Mining Patterns and Violations using Concept Analysis

Christian Lindig
Saarland University
Department of Computer Science
Saarbrücken, Germany
lindig@cs.uni-sb.de

ABSTRACT

Large programs develop patterns in their implementation and behavior that can be used for defect mining. Previous work used frequent itemset mining to detect such patterns and their violations, which correlate with defects. However, frequent itemset mining gives much more attention to patterns than to the instances of these patterns. We are proposing a more general framework to understand and mine purely structural patterns and violations. By combining patterns and their instances into blocks, we gain access to the rich theory of formal concepts. This results in a novel geometric interpretation, which helps to understand previous mining approaches. Blocks form a hierarchy in which each block corresponds to a pattern and neighboring blocks to a violation. Furthermore, blocks may be computed efficiently and searched for violations. Using our open-source tool Colibri/ML, we mined patterns and violations from five open-source projects in less than a minute each, including the Linux kernel.

Categories and Subject Descriptors

D.2.4 [**Requirements/Specifications**]: Statistical methods; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Documentation; D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Algorithms, Documentation, Reliability

1. INTRODUCTION

While classifying something as a software defect requires a specification, we can *find potential defects without a specification*. This is based on the observation that large software systems exhibit *patterns* in their implementation or behavior and that deviations from these patterns correlate with defects (Engler et al., 2001). An automatic analysis of such deviations is practical for large systems and is especially suited to find latent bugs.

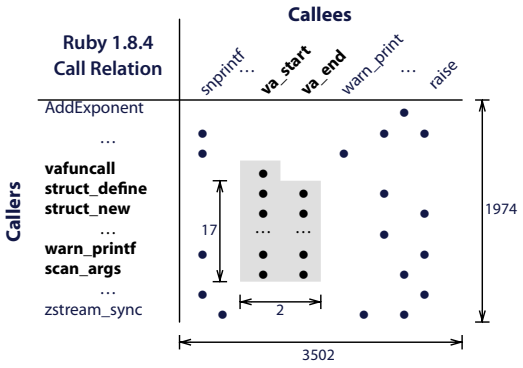

Figure 1: Call relation for Ruby 1.8.4. The *pattern* {`va_start`, `va_end`} becomes visible as a *block*. It is violated by function `vafuncall`. This *violation* becomes visible as an *imperfect block*.

Patterns in code and behavior are a consequence of small and orthogonal interfaces. They force clients to combine functions to implement a certain functionality. For example, implementing in C a function with a varying number of arguments (like `printf`) requires the concerted use of macros `va_start`, `va_arg`, and `va_end`. Hence, we see many functions that call both `va_start` and `va_end`. For example, the source code for the Ruby 1.8.4 interpreter includes seventeen such functions. But it also includes one function (`vafuncall`) that calls `va_start` but not `va_end`. This deviation is indeed a bug that was corrected in a later release.

Mining software for structural patterns and their violations was pioneered by Li and Zhou (2005) with PR-Miner[1], a tool that mines programming rules from source code and flags violations. Patterns are not limited to a known set of patterns or names but are *purely structural*. Li and Zhou demonstrated effectiveness and efficiency of this approach by reporting 27 previously unknown bugs in the Linux kernel, PostgreSQL, and Apache HTTP server. PR-Miner uses frequent itemset mining to detect patterns and their violations. The authors also note that *"frequent itemset mining algorithms were not designed exactly for this purpose"* and develop some ad-hoc mechanisms like applying frequent-item mining twice.

[1]Programming Rule Miner

The goal of this paper is *not* to improve upon the excellent results of PR-Miner but to *improve the foundation for detecting structural patterns and their violations*. Our hope is that this will lead to new applications of the idea that stands behind PR-Miner. In particular, we propose a unified representation for patterns and their instances that uncovers their hierarchical nature and provides an intuitive geometric interpretation.

Our formalism is based on the following insight: any binary relation (like a call relation) can be represented as a cross table like in Figure 1, which sketches the call relation of Ruby 1.8.4. A caller f and a callee g are related (marked with a dot), if f calls g. In such a table rows (callers) and columns (callees) may be permuted without changing the underlying relation. By picking a suitable permutation, we can *make a pattern visible as a block*. Figure 1 shows the block for *pattern* {`va_start`, `va_end`} as well as the seventeen functions that are *instances* of this pattern. In addition, the *violation* of this pattern by function `vafuncall` becomes visible as an *imperfect block*: `vafuncall` calls `va_start` but not `va_end`, which leaves a *gap* in the block.

Mining patterns from a relation can be understood as finding the blocks of the relation. Analogously, detecting violations of patterns can be understood as finding imperfect blocks. *Patterns and violations can be mined from any binary relation*, not just a call relation. However, for illustration we shall stick with the call relation as an example for the most part of the paper and present another application in Section 8.

1.1 Contributions

This paper makes the following contributions:

- *Blocks unify patterns and their instances*, which were previously treated seperately and ad-hoc. Furthermore, blocks provide a *geometric interpretation* of patterns and violations.
- A *block hierarchy* captures the recursive relation of blocks and violations: patterns correspond to blocks and violations correspond to neighboring blocks.
- Case studies show the *efficiency and practicality* of the proposed formalism. Call patterns and their violations could be identified statically for the Python interpreter within twenty seconds, and for the Linux kernel within one minute.
- We draw the connection between patterns, their instances and violations, and formal concept analysis (Ganter and Wille, 1999), which provides a theory to study them.

The remainder of this paper is organized as follows: Section 2 introduces the relation between patterns and blocks, where Section 3 shows how to compute them from an input relation. Analogously, Section 4 introduces violations of patterns and Section 5 shows how to identify them efficiently. Section 6 explores the recursive relation of patterns and violations. Section 7 reports performance numbers gathered from the analysis of open-source projects. Section 8 and Section 9 demonstrate the versatility of the binary relation in program analysis. The paper closes with a discussion of related work in Section 10 and our conclusions in Section 11.

Project	Supp.	Pattern
Ruby 1.8.4	17	`va_start`, `va_end`
Apache HTTP 2.2.2	20	`va_start`, `va_end`
	29	`apr_thread_mutex_lock` `apr_thread_mutex_unlock`
Linux 2.6.10	28	`add_wait_queue`, `remove_wait_queue`
	53	`acpi_ut_acquire_mutex`, `acpi_ut_release_mutex`
	27	`journal_begin`, `journal_end`
Linux 2.6.17	31	`kmalloc`, `copy_from_user`, `kfree`
Phyton 2006-06-20	59	`PyEval_SaveThread`, `PyEval_RestoreThread`

Table 1: Patterns found in open-source projects.

2. PATTERNS AND BLOCKS

A relation associates *objects* and their *features*, like callers and callees in the example above. A *pattern* is a set of features shared by objects. These objects are called the *instances* of the pattern. For defect detection, the goal is to find patterns that have many instances because these patterns are likely to capture a universal principle. As we shall see, patterns and instances are unified by blocks.

For example, the Ruby interpreter contains the following pattern: functions `raise` and `int2inum` are called together from 107 different functions. These functions are the *instances* of *pattern* {`raise`, `int2inum`}. The number of instances (107) is called the *support* for the pattern.

Table 1 illustrates more patterns and their support that we mined from the call relation of systems implemented in C. Most of them show the familiar pattern of allocation and deallocation of a resource. The interesting fact is not that they exist but that we were able to find them without knowing the names of these functions in advance.

Formally, a relation $R \subseteq \mathcal{O} \times \mathcal{F}$ is a set of pairs. Each pair (o, f) relates an *object* $o \in \mathcal{O}$ and a feature $f \in \mathcal{F}$. A pattern is a set of features $F \subseteq \mathcal{F}$ and its instances are a set of objects $O \subseteq \mathcal{O}$. Given a set of objects O we can ask what features these objects share; likewise, given a set of features F we can ask for its instances. Both answers are expressed with the prime operator $'$ (which one can think of as the derivative of a set).

Definition 1 (Features, Instances). *Given a relation $R \subseteq \mathcal{O} \times \mathcal{F}$ and a set of objects $O \subseteq \mathcal{O}$, objects* share *the set $O' \subseteq \mathcal{F}$ of features. Likewise, a set of features $F \subseteq \mathcal{F}$ has* instances *$F' \subseteq \mathcal{O}$, defined as follows:*

$$O' = \{f \in \mathcal{F} \mid (o, f) \in R \text{ for all } o \in O\}$$
$$F' = \{o \in \mathcal{O} \mid (o, f) \in R \text{ for all } f \in F\}$$

A pattern (a set of features) corresponds to a maximal block in a cross table—see Figure 2. A block is characterized by two sets: a pattern and its instances, which form the sides of the block. The formal definition interlocks a pattern and its instances:

Definition 2 (Block). *For a relation $R \subseteq \mathcal{O} \times \mathcal{F}$, a* block *is defined as a pair (O, F) of objects and their features such that $O' = F$ and $F' = O$ holds. The cardinalities $|O|$ and $|F|$ are called* support *and* pattern width*, respectively.*

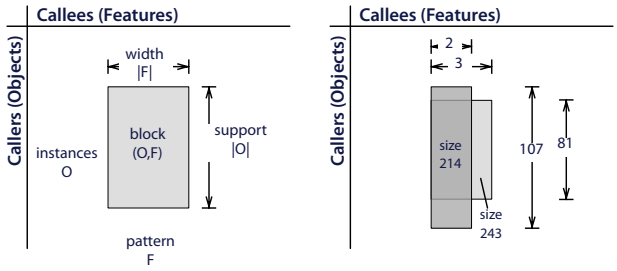

Figure 2: A block is a pair (O, F) of a pattern F and its instances O. Overlapping patterns lead to overlapping blocks, where large patterns have fewer instances and vice versa. The size of a block can be used to identify interesting patterns.

Note that a block is defined as a pair of two *sets* and therefore objects and features are unordered. However, to visualize a block (O, F) in a cross table we have to put the elements of O and F next to each other. For this reason, typically not all blocks can be made visible in a table at the same time.

Because patterns are sets, a subset relation may hold between them. For example, the Ruby interpreter exhibits the pattern {`raise`, `int2inum`}, which has 107 instances. Of these 107 instances, a subset of 81 instances also call function `funcall2`. These 81 instances thus form a *wider* pattern {`raise`, `int2inum`, `funcall2`} with *fewer* instances.

Patterns in a subset relation correspond to overlapping blocks (see Figure 2). Pattern {`raise`, `int2inum`} is represented by a tall but slim block, whereas the larger pattern {`raise`, `int2inum`, `funcall2`} is represented by a wider but shorter block. The *size* $|O| \times |F|$ of a block (O, F) can be used as a criterion to find interesting patterns—large blocks are good candidates.

Blocks unify patterns and their instances.

3. COMPUTING ALL BLOCKS

Finding patterns requires to identify the blocks of a relation. The crucial question is how to do this efficiently, at least for the blocks that we are most interested in.

The problem of computing all blocks of a relation is solved by *formal concept analysis* (Ganter and Wille, 1999). The definition of a block corresponds to a so-called *formal concept*. Concepts (and hence blocks) form a hierarchy which is defined by $(O_1, F_1) \leq (O_2, F_2) \Leftrightarrow O_1 \subseteq O_2$. Indeed, the hierarchy is a lattice (see Figure 4). This means, among other things, that any two blocks have a unique common sub block and any intersection of two blocks is a block in the hierarchy as well.

The call relation for the Ruby interpreter has 7280 blocks. Most blocks are small, as can be seen from the frequency distribution for block size ($|O| \times |F|$) in Figure 3. A bar in the diagram for size s represents the number of blocks whose size is in an interval of width 10 that is centered at s. There are 6430 blocks of size 20 or less and 88 blocks of size 100 or more. Likewise, 7043 patterns have a support of 20 or less and 24 patterns have support of 100 or more. We are most interested in large blocks that exceed a minimum support

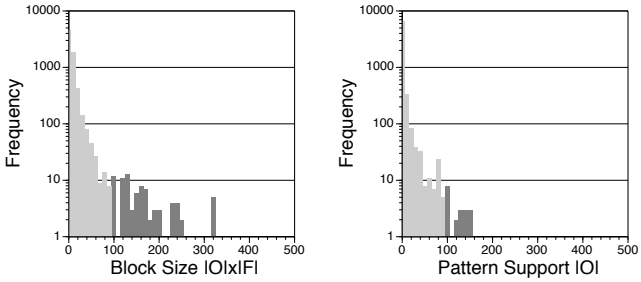

Figure 3: Distribution of block size $|O| \times |F|$ and pattern support $|O|$ in Ruby 1.8.4. From 7280 blocks, 88 blocks are of size 100 or bigger and 24 patterns have support 100 or higher.

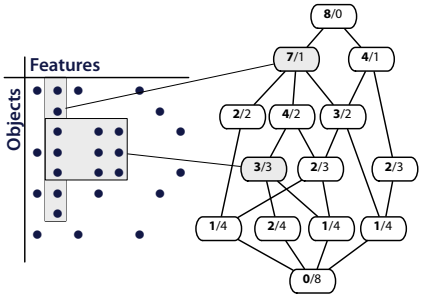

Figure 4: The blocks of a relation form a lattice. Each block corresponds to a formal concept—two such correspondences are shown. The numbers inside each concept denote $|O|/|F|$: support and width of a rule.

because they are likely to represent a regularity in the Ruby implementation.

A relation $R \subseteq \mathcal{O} \times \mathcal{F}$ may have up to 2^n blocks where $n = \min(|\mathcal{O}|, |\mathcal{F}|)$. The actual number of blocks strongly depends on the *density* $|R|/(|\mathcal{O}| \times |\mathcal{F}|)$ of the relation (or table). The exponential case only holds for extremely dense tables. The density of the call relation for Ruby (and other systems—see Table 2) is below one percent, which is why the number of blocks is typically dominated by $O(|R|^3)$.

Since we are most interested in the fraction of patterns (or blocks) with high support and large size, it would be wasteful to compute all blocks of a relation. The key observation for an efficient algorithm is that the blocks highest in the hierarchy exhibit the highest support (see Figure 4). In other words: as we move down in the hierarchy, support

	Call Relation			
Project	$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{F}\rvert$	Density	Blocks
Ruby 1.8.4	3502	1974	0.002	7280
Linux 2.6.0	11131	7176	< 0.001	11308
Python 2.4.3	2624	1627	0.002	4870
Lua 5.1	766	664	0.005	1523
Apache 2.2.2	2256	1576	0.002	3301

Table 2: Statistics for the call relation of open-source projects.

$|O|$ decreases monotonically while $|F|$ increases. The size $|O| \times |F|$ of blocks maximizes towards the middle of the hierarchy. They are interesting because they combine wide patterns that still have relatively high support.

3.1 Algorithm in a Nutshell

The best-known algorithm for concept analysis is by Ganter and Wille (1999); it computes efficiently the *set* of all concepts. However, it does not compute the lattice of concepts explicitly, nor does it work breadth-first. Taken together, this makes it less suitable for the exploration of only the topmost concepts in the lattice. We sketch a simple yet efficient algorithm below, more details can be found in Lindig (2000).

The top concept (or block) for a relation $R \subseteq \mathcal{O} \times \mathcal{F}$ is $(\{\}', \{\}'')$ and serves as a starting point. Given any concept (O, F), we can compute a sub concept (O_f, F_f) for each feature $f \in \mathcal{F} \setminus F$ that is not already part of (O, F): $(O_f, F_f) = ((F \cup \{f\})', (F \cup \{f\})'')$. This list of sub concepts contains all lower neighbors of (O, F) but may also contain additional concepts. The following criterion holds only for lower neighbors and is used to identify them: (O_f, F_f) is a lower neighbor if and only if for all $x \in F_f \setminus F$ the following holds: $(F \cup \{x\})'' = (F \cup \{f\})''$.

The above algorithm is implemented in Colibri/ML, a command-line tool for concept analysis (Lindig, 2007). It takes a textual representation of a relation and computes all blocks and block violations. As sketched above, Colibri/ML avoids computing all blocks by starting from the top block and then moving to lower blocks breadth-first while blocks still exceed a given minimum support.

Colibri/ML worked well for our cases studies (see Section 7 for its performance). For very large systems ($|\mathcal{O}| > 20\,000$) the more advanced algorithm by Stumme et al. (2002) could provide an alternative, as it is explicitly designed for extreme scalability.

Formal concept analysis computes all blocks from a relation.

4. VIOLATIONS

When a pattern is represented as a block, a violation of such a pattern is represented by an imperfect block. The initial example in Figure 1 shows such an imperfect block formed by pattern {`va_start`, `va_end`}, its instances, and one function that only calls `va_start`. Adding the missing call to `va_end` would remove the violation and make the block perfect.

A similar situation is shown more schematically on the left in Figure 5. Closer inspection reveals that an imperfect block is really a composition of two blocks. Block A represents a pattern; this pattern is violated by a (small) number of violators belonging to a subset of block B, where the patterns of A and B overlap. This is equivalent to B being a super block of A in the block hierarchy (shown on the right of Figure 5). Together they leave a *gap* in a block as wide as block A and as tall as block B. The width of the gap is the number of corrections necesary in any violator to remove the violation.

Just like not every block constitutes an interesting pattern that captures a universal quality, not every gap constitutes an interesting violation of a pattern. We are only interested in gaps within blocks that we already have found interesting.

Project	Supp.	Conf.	Violated Pattern
Linux 2.6.17	141	0.97	`mutex_lock`, `mutex_unlock`
Linux 2.6.16	48	0.98	`down_failed`, `up_wakeup`
Linux 2.6.0	44	0.96	`kmalloc`, `vmalloc`
Linux 2.6.0	68	0.99	`printk`, `dump_stack`
Python[1]	59	0.98	`PyEval_RestoreThread`, `PyEval_SaveThread`
Ruby[2]	24	0.96	`id_each`, `rb_block_call`

[1] SVN 2006-06-20 [2] CVS 2006-06-20

Table 3: Some pattern violations; the underlined call was missing.

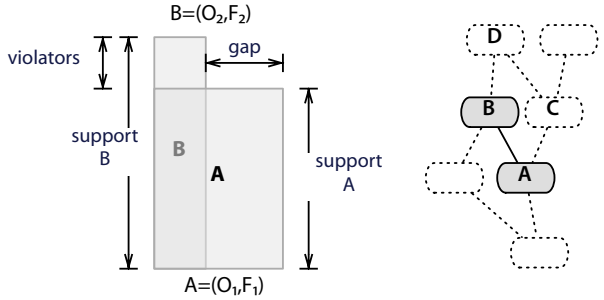

Figure 5: A pattern and its violation are represented by two blocks that are neighbors in the lattice: block A represents a pattern which is violated by block B. Our confidence that such a violation is genuine depends on the support of both blocks.

This typically means that we demand a minimum support for block A before we would consider it. In addition, we believe that fewer violations of a pattern make these violations more credible. This is expressed in the *confidence* for a violation.

Definition 3 (Violation, Confidence). *Given a pattern represented by block $A = (O_1, F_1)$ and a second block $B = (O_2, F_2)$ with $A < B$, the objects $O_2 \setminus O_1$ violate pattern F_1. The* confidence *that these violations are genuine is* $|O_1|/|O_2|$.

Confidence is the probability that any object that exhibits features F_1 also exhibits features F_2. A rule with a support of 100 instances and two violations yields a confidence of $100/102 = 0.98$. In the initial example from Ruby 1.8.4, rule {`va_start`, `va_end`} has support 17 and one violation. This results in a confidence of $17/18 = 0.94$. Table 3 shows some additional violated patterns from open-source projects.

A violation is a composition of two blocks.

5. FINDING VIOLATIONS

An imperfect block like on the left side of Figure 5 can be constructed from block A and any super block. In the partial block hierarchy on the right of Figure 5, these are blocks B, C, and D, as well as all their super blocks.

The violations of block A with the highest confidence are those represented by the upper neighbors of A in the block

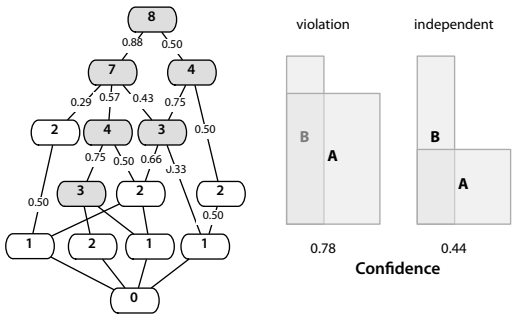

Figure 6: Block hierarchy for the example from Figure 1. Each block is marked with its support; shaded blocks have support of 3 or greater and edge labels indicate confidence for pattern violations.

hierarchy: blocks B and C in Figure 5. The reason is that, as we move up in the hierarchy, blocks become slimmer but taller. Since confidence essentially expresses the height ratio of two blocks and we are looking for blocks of almost equal height, *immediate neighbors represent pattern violations with the highest confidence.*

Figure 6 shows the block hierarchy from the example in Figure 1; the number inside each block indicates the support for the pattern represented by that block. Links between blocks represent violations—some are labeled with the confidence of the violation. As we have observed above, support decreases monotonically as we move down in the hierarchy. On the other hand, *confidence is non-monotonic.* There is no obvious algorithm to identify only the violations with the highest confidence.

A pragmatic approach to identify violations is to consider only those that violate patterns exceeding a minimum support. These are represented by the top blocks in a hierarchy; in Figure 6 all blocks with support of at least three are shaded. Traversing all edges of the lattice breadth-first, starting from the top element then will find all interesting violations. This is most efficient with an algorithm that computes blocks and their neighbors on demand, rather than all blocks beforehand.

Violations correspond to neighboring blocks.

6. TWO PATTERNS OR ONE VIOLATION?

The recursive nature of a block hierarchy causes a dilemma: whether a block is a pattern or contributes to the violation of another pattern is a matter of interpretation.

When two blocks A and B with $A < B$ have almost the same support, the confidence for a violation of B is close to one. This is the situation presented above in Figure 5 and in the middle of Figure 6. In that case we regard B as a block that contributes to a violation of block A.

An alternative situation is shown in Figure 6 on the right: two blocks A and B with $A < B$ where B has about twice the support of A. Considering B as violating A would result in a low confidence. It is thus more sensible to assume that A and B are overlapping but otherwise independent patterns. This would mean that both A and B represent a correct usage, even though one pattern (B) is a subset of the other (A).

	Patterns[1]		Violated Patterns[2]		
Project	#	Width	#	Violators	Gap
Ruby 1.8.4	143	2.67	39	1.49	2.26
Linux 2.6.0	112	2.52	19	1.21	1.05
Python 2.4.3	163	2.32	8	1.00	1.62
Lua 5.1	5	2.00	0	0.00	0.00
Apache 2.2.2	25	2.08	1	1.00	1.00

[1] with support ≥ 20 [2] with confidence ≥ 0.95

Table 5: Patterns and violations in the call relation of C Programs.

We analyzed the call relations of the projects in Table 2 for independent but overlapping patterns. We considered all patterns with support of at least 20 and a violation confidence below 60%. We found no such patterns in the Linux 2.6.0 kernel, none in Lua 5.1, one in Apache HTTP, but 59 such patterns in Python 2.4.3, and 49 in Ruby 1.8.4. For example, a typical pair of patterns in Python is {`PyType_IsSubtype`, `PyErr_SetString`, `PyErr_Format`} with support 42 and {`PyType_IsSubtype`, `PyErr_SetString`} with support 202. We have no immediate explanation why some systems show many overlapping patterns, while others show none at all. Both systems that show them are interpreters and we suspect that these include a considerable number of functions which call many functions such that overlapping patterns can emerge.

In addition to confidence, we may use a second criterion to classify two blocks as either independent or violating: the width of the gap (see Figure 5), which is the number of corrections needed to make an object an instance of the violated pattern. If a pattern has width 5, it is likely that a true error misses one call, rather than two or three. We thus could demand that a violation should have a small gap. As a consequence, we would only consider one block violating another if both blocks have about the same height *and* about the same width.

Using gap width to identify violations requires patterns of a considerable width. Otherwise gap width is too bound to be useful as a criterion. This is the case for patterns that we found in C programs, where most patterns have width two.

Table 5 presents some statistics for open-source projects to support this: columns under *Patterns* indicate the number of patterns with support of at least 20 and their average width. Columns under *Violated Patterns* indicate how often these were violated, by how many functions (column *Violators*), and the average number of missing calls (column *Gap*). Because the average gap width is between one and two, it cannot be used as a criterion to classify blocks as violations or patterns.

Patterns and violations are recursive.

7. PERFORMANCE

Thinking about patterns and their violations as a hierarchy of blocks is not just a theoretical model but is also well suited for an implementation. We outlined in Sections 3 and 4 efficient algorithms to compute all blocks and to find violations of patterns above a minimal support. Here we report some performance numbers gathered with COLIBRI/ML, a

Support	≥ 20				≥ 30				≥ 40			
Confidence	0.80	0.85	0.90	0.95	0.80	0.85	0.90	0.95	0.80	0.85	0.90	0.95
Ruby 1.8.4	10.8	10.7	9.9	10.7	9.2	8.3	9.1	8.4	8.3	7.6	8.3	7.6
Linux 2.6.0	68.9	73.4	68.7	73.4	50.7	55.1	55.1	50.7	43.4	47.6	43.4	46.8
Python 2.4.3	19.3	17.8	19.3	19.4	15.7	14.3	15.7	14.3	14.2	12.9	14.3	12.9
Lua 5.1	0.3	0.3	0.3	0.3	0.3	0.3	0.2	0.3	0.2	0.2	0.2	0.2
Apache 2.2.2	3.1	3.1	2.8	2.9	2.8	2.5	2.8	2.5	2.7	2.4	2.7	2.7

Table 4: Time in seconds to analyze call relations for pattern violations with Colibri/ML on a 2 GHz AMD-64.

command-line tool for concept analysis implemented in Objective Caml[2].

Our test subjects were the open-source applications written in C that we have used throughout the paper. These range from the small Lua interpreter (12 kLOC of C code[3]), over medium-sized systems like the Apache HTTP server (207 kLOC), the Python and Ruby interpreters (300 kLOC, 209 kLOC), to the Linux 2.6 kernel (3.6 MLOC). For the Linux kernel the actual size of the system depends strongly on the configuration because drivers can be included into the kernel, compiled into modules, or not be used at all. We configured a kernel for a small server where all relevant modules are integrated into the kernel.

From each application we extracted the call relation and analyzed it for violations of patterns. We extracted the call relation by building a binary of each application, disassembling it, and analyzing labels and jumps with a small script. This static analysis is fast but misses computed jumps and calls from function pointers. While these are rare in C applications, this simple technique would not work for C++ where methods are invoked via jump tables.

Table 4 reports wall clock times in seconds for the analysis of pattern violations. The analysis was run on a 2 GHz AMD-64 on Linux for several levels of minimum support and confidence. For example, analyzing Python for all violations with support of at least 20 instances and confidence ≥ 0.85 took 17.8 seconds. The analysis of the Linux kernel took about a minute, while smaller systems could be analyzed within less than 20 seconds. The analysis is faster for higher confidence and support levels because it must consider fewer blocks; however, the overall impact of these parameters are not prohibitive in any way. The memory requirement was about 100 MB for the analysis of Linux and 20 MB for the other systems.

Finding pattern violations is efficient.

8. ENCODING ORDER

Our analysis of the call relation is control-flow insensitive: all calls from a function are considered, ignoring their order and whether they are actually possible. This is a good fit for our framework because it is based on sets. However, we briefly like to demonstrate that a flow-sensitive analysis can be encoded as well by discussing the approach by Wasylkowski (2007). Using our framework, he discovered the previously unknown bug #165631 in the AspectJ compiler.

Wasylkowski observes statically for objects of a Java class C sequences of incoming method calls. The order of these calls is encoded as $C.a \prec C.b$, denoting "call $C.a$ *may* precede call $C.b$". Each observation is local to a method that uses an instance of C as a parameter or local variable. The result is a relation R over methods (that use class C) and pairs ($C.a \prec C.b$) of methods from class C. An analysis of this relation reveals methods that use class C in an unusual way: for instance, the bug found in AspectJ was detected because the buggy method never called `C.hasNext()` after calling `C.next()`, which was a common observation in other methods. Overall, he analyzed within minutes over 35 000 methods that used almost 3000 classes.

The example shows that sequences and graphs, which lend themself not to characterization by feature sets, may be analyzed for patterns using an appropriate encoding. This particular encoding, however, may grow exponentially and is thus best suited for small graphs or sequences. An alternative is an encoding $\prec_n$ that considers only nodes or events whose distance is bound by n.

Sequences may be encoded as relations to facilitate their analysis.

9. INLINING

When a function f calls `lock` but not `unlock` this is not necessarily an error: f may call g, which in turn calls `unlock`. Hence, f calls `unlock` *indirectly*, as sketched in Figure 7. Both f and g violate pattern {`open`, `close`} but we can avoid this false positive by applying *inlining*. Inlining works for any relation $R \subseteq \mathcal{O} \times \mathcal{F}$ where $\mathcal{O} = \mathcal{F}$ holds, like in a call relation. Li and Zhou (2005) explain inlining in terms of data flow analysis; we provide here an alternative explanation that solely works on the input relation.

Inlining derives from an existing relation $R^0 \subseteq X \times X$ a new relation $R^1 \supseteq R^0$ according to the following rules:

$$\begin{aligned} (f,g) \in R^0 &\Rightarrow (f,g) \in R^1 \\ (f,g) \in R^0 \wedge (g,h) \in R^0 &\Rightarrow (f,h) \in R^1 \end{aligned}$$

In the derived relation R^1 a function f is related with the function it calls directly (g), as well as those that it calls indirectly (h) through *one* intermediate function. Inlining may be repeated to capture indirect calls through two intermediate functions, and so on, to account for even more indirect calls.

So far we have fixed f by attributing `close` to it, but have not fixed g yet by attributing `open` to it. This can be easily expressed using the prime operator:

$$(f,g) \in R^0 \Rightarrow (g,x) \in R^1 \quad \text{for } x \in \{g\}''$$

[2]Colibri/Java is forthcoming (Götzmann, 2007)

[3]as reported by David A. Wheeler's SLOCCount

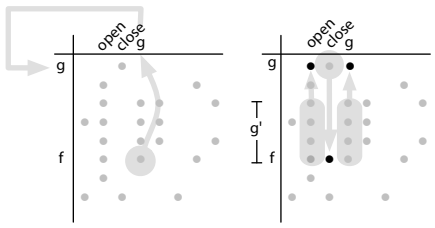

Figure 7: Function f calls `open` directly, but `close` indirectly through g. Inlining attributes the indirect calls of `close` to f. Likewise, all functions that are called by *all* callers of g are attributed to g as well.

Object set g' is the set of all functions calling g and g'' is the set of all functions called by all callers of g. These are attributed to g in R^1.

> **Inlining can be expressed solely on the input relation.**

10. RELATED WORK

There are many ways to find software defects. The best way is by checking a program against an *independent* specification or test. A failing test then can be used to locate the defect automatically (Cleve and Zeller, 2005). We are focusing on a scenario without such external references. Instead, we aim to identify intrinsic *patterns* in a software system's implementation or behavior and *deviations* from these patterns. Such deviations are then suggested as potential defects.

10.1 Mining Patterns

Mining patterns from programs for program understanding, specification, or documentation has inspired many researchers, especially in the domain of temporal behavior. The following approaches *do not* develop a notion of deviation and therefore are only interesting in so far as they could provide relations that could be mined for deviations.

Finite Automata. Cook and Wolf (1998) have written the seminal work about learning *finite-state machines* from event sequences. ADABU by Dallmeier et al. (2006) dynamically mines automata from Java classes that characterize the state of objects as seen through observer methods provided by their interface. A similar approach is by Xie and Notkin (2004) where object state is observed trough the return values of methods. This leads to more detailed but also less general automata. In constrast to these dynamic approaches, Henzinger et al. (2005) learn *permissive interfaces* by repeatedly generating candidate automata that capture legal method sequences and checking them against an abstract program interpretation. While elegant, it works only for a subset of Java.

Dynamic Invariants. Dynamic invariants, as conceived by Ernst et al. (2001) and mined with DAIKON, represent logical relations between data that held during test executions. Observed relations like $a < b$ are suggested as program invariants. DAIKON works by checking a list of fixed relations between pairs of variables and field and thus cannot infer new invariants. However, by checking a long list of relations between many pairs, a considerable variety of patterns can be mined. A simpler variation of DAIKON was proposed by Hangal and Lam (2002).

10.2 Mining Violations

The most formal and well established systems for the notion of consistency in software are type systems, and type inference in particular (Pierce, 2002). Undoubtedly, they prevent the introduction of bugs on a routine basis. However, advances in type theory only benefit future programming languages and type systems of existing languages are often too weak to express consistency. Hence, there is a strong interest in mining patterns and violations in *existing software* with the goal to identify defects.

Sets of Sequences. Hofmeyr et al. (1998) observe sequences of system calls for intrusion detection. Normal behavior is characterized by a set of short overlapping sequences. Abnormal behavior is detected when unknown sequences occur. This approach was refined by Dallmeier et al. (2005) for defect localization in Java programs: the AMPLE tool compares sequences of method calls issued by objects across passing and failing test cases. The class that shows the largest deviation in behavior between passing and failing test cases is suggested as a culprit. Hence, this violation does not imply a detaild fix, unlike the method proposed here.

Cluster Analysis. Dickinson et al. (2001) employ cluster analysis to separate normal program traces from traces triggering bugs. While this can capture a very wide range of behavioral patterns, cluster analysis has very little explanatory power, unlike the patterns and violations we propose here.

Mining Correlations. Liblit et al. (2005) mine violations in an abstract sense. They observe a statistical correlation of program failure with return values of functions (which are then used in control-flow statements) and predicates. This correlation has high explanatory power but depends on a high number of varying program executions to develop a statistical notion of normal behavior.

Pairs of Functions. Weimer and Necula (2005) learn pairs of function calls (like `open`/`close`) from program traces. They look specifically for violations of these pattern in error-handling code that misses the call to the second function. They detect a considerable number of defects. The paper is also remarkable for its comparison with other defect localization approaches. Conceptually, this approach learns purely structural patterns and does not depend on prior knowledge—like us. Unlike us, patterns are ordered pairs whereas we consider unordered sets of any size.

Checking known Patterns. Engler et al. (2001) coined the slogan of bugs as deviant behavior and introduced a tool that searches for bug patterns. Each instance of such a pattern expresses an inconsistent view of the state of a program and hence a likely bug. The difference to our formalism is that Engler et al. can only detect known patterns of inconsistency but not find new ones. On the other hand, searching for known patterns results in high precision.

Mining Version History. Livshits and Zimmermann (2005) mine patterns from the development history which

they represent as a sequence of transactions that add new function calls. For each transaction they mine (using frequent itemset mining) usage patterns of method calls being added together. These patterns are presented to the user for confirmation as being correct; based on them, dynamic tests search for violations of these patterns. The static mining step for patterns is similar to our approach (and could have used it), whereas violation detection is done dynamically using program instrumentation and test cases. The detection of test cases is limited to pairs of functions whereas we can detect violations of any pattern.

10.3 PR-Miner

PR-Miner by Li and Zhou (2005) inspired us to propose concept analysis as a better foundation for their analysis that identifies purely structural sets of features and their violations. PR-Miner is based on frequent itemset mining and mines *closed* feature sets. A violation (called a rule) is represented as an implication $A \Rightarrow B$ where A and B are closed feature sets.

A closed itemset corresponds to a pattern in our formalism and also to a block, which has the additional benefit that it includes the instances of the pattern. A rule corresponds to neighboring blocks in our formalism, again with the benefit of also representing all instances and thus making theory and implementation more uniform. The notion of confidence in both formalisms is equivalent.

The short characterization of PR-Miner above might suggest that blocks and formal concepts provide no added benefit. However, we like to argue that a precise understanding of what PR-Miner does is greatly enhanced by the theory of formal concept analysis. This seems evident both from our simpler and shorter explanation of mining, algorithms, as well as the discussion of the block hierarchy and its size. Combining patterns and instances into blocks gives access to a rich algebra and intuitive geometric interpretation which simply does not exist for closed itemsets in isolation.

Li and Zhou report impressive performance numbers using an off-the-shelf implementation for frequent itemset mining. They clearly benefit from years of development of these tools in the data-mining community. However, we believe that the performance of Colibri/ML provides viable alternative for practical problems.

PR-Miner implements some data flow analysis to minimize false positives. It is based on the insight that a pattern $\{a, b\}$ might be implemented not just by calling a and b directly, but by calling a and c, which in turn calls b. This analysis is independent of the mining and can be implemented for either system. Indeed, what is expressed as a data flow analysis by Li and Zhou (2005) can be also expressed as operation on the input relation R as in Section **??**.

PR-Miner analyzes variable declarations in addition to function calls. Again, this is not inherent to the mining and can be implemented for any system by making these part of the input relation.

11. CONCLUSIONS

Formal concept analysis provides a practical and theoretical framework to identify structural patterns and violations in binary relations. The analysis assumes no *a priori* knowledge like names or pre-defined patterns—unlike many previous approaches. Pattern violations have been shown to correlate with bugs in software systems. The main benefit over classical frequent itemset mining (Agrawal and Srikant, 1994) is that blocks (or concepts) unify a pattern and its instances. Together they form a rich and well-studied algebra; furthermore, they offer a geometric interpretation which provides intuition: violations correspond to imperfect blocks in a cross table.

A relation (like a call relation) induces a block hierarchy. Each block corresponds to a pattern and neighboring blocks correspond either to independent patterns, or a violation—depending on the associated confidence. This is the main conceptual result of this paper.

Formal concept analysis gives us complexity results for pattern mining: the number of blocks (or patterns) induced by a relation may grow exponentially. This happens only for dense relations; call relations, at least, tend to be sparse. On top of that, only a small fraction of blocks exceeding a minimum support are of interest and can be computed efficiently using an implementation that we provide (Lindig, 2007).

Algorithms for formal concept analysis are practical although they lack the performance tuning that went into algorithms and implementations for frequent itemset mining (Hipp et al., 2000). We provide an open-source implementation Colibri/ML that was able to analyze the call relation of the Linux kernel within one minute, and smaller systems like the Python interpreter in under twenty seconds.

Earlier work on detecting anomalies often had a special focus on pairs of function calls (Weimer and Necula, 2005; Yang and Evans, 2004), rather than the more general patterns we studied. However, we found that most call patterns in open-source projects implemented in C have width between two and three (see Table 5). This is *a posteriori* a justification for the special interest in such pairs.

The starting point of our analysis is a binary relation. This implies that we analyze *sets* of features related with objects. This seeming limitation may be overcome using clever encodings, as demonstrated by Wasylkowski (2007). We are also encouraged by the success of code query languages like CodeQuest (Hajiyev et al., 2006), which represent software at their core using relations. By extending them with our analysis would become possible to mine many more source code relations for patterns and violations.

Future Work. Patterns and violations as we mine them do not capture intrinsically the notion of program execution. We like to take advantage of this for providing better support of non-executable code. By this we mean configuration files for services like mail, HTTP, or firewalls. They control security-critical applications but almost no support for them exists beyond syntax checkers and syntax highlighting. Patterns can capture best practices that can be learned from existing configuration files. For example, a pattern may represent a combination of flags in a firewall rule. A system administrator could be warned before deploying an unusal flag combination in a firewall rule. We believe that this kind of support could be provided by editors on a routine basis as they do it today for syntax highlighting.

All in all, we have shown that formal concept analysis provides an appealing theoretical and practical framework to identify structural patterns and their violations.

Acknowledgements. Discussions with Silvia Breu, David Schuler, and Valentin Dallmeier improved this paper.

References

R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *20th Intern. Conf. on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann Publishers, 1994.

Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. 27th Intern. Conf. of Software Engineering (ICSE 2005)*, St. Louis, USA, 2005. To appear.

J. Cook and A. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew Black, editor, *European Conf. on Object-Oriented Programming (ECOOP)*, pages 528–550, 2005.

Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *Proc. of the 2006 Intern. Workshop on Dynamic System Analysis (WODA)*, pages 17–24. ACM Press, 2006.

William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. of the 23rd Intern. Conf. on Software Engineering, ICSE 2001*, pages 339–348. IEEE Computer Society, May 2001.

Dawson Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 57–72, New York, October 21–24 2001. ACM Press.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin – Heidelberg – New York, 1999.

Daniel Götzmann. Formal concept analysis in Java. Bachelor thesis, Saarland University, Computer Science Department, 2007.

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. codequest: Scalable source code queries with datalog. In *European Conf. on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th Intern. Conf. on Software Engineering (ICSE-02)*, pages 291–301, New York, May 19–25 2002. ACM Press.

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proc. of the 10th European Software Engineering Conf. ESEC/SIGSOFT FSE*, pages 31–40. ACM, 2005.

Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining—A general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.

Steven A. Hofmeyr, Stephanie Forrest, and Somayaji Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of the 10th European Software Engineering Conf. ESEC/SIGSOFT FSE*, pages 306–315. ACM, September 2005.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 15–26, June 2005.

Christian Lindig. Colibri/ML. `http//code.google.com/p/colibri-ml/`, 2007. Open-source tool for concept analysis, implements algorithm from Lindig (2000).

Christian Lindig. Fast concept analysis. In Gerhard Stumme, editor, *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 152–161, Aachen, Germany, August 2000. Shaker Verlag.

V. Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proc. of the 10th European Software Engineering Conf. ESEC/SIGSOFT FSE*, pages 296–305. ACM, 2005.

Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002.

Gerd Stumme, Rafik Taouil, Yves Bastide, Nicolas Pasquier, and Lotfi Lakhal. Computing iceberg concept lattices with Titanic. *Data Knowl. Eng*, 42(2):189–222, 2002.

Andrzej Wasylkowski. Mining object usage models (doctoral symposium). In *Proc. of the 29th Intern. Conf. on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA, May 2007.

Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 461–476. Springer, 2005.

Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. of the 6th Intern. Conf. on Formal Engineering Methods (ICFEM 2004)*, pages 290–305, November 2004.

Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. In *Intern. Symposium on Software Reliability Engineering*, pages 340–351. IEEE Computer Society, 2004.