parameter is best set to 1, so that we find patterns of all possible sizes. The reason for this is that even if a pattern consists of just one sequential constraint, it represents a relationship between two events, and any such relationship might be of interest.

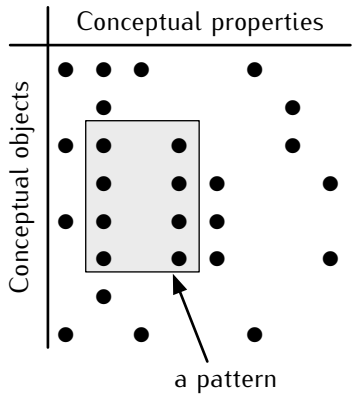



Figure 3.4: Sample cross table input to a formal concept analysis. The (noncontiguous) rectangle in the cross table represents a sample pattern (Lindig 2007).

*Minimum support* specifies the minimum number of conceptual objects (in our case—methods) that must exhibit a pattern if it is to be reported to the user. Finding the optimal value of minimum support to use is difficult, if at all possible. The problem here is that judging a pattern's usefulness by the number of times it occurs is misleading. There can be very important, strong, and useful patterns that occur very rarely simply because they pertain to a very specific event that does not occur very often. Unfortunately, we cannot simply set the minimum support to 1, as we do with minimum size, because there would be too many useless patterns in the results. We do not have a good solution to this problem, and we have arbitrarily decided to use 20 as the minimum support value in our experiments. We will later show that this number is good enough for our purposes (i.e., detecting methods that violate one or more patterns in the hope that these are defective; see Section 3.4.3).

On the other hand, there exist events that happen almost everywhere (such as calls to `StringBuffer.toString ()`), yet patterns these frequent occurrences yield are useless from a practical point of view. The phenomenon here is that those patterns come to be not because this is the way those APIs are supposed to be used. Instead, these patterns are due to the *scale effect*: these APIs are used so often that almost every possible combination of sequential constraints related to them has a large chance of occurring often enough to become a pattern. One idea to deal with fake patterns related to too frequently occurring events is to simply ignore such events when calculating the sequential constraints abstraction. We can identify too frequently occurring events by calculating how often (i.e., in how many object usage

Table 3.1: Patterns found in the case study subjects.

| Program | Sequential constraints | Patterns | Time (mm:ss) |
|---|---|---|---|
| Vuze 3.1.1.0 | 178,034 | 2,887 | 0:19 |
| AspectJ 1.5.3 | 290,041 | 583 | 0:18 |
| Apache Tomcat 6.0.18 | 89,793 | 98 | 0:09 |
| ArgoUML 0.26 | 133,575 | 387 | 0:10 |
| Columba 1.4 | 58,936 | 245 | 0:06 |
| Act-Rbot 0.8.2 | 27,557 | 56 | 0:05 |

models) each event occurs and finding the outliers. Unfortunately, just as in the case of minimum support, there exist events that occur frequently, but always (or most of the time) follow the same usage pattern. During preliminary experiments we have noticed that there are three Java classes that contain mostly methods of the kind described above: `StringBuffer`, `String`, and `StringBuilder`. Intuitively, the reason for this is that these classes are not really stateful (i.e., they act more like data containers that can be accessed any time in any way). It seems very likely that it is this characteristic that separates classes that do have object usage patterns from those that do not. Unfortunately, finding such classes automatically is a problem, if only because we need a precise definition of what we are looking for. We have decided to simply ignore all the events related to one of the three classes mentioned above. Better solutions are definitely possible, but we will leave them as potential future extensions of this work.

### 3.3.3 Case Study

We have implemented the approach described above and applied it to the projects listed in Table 2.1. As a formal concept analysis tool, we have used Colibri/Java (Götzmann 2007). Table 3.1 lists all the projects and the results obtained for them: number of distinct sequential constraints found, number of patterns found, and the time (wall clock time, averaged over ten consecutive runs) that was needed to perform the analysis on a 2.53 GHz Intel Core 2 Duo machine with 4 GB of RAM.

Detecting patterns is fast, with less than half a minute needed per project. The number of patterns found varies from relatively few (e.g., 56 for Act-Rbot) to quite many relative to the size of the project (e.g., 2887 for Vuze compared to 583 for AspectJ). As we will see later (see Section 3.4.3), the less patterns found, the better—at least when it comes to defect detection capabilities. The intuitive explanation of this fact is that the less different patterns, the more

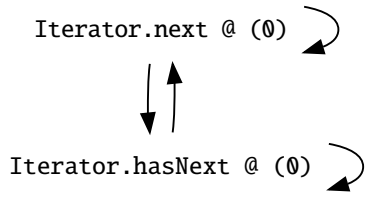



Figure 3.5: The "iterator" pattern found in all projects from Table 3.1.

homogenous the project, and the bigger the probability that any deviation from a pattern is a real defect as opposed to just being an irregularity.

Let us now introduce a graphical representation of patterns and then take a look at some of the patterns found. Patterns are sets of sequential constraints (cf. definition 3.4), but representing them in this way makes them difficult to understand. Instead, we will use a graphical representation that will make the data flow represented by the pattern easier to follow. Consider a pattern $P = c_1, \ldots, c_n$ with each $c_i$ being a sequential constraint $\sigma_{i1} \prec \sigma_{i2}$. We will represent a pattern as a directed graph $G = (V, E)$, where $V = \{\sigma : \exists i, \sigma' . c_i \in P \text{ and } (c_i = \sigma \prec \sigma' \text{ or } c_i = \sigma' \prec \sigma)\}$ and $E = \{(\sigma_a, \sigma_b) : (\sigma_a \prec \sigma_b) \in P\}$. Let us consider the following pattern:

```
Iterator.hasNext @ (0) ≺ Iterator.hasNext @ (0)
Iterator.hasNext @ (0) ≺ Iterator.next @ (0)
Iterator.next @ (0) ≺ Iterator.hasNext @ (0)
Iterator.next @ (0) ≺ Iterator.next @ (0)
```

If we assume $\sigma_1$ = `Iterator.hasNext @ (0)` and $\sigma_2$ = `Iterator.next @ (0)`, then this pattern can be represented as a directed graph $G = (V, E)$, where $V = \{\sigma_1, \sigma_2\}$ and $E = \{(\sigma_1, \sigma_1), (\sigma_1, \sigma_2), (\sigma_2, \sigma_1), (\sigma_2, \sigma_2)\}$. This graph (with full names of nodes instead of symbols) is shown in Figure 3.5. The pattern it represents can be found in all projects listed in Table 3.1 and it occurs very frequently. Its support (see definition 3.4) ranges from 176 for Act-Rbot to 876 for ArgoUML. This pattern illustrates the most common way to use iterators, namely combining calls to `hasNext()` with calls to `next()`. It also shows one of the weaknesses of the sequential constraints abstraction: information about the alternating character of these calls is lost.

Figure 3.6 shows one of the patterns found in Vuze. This pattern (with support of 77) illustrates how instances of the `DERSequence` class are created. One of the constructors of `DERSequence` takes as a parameter an object of type `DEREncodableVector`, and a call to exactly this constructor is part

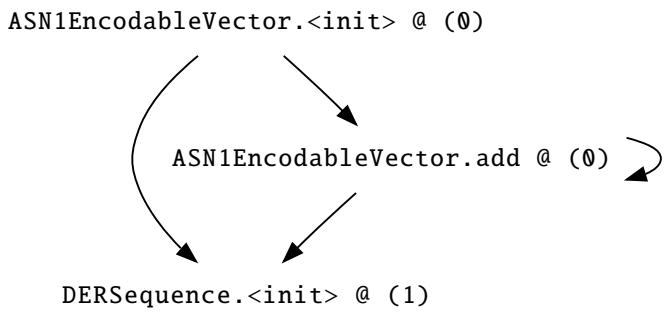



Figure 3.6: Pattern from Vuze illustrating how instances of the `DERSequence` class are created.

of the pattern. However, the pattern shows that it is an instance of the `ASN1EncodableVector` that is passed to the constructor instead. A quick look at the source code of `DEREncodableVector` explains everything: There is a comment stating that this class has been superseded by the `ASN1EncodableVector` class, and thus the latter should be used in preference to the former.

Figure 3.7 shows a pattern with support 52 found in Act-Rbot. This pattern illustrates how some parts of the database API (executing a query and accessing its results) are intended to be used. As can be seen, the pattern is very complicated, but we can simplify it a little bit by discarding events that represent method calls ending in an exception being thrown. The remaining part of the pattern is shown in Figure 3.8. This pattern consists of four clearly separated "subpatterns"[3]:

- The connection that is used to create the statement to be executed (via the call to `createStatement()` of the `Connection` class) must be freed by passing it as the second parameter to the `freeConnection()` method of the `DBConnectionManager` class.
- The `DBConnectionManager` class is a singleton, and we can get its instance by calling the static method `getInstance()` on it.
- After creating the statement to be executed (via the call to `createStatement()`), we execute it (by calling `executeQuery()` on it), and must close it afterwards (by calling `close()` on it).

[3]We use the term informally here. We treat each connected component of the graphical representation of a pattern as a subpattern.

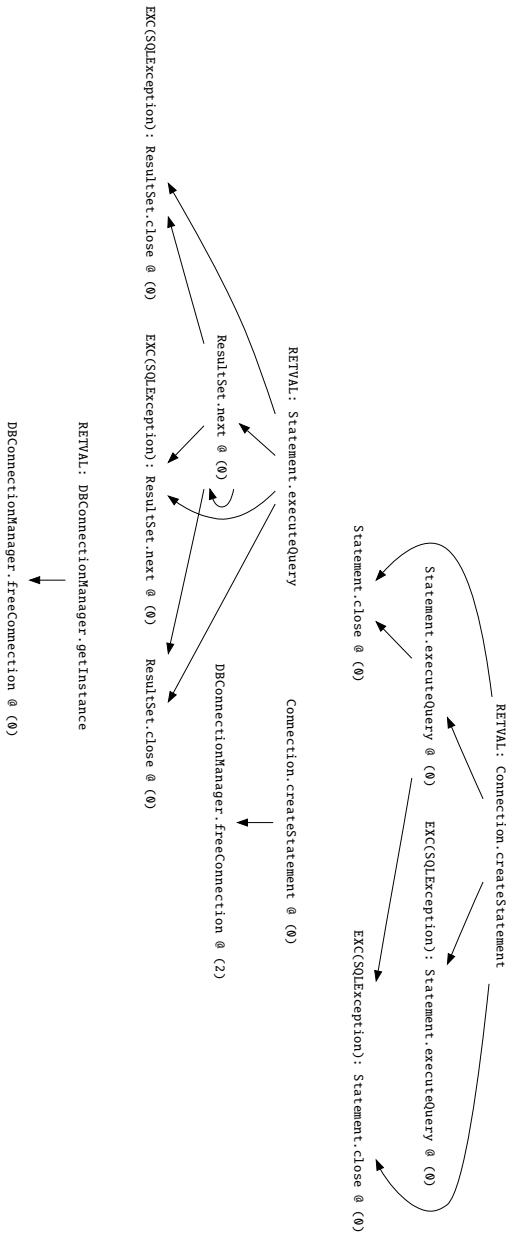



Figure 3.7: Pattern from Act-Rbot illustrating database API usage.

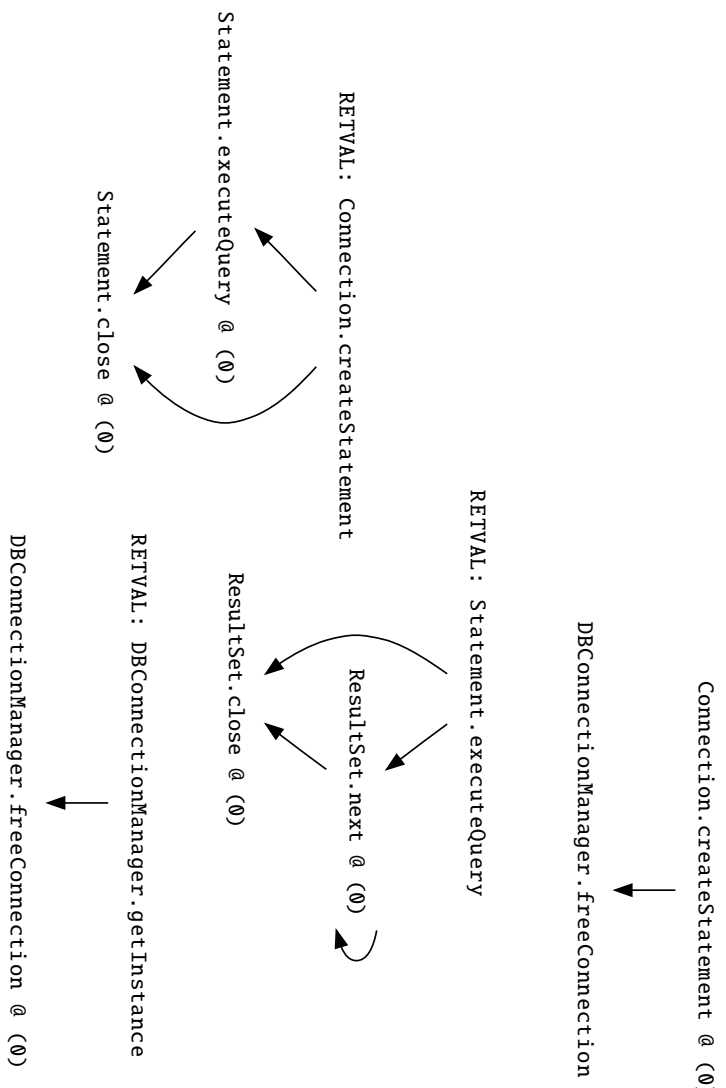



Figure 3.8: Part of the pattern shown in Figure 3.7, illustrating database API usage. Events representing method calls that end in an exception being thrown were removed to make the pattern more comprehensible.

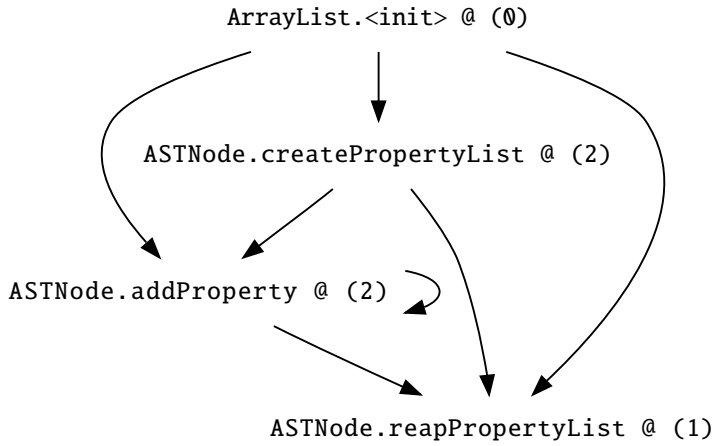



Figure 3.9: Pattern from AspectJ illustrating how property lists are constructed.

- The call to `executeQuery()` returns an instance of the `ResultSet` class. We can iterate through the rows returned by repeatedly calling `next()` on the result set. After extracting the results we must close the result set (via the call to `close()` on it).

As we can see, this pattern is very informative and contains a lot of information on the interplay between different classes and their methods.

Figure 3.9 shows a pattern from AspectJ illustrating how property lists are created. This pattern's support is 71. Incidentally, the Java method shown in Figure 2.2 is an example of code that adheres to this pattern.

All the patterns were mined using 20 as minimum support, 1 as minimum size, and with filtering of events related to `StringBuffer`, `String`, and `StringBuilder` turned on (cf. Section 3.3.2). We will use these settings throughout this dissertation, but before we proceed further, let us take a look at how the number of patterns changes if we change those values. Figure 3.10 shows the influence of the minimum support on the number of patterns found in AspectJ. Minimum size was fixed at its default value—1, and filtering was turned on as well. We can see that the number of patterns decreases exponentially with the increase in the minimum support value. Overall, a minimum support of 10 results in 2324, and a minimum support of 30 in 290 patterns being found. Figure 3.11 shows the influence of minimum size on the number of patterns found in AspectJ. Minimum support was fixed at its default value—20,

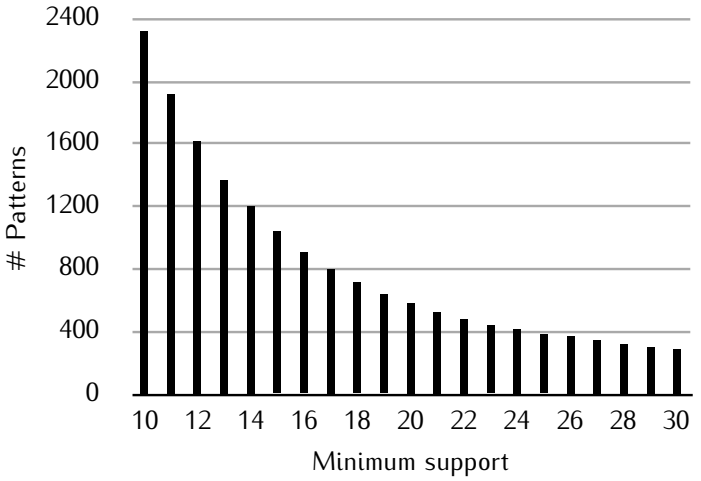



Figure 3.10: Influence of minimum support on the number of patterns found in AspectJ (minimum size fixed at 1).

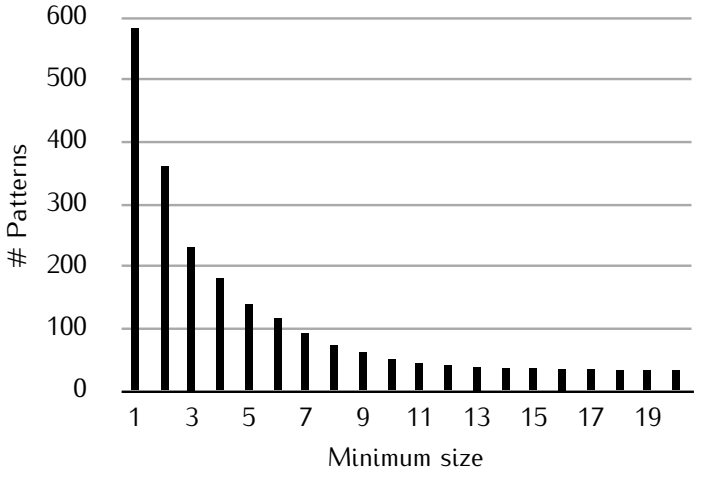



Figure 3.11: Influence of minimum size on the number of patterns found in AspectJ (minimum support fixed at 20).

and filtering was turned on as well. Again, the number of patterns decreases exponentially with the increase in the minimum size value, but this time the effect is much more pronounced. Overall, a minimum size of 1 (the default value) results in 583, and a minimum size of 20 in 33 patterns being found. When it comes to filtering, after turning it off (with minimum support and minimum size fixed at their default values) the number of patterns found in AspectJ goes up to 1906, which is a more than threefold increase.

## 3.4 Detecting Anomalous Methods

### 3.4.1 Anomalies as Missing Functionality

Having found patterns, we can now turn to the task we wanted to focus on from the very beginning: detecting methods that *violate* one or more patterns. Our hope here is that these methods will be defective and that the violations they exhibit will pinpoint the problem. More precisely, our sequential constraints abstraction abstracts each method in the program into a set of sequential constraints (cf. definition 3.3), and a pattern is a set of sequential constraints that are common to a number of methods. Now, given a pattern, we may ask the following question: *which methods violate this pattern?* Of course we must assume that there is something that the pattern and the method have in common (i.e., there must be sequential constraints that are present both in the pattern and in the method's sequential constraints abstraction). Otherwise there is no violation, but just a plain disjunction. Hence, a violation of a pattern occurs when there are sequential constraints that are common both to the pattern and the violating method, and there are also constraints that are present in the pattern, but not in the method:

**Definition 3.5** (Violation, deviation, deviation level)**.** Let $m$ be a method and $P$ be a pattern. Let $sca(m)$ be $m$'s sequential constraints abstraction (see definition 3.3). $m$ *violates* $P$ iff $P \cap sca(m) \neq \emptyset$, and there exists $p \in P$ such that $p \notin sca(m)$. $D(P, m) = P \setminus sca(m)$ is the *deviation* of the violation. $|D(P, m)|$ is the *deviation level* of the violation.

Consider the `replaceName()` method shown in Figure 3.12. Its sequential constraints abstraction looks as follows:

```
ASTNode.changeName @ (0) ≺ ASTNode.postReplaceChild @ (0)
ASTNode.preReplaceChild @ (0) ≺ ASTNode.changeName @ (0)
ASTNode.preReplaceChild @ (0) ≺ ASTNode.postReplaceChild @ (0)
ASTNode.preReplaceChild @ (1) ≺ ASTNode.postReplaceChild @ (1)
ASTNode.preReplaceChild @ (1) ≺ ASTNode.postReplaceChild @ (2)
ASTNode.preReplaceChild @ (2) ≺ ASTNode.changeName @ (1)
```

```
public void replaceName (ASTNode root, ASTNode old, ASTNode new) {
    root.preReplaceChild (old, new, NAME_PROPERTY);
    root.changeName (new);
    root.postReplaceChild (old, old, NAME_PROPERTY);
}
```

Figure 3.12: Sample source code using `pre-` and `postReplaceChild()` methods in a way that violates a pattern.

Now consider the following pattern $P_{pre\_post}$:

```
ASTNode.preReplaceChild @ (0) ≺ ASTNode.postReplaceChild @ (0)
ASTNode.preReplaceChild @ (1) ≺ ASTNode.postReplaceChild @ (1)
ASTNode.preReplaceChild @ (2) ≺ ASTNode.postReplaceChild @ (2)
```

Let us represent the three sequential constraints that constitute $P_{pre\_post}$ as $\sigma_1, \sigma_2, \sigma_3$, respectively (so $P_{pre\_post} = \{\sigma_1, \sigma_2, \sigma_3\}$). The `replaceName()` method violates the pattern $P_{pre\_post}$, because $P_{pre\_post} \cap sca(\texttt{replaceName()}) = \{\sigma_1, \sigma_2\} \neq \emptyset$, and there exists $p = \sigma_3 \in P_{pre\_post}$ such that $p \notin sca(\texttt{replaceName()})$. The *deviation* of this violation is $D(P_{pre\_post}, \texttt{replaceName()}) = P_{pre\_post} \setminus sca(\texttt{replaceName()}) = \{\sigma_3\}$. Since this set has one element, the *deviation level* of this violation is 1.

The definition we have just given allows us not only to identify violations but also gives us a way to calculate the *deviation level* of a violation which is a measure of how strongly a method deviates from a pattern. There is also another important property, which we have not mentioned yet: it is the *confidence* of the deviation. As its name implies, it is supposed to give us a measure of confidence in that the deviation is a real anomaly and a potential defect, and not a harmless difference:

**Definition 3.6** (Confidence)**.** Let $m_1, \ldots, m_n$ be methods and $P$ be a pattern supported by $s$ methods. Let $m_i$ violate $P$ and $D(P, m_i)$ be the deviation of the violation. Let $n_v = |\{m_j : D(P, m_j) = D(P, m_i)\}|$ be the number of methods that violate the same pattern in the same way. $s/(s + n_v)$ is the *confidence* of the deviation $D(P, m_i)$.

Consider again the pattern $P_{pre\_post}$ shown above. If the support of $P_{pre\_post}$ is 133 (i.e., there are 133 methods that adhere to it), and there are seven methods, $m_1, \ldots, m_7$, for which $D(P_{pre\_post}, m_i) = D(P_{pre\_post}, \texttt{replaceName()})$ (i.e., those seven methods violate $P_{pre\_post}$ in the same way as the `replaceName()` method shown in Figure 3.12[4]), then the confidence of the deviation $D(P_{pre\_post}, \texttt{replaceName()})$ is equal to $133/(133 + 7) = 133/140 = 0.95$.

[4] `replaceName()` is one of them, so there must be 6 *other methods* with the same violation.

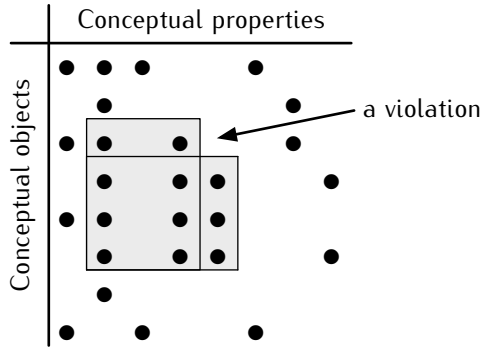



Figure 3.13: Sample cross table input to a formal concept analysis. The two (non-contiguous) rectangles in the cross table represent a violation (Lindig 2007).

Confidence of a deviation tells us how often the pattern is adhered to relative to how often it is being violated in the same way. This extends straightforwardly to violations as well, giving us the notion of a confidence of a violation. Confidence and deviation level give us some quantitative data about each violation.

To detect violations as we have defined them, we can use the results returned by concept analysis. The idea, first presented by Lindig (2007), is as follows: if we have two closed patterns $P_1$ and $P_2$, with $P_1$ being a proper subset of $P_2$, the support of $P_1$ must be strictly greater than support of $P_2$ (otherwise $P_1$ is not closed; cf. definition 3.4). This means that there are methods that adhere to $P_1$ but do not adhere to $P_2$. In other words, properties from the set $P_1$ are present in those methods' sequential constraints abstractions, but those from the set $P_2 \setminus P_1$ are not. According to the definition of a violation given above (definition 3.5), these methods violate the pattern $P_2$. Similarly we can show that if a set of methods violates a pattern $P_2$, there must exist a pattern $P_1 \subset P_2$ such that only those methods adhere to it.

Consider the cross table shown in Figure 3.13. There are two patterns marked as rectangles in this cross table, with one pattern being a proper subset of another. These two patterns induce the violation shown in the Figure. The deviation level of this violation is 1. This is because there is only one conceptual property that is present in one pattern, but not the other. The confidence of the violation is 0.75. This is because the pattern being violated (the one with the bigger size [i.e., wider] and the smaller support [i.e., smaller]) is supported by three objects, and there is only one conceptual object violating this pattern. Therefore, the confidence of the violation is $3/(3+1) = 0.75$. We will not go into more detail here, the reader can consult the work by Lindig (2007) for a thorough discussion of this topic.

### 3.4.2 Ranking violations

The procedure described above can lead to quite a lot of violations being found, not all of them being worthy of investigation. The reason for this is that it is based on purely statistical reasoning: *anomalous behavior is likely to be erroneous behavior* (Engler et al. 2001). But it can also happen that anomalous behavior is just that: a harmless anomaly. To mitigate this problem, we rank violations in the hope that the ones that point to erroneous behavior will be placed higher than the ones that are harmless.

To rank violations, we treat them as so-called "association rules". Originally this term was introduced by Agrawal, Imieliński, and Swami (1993) in a work on analyzing market-basket data. Agrawal, Imieliński, and Swami's idea was to find rules of the form $X \Rightarrow y$, where $X$ is a set of items and $i$ is an item. The meaning of the rule is "clients that buy items from the set $X$ typically also buy an item $y$". The word "typically" is used to represent a modifiable threshold (typically confidence) that is used for mining such association rules. The original definition of an association rule by Agrawal, Imieliński, and Swami is as follows:

> "Let $\mathcal{I} = I_1, I_2, \ldots, I_m$ be a set of binary attributes, called items. Let $T$ be a database of transactions. Each transaction $t$ is represented as a binary vector, with $t[k] = 1$ if $t$ bought the item $I_k$, and $t[k] = 0$ otherwise. There is one tuple in the database for each transaction. Let $X$ be a set of some items in $\mathcal{I}$. We say that a transaction $t$ *satisfies* $X$ if for all items $I_k$ in $X$, $t[k] = 1$.
>
> By an *association rule*, we mean an implication of the form $X \Rightarrow I_j$, where $X$ is a set of some items in $\mathcal{I}$, and $I_j$ is a single item in $\mathcal{I}$ that is not present in $X$."

We can easily notice a very close similarity between the definition of an association rule above and our definition of a violation (definition 3.5). The only important difference is that Agrawal, Imieliński, and Swami—to use the terminology used in this dissertation—only allow a deviation level of 1. It is therefore only natural to treat violations as "extended" association rules $X \Rightarrow Y$, where both $X$ and $Y$ are sets. Let us give the precise definition for this "translation":

**Definition 3.7** (Violation's association rule)**.** Let $m$ be a method and $P$ be a pattern. Further, let us assume a violation of $P$ in $m$, and let $D(P, m)$ be the deviation of this violation. The *violation's association rule* is $X \Rightarrow Y$, where $X = P \setminus D(P, m)$ and $Y = D(P, m)$.

We can now treat every violation as its association rule, as defined above. Also, it is easy to show that an association rule uniquely defines a violation.

This allows us to approach our problem (ranking violations) from a slightly different viewpoint: instead of ranking violations, we can rank their association rules. One effective method of ranking was proposed by Brin et al. (1997): using so-called *conviction* measure. Original definition of conviction by Brin et al. is as follows: $conviction(X \Rightarrow Y) = P(X)P(\neg Y)/P(X, \neg Y)$, where $P(X)$ is the probability of $X$ occurring. Conviction measures the deviation of $X \wedge \neg Y$ from independence (i.e., how much more often would the pattern be violated if $X$ and $Y$ were independent; note that, logically, $X \Rightarrow Y$ is equivalent to $\neg(X \wedge \neg Y)$). The original formula can be alternatively expressed as $conviction(X \Rightarrow Y) = (n - s_Y)/(n * (1 - c_{XY}))$, where $s_Y$ is the support of the set $Y$, $c_{XY}$ is the confidence of the violation, and $n$ is the normalization factor (for sequential constraints abstraction: the number of methods in the program being analyzed). We rank all violations according to their conviction measures such that the user can focus on the top-ranked violations instead of investigating all of them.

After ranking, we filter away all violations with conviction values $\leq 1.25$. This threshold is quite arbitrary, but the same value was used by Brin et al.. Next, we remove violations that look like other, higher-ranked, violations. We follow here again the approach of Brin et al.: Let $X_1 \Rightarrow Y$ and $X_2 \Rightarrow Y$ be violations' association rules with $X_1 \subset X_2$. If conviction value of $X_1 \Rightarrow B$ is at least as high as conviction value of $X_2 \Rightarrow B$, the latter will be removed. This step compares *all pairs of violations*, independent of the method that exhibits the violation. Therefore, it can happen that a violation exhibited by method $m_1$ will get filtered away because another method, $m_2$ exhibits a similar (in the sense of the description above), but higher-ranked violation. This can lead to defect-uncovering violations being removed, and we have indeed observed that such things happen. However, most of the time the removed violations are either duplicates (if the methods are the same) or false positives (if they are different). In cases where true positives are filtered away, most of the time the higher-ranked violation (that was kept) turns out to be a true positive as well—so fixing the problem and then rerunning the analysis uncovers the next-ranked violation. In any case, we feel that missing a few true positives is a small price to pay for a significantly higher true positives rate.

### 3.4.3 Experimental results

We have implemented the procedure described above in a tool called JADET[5] and run it on all the projects shown in Table 2.1. Summary of the results can be found in Table 3.2. For each project we report on the total number of violations found and on the time (wall clock time, averaged over ten con-

[5]Java Anomaly DETector

Table 3.2: Violations found in the case study subjects.

| Program | Violations | Time (mm:ss) |
|---|---|---|
| Vuze 3.1.1.0 | 890 | 0:33 |
| AspectJ 1.5.3 | 163 | 0:20 |
| Apache Tomcat 6.0.18 | 16 | 0:09 |
| ArgoUML 0.26 | 128 | 0:11 |
| Columba 1.4 | 113 | 0:07 |
| Act-Rbot 0.8.2 | 11 | 0:05 |

secutive runs) that was needed to perform the analysis on a 2.53 GHz Intel Core 2 Duo machine with 4 GB of RAM. As we can see, there is a large difference between the number of violations found in different projects, even for projects that are of similar size (like Apache Tomcat and ArgoUML, cf. Table 2.1). Ideally we would investigate every single violation found in each project and classify it to find out how many of those are true positives (more on that below). While for Apache Tomcat and Act-Rbot it is entirely possible to investigate all the violations, for Vuze 3.1.1.0 the number of violations found is so large that this would be a time-consuming and mundane task. Therefore, we have investigated 10 top-ranked and 10% top-ranked violations for each project and classified them into three 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. One example is if a comment in the base class states that a particular method accepts null values passed as a parameter, but the implementation of that method in the derived class does not. If this method is public, we still mark it as defective, because someone may cause it to fail by following the contract of the base class. Example of a code classified as a defect can be found in Figure 3.14.

**Code smells.** This category contains all violations that are not defects, but the violating methods have properties indicating that something may go wrong (Fowler 1999) 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 (See Figure 3.15).

```
public void execute (..., List args)
{
    if (args.isEmpty())
    {
        ...
        return;
    }
    ...
    String arg = (String)args.remove(0);
    if (args.isEmpty() && "list".equalsIgnoreCase(arg))
    {
        ...
        return;
    }
    File path = new File ((String)args.get(0));
    ...
}
```

Figure 3.14: A Vuze defect found by JADET. If the `args` list contains just one element, different than "`list`", this method will throw an `IndexOutOfBoundsException`.

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

Figure 3.15: Example of a code smell, coming from AspectJ. The loop body is executed at most once—but this is not a defect, since the collection iterated through can have at most one element.

Table 3.3: Classification results for top 10 violations in each project. "CSs" stands for the number of code smells. "FPs" stands for the number of false positives.

| Program | Classified | Defects | CSs | FPs | Effectiveness |
|---|---|---|---|---|---|
| Vuze 3.1.1.0 | 10 | 1 | 4 | 5 | **50%** |
| AspectJ 1.5.3 | 10 | 1 | 2 | 7 | **30%** |
| Apache Tomcat 6.0.18 | 10 | 0 | 4 | 6 | **40%** |
| ArgoUML 0.26 | 10 | 0 | 5 | 5 | **50%** |
| Columba 1.4 | 10 | 2 | 4 | 4 | **60%** |
| Act-Rbot 0.8.2 | 10 | 1 | 4 | 5 | **50%** |
| **Overall** | 60 | 5 | 23 | 32 | **47%** |

Table 3.4: Classification results for top 10% violations in each project. "CSs" stands for the number of code smells. "FPs" stands for the number of false positives.

| Program | Classified | Defects | CSs | FPs | Effectiveness |
|---|---|---|---|---|---|
| Vuze 3.1.1.0 | 89 | 1 | 17 | 71 | **20%** |
| AspectJ 1.5.3 | 16 | 1 | 4 | 11 | **31%** |
| Apache Tomcat 6.0.18 | 1 | 0 | 0 | 1 | **0%** |
| ArgoUML 0.26 | 12 | 0 | 6 | 6 | **50%** |
| Columba 1.4 | 12 | 2 | 4 | 6 | **50%** |
| Act-Rbot 0.8.2 | 1 | 1 | 0 | 0 | **100%** |
| **Overall** | 131 | 5 | 31 | 95 | **27%** |

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

The results of this classification can be found in Tables 3.3 (for 10 top-ranked violations in each project) and 3.4 (for 10% top-ranked violations in each project). For each project we report on the number of violations that were classified[6], the number of defects, code smells (CSs), false positives (FPs), and the effectiveness (i.e., the percentage of violations that were defects or code smells). We also report on the overall effectiveness.

We can see at a glance that, overall, investigating top 10 violations is much more efficient than investigating top 10% violations. Not only is the overall effectiveness for top 10 violations higher (47% vs. 27%), but the increase in

[6]In Table 3.4 this is sometimes more than exactly 10%. The reason for this is that some violations have the same ranking (i.e., the same conviction value), so we had to include all such equally-ranked violations.

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

    indexOfInt(o, index);
    arrayrefOfArrayType(o, arrayref);
}
```

Figure 3.16: The defect found by JADET in AspectJ. This method verifies a `CALOAD` bytecode instruction, but misses one check: if the array contains elements of type `char`.

the number of defects and code smells found is much smaller than the increase in the number of violations that had to be classified (only 8 additional true positives for 81 additionally classified violations), so we get diminishing returns on our investigations. This is actually a good thing, because it suggests that our ranking system is effective in placing true positives high in the ranking. Let us now take a look at some of the violations we have classified.

Figure 3.14, presented earlier as an example of a defect, shows a skeleton of the defect found in Vuze. This code is responsible for parsing a command line, and `args` is a list of arguments. The method is full of checks for errors and special cases to make sure that a helpful error message can be shown if needed. However, if the arguments list contains just one argument, and this argument is different than "list", then the method will throw an `IndexOutOfBoundsException`. This special case was overlooked by the programmers.

Figure 3.16 shows the defect found in AspectJ. The task of the `visitCALOAD` method is to verify a `CALOAD` bytecode instruction. This instruction is part of the family of array loading instructions (`AALOAD`, `BALOAD`, `CALOAD`, etc.)[7], 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. The `visitCALOAD` 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 it does not check the second half of the first condition—if the array is of an appropriate type. This makes the `visitCALOAD` method positively verify illegal `CALOAD`

[7]Details can be found in the Java VM specification (Lindholm and Yellin 1999).

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

Figure 3.17: One of the defects found by JADET in Columba. The panel uses the default flow layout, but an element is added to it with a border layout constraint.

instructions. This defect was found by JADET, because JADET found out that the `visitCALOAD` method misses the call to `constraintsViolated()`—and it is this call that would have to be issued if the method discovered that the array is not of the correct type.

Figure 3.17 shows one of the defects found in Columba. The panel `rightPanel` is created using a default constructor, which amounts to it using the default layout (so-called "flow layout") for laying out components that will be added to it. However, a component is added to the panel with a layout constraint applicable to border layout only (`BorderLayout.SOUTH`). This code is defective, and the only reason it does not crash is because flow layout ignores all constraints: the way it lays out components cannot be parametrized. JADET found this defect because, as a rule, if a panel is created using the default constructor, and the two-parameter version of the `add()` method is used (i.e., the one that takes a layout constraint as the second parameter), then there should be a call to `setLayout()` somewhere in-between, which is not the case here.

The second defect found in Columba is shown in Figure 3.18. The panel `mainPanel` is created using the constructor that takes the layout as a parameter. However, later the layout is set again (via the call to `setLayout()`), and the layout used in this call is different than the one used in the constructor call. This code works, but it is clearly wrong, in addition to being misleading. JADET found this defect because of the same rule as in the last case, albeit violated in a different way: if a two-parameter version of the `add()` method is used (i.e., the one that takes a layout constraint as the second parameter), and there is a call to `setLayout()`, then the constructor should be the default one (i.e., without a layout specified).

Figure 3.19 shows the defect found in Act-Rbot. This code looks fine at first glance. However, it turns out that the loop will terminate during the first iteration with the `NullPointerException` being thrown. The reason for this is that the weakest link is initialized to be `null`, and in the first iteration of the loop `getActivation()` gets called on it. This defect was found by JADET

```
private void layoutComponents() {
    ...
    JPanel mainPanel = new JPanel(new BorderLayout());
    ...
    FormLayout layout = ...;

    CellConstraints cc = new CellConstraints();
    mainPanel.setLayout(layout);

    mainPanel.add(createGroupNamePanel(), cc.xy(1, 1));
    ...
}
```

Figure 3.18: Another one of the defects found by JADET in Columba. The panel gets its layout set twice, and to a different one at that.

```
public Link getWeakestLink() {
    Link weakestLink = null;
    for (Link link : links) {
        weakestLink = (weakestLink.getActivation() >
            link.getActivation() ? link : weakestLink);
    }
    return weakestLink;
}
```

Figure 3.19: The defect found by JADET in Act-Rbot. The loop will terminate during the first iteration with the `NullPointerException` being thrown.

```
private List<String> buildOptions() {
    Object asc = ...;
    ...

    // Get the ends from the association ...

    Iterator iter = Model.getFacade().getConnections(asc).iterator();

    Object ae0 = iter.next();
    Object ae1 = iter.next();

    Object cls0 = Model.getFacade().getType(ae0);
    Object cls1 = Model.getFacade().getType(ae1);
    ...
    if (...) {
        start = Model.getFacade().getName(cls0);
    }
    if (...) {
        end = Model.getFacade().getName(cls1);
    }
    ...
}
```

Figure 3.20: One of the code smells found by JADET in ArgoUML.

because the iterator created implicitly by the for loop is used to retrieve just the first element and this happens very rarely (typically, iterators are used just for that: iterating through multiple elements).

As a last example, consider the code smell found in ArgoUML (shown in Figure 3.20). Part of what the `buildOptions()` method does is getting the two ends from the association (cf. the comment in the method). As each association is directional, the two ends are not equivalent; there is a clear distinction between a "start" and "end" end, and this distinction is made by the method as well (cf. the variables `start` and `end`). However, the `getConnections()` method returns a collection, and collections in Java *do not guarantee any iteration order*. Normally this means that the code that depends on such order—like the one shown in Figure 3.20—is buggy. Why did we mark it as a code smell only, then? Because the `getConnections()` method is only *declared* as returning a collection. Its implementation actually returns a list, so the code shown works fine. This is an example of a very badly designed interface, because users of the API make use of assumptions that are not exported by the API, but are true by virtue of the implementation.

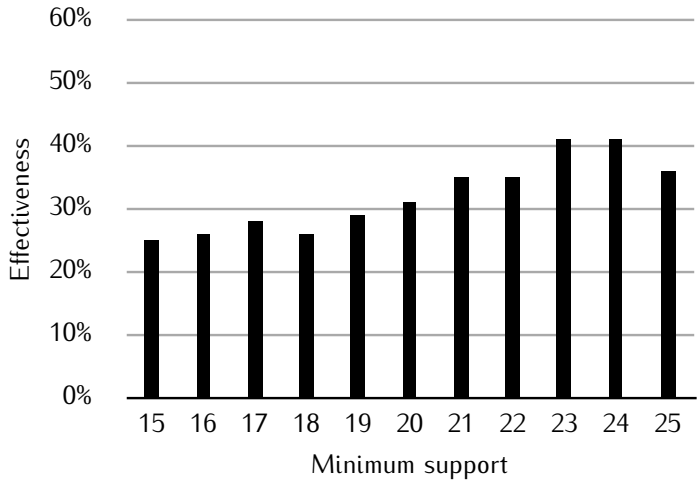



Figure 3.21: Influence of minimum support on the effectiveness of JADET for AspectJ (other parameters fixed at their default values).

**Sensitivity Analysis**

The results presented above prove that our approach is effective in finding defects and code smells in software fully automatically: by learning patterns and looking for where they are violated. However, it is possible that we might get much better or much worse results by small changes to parameters such as minimum support or the number of violations investigated. We have therefore investigated the influence of small changes to minimum support, minimum confidence, and the number of classified violations on the effectiveness of JADET on AspectJ. While manipulating one parameter, we kept all others at their default values (when it comes to the number of classified violations, we used 10% as the default value). Results of these investigations can be found in Figures 3.21, 3.22, and 3.23, respectively.

We can see that effectiveness is slightly sensitive to small changes to minimum support; less so for small changes to minimum confidence. We can also see that our choice of these two parameters was not the best possible (for AspectJ; for other projects things might look differently). This suggests that the JADET's effectiveness can be improved by choosing better parameters' values. When it comes to sensitivity of effectiveness to the number of violations classified, the picture looks roughly like we want it: as the number of classified violations goes up, effectiveness drops. This means that our ranking method is effective in giving true positives higher ranking.

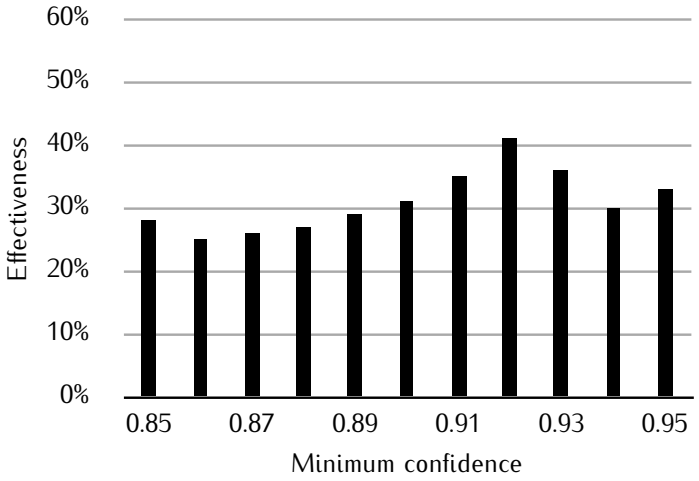



Figure 3.22: Influence of minimum confidence on the effectiveness of JADET for AspectJ (other parameters fixed at their default values).

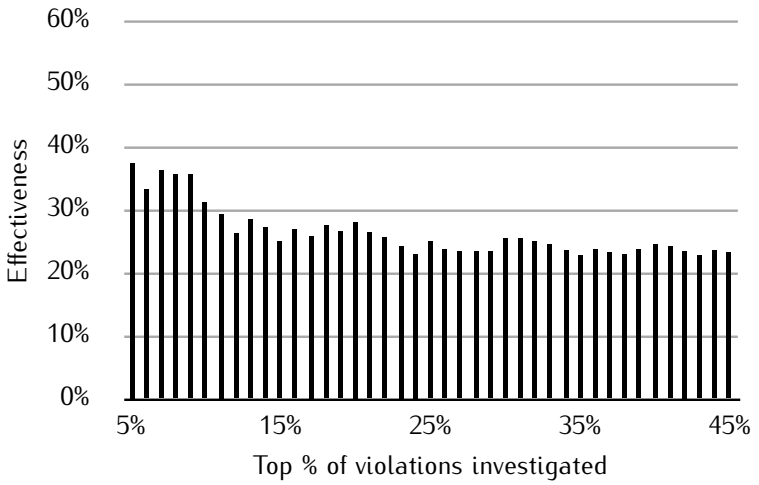



Figure 3.23: Influence of the number of violations classified on the effectiveness of JADET for AspectJ (other parameters fixed at their default values).

```
static int dcc_listen_init(struct DCC *dcc, sess *sess) {
    socklen_t len;
    dcc->sok = socket(AF_INET, S_STREAM, 0);
    if (send_port > 0) {
        i = 0;
        while (ls_port > ntohs(SAdr.sin_port) && br == -1) {
            i++;
            bind(dcc->sok, &SAdr, sizeof(SAdr)));
        }
        setsockopt(dcc->sok, SOL_SOCKET, SO_REUSE_ADDRESS,
            (char *)&len, sizeof(len));
    }
    listen(dcc->sok, 1);
    upnp_add_redir(inet_ntoa(addr), dcc->port);
}
```

Figure 3.24: Sample code from Conspire 0.20. The call to `setsockopt()` is issued too late to have any influence on the `bind()`'s behavior.

## 3.5 Scaling Up to Many Projects

Up to now we have shown how to mine patterns and anomalies from one program at a time. This allows us to find out how objects of certain classes should be used, but only if there are enough usages to learn from. This is not always the case. Consider, for example, a server that accepts remote requests from clients. Typically, the code that handles listening for incoming connections and accepting them will be found in one place in the server. This means that by analyzing the server's code we cannot really learn how to work with connections, because there are not enough examples to allow us to draw meaningful conclusions. As an example, consider the C function shown in Figure 3.24. This function, taken from Conspire 0.20—an open-source IRC client—creates a socket (the call to `socket()`), and binds it to a specific address (the call to `bind()`). Later it calls `setsockopt()` to allow binding to an already-in-use address, but it is, alas, too late. For the code to be correct the call to `setsockopt()` must come before the call to `bind()`. The pattern being violated consists of the following sequential constraints:

$$\begin{aligned}
&\texttt{setsockopt @ (1)} \prec \texttt{bind @ (1)}\\
&\texttt{setsockopt @ (1)} \prec \texttt{listen @ (1)}\\
&\texttt{bind @ (1)} \prec \texttt{listen @ (1)}
\end{aligned}$$

Table 3.5: Projects used as case study subjects for cross-project analysis (adapted from Gruska, Wasylkowski, and Zeller (2010))

| | | Analysis time | | |
|---|---|---|---|---|
| Project | SLOC | Parsing | Total | Violations |
| cacao-0.95 | 91,226 | 0:08 | 3:13 | 0 |
| cksfv-1.3.13 | 784 | 0:01 | 0:04 | 1 |
| concentration-1.2 | 1,715 | 0:01 | 0:01 | 0 |
| daudio-0.3 | 1,476 | 0:01 | 0:05 | 0 |
| dhcpdump-1.8 | 478 | 0:01 | 0:01 | 0 |
| ggv-2.12.0 | 13,149 | 0:02 | 3:22 | 3 |
| gimp-2.6.6 | 595,664 | 1:54 | 17:49 | 61 |
| glade3-3.6.4 | 53,159 | 0:07 | 4:04 | 18 |
| httrack-3.43-4 | 41,017 | 0:03 | 2:32 | 8 |
| LDL-2.0.1 | 904 | 0:01 | 0:01 | 0 |
| memcached-1.3.3 | 5,412 | 0:01 | 0:07 | 0 |
| mpich-1.2.7p1 | 196,609 | 0:13 | 5:20 | 12 |
| otp_src_R13B | 201,553 | 0:13 | 5:18 | 14 |
| psycopg-1.1.15 | 3,160 | 0:01 | 0:03 | 6 |
| python-scw-0.4.7 | 69 | 0:01 | 0:01 | 0 |
| tclxml-2.4 | 12,354 | 0:01 | 0:05 | 8 |
| vdr-arghdirector-0.2.6 | 1,109 | 0:01 | 0:01 | 0 |
| viewres-1.0.1 | 927 | 0:01 | 0:02 | 1 |
| xf86-video-savage-2.2.1 | 10,950 | 0:01 | 0:03 | 0 |
| Yap-5.1.3 | 124,410 | 0:09 | 4:53 | 4 |

This pattern cannot be found by analyzing Conspire, because there are only *two* calls to `setsockopt()` in this project. The solution is to learn usages from multiple *reference projects* before looking for anomalies. However, here we quickly run into scalability issues. While the analysis we have presented scales very well to large projects, scaling it up to hundreds or even thousands of reference projects would have to render it useless. The solution is to use a *lightweight analysis* that will be able to handle large bodies of code quickly. If we compare the time spend on mining object usage models (Table 2.2) with the time spend on looking for patterns and anomalies (Table 3.2) it becomes clear that finding patterns and anomalies is very fast, and it is just the static analysis that needs to be replaced.

This task was solved by Natalie Gruska, who built a lightweight parser of C code (Gruska 2009). The parser focuses just on basic control flow and function calls and outputs function models (akin to object usage models, but

Table 3.6: Classification results for top 25% violations found using cross-project analysis. Only projects with at least one violation found are listed. "CSs" stands for the number of code smells. "FPs" stands for the number of false positives (adapted from Gruska, Wasylkowski, and Zeller (2010))

| Program | Classified | Defects | CSs | FPs | Effectiveness |
|---|---|---|---|---|---|
| cksfv-1.3.13 | 1 | 1 | 0 | 0 | **100%** |
| ggv-2.12.0 | 1 | 0 | 0 | 1 | **0%** |
| gimp-2.6.6 | 22 | 1 | 2 | 19 | **14%** |
| glade3-3.6.4 | 5 | 2 | 0 | 3 | **40%** |
| httrack-3.43-4 | 2 | 0 | 1 | 1 | **50%** |
| mpich-1.2.7p1 | 5 | 0 | 4 | 1 | **80%** |
| otp_src_R13B | 4 | 0 | 0 | 4 | **0%** |
| psycopg-1.1.15 | 6 | 0 | 0 | 6 | **0%** |
| tclxml-2.4 | 2 | 0 | 0 | 2 | **0%** |
| viewres-1.0.1 | 1 | 0 | 0 | 1 | **0%** |
| Yap-5.1.3 | 1 | 0 | 0 | 1 | **0%** |
| **Overall** | 50 | 4 | 7 | 39 | **22%** |

focusing on whole functions instead of objects), which can later be used to create functions' sequential constraints abstractions. This allows us to use the techniques described in this chapter and actually find the defect in Conspire.

In our work (Gruska, Wasylkowski, and Zeller 2010), we have created a reference database consisting of sequential constraints abstractions of functions coming from more than *6000* C projects from Gentoo Linux distribution. This meant parsing over *200 million* lines of code, and took 18 hours, with an average analysis time per project of less than *11 seconds* using a single 2.9 GHz Intel Xeon core. We have then chosen 20 random projects (see Table 3.5) and proceeded to look for anomalies in them. The results of our investigations can be seen in Tables 3.5 and 3.6. The true positive rate is lower than for JADET, but we were able to find defects and code smells that require such cross-project analysis due to scarcity of examples on how to use certain functions.

We have also created a Web site, `checkmycode.org`, where a programmer can upload her own code and have it checked against the projects from the Gentoo Linux distribution, the same we have used for our experiments. (See Figure 3.25 for a screenshot). We will not go into more detail here; an interested reader can consult other sources (Gruska 2009; Gruska, Wasylkowski, and Zeller 2010), as well as simply use the `checkmycode.org` Web site and the tutorial provided there.

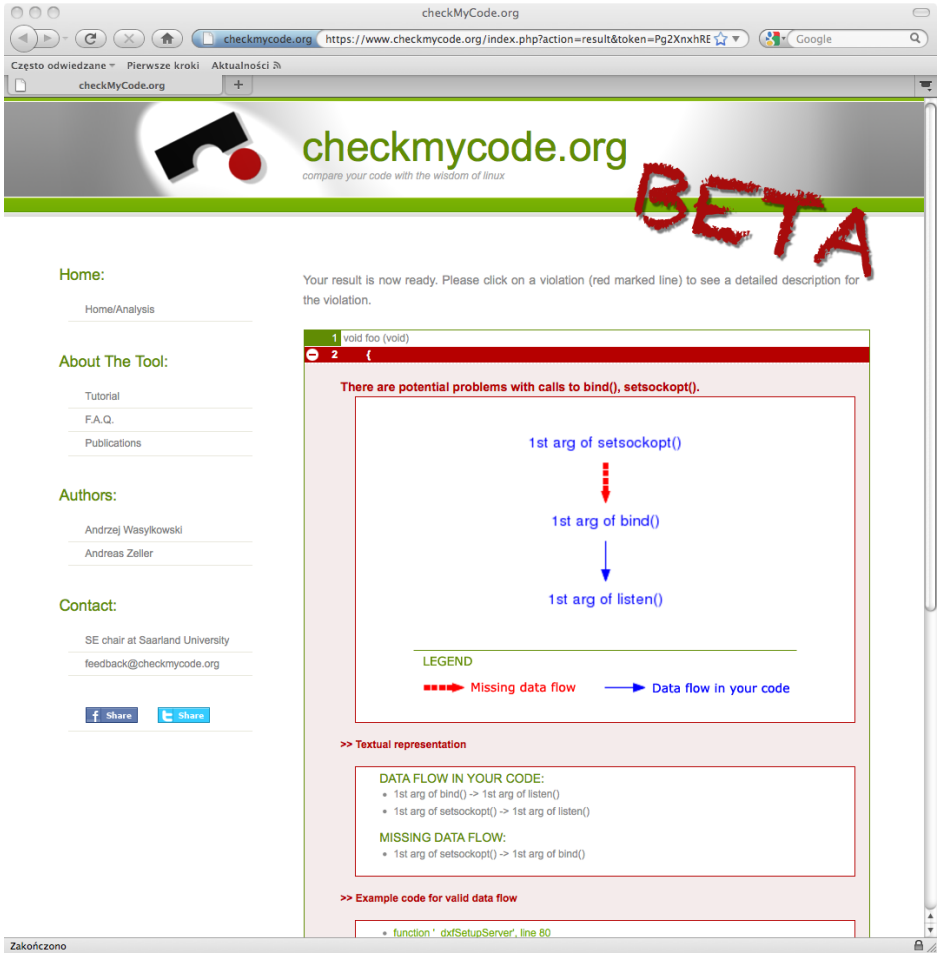



Figure 3.25: Screenshot of the `checkmycode.org` Web site. Here programmers can have their code checked against "the wisdom of Linux code" (adapted from Gruska, Wasylkowski, and Zeller (2010))

## 3.6 Related Work

Our work on mining patterns and anomalies (Wasylkowski 2007; Wasylkowski, Zeller, and Lindig 2007) has been preceded by works of other researchers. There is also a large body of contemporary work on this and similar topics. Some of them have been described by us already in Section 2.5; we will now

present other works that have more to do with finding anomalies.

The idea that what is common is typically correct, and what is uncommon is likely to be defective comes from Engler et al. (2001). Our own work on mining pattern patterns and anomalies was motivated by the PR-Miner tool Li and Zhou (2005). PR-Miner builds on Engler et al.'s idea, too: it looks for entities (function calls, variables, types) that frequently occur together. Such sets of frequently occurring entities form patterns, and their violations are reported to the user. We build directly on this idea, but with a few improvements: First, we use sequential constraints, and this allows us to represent ordering between method calls (Kagdi, Collard, and Maletic (2007b) performed an evaluation showing that providing ordering information is worth the additional cost). Second, we model data flow and make use of it to find which method calls are related (and how) and which are not. Third, we use concept analysis for finding patterns and their violations, which is a cleaner (though equivalent) solution than using two passes of frequent itemset mining, as Li and Zhou did.

GrouMiner by Nguyen et al. (2009) mines groums (which are directed acyclic graphs; see Section 2.5.2) from source code and finds usage patterns—being frequently occurring subgraphs—and their violations. GrouMiner has access to much more information that JADET, but it's true positive rate is worse than JADET's (between 7% and 20% for top 15 violations investigated in eight out of nine case study subjects for GrouMiner[8] compared to between 30% and 60% for top 10 violations in JADET's case). This is an unexpected result and it would be interesting to see why this is the case.

Ramanathan, Grama, and Jagannathan presented Chronicler (Ramanathan, Grama, and Jagannathan 2007a)—a tool for learning patterns consisting of precedence relationships between function calls. Chronicler's static analysis is path sensitive, increasing its accuracy, but the precedence relationships do not take data flow into account, so that unrelated functions can become part of one pattern just because they occur often together. These authors presented also another work (Ramanathan, Grama, and Jagannathan 2007b), where in addition to precedence relationships dataflow properties are used (like "`x` must be non-null"). This allows for more expressivity, but the precedence relationships still do not take data flow into account. A bit in the same direction, Kagdi, Collard, and Maletic presented an approach where sequencing data are combined with syntactic context to increase patterns' expressiveness (Kagdi, Collard, and Maletic 2007a).

Gabel and Su (2010) presented OCD, a tool using dynamic analysis to infer patterns of method calls and violations of those patterns. The patterns are

[8]The authors investigated all 64 violations reported for the ninth case study subject, so we do not know the true positive rate for the top 15 violations

based on templates provided by the user and instantiated by mapping placeholders in the templates to method calls from the execution trace. This tool is similar in spirit to JADET with the biggest difference being using dynamic, instead of static, analysis.

Weimer and Necula (2005) mine pairs of matching events (like calls to `open()` and `close()`) from traces, but they require the second of the events to occur at least once in exception-handling code. Their idea is that "if the policy is important to the programmer, language-level error handling will be used at least once to enforce it." These specifications are then fed to a program verification tool for the purposes of defect detection. Thummalapenta and Xie (2009b) presented a tool called CAR-Miner, which is similar in spirit in that it focuses on exception-handling code.

Chang, Podgurski, and Yang (2007) presented an approach for discovering neglected conditions. They build system dependence graphs and use frequent subgraph mining to find frequently occurring graph minors that become patterns, focusing on patterns that contain a control point (branch predicate of a conditional statement). Violations of patterns are reported to the user for investigation.

Thummalapenta and Xie (2009a) describe Alattin: an approach for finding neglected conditions by comparing the code with sample code obtained using a code search engine. The main advantage of Alattin, and one that could theoretically be integrated into our approach, is that it recognizes so-called "infrequent patterns". One major drawback (which is true for all approaches that use code-search engines) is that the user cannot control the quality of code that is used for learning, because she does not have influence on the code that is being returned by code-search engines.

Yang et al. (2006) presented Perracotta: a tool that mines temporal rules of program behavior fitting into templates (such as alternation) provided by the user. Perracotta is based on dynamic analysis and the templates just contain function calls—data flow is not taken into account. This means that unrelated functions can become part of one pattern just because they occur often together. The rules found can later be fed to a verifier to check if a program conforms to them.

Acharya, Xie, and Xu (2006) presented an approach for mining specifications (in the form of FSAs) for functions. Their idea is that the user provides general "robustness property" (e.g., "check before use") as a FSA, and this is then instantiated by a model checker analyzing a given program that uses a given function. The resulting specification shows how to use a single function in a way that avoids potential problems (e.g., the specification may state that after calling `malloc()` one should first *check* if the return value is not `NULL` and only then *use* it). Later, a program can be checked for confor-

mance with these specifications by using a model checker, and "robustness violations" found will be reported to the user.

Liu, Ye, and Richardson (2006) developed LtRules, an Eclipse plug-in that can extract usage rules from C programs. LtRules requires the user to provide a set of functions he is interested in, and applies them to a set of six templates, thus creating rule candidates, and uses a model checker to find rule candidates that are followed by the given set of C programs. Rules confirmed in this way are reported to the user.

Le Goues and Weimer (2009) proposed a technique for making specification mining approaches more accurate. Their idea is to assign trustworthiness to code fragments (depending on things like author rank, code churn, or code readability), and thus influence the set of specifications that are generated. They show that this technique helps in finding more real specifications and filtering false ones.

Olender and Osterweil (1992) introduced Cesar, a system that allows the user to provide sequencing constraints (as regular expressions) for an abstract type, and checks the program for conformance with those constraints.

Hovemeyer and Pugh (2004) presented FindBugs, a tool for finding bugs in Java programs. The idea behind FindBugs is that there is a database of "bug patterns" (e.g., "Null Pointer Dereference"), and if the program contains an instance of such a pattern (e.g., a `null` value might be dereferenced at some point), the corresponding place in the program code gets reported to the user as potentially buggy. One very important difference between FindBugs and JADET is that FindBugs is limited to a priori specified bug patterns, whereas JADET infers specification from the program to be checked, and can thus detect bugs that are program-, project-, or library-specific.

Livshits and Zimmermann (2005) presented DynaMine: a tool that uses software repositories to find highly-correlated method calls. These form patterns that are later checked using dynamic analysis.

## 3.7 Summary

This chapter makes the following contributions:

- We have introduced the notion of a *sequential constraints abstraction*. This allows us to abstract a method from a program into a set of sequential constraints that show how objects are being used in that method.

- We have shown how we can apply formal concept analysis to efficiently find patterns of object usage (frequently occurring sets of sequential

constraints). Our analysis scales very well: It takes less than *half a minute* for a large project like Vuze (345 K SLOC, 35,363 methods in 5532 classes).

- We have shown how we can quickly find violations of patterns. Our analysis takes less than *half a minute* for a large project like Vuze. Violations of patterns consisting of sequential constraints have the benefit that they show *what* is missing, thus helping the programmer fix the code, if it turns out to be defective.
- We have implemented all the techniques above in a tool called JADET and evaluated it on six open-source projects. JADET found defects or code smells in all of those projects. In total, investigating top 10 violations for all the projects resulted in finding 5 defects and 23 code smells—for a true positive rate of 47%.
- We have shown that—by using lightweight static analysis—we are able to reuse all the techniques shown above to scale our approach to handle thousands of projects and thus be able to perform cross-project analysis. Early results indicate that cross-project analysis can find subtle defects that are not detectable using single-project analysis.

To learn more about our work on finding patterns and anomalies, and on related topics, see:

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

# Chapter 4

# Operational Preconditions

## 4.1 Introduction

In the preceding chapter we have shown how we can mine patterns and anomalies in object usage. If we take a closer look at some of the patterns found, we will discover that they are akin to specifications, but have a much more limited expressivity than real, formal specifications. In Section 3.2.2 we have given a few examples on how sequential constraints abstraction could be made more expressive, but there is only so much that can be improved. However, the sequential constraints abstraction was simple enough to allow us to gently introduce the concept of patterns and anomalies, and show how can these be found fully automatically. In this chapter, we will build on this knowledge and show how we can use a much better and much more expressive abstraction while using the same mechanism for finding patterns and anomalies as before.

## 4.2 Operational Preconditions

### 4.2.1 The Concept of Operational Preconditions

When calling a method, the caller must ensure that the function's *precondition* is satisfied—the condition that has to be met before its execution. An example of a precondition might be "The parameter passed to the `sqrt()` method must be non-negative". This simple precondition is not only easy to understand and easy to check, but it also has a very important property: It is easy to implement. By this we mean that it is easy for the programmer to write code that satisfies the precondition. This is not always true.

```
/**
 * Internal helper method that completes the building of
 * a node type's structural property descriptor list.
 *
 * @param propertyList list beginning with the AST node class
 * followed by accumulated structural property descriptors
 * @return unmodifiable list of structural property descriptors
 * (element type: <code>StructuralPropertyDescriptor</code>)
 */
static List reapPropertyList(List propertyList) {
    propertyList.remove(0); // remove nodeClass
    // compact
    ArrayList a = new ArrayList(propertyList.size());
    a.addAll(propertyList);
    return Collections.unmodifiableList(a);
}
```

Figure 4.1: The `reapPropertyList()` method from AspectJ.

Consider the `reapPropertyList()` method shown in Figure 4.1. This method comes from AspectJ; we show its body and the accompanying comment. What is the precondition of this method, or—more precisely—what is the precondition for the parameter `propertyList` of this method? If we only look at its source code, we can state it like this:

```
@requires propertyList.size() >= 1
```

This is a JML[1] specification of `reapPropertyList()` stating that `propertyList` must have at least one element. From a strictly formal point of view this is a complete precondition: `reapPropertyList()` will work correctly if given a list with at least one element. However, this is not how the list *should* look like, as the JavaDoc comment accompanying the method proves. Let us take the comment into account and write a new precondition:

```
@requires propertyList.size() >= 1
@requires propertyList.get(0) instanceof Class
@requires \forall int i; 0 < i && i < propertyList.size();
    propertyList.get(i) instanceof StructuralPropertyDescriptor
```

It might seem like we are done, but unfortunately this is not the case. The comment misses one important point: there is a relationship between the

[1] http://www.eecs.ucf.edu/~leavens/JML/index.shtml

first element of the list and all the others that must hold for the list to be properly constructed. The `reapPropertyList()` method does not check for this relationship, and the comment does not mention it, because the designers assume that the list will be constructed using helper methods provided by AspectJ, and not directly. Let us rewrite our precondition to capture this relationship:

```
@requires propertyList.size() >= 1
@requires propertyList.get(0) instanceof Class
@requires \forall int i; 0 < i && i < propertyList.size();
    propertyList.get(i) instanceof StructuralPropertyDescriptor
    && ((StructuralPropertyDescriptor)propertyList.get(i)).
        getNodeClass() == propertyList.get(0)
```

This precondition is not difficult to understand, but it lacks the property that the `sqrt()` precondition given above has: it is not easy to implement. Creating a list and inserting a `Class` object into it is nothing too complicated, but what about creating structural property descriptors having an appropriate node class? We probably need to take a look at the implementation of the `StructuralPropertyDescriptor` class and find an appropriate constructor or an appropriate setter method to do the job.

So, while the precondition above answers the "What is the state that `propertyList` must be in?" question, it does not answer the "How to achieve this?" question. For the programmer, the answer to the latter is more important than the answer to the former. We will call traditional "what" preconditions *axiomatic preconditions* and the "how" preconditions *operational preconditions*. In the case of the `reapPropertyList()` method, its operational precondition can be informally stated as follows:

> Create an empty list. Call the `createPropertyList()` method, passing the list to this method as the second parameter. Call the `addProperty` method as many times as needed, passing the list to this method as the second parameter. Call `reapPropertyList()` passing the list as the first parameter.

This operational precondition helps the programmer see what needs to be done in order to pass the correct parameter to `reapPropertyList()`, and it also provides him with information on related methods.

### 4.2.2 Operational Preconditions as Temporal Logic Formulas

Traditional, axiomatic preconditions are usually expressed using some formalism, such as logic formulas. This is because natural language has two

important disadvantages: First, it can be ambiguous; second, it does not lend itself to automatic verification, whereby a program can be checked against preconditions, and places, where these are violated, can be reported to the user. Operational precondition expressed using natural language suffer from the same problems. Therefore, we need to propose a formalism to use for expressing them. Axiomatic preconditions use propositional logic, which lends itself well to describing *state*. However, it is unsuitable for operational preconditions, where it is *process*, not state, that needs to be described. For this purpose, temporal logics are best, because they allow for qualification in terms of time (as in "call `reapPropertyList()` *after* calling `createPropertyList()`"). In order to also incorporate possible branching (as in "call `addProperty` as many times as needed"), we will use a branching-time logic—a variant of Computation Tree Logic (CTL; Clarke and Emerson (1982)) called *Fair Computation Tree Logic* ($CTL^F$; Clarke, Emerson, and Sistla (1986))[2].

Figure 4.2 contains a brief overview of $CTL^F$ and model checking; as can be seen, $CTL^F$ formulas can only be evaluated as true or false *for a given model of a system we are interested in* (the model being a Kripke structure). On the other hand, a Kripke structure must be *over a set of atomic propositions*. These are used to label states and are used as propositions in the $CTL^F$ formulas. Since we want to use formulas such as "always call `createPropertyList()` passing the list as the second parameter", our atomic propositions will be events (see definition 2.1). Using $CTL^F$ with events as atomic propositions, we can represent the operational precondition for the only parameter of `reapPropertyList()` as follows:

$$
\begin{aligned}
&\mathrm{AG}\,\texttt{ArrayList.<init> @ (0)} \;\wedge\\
&\mathrm{AG}\,(\texttt{ArrayList.<init> @ (0)} \Rightarrow\\
&\quad \mathrm{AF}\,\texttt{ASTNode.createPropertyList @ (2)}) \;\wedge\\
&\mathrm{AG}\,(\texttt{ASTNode.createPropertyList @ (2)} \Rightarrow\\
&\quad \mathrm{AF}\,\texttt{ASTNode.reapPropertyList @ (1)}) \;\wedge\\
&\mathrm{AG}\,(\texttt{ASTNode.addProperty @ (2)} \Rightarrow\\
&\quad \mathrm{AF}\,\texttt{ASTNode.reapPropertyList @ (1)})
\end{aligned}
$$

The operational precondition is here a conjunction of subformulas. The first subformula states that the list needs to be created. The second subformula states that the list needs to be passed to `createPropertyList()` as the second parameter after being created. The third subformula states that the list needs to be passed as the first parameter to `reapPropertyList()` after it is passed as the second parameter to `createPropertyList()`. The last subformula states that the list needs to be passed as the first parameter to `reapPropertyList()` after it is passed as the second parameter to `addProperty()` (this makes sure

[2] See Section 4.3.2 for a discussion on why we use $CTL^F$ and not CTL

Temporal logic model checking (Clarke, Emerson, and Sistla 1986) is, in general, a technique for verifying that a given system satisfies the specification given as a temporal logic formula. We use Kripke structures as the model of the system, and $\text{CTL}^{\text{F}}$ (*Fair Computation Tree Logic*; Clarke, Emerson, and Sistla (1986)) as the language for representing specifications.

Let $AP$ be a set of *atomic propositions*. A Kripke structure over $AP$ is a tuple $M = (S, I, R, L)$, where $S$ is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a left-total[a] transition relation, and $L : S \to 2^{AP}$ is a labeling function. Atomic propositions are used to describe the state the system is in, and the Kripke structure represents transitions between the states of the system. Because $R$ is a left-total relation, all the behaviors of the system are infinite.

$\text{CTL}^{\text{F}}$ is a temporal logic used to predicate over the behaviors represented by the Kripke structure. It is defined over the same set $AP$ of atomic propositions that the Kripke structure uses:

1. *true* and *false* are $\text{CTL}^{\text{F}}$ formulas
2. Every atomic proposition $p \in AP$ is a $\text{CTL}^{\text{F}}$ formula
3. If $f_1$ and $f_2$ are $\text{CTL}^{\text{F}}$ formulas, then so are $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $f_1 \Rightarrow f_2$, and $f_1 \Leftrightarrow f_2$.
4. If $f_1$ and $f_2$ are $\text{CTL}^{\text{F}}$ formulas, then so are $\text{AX}\, f_1$, $\text{EX}\, f_1$, $\text{AF}\, f_1$, $\text{EF}\, f_1$, $\text{AG}\, f_1$, $\text{EG}\, f_1$, $\text{A}[f_1 \text{U} f_2]$, $\text{E}[f_1 \text{U} f_2]$.

A means "for all fair paths", and E means "there exists a fair path". X stands for "next", F stands for "finally", G stands for "globally", and U stands for "until". A path is fair if a certain (arbitrary, but fixed) predicate holds infinitely often along it.

The intuitive meaning of some $\text{CTL}^{\text{F}}$ formulas is as follows: $\text{AX}\, f_1$ means that "for each state $s_0 \in I$, for all (A) fair paths starting in $s_0$, $f_1$ holds in the next (X) state". $\text{EF}\, f_1$ means that "for each state $s_0 \in I$, there exists (E) a fair path, where $f_1$ holds somewhere along (F) this path". $\text{AG}\, f_1$ means that "for each state $s_0 \in I$, for all (A) fair paths starting in $s_0$, $f_1$ holds in all states along (G) the path". $\text{A}[f_1 \text{U} f_2]$ means that "for each state $s_0 \in I$, for all (A) fair paths starting in $s_0$, $f_1$ holds until (U) $f_2$ holds" (i.e., $f_2$ must hold somewhere along the path, and until then, $f_1$ must always hold). An atomic proposition $p$ holds in a given state iff this state is labeled with $p$.

Model checking a given $\text{CTL}^{\text{F}}$ formula $f$ against a given Kripke structure $M = (S, I, R, L)$ is equivalent to asking if $f$ holds for each $s_0 \in I$. If it does, $f$ is said to be true for $M$; if it does not, $f$ is said to be false for $M$.

[a] $\forall s_1 \in S . \exists s_2 \in S . (s_1, s_2) \in R$

Figure 4.2: $\text{CTL}^{\text{F}}$ and model checking in a nutshell.

that each call to `addProperty()` [optional, as there is no formula that would enforce its presence] comes before the call to `reapPropertyList()`).

An attentive reader will notice that the operational precondition shown above just states what is a must-have; it does not state what is a must-not-have (such as "do not call `createPropertyList()` more than once"). This is actually consistent with our definition of an operational precondition as answering the "How to achieve this?" question—only giving the information about what must be done. It is possible to extend the definition to make operational preconditions include must-not-have's as well, but we will leave this task as a possible extension of our work. The definition we have given, however, allows us to fully automatically mine operational preconditions and discover their violations in a program by following a procedure similar to the one described in the previous chapter—making operational preconditions a much more expressive cousin of sequential constraints abstraction. The remainder of this chapter will show how this can be done and what can be thereby achieved.

## 4.3 Mining Operational Preconditions

Operational preconditions of a parameter $p$ of method $m$ can be discovered manually by investigating sample code snippets containing a call to $m$ and looking at how $p$ is prepared. For example, to find the operational precondition for the `propertyList` parameter of the `reapPropertyList()` method we can investigate callers of `reapPropertyList()`. It turns out that most of them have the following structure:

```
List propertyList = new ArrayList();
createPropertyList(..., propertyList);
addProperty(..., propertyList);  // possibly multiple times
... = reapPropertyList(propertyList);
```

From this structure we can easily deduce the operational precondition presented in the preceding section. This structure can be automatically discovered and formalized by mining object usage models. Let us consider the sample method shown in Figure 4.3. The object usage model of `list` looks as shown in Figure 4.4. If we now investigate object usage models of all objets of type `List` that get passed as the `propertyList` parameter to `reapPropertyList()` and find out that they all look similar, we will be able to deduce that they represent the operational precondition of that parameter of that method. The principle at work here is the same that we have used in the preceding chapter, when we have described the sequential constraints abstraction. Thus, to be able to reuse the technique from that chapter, we need

```
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 () == 1)
        Debug.log ("Empty property list");
    return list;
}
```

Figure 4.3: Sample source code containing a call to `reapPropertyList()`.

to find an abstraction of an object usage model that satisfies the following conditions:

1. An object usage model is abstracted into a set of "properties" characteristic to it. This will make automatic pattern and anomaly detection via formal concept analysis possible (see Section 3.3.1).

2. An operational precondition can be represented as a set of "properties", and vice versa. This will make the patterns found by formal concept analysis represent operational preconditions.

Recall that an operational precondition is formally represented as a $CTL^F$ formula. To fulfill the second condition above we need to be able to represent a $CTL^F$ formula as a set of "properties". We have decided to limit our operational preconditions to be *conjunctions* of formulas. Thus, we choose our properties to be $CTL^F$ formulas. This fulfills the second condition: an operational precondition can be represented as a set of $CTL^F$ formulas (being interpreted as their conjunction), and a set of $CTL^F$ formulas represents an operational precondition (their conjunction is still a $CTL^F$ formula, so it is a potential operational precondition).

To fulfill the first condition above, we now need to be able to abstract an object usage model into a set of $CTL^F$ formulas. Generally, our idea is as follows: First, transform an object usage model into a Kripke structure (see Figure 4.2). Second, abstract a Kripke structure into a set of $CTL^F$ formulas. Let us now look into those two steps in detail.

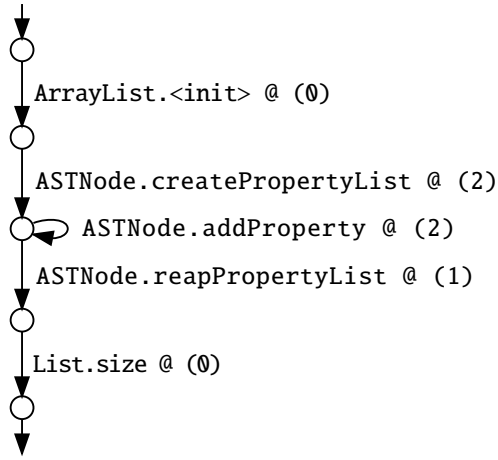



Figure 4.4: OUM for the list created by the method from Figure 4.3.

### 4.3.1 Creating Kripke Structures

Object usage models provide a representation of the way a single object is being passed to various methods, and we want to discover operational preconditions for each parameter of each method separately. To be able to do this, we must transform each object usage model into a set of Kripke structures, each focusing on a specific combination of a method and its parameter—a specific target event $e$ (see definition 2.1) in the object usage model. There are two problems to be solved here:

1. We need to transform transitions from the object usage model into states in the Kripke structure, and have transitions in the Kripke structure reflect the way transitions in the object usage model can follow one another.

2. Since we want to discover *preconditions*, we are only interested in what happens before $e$. Thus, we must prune the object usage model by removing some states and transitions.

Jonsson, Khan, and Parrow faced a problem almost identical to the first problem above. Let us assume (for clarity) that we call transitions in object usage models "edges" and states in object usage models "nodes", and use the names "transition" and "state" only in relation to Kripke structures. Basically, Jonsson, Khan, and Parrow's idea is that each edge $s_1 \xrightarrow{e} s_2$ is transformed

into a separate state in the Kripke structure, and the transition relation is built in such a way that there is a transition between states corresponding to $s_1 \xrightarrow{e_1} s_2$ and $s_3 \xrightarrow{e_2} s_4$ iff $s2 = s3$ (Jonsson, Khan, and Parrow 1990). Labels are given as follows: State corresponding to the edge $s_1 \xrightarrow{e} s_2$ gets labeled with $e$. Also, additional initial and "sink" states (both unlabeled) are introduced.[3]

The second problem can be solved with the help of the "sink" state that—in the procedure of Jonsson, Khan, and Parrow—denotes the "final" state and transitions to it are created only from states corresponding to $s_1 \xrightarrow{e_1} s_2$ where $s_2$ is terminal (i.e., it has no outgoing edges). In our case, where we are only interested in what happens *before* the target event $e$, we can modify Jonsson, Khan, and Parrow's procedure in the following way: Transitions to the "sink" state are created only from states corresponding to $s_1 \xrightarrow{e_1} s_2$ where $e_1$ is an edge labeled with the target event $e$. Now the only thing that needs to be done is to remove from the Kripke structure all states, from which it is not possible to reach the "sink" state.[4] Of course transitions incidental to these states need to be removed, too. This makes the Kripke structure describe the way the object is used under the assumption that the event $e$ always finally happens (we will take a closer look at this statement in the next section). This in turn allows us in the end to discover operational preconditions of $e$.

Let us now give a precise definition of how to create a Kripke structure given an object usage model and an event of interest (i.e., the combination of a method and its parameter, for which we want to mine operational preconditions).

**Definition 4.1** (Kripke structure induced by an OUM and an event)**.** Let $oum = (Q, \Sigma, T, q_0, F, Exc)$ be an object usage model (see definition 2.2) and $e$ be an event (see definition 2.1). Let $S$ be a finite set of states defined as follows:

$$S = \{q \xrightarrow{\sigma} q' : q' \in T(q, \sigma)\} \cup \{s_i, s_f\}$$

Let $I$ be a set of initial states defined as $I = \{s_i\}$. Let $R \subseteq S \times S$ be a transition relation defined as follows:

$$\begin{aligned} R &= \{(q \xrightarrow{\sigma} q', q' \xrightarrow{\sigma'} q'') : q \xrightarrow{\sigma} q' \in S \text{ and } q' \xrightarrow{\sigma'} q'' \in S\} \\ &\cup \{(s_i, q_0 \xrightarrow{\sigma} q) : q_0 \xrightarrow{\sigma} q \in S\} \\ &\cup \{(q \xrightarrow{e} q', s_f) : q \xrightarrow{e} q' \in S\} \\ &\cup \{(s_f, s_f)\} \end{aligned}$$

[3] The "sink" state is needed to make sure that the transition relation is left-total.

[4] It can be easily shown that if the "sink" state is not reachable, then the state corresponding to the target event is also not reachable, and vice versa.

Let $AP$ be the set of all events occurring in *oum*:

$$AP = \{\sigma : \exists q, q' . q' \in T(q, \sigma)\}$$

Let $L : S \to 2^{AP}$ be a labeling function defined as follows:

$$L(s) = \begin{cases} \{\sigma\} & \text{iff } s = q \xrightarrow{\sigma} q' \\ \emptyset & \text{iff } s = s_i \text{ or } s = s_f \end{cases}$$

Let $R_t \subseteq S \times S$ be a transitive closure of $\{(s, s') : s = s' \text{ or } (s, s') \in R\}$. The Kripke structure induced by the OUM *oum* and the event $e$ is a tuple $\text{ks}(oum, e) = (S', I', R', L')$ where $S' \subseteq S, I' \subseteq I, R' \subseteq R,$ and $L' : S' \to 2^{AP}$ are defined as follows:

$$\begin{aligned} S' &= \{s \in S : (s, s_f) \in R_t\} \\ I' &= I \cap S' \\ R' &= \{(s, s') \in R : (s, s_f) \in R_t \text{ and } (s', s_f) \in R_t\} \\ L' &= L|_{S'} \end{aligned}$$

If $S' = \{s_f\}$, we call the Kripke structure *empty*.[5]

By following this definition, the Kripke structure induced by the object usage model from Figure 4.4 and the event `ASTNode.reapPropertyList @ (1)` looks as shown Figure 4.5. The intuitive meaning of the state labeling is "what is the event that has happened most recently?" This allows us to learn $\text{CTL}^{\text{F}}$ formulas that describe the temporal relation between events the object must go through before participating in the target event.

If we want to find an operational precondition of a specific event $e$, we first generate all Kripke structures induced by all object usage models and the event $e$, while discarding all empty Kripke structures (see definition 4.1). Formally, the set of Kripke structures *pertaining to* the event $e$ is defined as follows:

**Definition 4.2** (Set of Kripke structures pertaining to an event)**.** Let $e$ be an event, and $m_1, \ldots, m_n$ be methods. Let $\text{Obj}(m_i)$ be the set of abstract objects used by $m_i$ (see definition 2.4), and let $oum(x)$ be defined as an object usage model of an abstract object $x$. The set of Kripke structures pertaining to the event $e$ is defined as $pertaining\_kss(e, \{m_1, \ldots m_n\}) = \{ks : \exists i, x . 1 \leq i \leq n \text{ and } x \in \text{Obj}(m_i) \text{ and } ks = \text{ks}(oum(x), e) \text{ and } ks \text{ is not empty}\}$

[5]This may happen if the event $e$ does not occur in the object usage model *oum* (i.e., no transition in *oum* is labeled with $e$).

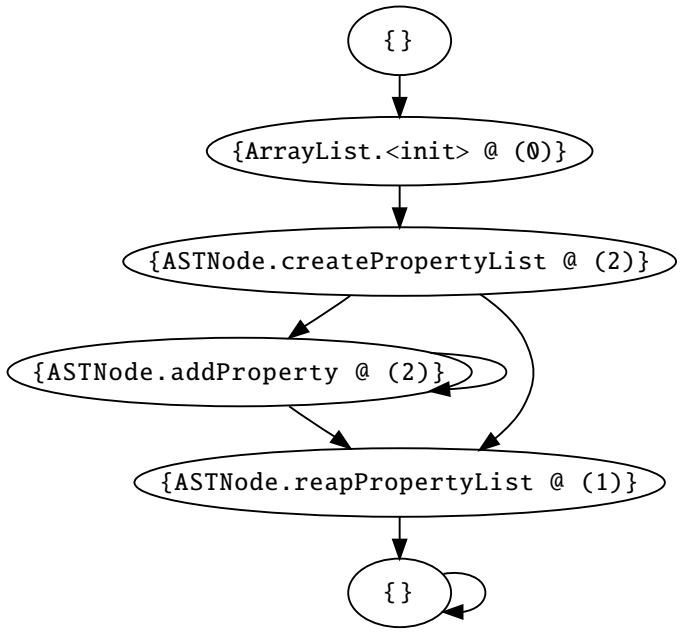



Figure 4.5: Kripke structure induced by the object usage model from Figure 4.4 and the event `ASTNode.reapPropertyList @ (1)`.

### 4.3.2 From Kripke structures to CTL$^F$ formulas

Having created a Kripke structure, we can now try to abstract it into a set of CTL$^F$ formulas. Our idea is to generate many plausible CTL$^F$ formulas and model check them to discover those that are true. (See Figure 4.2 for a brief overview of CTL$^F$ and model checking).

#### Generating CTL$^F$ formulas

If we want to generate CTL$^F$ formulas, then the most accurate approach would be to generate all possible CTL$^F$ formulas up to a certain depth.[6] Unfortunately, this is infeasible due to a combinatorial explosion in the number of formulas. Even assuming that there are only four atomic propositions (as is the case in Figure 4.5), and limiting ourselves to only a minimal, complete set of CTL$^F$ operators (say, the *existential normal form* with only *true*, $\wedge$, $\neg$, EX, EU, and EG operators), the number of CTL$^F$ formulas increases very

[6]The depth of a CTL$^F$ formula is the depth of its abstract syntax tree.

rapidly: There are 5 formulas of depth 0, 65 formulas of depth 1, almost 10,000 formulas of depth 2, and over *200,000,000* formulas of depth 3. While many of those are equivalent, the number of formulas still remains too large.

Therefore, we have decided to limit ourselves to only a set of $\mathrm{CTL}^{\mathrm{F}}$ formulas that can be generated from *predefined templates.* These templates can be easily adapted to the needs of the user. We did not investigate in detail the problem of choosing best templates; this is left as an extension of this work. Instead, we settled on a simple set of templates that would allow for expressing two things: the fact that an event may/must occur, and the fact that an event may/must occur after another event. Our set of templates around atomic propositions ($p_i$) is as follows:

$\mathrm{AF}\, p_1$ : The object *must* eventually participate in the event denoted by $p_1$. For example, $\mathrm{AF}$ `createPropertyList @ (2)` means that the object *must* eventually be passed as the second parameter to `createPropertyList()`.

$\mathrm{EF}\, p_1$ : It *must be possible* for the object to participate in the event denoted by $p_1$. For example, $\mathrm{EF}$ `addProperty @ (2)` means that *it is possible* that the object is passed as the second parameter to `addProperty()`.

$\mathrm{AG}\,(p_1 \Rightarrow \mathrm{AX}\,\mathrm{AF}\, p_2)$ : Whenever the object participates in the event denoted by $p_1$, it *must* at some later point participate in the event denoted by $p_2$. For example, $\mathrm{AG}\,($`lock @ (1)` $\Rightarrow \mathrm{AX}\,\mathrm{AF}$ `unlock @ (1)`$)$ means that after being passed as the first parameter to `lock()`, an object *must* eventually be passed as the first parameter to `unlock()`.

$\mathrm{AG}\,(p_1 \Rightarrow \mathrm{EX}\,\mathrm{EF}\, p_2)$ : Whenever the object participates in the event denoted by $p_1$, it *must be possible* for the object to at some later point participate in the event denoted by $p_2$. For example, $\mathrm{AG}\,($`ArrayList.<init> @ (0)` $\Rightarrow \mathrm{EX}\,\mathrm{EF}$ `addProperty @ (2)`$)$ means that after creating an array list there *exists a path* through the program such that the list is *eventually* passed as the second parameter to `addProperty()`.

We use $\mathrm{AX}\,\mathrm{AF}$ and $\mathrm{EX}\,\mathrm{EF}$ instead of $\mathrm{AF}$ and $\mathrm{EF}$, respectively, because we want to avoid tautologies such as $\mathrm{AG}\,(p_1 \Rightarrow \mathrm{AF}\, p_1)$ and $\mathrm{AG}\,(p_1 \Rightarrow \mathrm{EF}\, p_1)$. The fact that these are tautologies means that just using $\mathrm{AF}$ and $\mathrm{EF}$ does not really allow for expressing the fact that a certain event can happen more than once (which is possible when $\mathrm{AX}\,\mathrm{AF}$ and $\mathrm{EX}\,\mathrm{EF}$ are used instead).

Let us now give a formal definition of the set of $\mathrm{CTL}^{\mathrm{F}}$ formulas that are generated from a given Kripke structure:

**Definition 4.3** ($\mathrm{CTL}^{\mathrm{F}}$ formulas induced by a Kripke structure)**.** Let $ks = (S, I, R, L)$ be a Kripke structure. Let $\Sigma$ be the set of events from $ks$ defined as follows:

$$\Sigma = \{\sigma : \exists\, s \in S \,.\, \sigma \in L(s)\}$$

The set of CTL$^F$ formulas induced by the Kripke structure *ks* is defined as follows:

$$\begin{aligned} \mathit{ctlf\_fs}(\mathit{ks}) &= \{\mathrm{AF}\,\sigma : \sigma \in \Sigma\} \\ &\cup \{\mathrm{EF}\,\sigma : \sigma \in \Sigma\} \\ &\cup \{\mathrm{AG}\,(\sigma_1 \Rightarrow \mathrm{AX}\,\mathrm{AF}\,\sigma_2) : \sigma_1, \sigma_2 \in \Sigma\} \\ &\cup \{\mathrm{AG}\,(\sigma_1 \Rightarrow \mathrm{EX}\,\mathrm{EF}\,\sigma_2) : \sigma_1, \sigma_2 \in \Sigma\} \end{aligned}$$

Before we move further, let us make a slight digression here. The templates we have proposed effectively reduce the set of generated CTL$^F$ formulas. Very long methods, though, can still result in large object usage models and hence tens of thousands of CTL$^F$ formulas. Therefore, in practice, we limit ourselves to using only those atomic propositions, that occur more often than a specified minimum number of times (see the description of *minimum support* in Section 4.3.3). This drastically reduces the number of formulas while not influencing our final results. We have not included this optimization in the definition above because it works only if the templates do not contain any negations. This is the case for the set proposed by us, but—in general—is not required.

### Model checking CTL$^F$ formulas

Having created a Kripke structure and generated a set of CTL$^F$ formulas, we now must discover which CTL$^F$ formulas are true, and which ones are false. We do this using *model checking*. Model checking CTL$^F$ is particularly easy: We use the algorithm by Clarke, Emerson, and Sistla (1986). It treats a CTL$^F$ formula as a tree (with subformulas as subtrees), and then travels up the tree and model checks each subformula in each state of the Kripke structure. In the end, if the whole formula holds for all initial states of the Kripke structure, it is marked as being true.

This algorithm not only has polynomial complexity, it also facilitates reuse of the results obtained earlier. Some of the subformulas occur in more than one formula, and in that case they do not need to be reevaluated. These are very important properties, that allow us to achieve a very good run-time performance, and thus scalability to real-life programs.

One important point is that we use CTL$^F$ instead of the better-known CTL. The problem we have with CTL$^F$ is that all loops within the Kripke structure under consideration are interpreted as potentially infinite. However, when we consider source code, we assume that most loops are finite: Consider the source code shown in Figure 4.3 and the formula AF `reapPropertyList @ (1)`. Depending on the iterator implementation, the while-loop could be finite or

infinite, and the formula could be true or false, but our intuition tells us that the formula is true, because we implicitly assume that the loop will end after a finite number of iterations. CTL$^F$ allows us to solve this problem by introducing the notion of fairness: only paths that are "fair" are taken into account. As stated in Figure 4.2, a path is fair if a certain (arbitrary, but fixed) predicate holds infinitely often along it. We use the predicate meaning "being in the sink state" and require it to hold infinitely often along a path for the path to be considered fair. As a result, every fair path through a Kripke structure contains only a finite number of states other than the sink state, and thus contains a finite number of iterations of each loop (of course apart from the cycle in the sink state) and this is exactly what we want.

We can now formally define what an event-induced operational precondition abstraction of an object usage model is:

**Definition 4.4** (Event-induced operational precondition abstraction)**.** Let $e$ be an event. Let $oum = (Q, \Sigma, T, q_0, F, Exc)$ an an object usage model (see definition 2.2), $ks = ks(oum, e)$ be the Kripke structure induced by $oum$ and $e$ (see definition 4.1), and $\Phi = ctlf\_fs(ks)$ be the set of CTL$^F$ formulas induced by $ks$. $e$-induced operational precondition abstraction of $oum$ is defined as $\text{opa}(oum, e) = \{\phi : \phi \in \Phi \text{ and } ks \models \phi\}$.

Let us illustrate the definition of an event-induced operational precondition abstraction using an example. Consider the object usage model of a `Stack` object shown in Figure 4.6. The "`Stack.push @ (0)`"-induced operational precondition abstraction of that model is the following set of CTL$^F$ formulas:

$$\begin{aligned}
&\text{AF } \texttt{Stack.<init> @ (0)}\\
&\text{AF } \texttt{Stack.push @ (0)}\\
&\text{EF } \texttt{Stack.<init> @ (0)}\\
&\text{EF } \texttt{Stack.push @ (0)}\\
&\text{AG}(\texttt{Stack.<init> @ (0)} \Rightarrow \text{AX AF } \texttt{Stack.push @ (0)})\\
&\text{AG}(\texttt{Stack.<init> @ (0)} \Rightarrow \text{EX EF } \texttt{Stack.push @ (0)})\\
&\text{AG}(\texttt{Stack.push @ (0)} \Rightarrow \text{EX EF } \texttt{Stack.push @ (0)})
\end{aligned}$$

We can now demonstrate that operational precondition abstraction is more expressive than sequential constraints abstraction. Consider another object usage model of a `Stack` object, shown in Figure 4.7. Its "`Stack.push @ (0)`"-induced operational precondition abstraction of that model is the following set of CTL$^F$ formulas:[7]

[7] We again assume here that all events from the object usage model will actually be used to generate CTL$^F$ formulas.

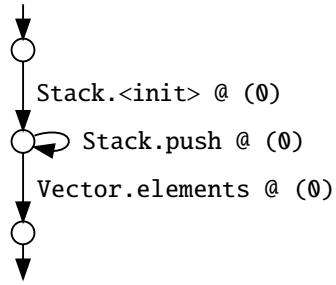



Figure 4.6: OUM for a Stack object.

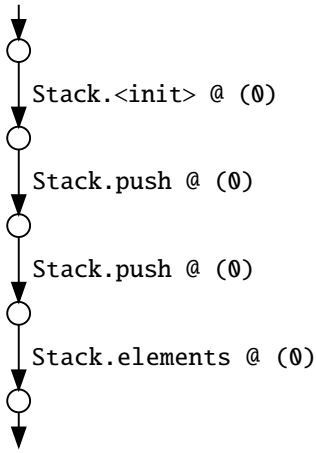



Figure 4.7: Hypothetical OUM for a Stack object with different operational precondition abstraction, but the same sequential constraints abstraction as the OUM shown in Figure 4.6.

$\mathrm{AF}$ `Stack.<init> @ (0)`
$\mathrm{AF}$ `Stack.push @ (0)`
$\mathrm{EF}$ `Stack.<init> @ (0)`
$\mathrm{EF}$ `Stack.push @ (0)`
$\mathrm{AG}$ (`Stack.<init> @ (0)` $\Rightarrow \mathrm{AX}\,\mathrm{AF}$ `Stack.push @ (0)`)
$\mathrm{AG}$ (`Stack.<init> @ (0)` $\Rightarrow \mathrm{EX}\,\mathrm{EF}$ `Stack.push @ (0)`)

This operational precondition abstraction and the one shown earlier differ—the last $\mathrm{CTL}^{\mathrm{F}}$ formula from the latter does not appear in the former. However, sequential constraints abstractions of these two object usage models are identical (see Section 3.2.1), which means that the operational precondition abstraction caught a difference between a loop and several consecutive calls, while the sequential constraints abstraction was not able to do this.

### 4.3.3 Mining Operational Preconditions and their Violations

Now that we are able to abstract each object usage model into an (event-induced) operational precondition abstraction, we can apply a slightly modified version of the techniques presented in Sections 3.3.1 and 3.4.1 to mine operational preconditions and their violations. In these sections, we have shown how we can find patterns and their violations based on methods' sequential constraints abstractions. For mining operational preconditions, we have to take into account an important difference between the two abstractions: sequential constraints abstraction was unfocused (i.e., we simply looked for characteristics of methods), while operational precondition abstraction focuses on one target event at a time (this is the event we want to discover operational preconditions of). This means that we must modify the aforementioned techniques in the following ways:

- There is no longer just one cross table representing sequential constraints occurring in methods. The operational precondition abstraction of each object usage model depends on the choice of the target event, and thus for each event, for which we want to mine operational preconditions, we have to create a separate cross table.

- Conceptual objects in the cross table cannot be methods anymore. This is because a pattern now is an operational preconditions—which is a description on how *an object* needs to prepared before being used in a

certain event. Likewise, violations of operational preconditions are *objects* that are prepared incorrectly. So we must use objects as conceptual objects.

- Conceptual properties are not sequential constraints, but $\text{CTL}^{\text{F}}$ formulas. This is a straightforward and obvious modification.

If we apply the aforementioned modifications, we will have to change the definitions of the pattern (see definition 3.4)—this one becomes an operational precondition, violation (see definition 3.5), and confidence (see definition 3.6). The new definitions are:

**Definition 4.5** (Operational precondition, support, size, closed operational precondition)**.** Let $e$ be an event, and $m_1, \ldots, m_n$ be methods. Let $oums(m_i)$ be the set of object usage models stemming from $m_i$, defined as $oums(m) = \{oum(x) : x \in \text{Obj}(m)\}$. $P$ is an *operational precondition* of $e$ supported by $s$ object usage models iff $|\{oum : \exists i \,.\, 1 \leq i \leq n \text{ and } oum \in oums(m_i) \text{ and } P \subseteq \text{opa}(oum, e)\}| = s$. $s$ is called the *support* of $P$. $|P|$ is called the size of $P$. $P$ is a *closed operational precondition* iff for all $P' \supset P$ we have $|\{oum : \exists i \,.\, 1 \leq i \leq n \text{ and } oum \in oums(m_i) \text{ and } P' \subseteq \text{opa}(oum, e)\}| < s$.

We will normally use the term "operational precondition" to mean "closed operational precondition", and all exceptions will be explicitly stated. Just as with sequential constraints abstraction, the number and size of patterns that will be discovered can be influenced by adjusting the *minimum size* and *minimum support* (see Section 3.3.2). We set the minimum support to the same value as before: 20. However, the minimum size we use for mining operational preconditions is 3 instead of 1. The reason for this is that every single event $e$ has an operational precondition of size 2, consisting of the formulas AF $e$ and EF $e$. These are trivial and not interesting, so we do not want them mined. When it comes to filtering, when creating sequential constraints abstraction we have ignored all events related to one of the following classes: `StringBuffer`, `String`, and `StringBuilder` (see Section 3.3.2). Now, we do not filter any events, but we do not look for operational preconditions of events that are calls to methods from these classes.

**Definition 4.6** (Violation, deviation, deviation level)**.** Let $e$ be an event, $oum$ be an object usage model, and $P$ be an operational precondition. Let $\text{opa}(oum, e)$ be $oum$'s operational precondition abstraction (see definition 4.4). *$oum$ violates $P$* iff $P \cap \text{opa}(oum, e) \neq \emptyset$, and there exists $p \in P$ such that $p \notin \text{opa}(oum, e)$. $D(P, oum, e) = P \setminus \text{opa}(oum, e)$ is the *deviation* of the violation. $|D(P, oum, e)|$ is the *deviation level* of the violation.

**Definition 4.7** (Confidence). Let $e$ be an event, $m_1, \ldots, m_n$ be methods, and $P$ be an operational precondition supported by $s$ object usage models. Let $oums(m_i)$ be the set of object usage models stemming from $m_i$, defined as $oums(m) = \{oum(x) : x \in \text{Obj}(m)\}$. Let $oum$ be an object usage model that violates $P$ and $D(P, oum, e)$ be the deviation of the violation. Let $n_v = |\{oum' : \exists i \,.\, 1 \leq i \leq n \text{ and } oum \in oums(m_i) \text{ and } D(P, oum', e) = D(P, oum, e)\}|$ be the number of object usage models that violate the same operational precondition in the same way. $s/(s + n_v)$ is the *confidence* of the deviation $D(P, oum, e)$.

After mining violations of operational preconditions they are ranked exactly as described in Section 3.4.2. One difference introduced by needs of operational preconditions abstraction to this scheme is a different value used as the normalization factor when calculating conviction. Since we will at the same time report violations of all operational preconditions found, we need our ranking scheme to take into account the fact that we have multiple cross tables with different sizes. Therefore, for each violation we use the number of times the target event of the operational precondition occurs in the program being analyzed as the normalization factor.

## 4.4 Operational Preconditions: A Case Study

We have implemented finding operational preconditions and their violations in a tool called *Tikanga*[8]. First, we used Tikanga to mine operational preconditions from projects listed in Table 2.1. Table 4.1 lists all the projects and the results obtained for them: number of events, for which operational preconditions were found ("Events with OPs"), total number of operational preconditions found ("Total OPs"), and the time (wall clock time, averaged over ten consecutive runs) that was needed to perform the analysis on a 2.53 GHz Intel Core 2 Duo machine with 4 GB of RAM.

As we can see, mining operational preconditions takes more time that mining patterns consisting of sequential constraints, but the difference is not that large considering that model-checking is time-consuming. The number of operational preconditions found varies from project to project, but not as much as in the case of sequential constraints, and the number of patterns consisting of sequential constraints found for a project is not correlated with the number of operational preconditions found (for example, there are 2887 patterns found in Vuze, but only 1034 operational preconditions; on the other hand, there are 56 patterns found in Act-Rbot, but as many as 225 operational preconditions).

[8] "Tikanga" is the Māori word for "correct procedure".

Table 4.1: Operational preconditions found in the case study subjects.

| Program | Events with OPs | Total OPs | Time (mm:ss) |
|---|---|---|---|
| Vuze 3.1.1.0 | 265 | 1,034 | 0:58 |
| AspectJ 1.5.3 | 372 | 1,162 | 0:59 |
| Apache Tomcat 6.0.18 | 154 | 442 | 0:29 |
| ArgoUML 0.26 | 182 | 369 | 0:26 |
| Columba 1.4 | 97 | 185 | 0:16 |
| Act-Rbot 0.8.2 | 96 | 225 | 0:17 |

$$\mathrm{AF}\,\texttt{Label.<init> @ (0)} \tag{4.1}$$
$$\mathrm{EF}\,\texttt{Label.<init> @ (0)} \tag{4.2}$$
$$\mathrm{AF}\,\texttt{Label.place @ (0)} \tag{4.3}$$
$$\mathrm{EF}\,\texttt{Label.place @ (0)} \tag{4.4}$$
$$\mathrm{EF}\,\texttt{CodeStream.goto\_ (Label) @ 1} \tag{4.5}$$
$$\mathrm{AG}\,(\texttt{Label.<init> @ (0)} \Rightarrow \mathrm{EX}\,\mathrm{EF}\,\texttt{CodeStream.goto\_ @ (1)}) \tag{4.6}$$
$$\mathrm{AG}\,(\texttt{Label.<init> @ (0)} \Rightarrow \mathrm{AX}\,\mathrm{AF}\,\texttt{Label.place @ (0)}) \tag{4.7}$$
$$\mathrm{AG}\,(\texttt{Label.<init> @ (0)} \Rightarrow \mathrm{EX}\,\mathrm{EF}\,\texttt{Label.place @ (0)}) \tag{4.8}$$
$$\mathrm{AG}\,(\texttt{CodeStream.goto\_ @ (1)} \Rightarrow \mathrm{AX}\,\mathrm{AF}\,\texttt{Label.place @ (0)}) \tag{4.9}$$
$$\mathrm{AG}\,(\texttt{CodeStream.goto\_ @ (1)} \Rightarrow \mathrm{EX}\,\mathrm{EF}\,\texttt{Label.place @ (0)}) \tag{4.10}$$

Figure 4.8: Operational precondition for the target object of a call to `Label.place()`. See Section 4.4 for a discussion.

As an example of an operational precondition, consider the one shown in Figure 4.8 for the target object of a call to `Label.place()`. This operational precondition has been extracted from AspectJ. The code that this operational precondition stems from is responsible for generating bytecode instructions, among them `goto` statements, which jump unconditionally to a specific label. The operational precondition contains the following rules:

1. A label is always created by calling its constructor before being placed (which is not that obvious, as there may also be a factory method to be used instead) (4.1, 4.2)
2. A label is always placed (which is obvious, as this operation is the target of the operational precondition) (4.3, 4.4)
3. A label *can* be used as the target of a goto statement (4.5)

Table 4.2: Violations found in the case study subjects.

| Program | Violations | Time (mm:ss) |
|---|---|---|
| Vuze 3.1.1.0 | 224 | 0:58 |
| AspectJ 1.5.3 | 169 | 0:59 |
| Apache Tomcat 6.0.18 | 14 | 0:28 |
| ArgoUML 0.26 | 51 | 0:25 |
| Columba 1.4 | 22 | 0:16 |
| Act-Rbot 0.8.2 | 15 | 0:16 |

4. After a label has been created, it *can* be used as the target of a goto statement (4.6)

5. After a label is created, it is always placed (4.7, 4.8)

6. After a label is used as the target of a goto statement, it is always placed (4.9, 4.10)

Note that AspectJ apparently may produce labels that are not referenced by a `goto` statement; we do not see this as a problem.

## 4.5 Operational Preconditions' Violations: Experiments

In the next experiment, we run Tikanga again on all the projects shown in Table 2.1, this time for the purpose of finding violations of operational preconditions. Summary of the results can be found in Table 4.2. For each project we report on the total number of violations found and on the time (wall clock time, averaged over ten consecutive runs) that was needed to perform the analysis on a 2.53 GHz Intel Core 2 Duo machine with 4 GB of RAM. As we can see, there is a large difference between the number of violations found in different projects, even for projects that are of similar size (like Apache Tomcat and ArgoUML, cf. Table 2.1). Ideally we would investigate every single violation found in each project and classify it to find out how many of those are true positives (more on that below). While for Apache Tomcat and Act-Rbot it is entirely possible to investigate all the violations, for Vuze 3.1.1.0 the number of violations found is large enough to make this a time-consuming and mundane task. However, since in total the number of violations is not as large as it was the case for the sequential constraints abstraction, we have decided to investigate 25% top-ranked violations for each

Table 4.3: Classification results for top 25% violations in each project. "CSs" stands for the number of code smells. "FPs" stands for the number of false positives.

| Program | Classified | Defects | CSs | FPs | Effectiveness |
|---|---|---|---|---|---|
| Vuze 3.1.1.0 | 56 | 0 | 11 | 45 | **19%** |
| AspectJ 1.5.3 | 42 | 9 | 13 | 20 | **52%** |
| Apache Tomcat 6.0.18 | 3 | 0 | 2 | 1 | **66%** |
| ArgoUML 0.26 | 12 | 1 | 6 | 5 | **58%** |
| Columba 1.4 | 5 | 1 | 4 | 0 | **100%** |
| Act-Rbot 0.8.2 | 3 | 1 | 0 | 2 | **33%** |
| **Overall** | 121 | 12 | 36 | 73 | **39%** |

project. We classified all the violations into the same three categories as before: defects, code smells, and false positives (see Section 3.4.3. The results of this classification can be found in Table 4.3. For each project we report on the number of violations that were classified[9], the number of defects, code smells (CSs), false positives (FPs), and the effectiveness (i.e., the percentage of violations that were defects or code smells). We also report on the overall effectiveness.

We can see that, in general, even though we have investigated roughly the same absolute number of violations as in the top 10% violations found for sequential constraints abstraction, the results obtained by Tikanga are much better than the results obtained by JADET (Section 3.4.3), and the number of defects found went up from 5 to 12 (see Table 3.4). Let us take a look at some of the violations we have classified.

Out of nine violations classified as defects in AspectJ, one is severe enough to cause a compiler crash (previously reported as bug #218,167). It is a simple typo resulting in a violation of an operational precondition of the `Iterator.next()` method. The skeleton of the defective code is shown in Figure 4.9. Iterator `it2` violates the operational precondition, because `hasNext()` is not being called before `next()` is called. This defect could not possibly be found using a sequential constraints abstraction, because the method, where the defect occurs, uses multiple iterators, and the correct usages overshadow the incorrect usage.

Two of the violations classified as defects in AspectJ occur in code that uses progress monitors incorrectly. Documentation of the `IProgressMonitor`

[9]This is sometimes more than exactly 25%. The reason for this is that some violations have the same ranking (i.e., the same conviction value), so we had to include all such equally-ranked violations.

```
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 4.9: One of the defects found by Tikanga in AspectJ. In this method an inner loop checks the iterator of the outer loop.

interface states that one has to call `beginTask()` before calling `worked()` and `done()` on the monitor, and in these two violations this is not the case. This rule was mined as an operational precondition of both the `worked()` and `done()` method, and that is why Tikanga found these defects.

Five of the violations classified as defects in AspectJ are located in methods that violate the contract of the method they override. All 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 the method `IProgressMonitor.done()` states that the monitor should be the return value from the factory method `Policy.monitorFor()`. 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`, they throw a `NullPointerException` if they are called with `null` as the progress monitor.

Figure 4.10 shows a skeleton of the remaining defect found in AspectJ (previously reported as bug #165,631). The loop in this code processes only the first element returned by the iterator, even though it should process all of them. As a result of this omission, in some cases AspectJ performs an error-free compilation of type-incorrect code.

Figure 4.11 shows the skeleton of the defect found by Tikanga in ArgoUML. The problem with this code is that it just processes the first dependency from the list of dependencies. This method is used when assigning a component to a diagram. If this component is already connected to multiple other components in that diagram (via specialization/generalization), only one connection is going to get added instead of all of them. This defect was detected by Tikanga, because the `Iterator` object that is being implicitly used here violates an operational precondition of the `hasNext()` method.

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

Figure 4.10: One of the defects found by Tikanga in AspectJ. The loop body is executed at most once.

```
        public void addNodeRelatedEdges(Object node) {
            ...
            if (Model.getFacade().isAModelElement(node)) {
                List dependencies = ...;
                dependencies.addAll (...);
                for (Object dependency : dependencies) {
                    if (canAddEdge(dependency)) {
                        addEdge(dependency);
                    }
                    return;
                }
            }
        }
```

Figure 4.11: The defect found by Tikanga in ArgoUML. This code misses dependencies while adding edges related to a node.

```
    public Link getStrongestLink() {
        Link strongestLink = null;
        for (Link link : links) {
            strongestLink = (strongestLink.getActivation() >
                link.getActivation() ? link : strongestLink);
        }
        return strongestLink;
    }
```

Figure 4.12: The defect found by Tikanga in Act-Rbot. The loop will terminate during the first iteration with the `NullPointerException` being thrown.

```
SimpleName name = ...;
...
recordNodes (name, ...);
...
name.setSourceRange (..., ...);
```

Figure 4.13: Example of a code smell in AspectJ. For preventive reasons, `setSourceRange()` should be called before `recordNodes()`.

The defect found by Tikanga in Columba is the same as the one found by JADET (see Figure 3.17). Figure 4.12 shows the defect found in Act-Rbot; it is identical to the defect found by JADET (Figure 3.19), but it occurs in a different method.

As an example of a code smell, consider the one found in AspectJ (shown in Figure 4.13). This code creates an object of type `SimpleName`, puts it in some data structure (using `recordNodes()`) and then modifies the object by calling `setSourceRange()` on it. This order of calls (`recordNodes()` before `setSourceRange()`) is very uncommon in AspectJ, and for a reason: `recordNodes()` puts the object into a hashtable, and for this it needs the object's hashcode. Currently, implementation of `SimpleName` uses the default implementation of the `hashCode()` method, but if developers of AspectJ decide at some point to implement the method themselves (which would be recommended for classes representing AST nodes, such as `SimpleName`), they will most probably use object's fields' values to create the hashcode, and source range of a simple name is stored in such fields. This would lead to a difficult-to-discover bug in the code shown. Setting up the object before acting on it is a much safer—and therefore preferred—option.

### Sensitivity Analysis

Just as with sequential constraints abstraction, we would like to see how sensitive our results are to small changes to parameters such as minimum support or the number of violations investigated. Therefore, we have investigated the influence of small changes to minimum support, minimum confidence, and the number of classified violations on the effectiveness of Tikanga on AspectJ. While manipulating one parameter, we kept all others at their default values. Results of these investigations can be found in Figures 4.14, 4.15, and 4.16, respectively.

We can see that effectiveness is fairly insensitive to small changes to minimum support, minimum confidence, and the number of top violations investigated. Overall, Tikanga is much more stable in this respect than JADET.

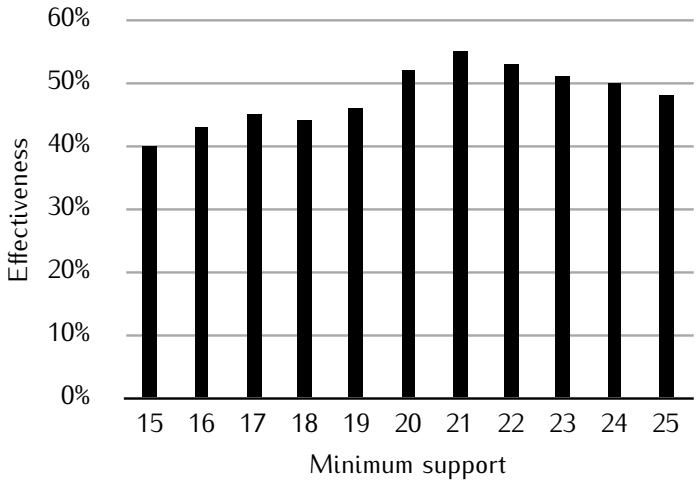



Figure 4.14: Influence of minimum support on the effectiveness of Tikanga for AspectJ (other parameters fixed at their default values).

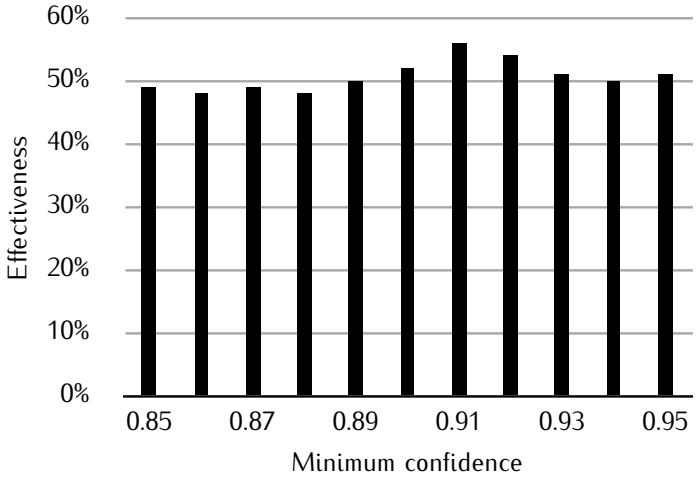



Figure 4.15: Influence of minimum confidence on the effectiveness of Tikanga for AspectJ (other parameters fixed at their default values).

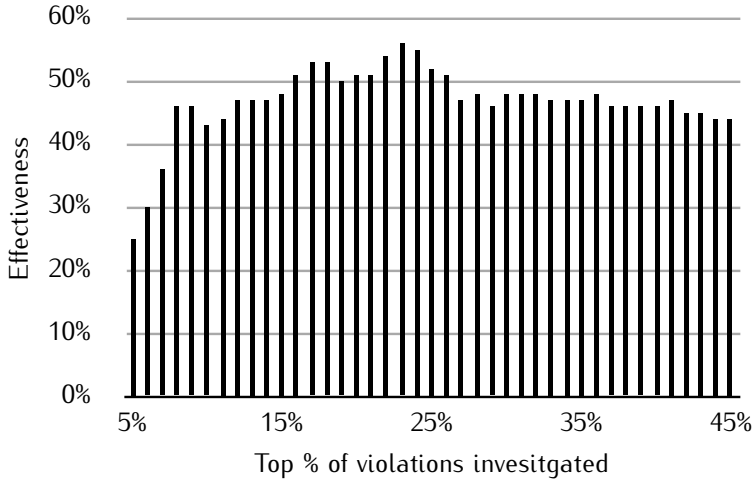



Figure 4.16: Influence of the number of violations classified on the effectiveness of Tikanga for AspectJ (other parameters fixed at their default values).

However, one important thing worth taking a closer look at is that effectiveness is fairly insensitive to small changes to the number of top violations investigated. This is quite unexpected, and shows that the ranking does not perform here as well as with JADET. On the other hand, this further illustrates Tikanga's advantage over JADET: Even with an underperforming ranking system, Tikanga is still much better. There is also an interesting artifact here: Effectiveness rises to a certain point, indicating that most interesting violations are not at the top, but near it.

## 4.6 Related Work

To the best of our knowledge, the present work is the first to take an operational view at preconditions, and the first to learn temporal logic specifications directly from program code, instead of requiring them to be provided by the user. However, learning CTL formulas from Kripke structures has been done before by Chan (2000). Chan introduced the concept of so-called temporal-logic queries, which are similar to our templates, but restricted to only one placeholder. This work was extended later to queries with multiple placeholders (Gurfinkel, Devereux, and Chechik 2002; Gurfinkel, Chechik, and Devereux 2003). Temporal logic queries with multiple placeholders are very close to our templates, but the technique of solving them is not directly applicable in our setting: Solutions to queries are the strongest possible for-

mulas, whereas we need all possible formulas in order to be able to look for frequently occurring ones later on.

When it comes to learning from existing code in general, or to automatic defect detection, all the related work has been mentioned in the preceding chapter and we refer the reader to Section 3.6 for details.

## 4.7 Summary

This chapter makes the following contributions:

- We have introduced the concept of *operational preconditions*. While traditional preconditions specify *what* the state must be for a method call to be successful, operational preconditions specify *how* to achieve that state. This directly helps the programmer wishing to know how to correctly call a method.
- We have proposed using temporal logic ($CTL^F$) for specifying operational preconditions.
- We have shown how we can combine model checking with formal concept analysis to automatically mine operational preconditions from programs. Our analysis scales very well: It takes less than *one minute* for a large project like Vuze (345 K SLOC, 35,363 methods in 5532 classes). This is the first time that real formal specifications in the form of temporal logic formulas are fully automatically mined from a program.
- We have shown how we can quickly find violations of operational preconditions. Our analysis takes less than *one minute* for a large project like Vuze. Violations of operational preconditions have the benefit that they show which $CTL^F$ formulas are not satisfied, thus helping the programmer fix the code, if it turns out to be defective.
- We have implemented all the techniques above in a tool called Tikanga and evaluated it on six open-source projects. Tikanga found defects or code smells in all of those projects. In total, investigating top 25% violations for all the projects resulted in finding 12 defects and 36 code smells—for a true positive rate of 39%.

To learn more about our work on operational preconditions and related topics, see:

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

# Chapter 5

# Conclusions and Future Work

In modern object-oriented programs, most complexity stems not from within the methods, but from method compositions. To correctly use those methods, the programmer needs to know how how are they supposed to be combined to achieve the required effect. This information should come from a documentation, or—even better—from documentation combined with formal specification, but these are often outdated, incomplete, ambiguous or simply missing. This is true especially when it comes to formal specifications, which are notoriously difficult to get right. To cope with these problems, programmers frequently resort to consulting code examples, but these can be defective, and the programmer—not knowing the API—is not in a position to decide if an example is correct or not. This dissertation makes the following contributions aimed at helping to solve this problem:

- We have introduced the notion of *object usage models*, and shown how we can mine them from programs using static analysis. Object usage models show how objects are being used *from the perspective of the programmer*, and are thus not limited to a fixed abstraction level—on the contrary, even objects of the same class can be modeled from different perspectives, if they were so used by the programmer. Our analysis scales to large programs: Analyzing Vuze (the largest program we used in our experiments, 345 K SLOC, 35,363 methods in 5532 classes) takes slightly more than three minutes.
- We have introduced the notion of *sequential constraints abstraction* and shown how it can be applied to find patterns and anomalies of

object usage. We have created a tool, JADET, that is able to find *object usage patterns* and their violations, each in less than *half a minute* for a large program like Vuze. JADET managed to discover 5 defects and 23 code smells in top 10 violations for six programs we used as case study subjects.

- We have shown how—by using lightweight static analysis—we can make the concepts used in JADET scale to handle *thousands* of projects at a time. Early results indicate that cross-project analysis can find subtle defects that are not detectable using single-project analysis. Our `checkmycode.org` Web site allows programmers to upload their code and have it checked against more than 6000 C projects from the Gentoo Linux distribution.

- We have introduced the concept of *operational preconditions*. While traditional preconditions specify *what* the state must be for a method call to be successful, operational preconditions specify *how* to achieve that state. This directly helps the programmer wishing to know how to correctly call a method. We have implemented a tool called Tikanga that can find such operational preconditions (as temporal logic [$CTL^F$] formulas), and their violations, fully automatically by analyzing a program—in less than *one minute* for a large program like Vuze. Tikanga managed to discover 12 defects and 36 code smells in top 25% violations for six programs we used as case study subjects. This is the first time that real formal specifications in the form of temporal logic formulas were fully automatically mined from a program.

However, there is still a lot that can be done to extend the ideas presented by this dissertation. Some of the topics that seem particularly worthy of investigation are:

**Improved abstraction.** The sequential constraints abstraction presented in Chapter 3 is effective, but quite limited. It would be interesting to see if extending it (e.g., in ways described in Section 3.2.2) would bring effectiveness improvements, or is it too limited, and we should use much more complicated abstractions, such as the one presented in Chapter 4.

**Negative examples.** Patterns and operational preconditions contain only "positive" information: this is what should be done, this is how the parameter should be prepared. However, having negative information (this is what you should not do, etc.) is valuable, too. Adding such negative information seems to be quite a challenge, but—if successful—could bring significant improvements.

**Early programmer support.** Patterns and operational preconditions mined from projects need not be used just for the purpose of finding violations. Integrating them into the programming environment, in order to warn the user about potential problems, could be an effective means for preventing defects, instead of detecting them after they have been introduced. Operational preconditions in particular could become part of the documentation, just as preconditions are typically explicitly enumerated.

**Better CTL$^F$ templates.** We have presented a rather simple set of CTL$^F$ templates used to construct operational preconditions, and yet obtained promising result. It would be interesting to see which templates are most effective in providing clear and useful operational preconditions, and in their defect detection ability.

**Cross-project mining of operational preconditions.** Our cross-project anomaly detection approach described in Section 3.5 builds on the sequential constraints abstraction. This was an obvious first choice, but it seems that mining operational preconditions from multiple projects at a time has a much greater potential.

To learn more about the work presented in this dissertation, as well as related topics, see:

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

# References

Acharya, Mithun, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 25–34. New York, NY: ACM.

Acharya, Mithun, Tao Xie, and Jun Xu. 2006. Mining interface specifications for generating checkable robustness properties. In *ISSRE 2006: Proceedings of the 17th International Symposium on Software Reliability Engineering*, 311–320. Los Alamitos, CA: IEEE Computer Society.

Agrawal, Rakesh, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *SIGMOD 1993: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 207–216. New York, NY: ACM.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1988. *Compilers: Principles, techniques, and tools.* Reading, MA: Addison-Wesley Publishing Company.

Allan, Chris, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Adding trace matching with free variables to AspectJ. In *OOPSLA 2005: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 345–364. New York, NY: ACM.

Alur, Rajeev, Pavol Černý, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 98–109. New York, NY: ACM.

Ammons, Glenn, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 4–16. New York, NY: ACM.

Bierhoff, Kevin, and Jonathan Aldrich. 2005. Lightweight object specification with typestates. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 217–226. New York, NY: ACM.

Bodden, Eric, Patrick Lam, and Laurie Hendren. 2008. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE 2008: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 36–47. New York, NY: ACM.

Brin, Sergey, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. 1997. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD 1997: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 255–264. New York, NY: ACM.

Chakrabarti, Arindam, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdzinski, and Freddy Y. C. Mang. 2002. Interface compatibility checking for software modules. In *CAV 2002: Proceedings of the 14th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, 428–441. Berlin: Springer-Verlag.

Chan, William. 2000. Temporal-logic queries. In *CAV 2000: Proceedings of the 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 1855, 450–463. Berlin: Springer-Verlag.

Chang, Ray-Yaung, Andy Podgurski, and Jiong Yang. 2007. Finding what's not there: A new approach to revealing neglected conditions in software. In *ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis*, 163–173. New York, NY: ACM.

Clarke, E. M., E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8, no. 2 (April): 244–263.

Clarke, Edmund M., and E. Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In *LOP 1982: Proceedings of the Workshop on Logics of Programs*, Lecture Notes in Computer Science 131, 52–71. Berlin: Springer-Verlag.

Cook, Jonathan E., and Alexander L. Wolf. 1995. Automating process discovery through event-data analysis. In *ICSE 1995: Proceedings of the 17th international conference on Software engineering*, 73–82. New York, NY: ACM.

———. 1998. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology* 7, no. 3 (July): 215–249.

Dallmeier, Valentin, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *WODA 2006: Proceedings of the 2006 international workshop on Dynamic systems analysis*, 17–24. New York, NY: ACM.

Das, Manuvir, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 57–68. New York, NY: ACM.

de Alfaro, Luca, and Thomas A. Henzinger. 2001. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 109–120. New York, NY: ACM.

DeLine, Robert, and Manuel Fähndrich. 2001. Enforcing high-level protocols in low-level software. In *PLDI 2001: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 59–69. New York, NY: ACM.

———. 2004. Typestates for objects. In *ECOOP 2004: Proceedings of the 18th European conference on object-oriented programming*, Lecture Notes in Computer Science 3086, 465–490. Berlin: Springer-Verlag.

Dunn, Robert H. 1984. *Software defect removal.* New York: McGraw-Hill.

Dwyer, Matthew B., Alex Kinneer, and Sebastian Elbaum. 2007. Adaptive online program analysis. In *ICSE 2007; Proceedings of the 29th international conference on Software Engineering*, 220–229. Los Alamitos, CA: IEEE Computer Society.

Dwyer, Matthew B., and Rahul Purandare. 2007. Residual dynamic typestate analysis: exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 124–133. New York, NY: ACM.

Eisenbarth, Thomas, Rainer Koschke, and Gunther Vogel. 2002. Static trace extraction. In *WCRE 2002: Proceedings of the Ninth Working Conference on Reverse Engineering*, 128–137. Los Alamitos, CA: IEEE Computer Society.

———. 2005. Static object trace extraction for programs with pointers. *Journal of Systems and Software* 77, no. 3 (September): 263–284.

Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP 2001: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 57–72. New York, NY: ACM.

Fink, Stephen J., Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology* 17, no. 2 (April): 1–34.

Fink, Stephen, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis*, 133–144. New York, NY: ACM.

Fowler, Martin. 1999. *Refactoring. Improving the design of existing code.* N.p.: Addison-Wesley.

Gabel, Mark, and Zhendong Su. 2008. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT 2008/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 339–349. New York, NY: ACM.

———. 2010. Online inference and enforcement of temporal properties. In *ICSE 2010: Proceedings of the 32nd International Conference on Software Engineering*, 15–24. New York, NY: ACM.

Ganter, Bernhard, and Rudolf Wille. 1999. *Formal concept analysis: Mathematical foundations.* Berlin: Springer-Verlag.

Ghezzi, Carlo, Andrea Mocci, and Mattia Monga. 2007. Efficient recovery of algebraic specifications for stateful components. In *IWPSE '07: Proceedings of the ninth international workshop on Principles of software evolution*, 98–105. New York, NY: ACM.

Götzmann, Daniel Norbert. 2007. Formale Begriffsanalyse in Java: Entwurf und Implementierung effizienter Algorithmen. Master's thesis, Saarland

University. Publication and software available from http://code.google.com/p/colibri-java/ (accessed 6 April 2010).

Gruska, Natalie. 2009. Language-independent sequential constraint mining. Master's thesis, Saarland University.

Gruska, Natalie, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *ISSTA 2010: Proceedings of the nineteenth international symposium on Software testing and analysis*. (At the time of writing this has not been published yet).

Guo, Philip J., Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. 2006. Dynamic inference of abstract types. In *ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis*, 255–265. New York, NY: ACM.

Gurfinkel, Arie, Marsha Chechik, and Benet Devereux. 2003. Temporal logic query checking: A tool for model exploration. *IEEE Transactions on Software Engineering* 29, no. 10 (October): 898–914.

Gurfinkel, Arie, Benet Devereux, and Marsha Chechik. 2002. Model exploration with temporal logic query checking. In *FSE 2002: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, 139–148. New York, NY: ACM.

Henkel, Johannes, and Amer Diwan. 2003. Discovering algebraic specifications from Java classes. In *ECOOP 2003: Proceedings of the 17th European conference on object-oriented programming*, Lecture Notes in Computer Science 2743, 431-456. Berlin: Springer-Verlag.

Henzinger, Thomas A., Ranjit Jhala, and Rupak Majumdar. 2005. Permissive interfaces. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 31–40. New York, NY: ACM.

Holmes, Reid, and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *ICSE 2005: Proceedings of the 27th international conference on Software engineering*, 117–125. New York, NY: ACM.

Hovemeyer, David, and William Pugh. 2004. Finding bugs is easy. In *OOPSLA 2004: Companion to the 19th annual ACM SIGPLAN conference*

*on Object-oriented programming systems, languages, and applications*, 132–136. New York, NY: ACM.

Java™ 2 Platform Standard Edition 5.0 API specification. http://java.sun.com/j2se/1.5.0/docs/api/ (accessed 2 October 2008).

Jonsson, Bengt, Ahmed Hussain Khan, and Joachim Parrow. 1990. Implementing a model checking algorithm by adapting existing automated tools. In *AVMFSS 1990: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, 179–188. Berlin: Springer-Verlag.

Kagdi, Huzefa, Michael L. Collard, and Jonathan I. Maletic. 2007a. An approach to mining call-usage patterns with syntactic context. In *ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 457–460. New York, NY: ACM.

———. 2007b. Comparing approaches to mining source code for call-usage patterns. In *MSR 2007: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 20. Los Alamitos, CA: IEEE Computer Society.

Le Goues, Claire, and Westley Weimer. 2009. Specification mining with few false positives. In *TACAS 2009: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, Lecture Notes in Computer Science 5505, 292–306. Berlin: Springer-Verlag.

Lee, Edward A., and Yuhong Xiong. 2001. System-level types for component-based design. In *EMSOFT 2001: Proceedings of the First International Workshop on Embedded Software*, Lecture Notes in Computer Science 2211, 237–253. Berlin: Springer-Verlag.

Li, Zhenmin, and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 306–315. New York, NY: ACM.

Lindholm, Tim, and Frank Yellin. 1999. *The Java™ virtual machine specification*. 2nd ed. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html (accessed 2 October 2008).

Lindig, Christian. 2007. Mining patterns and violations using concept analysis. Technical Report, Saarland University, Software Engineering Chair. Available from http://www.st.cs.uni-sb.de/publications/; the software is available from http://code.google.com/p/colibri-ml/ (accessed 7 October 2008).

Liu, Chang, En Ye, and Debra J. Richardson. 2006. LtRules: An automated software library usage rule extraction tool. In *ICSE 2006: Proceedings of the 28th international conference on Software engineering*, 823–826. New York, NY: ACM.

Livshits, Benjamin, and Thomas Zimmermann. 2005. DynaMine: Finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 296–305. New York, NY: ACM.

Lo, David, and Siau-Cheng Khoo. 2006. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE 2006: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 265–275. New York, NY: ACM.

Lo, David, Shahar Maoz, and Siau-Cheng Khoo. 2007. Mining modal scenario-based specifications from execution traces of reactive systems. In *ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 465–468. New York, NY: ACM.

Lorenzoli, Davide, Leonardo Mariani, and Mauro Pezzè. 2006. Inferring state-based behavior models. In *WODA 2006: Proceedings of the 2006 international workshop on Dynamic systems analysis*, 25–32. New York, NY: ACM.

———. 2008. Automatic generation of software behavioral models. In *ICSE 2008: Proceedings of the 30th international conference on Software engineering*, 501–510. New York, NY: ACM.

Mandelin, David, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: Helping to navigate the API jungle. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 48–61. New York, NY: ACM.

Mariani, Leonardo, and Mauro Pezzè. 2005. Behavior capture and test: Automated analysis of component integration. In *ICECCS 2005: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 292–301. Washington, DC, USA: Los Alamitos, CA.

Marlowe, T. J., and B. G. Ryder. 1990. Properties of data flow frameworks: A unified model. *Acta Informatica* 28, no. 2 (February): 121–163.

Naeem, Nomair A., and Ondřej Lhoták. 2008. Typestate-like analysis of multiple interacting objects. In *OOPSLA 2008: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 347–366. New York, NY: ACM.

Nguyen, Tung Thanh, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *ESEC/FSE 2009: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 383–392. New York, NY: ACM.

Nierstrasz, Oscar. 1993. Regular types for active objects. In *OOPSLA 1993: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1–15. New York, NY: ACM.

O'Callahan, Robert, and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *ICSE 1997: Proceedings of the 19th international conference on Software engineering*, 338–348. New York, NY: ACM.

Olender, Kurt M., and Leon J. Osterweil. 1992. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology* 1, no. 1 (January): 21–52.

Pradel, Michael, and Thomas R. Gross. 2009. Automatic generation of object usage specifications from large method traces. In *ASE 2009: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 371–382. Los Alamitos, CA: IEEE Computer Society.

Quante, Jochen, and Rainer Koschke. 2007. Dynamic protocol recovery. In *WCRE 2007: Proceedings of the 14th Working Conference on Reverse Engineering*, 219–228. Los Alamitos, CA: IEEE Computer Society.

Ramanathan, Murali Krishna, Ananth Grama, and Suresh Jagannathan. 2007a. Path-sensitive inference of function precedence protocols. In *ICSE 2007: Proceedings of the 29th international conference on Software Engineering*, 240–250. Los Alamitos, CA: IEEE Computer Society.

———. 2007b. Static specification inference using predicate mining. In *PLDI 2007: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 123–134. New York, NY: ACM.

Reiss, Steven P. 2005. Specifying and checking component usage. In *AADEBUG 2005: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 13–22. New York, NY: ACM.

Reiss, Steven P., and Manos Renieris. 2001. Encoding program executions. In *ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering*, 221–230. Los Alamitos, CA: IEEE Computer Society.

Rountev, Atanas, and Beth Harkness Connell. 2005. Object naming analysis for reverse-engineered sequence diagrams. In *ICSE 2005: Proceedings of the 27th international conference on Software engineering*, 254–263. New York, NY: ACM.

Rountev, Atanas, Olga Volgin, and Miriam Reddoch. 2005. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *PASTE 2005: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 96–102. New York, NY: ACM.

Sahavechaphan, Naiyana, and Kajal Claypool. 2006. XSnippet: Mining for sample code. In *OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 413–430. New York, NY: ACM.

Shoham, Sharon, Eran Yahav, Stephen J. Fink, and Marco Pistoia. 2008. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering* 34, no. 5 (September): 651–666.

Shoham, Sharon, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis*, 174–184. New York, NY: ACM.

Strom, Robert E., and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* 12, no. 1 (January): 157–171.

Systä, Tarja, Kai Koskimies, and Hausi Müller. 2001. Shimba—an environment for reverse engineering Java software systems. *Software—Practice and Experience* 31, no. 4 (April): 371–394.

Thummalapenta, Suresh, and Tao Xie. 2007. PARSEWeb: A programmer assistant for reusing open source code on the web. In *ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 204–213. New York, NY: ACM.

———. 2009a. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE 2009: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 283–294. Los Alamitos, CA: IEEE Computer Society.

———. 2009b. Mining exception-handling rules as sequence association rules. In *ICSE 2009: Proceedings of the 31st International Conference on Software Engineering*, 496–506. Los Alamitos, CA: IEEE Computer Society.

Wasylkowski, Andrzej. 2007. Mining object usage models. In *ICSE COMPANION 2007: Companion to the proceedings of the 29th International Conference on Software Engineering*, 93–94. Los Alamitos, CA: IEEE Computer Society.

Wasylkowski, Andrzej, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 35–44. New York, NY: ACM.

Weimer, Westley, and George C. Necula. 2005. Mining temporal specifications for error detection. In *TACAS 2005: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*, Lecture Notes in Computer Science 3440, 461–476. Berlin: Springer-Verlag.

Whaley, John, Michael C. Martin, and Monica S. Lam. 2002. Automatic extraction of object-oriented component interfaces. In *ISSTA 2002: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 218–228. New York, NY: ACM.

Williams, Chadd C., and Jeffrey K. Hollingsworth. 2005. Recovering system specific rules from software repositories. In *MSR 2005: Proceedings of the 2005 international workshop on Mining software repositories*, 1–5. New York, NY: ACM.

Xie, Tao, and David Notkin. 2004a. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *ICFEM 2004: Proceed-*

*ings of the 6th international conference on formal engineering methods*, Lecture Notes in Computer Science 3308, 290-305. Berlin: Springer-Verlag.

———. 2004b. Automatic extraction of sliced object state machines for component interfaces. In *SAVCBS 2004: Proceedings of the 3rd workshop on Specification and Verification of Component-Based Systems*, 39–46.

Xie, Tao, and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *MSR 2006: Proceedings of the 2006 international workshop on Mining software repositories*, 54–57. New York, NY, USA: ACM.

Yang, Jinlin, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE 2006: Proceedings of the 28th international conference on Software engineering*, 282–291. New York, NY: ACM.

Yellin, Daniel M., and Robert E. Strom. 1997. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems* 19, no. 2 (March): 292–333.

Yuan, Hai, and Tao Xie. 2005. Automatic extraction of abstract-object-state machines based on branch coverage. In *RETR 2005: Proceedings of the 1st international workshop on Reverse Engineering to Requirements*, 5–11.

Zhong, Hao, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009: Proc. 23rd European conference on object-oriented programming*, Lecture Notes in Computer Science 5653, 318–343. Berlin: Springer-Verlag.