# Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection

Natalie Gruska*
School of Computing, Queen's University
Kingston, Ontario, Canada
gruska@cs.queensu.ca

Andrzej Wasylkowski · Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
{wasylkowski, zeller}@cs.uni-saarland.de

## ABSTRACT

Real production code contains lots of knowledge—on the domain, on the architecture, and on the environment. How can we leverage this knowledge in new projects? Using a novel lightweight source code parser, we have mined more than 6,000 open source Linux projects (totaling 200,000,000 lines of code) to obtain 16,000,000 *temporal properties* reflecting normal interface usage. New projects can be checked against these rules to detect *anomalies*—that is, code that deviates from the wisdom of the crowds. In a sample of 20 projects, ∼25% of the top-ranked anomalies uncovered actual code smells or defects.

## Categories and Subject Descriptors

D.2.1 [**Software**]: Software Engineering—*Software/Program Verification, Requirements/Specifications*; D.3.4 [**Software**]: Programming Languages—*Processors*

## General Terms

Design, Experimentation, Languages, Verification

## Keywords

lightweight parsing, language independent parsing, mining specifications, temporal properties, formal concept analysis

## 1. INTRODUCTION

When interacting with existing code, programmers must follow the underlying interaction assumptions. These assumptions can come in the form of *formal rules,* such as function signatures, argument types, or explicit preconditions. Many assumptions, though, are informal, or even implicit; and thus, anomalies only show up at run time.

In earlier work [19,18], we have worked on extracting such assumptions from existing code. By observing how specific interfaces were normally used in a project, we would detect *anomalies*—that is, deviations from normal interface usage. While the approach was successful, it required usage examples to learn from—and a single project would not always contain a sufficient number of such examples. Would it be possible to learn from existing projects to detect anomalies in other projects? In this paper, we show that such a cross-project anomaly detection is indeed feasible. We have automatically mined usage examples from over 6,000 open source Linux projects, totaling more than 200 million lines of code. The resulting knowledge base contains 16 million temporal properties describing all interfaces used in these projects. This database can be used to detect anomalies—that is, code that deviates from the wisdom of the crowds.

Figure 1 shows such an anomaly in conspire-0.20, an IRC client[1].

```
dcc->sok = socket (..., ..., ...);
if (...) {
    while (...)
        bind (dcc->sok, ..., ...);
    setsockopt (dcc->sok, ..., SO_REUSEADDR, ..., ...);
}
listen (dcc->sok, ...);
```

**Figure 1: A defect: `setsockopt` has no effect.**

This code, which sets up a network connection using sockets, is not only an anomaly, but also wrong. The problem with this code is that it attempts (in a call to `setsockopt`) to allow the socket to be bound to the address that is already in use *after* attempting to bind the socket. This mistake is easy to make, because most of the `SO_*` options that can be passed to `setsockopt` are effective even after the call to `bind`. Our approach flags this as an anomaly because almost all other projects using `bind`, `setsockopt`, and `listen` call `setsockopt` before `bind`. Moreover, this defect cannot be detected by single-project anomaly detection: the violated pattern does not occur in conspire-0.20 at all.

Problems like this can only be discovered by comparing interface usage against normal usage—and this "normal" usage is precisely what we extract from thousands of projects. Our approach uses two major building blocks:

1. We introduce a *lightweight, language-independent parser* that is able to perform analysis of programs written in languages such as C, C++, Java, PHP, and others with a similar syntax. Figure 2 shows the individual steps of the parser, detailed in Section 2. To the

*Natalie Gruska was with Saarland University while carrying out this work.

[1]http://confluence.atheme.org/display/CON


*ISSTA'10,* July 12–16, 2010, Trento, Italy.

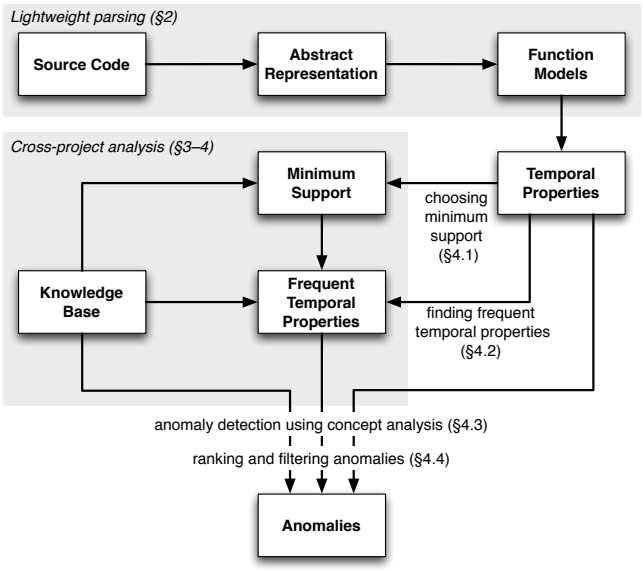


**Figure 2: From source code to anomalies: A lightweight parser extracts temporal properties from source code, which are then checked against the knowledge base to detect anomalies.**

best of our knowledge, this is the first parser that is lightweight, language-independent, and at the same time accurate enough to actually allow detecting defects in source code. Its usage makes our approach directly applicable to programs written in multiple languages and lets us successfully deal with problems associated with analyzing source code statically.

2. We leverage the JADET [19] approach to detect object usage anomalies, extending it to arbitrary languages with function calls. We demonstrate its scalability by learning a knowledge base (Section 3) from more than 6,000 C projects; and showing how new projects can be efficiently checked for anomalies (Section 4); again, Figure 2 shows the individual steps detailed in these Sections. To the best of our knowledge, this is the first time that interface usage has been mined across projects and leveraged at such a massive scale.

The evaluation results are very promising. In a sample of 20 projects, ∼25% of the top-ranked anomalies uncovered actual code smells or defects (Section 5); in less mature code, we would expect an even higher rate of defects and anomalies. After discussing the related work (Section 6), we present *checkmycode.org*, a public Web site where programmers can have their code checked against the wisdom of the crowds. Section 7 closes with conclusion and consequences.

## 2. LIGHTWEIGHT PARSING

Our first challenge was to analyze large amounts of source code. As other researchers before us [2], we found that source code retrieved from code archives or web repositories can be hard to analyze statically: The code may go through various preprocessing steps (such as macros expansion); and references across projects or packages are hard to resolve. We therefore designed and implemented a *lightweight parser* to extract information about function calls and code structure from source code. Since we are only interested in discovering function calls and structure, many of the details present in source code can be disregarded. This is what makes our parser lightweight: it only parses selected parts of source code and disregards irrelevant statements.

```
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,SQL_S,SO_RA,&len,sizeof(len));
   }
   listen(dcc->sok, 1);
   upnp_add_redir(inet_ntoa(addr), dcc->port);
}
```

**Figure 3: Sample function from conspire-0.20.**

```
FUNCTION dcc_listen_init (2) {
   dcc.sok : socket(AF_INET, S_STREAM, CONST);
   IF () {
      LOOP (ntohs(SAdr.sin_port)) {
         bind(dcc.sok, &SAdr, sizeof(SAdr));
      }
      setsockopt(dcc.sok,SQL_S,SO_RA,&len,sizeof(len));
   }
   listen(dcc.sok, CONST);
   upnp_add_redir(inet_ntoa(addr), dcc.port);
}
```

**Figure 4: Information extracted by the parser from the source code above.**

As an example, consider the source code fragment in Figure 3. Receiving this fragment as input, our parser extracts information about function calls and code structure. The extracted information is shown in Figure 4. The basic structure of the function, the `if`-statement and the `while`-loop are retained, as are all the function calls. However, instructions not containing function calls are unimportant for our purposes and therefore disregarded; this includes variable declarations such as `socklen_t len;` as well as the `if` condition which does not contain a function call. We use a specially designed *generic abstract representation* to store the extracted information. This representation is similar to an abstract syntax tree but differs in that it does not store all information that was present in the source code and it is not bound to a specific programming language.

Retaining code structure is important because it allows us to partially reconstruct control flow information with which we can determine the order in which function calls are made. This information is critical to the creation of *function models* which will be discussed in Section 2.4.

Since many programming languages share very similar syntax for function calls and basic structure (e.g. loops and `if`-statements), it is possible for our parser to work in a language-independent fashion. Our parser is not bound to a specific programming language but rather to a subset of syntax that many programming languages (such as C and Java) share. This lack of direct focus makes the parser applicable to programs in all languages with syntax similar to the ones mentioned above.

The parsing process can be broken down into three main steps: creating tokens, identifying structure and extracting function calls. *Creating tokens,* as its name suggests, takes source code, filters it and separates it into tokens. In the *structure identification* step, tokens are classified as certain types of statements; the final *extract calls* step performs the crucial identification and extraction of function calls. The three steps are discussed in more detail in the following sections. Full details about the implementation of the parser can be found elsewhere [6].

## 2.1 Creating Tokens

Our parser receives source code as input and starts with preliminary filtering, such as removing comments and replacing line breaks with spaces. The remaining code is separated into tokens based on the following delimiters: "{", "}", and ";". Our choice of delimiters was motivated by the syntax that the parser is based on: curly braces ("{}") to indicate blocks and semicolons (";") to indicate the end of an instruction. Thus, each token contains exactly one instruction or the beginning or end of a block[2]. For instance, during this step, the source code shown in Figure 3 is separated into 14 tokens—in this case each line is a token.

## 2.2 Identifying Structure

It is important for our parser to recognize code structure so that the majority of control flow information can be reconstructed. After the source code has been broken down into tokens, each token goes through a *classification process.* The purpose of this process is to determine the kind of statement that the token contains. This process consists of matching the token against several regular expressions constructed to match a specific type of statement: class definition, function definition, loop, `if`-statement, or `switch`-statement. Whichever regular expression the token matches it is treated as that kind of statement. These are the only types of structural statements the parser is able to recognize, which means that in some cases the extracted code structure may not fully reflect the actual code structure.

Note that most of the time a token will not contain an entire statement, only the part of the statement up to the first delimiter. For instance the token `if (send_port > 0){` from the source code in Figure 3 is classified as an `if`-statement; however, this token does not contain the whole `if`-statement, as the body of the statement is missing. To obtain the entire statement, more tokens are parsed until one that indicates the end of the statement is found. In our example, the body of the `if`-statement is identified by continually parsing more tokens until the token containing the matching closing curly brace ("}") is found. As they are parsed, the tokens representing the body of the statement recursively undergo the structure identification process. Through this recursion, we identify nested structures such as the `while`-loop and its body.

If a token is part of the body of a statement, but is not identified as a structure statement, it is assumed to be an *expression.* For instance, the token `listen(dcc->sok, 1);` is part of the function body in Figure 3 but is not a structure statement. Expressions are important because these are the parts of the source code that contain function calls. The extraction of function calls from these tokens is discussed in Section 2.3.

[2]Note that this is different than in conventional parsers.

In the case that a token is identified as a loop or as an `if`-statement (e.g. `if (send_port > 0)`), its condition is extracted and treated as a separate token (e.g. `send_port > 0`), which is considered an expression.

## 2.3 Extracting Function Calls

Function calls can be present in expressions (see Section 2.2) and are identified on a purely syntactic level. This means that when parsing an expression token the parser searches for the syntax of a function call: an identifier followed by a set of parentheses (`id(...)`). If such syntax is found, it is assumed to be a function call: no further analysis of whether such a function with the given parameters and name exists is performed. Once a function call is identified, our parser extracts the name of the function that is called, the arguments and, if applicable, the target. Each argument of the function call is then considered an independent token and treated as an expression.

Consider the following token that is classified as an expression:

```
upnp_ad_redir(inet_ntoa(addr), dcc->port);
```

The function call contained in this token is recognized by the parser and the function name `upnp_add_redir` and the arguments `inet_ntoa(addr)` and `dcc->port` are extracted. Then each argument is regarded as a separate token and parsed. The token `inet_ntoa(addr)` is also identified as a function call and its name `inet_ntoa` and arguments are extracted. The token `dcc->port` does not contain a function call and is marked as an identifier.

For the sake of conformity, all instances of the arrow operator ("->") in an identifier are replaced with the dot operator (".") This causes some inaccuracies since in C and C++. "->" is a shortcut for dereferencing and accessing a member. However, in other languages such as PHP this operator is a simple member access. It is not possible to distinguish these two cases in a language-independent fashion. Therefore, we perform additional filtering (see Section 2.4) and do not differentiate between referenced and dereferenced objects.

## 2.4 Function Models

Once the source code has been parsed and all relevant information has been stored in the abstract representation, function models are created. A function model represents all possible sequences of function calls in a specific function.

When creating function models, we assume that the arguments of calls as well as the arguments of binary operators are evaluated from left to right. This is only of importance if multiple arguments of a function call are return values of function calls because it then influences the order in which these functions are called.

For each function that was identified during the parsing process and stored in the abstract representation, one function model is created. The states of a function model represent locations in the code, and the transitions are labelled with *events.* Each event represents a function call; it stores the name of the function that is called—annotated with the number of parameters the function takes—and a list of all objects associated with the call. Objects associated with a function call are its arguments, return value, and if applicable, the target.[3]

[3]There is a risk of the parser confounding functions with the same name and number of arguments, but different types.

```
static void glade_editor_reset_toggled (
        GtkCellRendererToggle *cell, gchar *path_str,
        GtkTreeModel *model) {
    GtkTreePath *path =
        gtk_tree_path_new_from_string (path_str);
    GtkTreeIter  iter;
    gboolean     enabled;

    /* get toggled iter */
    gtk_tree_model_get_iter (model, &iter, path);
    gtk_tree_model_get (model,
        &iter, COLUMN_ENABLED, &nabled, -1);
    gtk_tree_store_set (GTK_TREE_STORE (model),
        &iter, COLUMN_ENABLED, !enabled, -1);
    gtk_tree_path_free (path);
}
```

**Figure 5: Sample function.**

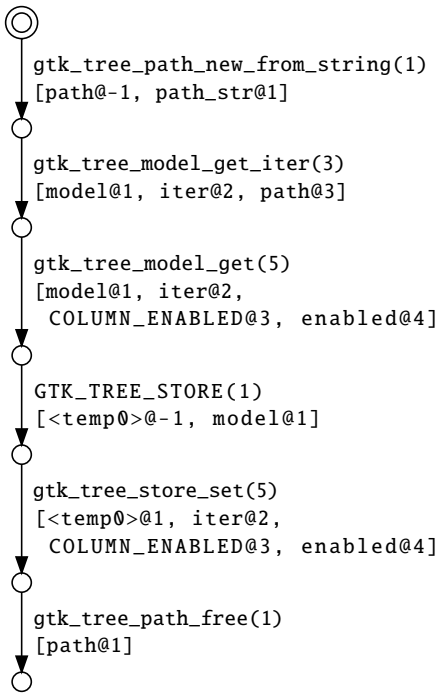


**Figure 6: Function model created by the parser from the function in Figure 5.**

Furthermore, each object is expressed through an identifier and annotated with information about how it is associated with the function call. An *annotation* is an "@" character followed by a number. This number indicates the relationship between function call and object. For a return value this number is -1, for a target 0, and for an argument, it is the number of the argument—the leftmost argument has number 1.

As a result of the conflicting semantics for the "->" operator discussed in Section 2.3 all instances of "*" and "&" are filtered from the identifiers. Hence we do not distinguish between a referenced and dereferenced object.

Consider the following line of code taken from the function presented in Figure 5:

```
GtkTreePath *path =
    gtk_tree_path_new_from_string (path_str);
```

We create an event for the function call `gtk_tree_path_new_from_string`. Since the function that is called takes one argument its name is annotated with "(1)". The list of associated objects contains the argument `path_str` and the return value `path`. Since `path_str` is the first—in this case the only—argument of the function call it is annotated with "@1", and since `path` is the return value of the function call it is annotated with "@-1". The event then looks as follows:

```
gtk_tree_path_new_from_string(1)
[path@-1, path_str@1]
```

The function model for the code in Figure 5 is presented in Figure 6. It shows the order, in which the function calls are made, and how they are related in terms of objects. For instance, the function `gtk_tree_path_new_from_string` is called before `gtk_tree_model_get_iter`. Notice the use of a variable named `<temp0>` that appears in the function model but is not present in the source code. This variable name is explicitly introduced to represent the return value of the call to `GTK_TREE_STORE(model)`. In the source code, this return value is not assigned anywhere but is immediately passed to the function `gtk_tree_store_set`. In cases like this it is necessary to introduce temporary variable names such as `<temp0>` to be able to express the fact that the return value of one function call is used as an argument for another.

# 3. THE KNOWLEDGE BASE

The capability to extract function models from massive amounts of source code allows us to create a *knowledge base* containing data about all the projects we want to learn from. Information from this knowledge base will then be used for cross-project anomaly detection (see Section 4). We store two types of information for each *reference project*: general information about the project—its *name*, and names and locations of *functions* that are part of that project; and interface usage information that will be of direct use when detecting anomalies—the set of *events* and *temporal properties* [19] that occur in the reference project's functions' models. Analysis of a single project consists of two steps: (1) applying the lightweight parser to extract function models from the project; and (2) extracting temporal properties from function models. The extracted data is then stored in the knowledge base.

## 3.1 Applying the Lightweight Parser

In the first step, we invoke the lightweight parser on every source file of the project to be analyzed. As a result of this analysis, we get a set of function models, one for each function identified by the parser. Information about the project and the functions found is inserted directly into the knowledge base.

It is possible to create only one knowledge base that will contain information about projects written in multiple programming languages. However, cross-language cross-project information does not seem nearly as useful as single-language cross-project information, so it makes more sense to create a separate knowledge base for each programming language of interest. Therefore, it suffices to invoke the parser on files that contain only source code written in this language (e.g., `.c` files for C).

In C, such usage is very rare; in other languages, we would still expect all instances of the same name to follow common usage rules.

## 3.2 Extracting Temporal Properties

In the second step, we transform each function model into a set of *temporal properties* that represent the flow of values between function calls. For this purpose, we leverage the notion of a temporal property from the JADET tool [19]. In JADET, temporal properties represent a pair of events that occur in a specific order (such as `lock` $\prec$ `unlock` meaning that a call to `lock` occurs before a call to `unlock`). We extend this definition to include information about the argument position in our events (e.g., the same value is first passed as the *first argument* to `lock` and then as the *first argument* to `unlock`). Thus, instead of only being able to express the ordering of function calls, we can also tell which argument is responsible for this ordering.

Let us consider the sample function shown in Figure 5. Its function model is shown in Figure 6. Because the model contains dataflow information (such as the fact that the return value of the call to `gtk_tree_path_new_from_string` was later used as the third argument of the call to `gtk_tree_model_get_iter`), we can extract temporal properties that will capture the order, in which values flow through function calls. One temporal property that we can extract is `gtk_tree_model_get_iter(3)@3` $\prec$ `gtk_tree_path_free(1)@1`, because the variable called `path` is passed as an argument to both those calls, and it is not redefined between them. Another is `GTK_TREE_STORE(1)@-1` $\prec$ `gtk_tree_store_set(5)@1`, because there is an unnamed value which, upon being returned as a return value of the first one of those calls, was passed as the first argument to the second one of those calls. Extracted temporal properties are inserted into the database, just like the events that form these properties.

It is important to note that it is not enough that a function call may occur before another function call for a temporal property to be constructed. As an example, consider the two calls to `gtk_tree_ store_set` and `gtk_tree_path_free` in the function model in Figure 6. These two functions are not directly related, and we can discover this fact by noticing that there is no variable that would be associated with both of them. Therefore, no temporal property gets constructed based on these two calls. In addition, a light dataflow analysis detects reassignments between function calls.

# 4. ANOMALY DETECTION

Once we have a knowledge base available, we can detect anomalies in a single project by comparing its interface usage with that of reference projects stored in the knowledge base. Figure 2 contains an overview of our cross-project anomaly detection approach. We combine the novel cross-project analysis (highlighted in Figure 2 by a grey background) with the anomaly detection mechanism used by JADET to achieve cross-project anomaly detection capabilities. The whole process consists of the following steps:

**Parsing.** The program $P$ to be analyzed is parsed by the lightweight parser resulting in function models being created. The process used here is the same as the one used when constructing the knowledge base (see Sections 2 and 3.1).

**Extracting temporal properties.** We extract temporal properties from the function models. The process used here is the same as the one used when constructing the reference database (see Section 3.2).

**Choosing minimum support.** The minimum support value to be used for anomaly detection is determined (see Section 4.1).

**Finding frequent temporal properties.** We find temporal properties that are related to the events occurring in $P$, and at the same time occur frequently enough in the reference projects (see Section 4.2).

**Anomaly detection.** We look for anomalies by applying concept analysis to the data we have both gathered from $P$ and extracted from the reference projects (see Section 4.3).

**Ranking and filtering.** We rank and filter anomalies before presenting them to the user (see Section 4.4).

## 4.1 Choosing Minimum Support

Minimum support is the number of functions that must adhere to a certain pattern (understood as a set of temporal properties) for this pattern to be acknowledged. For typical single-project anomaly detection this value is constant, but for cross-project anomaly detection choosing a constant value results in too much reference data (and thus loss of scalability) for larger projects. We deal with this problem by having a fixed constant range, in which the minimum support must lie, and applying binary search to find the lowest minimum support value that is in that range and at the same time keeps the amount of reference data[4] below a fixed threshold. The important thing here is that the threshold can be changed at will, so that our approach can actually use higher threshold (i.e., lower minimum support values) if the user is willing to pay the price in memory and time that will be spent on analysis. In practice, the higher the threshold value the better, and it should be chosen so as to maximize the amount of memory that is used by the process without causing swapping.

## 4.2 Finding Frequent Temporal Properties

To be able to do cross-project analysis, we need to take into account the temporal properties that are stored in the knowledge base. However, it does not make sense to consider all of them, because not all of them are useful when looking for anomalies in $P$. If some of them express relationships between functions that are part of a certain API that $P$ does not use, they are not going to be of any use for finding anomalies. Also, including them would make the whole approach infeasible, or at the very least applicable only in cases where the number of reference projects would be relatively small. Therefore, we limit ourselves to only those temporal properties that pertain to $P$.

We say that a temporal property *pertains* to $P$ if at least one of the events it consists of occurs in $P$. Such temporal properties are of the form $a \prec b$, where either $a$, $b$, or both $a$ and $b$, are events that occur in $P$. We would like to emphasize the fact that this rather conservative approach allows the analysis to make use of temporal properties that do *not* occur in $P$. This very important property allows us to find cases where the anomaly is caused not only by wrong ordering of function calls, but also by absence of a certain call or a set of calls.

However, not all temporal properties that pertain to $P$ are useful. If some of them occur in only a handful of functions

[4]expressed as the size of the final concept analysis matrix

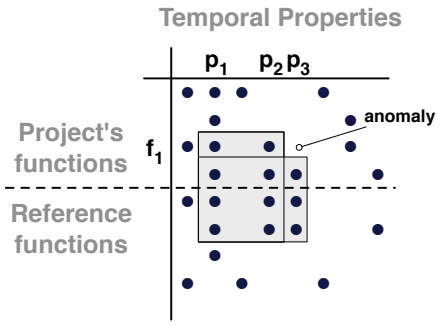


**Figure 7: Detecting anomalies via concept analysis. Each rectangle corresponds to a pattern; gaps indicate potential anomalies [10]. The matrix contains entries for functions of the analyzed project as well as for functions of reference projects.**

(including reference functions from the knowledge base in addition to those from $P$), we cannot really trust that these are not accidental. Therefore, we limit ourselves to only those properties that are frequent enough—understood as occurring in at least minimum-support-many functions.

## 4.3 Detecting Anomalies

The process of detecting anomalies consists of two phases: discovering *patterns*, and discovering *violations* of those patterns. The idea is to first find patterns—sets of temporal properties—that occur together in many functions, and then to look for anomalies (i.e., violations of those patterns). For this purpose, we use the Colibri-Java tool [7] that implements formal concept analysis [5]. We give Colibri-Java a binary matrix as an input, where rows are functions (those coming from $P$ as well as those stored in the knowledge base), and columns are temporal properties. The matrix contains an entry for a specific function and a specific temporal property if the property occurs in the function. Figure 7 shows a sample concept analysis matrix.

Colibri-Java first finds patterns in the matrix: these are the sets of temporal properties that occur in at least minimum-support-many functions. Intuitively, a pattern is a rectangle (not necessarily contiguous) in the matrix. Figure 7 shows two sample patterns. It is important to notice that we are actually able to find patterns induced by reference data; patterns that might not be present in $P$ at all.

After finding patterns, Colibri-Java looks for anomalies. The idea here is to find functions in $P$ that violate the patterns found earlier. A function violates a pattern if it exhibits some of its temporal properties, but not all of them. Detecting such violations is equivalent to finding "gaps" in the concept analysis matrix [10]. In Figure 7 we can see that the function $f_1$ violates the pattern represented by the set of temporal properties $\{p_1, p_2, p_3\}$, because it exhibits $p_1$ and $p_2$, but not $p_3$. For more details about formal concept analysis and using it for detecting patterns and anomalies the reader can consult the publications cited in this section as well as the work on JADET [19], which also uses formal concept analysis for detecting anomalies.

## 4.4 Ranking and Filtering Anomalies

Anomaly detection typically results in discovering a large number of anomalies, most of them being false positives. We deal with this problem by first ranking the anomalies, and then filtering them, so that the user has less anomalies to inspect and can focus only on the top-rated ones. For ranking anomalies, we compare their *lift* measures.

Formally, we can treat each anomaly (i.e., violation of a pattern) as a violation of an *association rule* $A \rightarrow B$, where $A$ and $B$ are sets of temporal properties, $A \cup B$ is the set of temporal properties that constitutes the pattern, and $B$ is the set of temporal properties that are missing in the function violating the pattern. Lift of an association rule $A \rightarrow B$ is a well-known metric that measures how many times more often $A$ and $B$ occur together than expected if they were statistically independent.

When it comes to filtering, we leave only one anomaly for each function—the highest ranked one. While this may lead us to miss some true positives, more often than not it results in simply removing duplicates. The reason for this is that it frequently happens that a certain function does not exhibit certain important temporal properties, thus violating multiple patterns in the same way (i.e., in all those anomalies the same properties are missing, and mostly the same properties are present).

# 5. EVALUATION

## 5.1 Evaluation Setup

To evaluate our approach in terms of efficiency and effectiveness, we use the following evaluation scheme:

1. We create a knowledge base consisting of data from C projects being part of the Gentoo Linux distribution. (see Section 5.2.) The reason for our choice is that Linux is widely used (the distribution would thus be expected to contain mature code to learn from) and contains the greatest possible variety of projects. We chose Gentoo because it is a distribution that provides the source code for each and every package it contains. The reason for focusing on C programs is that C is the dominant programming language in the Gentoo Linux distribution. However, we would like to emphasize the fact that our approach *is not limited to C*. C is our language of choice for the evaluation, but our approach works without any changes for other, similar programming languages, such as C++, Java, or PHP (see Section 2).

2. We randomly choose 20 C projects from the distribution and apply anomaly detection to them. We classify the anomalies found and calculate the true positive rate (Section 5.3). The rationale for choosing 20 projects is that this is a sample of sufficient size to minimize the risk of bias towards a specific type of project; also, 20 projects was the amount of code that we realistically could review for classification.

3. We apply anomaly detection to C projects from the Gentoo Linux distribution (i.e., all projects, whose data we have in our database) and use the total number of anomalies found and the true positive rate calculated earlier to find the expected number of true positives we could theoretically discover in all of the distribution. (Section 5.7)

When detecting anomalies, we applied two changes to the procedure described in Section 4:

First, for the time of the analysis, we excluded from the knowledge base the project that was being analyzed. This is because when analyzing a project that is at the same time a reference project (and thus has its temporal properties stored in the knowledge base) we do not want the temporal properties from this project count twice: as being part of the analyzed project, and as being part of a reference project.

Second, not all events and temporal properties extracted from reference projects are useful. It can happen that some events occur so often as to become "dead weight" that not only makes anomaly detection slower, but also causes false warnings to appear. One such example is for instance the family of `printf` functions. They occur many times in many projects, but the temporal properties in which they occur are typically not interesting. We deal with this issue by ignoring two kinds of events during the calculations:

- Events related to `printf`, `fprintf`, `sprintf`, as well as `scanf`, `fscanf`, `sscanf` are ignored.
- Events that occurred too frequently (relatively) are ignored. For this purpose, we calculated statistics for the events set in the reference projects, and ignored all events whose occurrence frequency was so large as to exceed the third quartile by more than 1.5 times the interquartile range (third quartile minus first quartile). This is one of the standard measures for identifying *outliers* in a data set; in practice this meant ignoring events occurring more than 5742 times.

  There were 262 such events (out of 3,592,375 events in total; less than 0.01%). The ignored events include for example `strncpy(3)@1` and `malloc(1)@-1`. These have the same characteristic as the events described above: they occur many times in many projects, but the temporal properties in which they occur are typically not interesting.[5]

The need to do this kind of filtering depends very much on the knowledge base. Ours is quite varied and contains many very different projects, so it is reasonable to assume that filtered events would really not be helpful. If the knowledge base is more homogenous, filtering might be unnecessary.

## 5.2 The Knowledge Base

We have included in our knowledge base almost all projects[6] that satisfy the following condition: The project is stored in one package, contains at least one `.c` file and the parser is able to create at least one function model out of it (interestingly, some files just contained data declarations).

This gives in total 6097 projects, with 5,985,193 function models created in the process. There are 3,592,375 different events and 15,803,766 different temporal properties in our database. The database itself takes around 5.9 GB of space under MySQL. The number of source lines (SLOC)[7] of the projects ranges from 7 (for openssl-blacklist_0.4.2 and openvpn-blacklist_0.3) to 5,491,951 (for linux-2.6.29), with the total SLOC of all projects being 201,321,237. This is over *200 million* lines of code of reference data. Creating the database took a little less than 18 hours, with an average time of less than *11 seconds* per project on a 16-core 2.9 GHz Intel Xeon machine with 24 GB of RAM. However, we used a single thread only, so the process can be sped up by actually exploiting parallelization. Also, this computation must be done only once; afterwards the knowledge base is easily used, and—if needed—easily and quickly updated.

[5]It is interesting that events of the first kind (`printf` and `scanf` family) are not detected as outliers. The reason for this is their large variety that results in a lot of events that are related to them, but that in themselves do not occur frequently enough to be classified as outliers. As an example, there are 1504 distinct events related to `printf`, with `printf` taking up to 47 arguments.

[6]29 small-sized projects have been used for testing purposes and were thus not included in the knowledge base.

[7]SLOC were calculated using David A. Wheeler's 'SLOCCount'; only `.c` files were taken into account.

**Table 1: Details of 20 analyzed projects.**

| Project | SLOC | Analysis time | | Anomalies |
|---|---|---|---|---|
| | | Parsing | Total | |
| 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 |

*We learn typical API usage from over 6000 projects, encompassing more than 200,000,000 lines of code, in less than 11 seconds per project.*

## 5.3 The Random Sample

We have randomly chosen 20 from the 6097 projects we had in our database and applied our anomaly detection mechanism to each one of them in turn. Table 1 shows for each of the projects the following data: its name, its size in SLOC[7], the time it took to do parsing and produce function models, the total time it took from parsing to outputting anomalies, and the number of anomalies found. As can be seen, the projects are quite varied, ranging from tiny (python-scw-0.4.7 with 69 SLOC) to large (gimp-2.6.6 with 595,664 SLOC). Analysis time was mostly less than *10 seconds* per project. The larger projects needed much more time, up to almost 18 minutes for gimp-2.6.6, but we deem this still an excellent value for cross-project analysis that has over 6000 reference projects at its disposal and makes full use of them. Notable is the fact that our lightweight parser can cope with almost all projects in several seconds.

*Cross-project anomaly detection is not only possible, but also fast enough to be of practical use.*

Table 2: Summary of results. Only projects with at least one anomaly are listed.

| Program | # Anomalies | | # Defects | # Code smells | # False positives | Effectiveness |
|---|---|---|---|---|---|---|
| | Total | Top ~25% | | | | |
| cksfv-1.3.13 | 1 | 1 | 1 | 0 | 0 | **100%** |
| ggv-2.12.0 | 3 | 1 | 0 | 0 | 1 | **0%** |
| gimp-2.6.6 | 61 | 22 | 1 | 2 | 19 | **14%** |
| glade3-3.6.4 | 18 | 5 | 2 | 0 | 3 | **40%** |
| httrack-3.43-4 | 8 | 2 | 0 | 1 | 1 | **50%** |
| mpich-1.2.7p1 | 12 | 5 | 0 | 4 | 1 | **80%** |
| otp_src_R13B | 14 | 4 | 0 | 0 | 4 | **0%** |
| psycopg-1.1.15 | 6 | 6 | 0 | 0 | 6 | **0%** |
| tclxml-2.4 | 8 | 2 | 0 | 0 | 2 | **0%** |
| viewres-1.0.1 | 1 | 1 | 0 | 0 | 1 | **0%** |
| Yap-5.1.3 | 4 | 1 | 0 | 0 | 1 | **0%** |
| | **136** | **50** | **4** | **7** | **39** | **22%** |

## 5.4 Classification

We have investigated the top 25% (rounded up) anomalies for each of the analyzed projects and classified them into three categories:

**Defects.** If code was defective, we marked it as such. An example of defective code is a resource leak: obtaining a resource (such as a file) and then forgetting to free it (in the case of a file—by closing it).

**Code smells.** This category contains all anomalies that are not defects, but the anomalous functions have properties indicating that something may go wrong [3] or they might be improved in a way that improves readability, maintainability or performance of the program. An example might be a function that uses a `for` loop to iterate through a collection and breaks unconditionally out of the first iteration. If the collection can have at most one element, this code will work, but it cannot be treated as fully correct.

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

We stayed on the conservative side when categorizing the anomalies. If we were not sure if the code was correct, we treated it as correct. We did not measure the time that was needed to classify the anomalies found in the analyzed projects, but in our past experiments the time measured was roughly two and a half minutes per anomaly on average [17].

## 5.5 Results

The result of our categorization can be seen in Table 2. For all analyzed projects we present the total number of anomalies, the number of anomalies the top 25% amounts to (these are the anomalies we investigated)[8], the number of defects, code smells, and false positives found by investigating them, and the effectiveness (i.e., the percentage of anomalies that were defects of code smells).

As can be seen, the largest projects also have the largest number of anomalies; however, only 11 out of the 20 projects actually show anomalies. We attribute this low number to two reasons:

- First, we would expect mature production code such as Linux distributions to show a *high level of standard conformance* anyway; its maturity, combined with open source peer review, naturally implies a low number of remaining defects, and thus a low number of anomalies as well. Still, 11 out of 50 violations (22%) pointed to code smells or new defects; Section 5.6 lists some of them. Applied to less mature or pre-production code, we would expect a much higher number of defects—and thus also a higher rate of true positives.

- Second, our *lower threshold* of 200 for the minimum support value causes several patterns with smaller support to go unnoticed—and likewise their violations. There is an obvious tradeoff involved, as lowering the threshold will increase the time and memory needed to do anomaly detection. An optimal threshold value thus depends on the resources being allocated for anomaly detection.

*Cross-project anomaly detection can detect previously unknown defects in mature software projects.*

## 5.6 Examples

Let us now take a more qualitative perspective and examine some of the anomalies detected. Figure 8 shows the skeleton of the defect found in cksfv-1.3.13. The problem with this code is that it indirectly (using `opendir`) creates a `DIR` structure allocating memory for it, but forgets to free it using `closedir`. The violated pattern is present in our knowledge base over 1700 times. The programmer was obviously aware of the problem, because she included a call to `rewinddir` at the very end of the function. Unfortunately, this is not the function that should be called. `find_file` will cause a memory leak each time it is called.

Figure 9 shows the skeleton of a code smell found in gimp-2.6.6. It uses GTK iterators to iterate through a tree model. The problem with this code is that it uses two different things for the same purpose: the `GTK_TREE_MODEL (priv->store)` expression and the `model` variable. Both are equivalent, as the first one gets assigned to the second near the

[8]Sometimes this is more than exactly 25%. The reason for this is that some anomalies have the same ranking, so we had to include all such equally-ranked anomalies.

```
static int find_file (...)
{
    DIR *dirp;
    struct dirent *dirinfo;
    ...
    dirp = opendir(".");
    if (dirp == NULL)
    {
        ...
    }
    while ((dirinfo = readdir(dirp)) != NULL)
    {
        ...
    }
    rewinddir(dirp);
    return 1;
}
```

**Figure 8: Skeleton of a defect in cksfv-1.3.13.**

```
static gboolean gimp_page_selector_item_width_idle (...)
{
    GimpPageSelectorPrivate *priv = ...;
    GtkTreeModel *model = GTK_TREE_MODEL (priv->store);
    GtkTreeIter iter;
    ...
    for (... = gtk_tree_model_get_iter_first (model,
                &iter);
            ...;
            ... = gtk_tree_model_iter_next (model,
                &iter))
    {
        ...
        gtk_tree_model_get (GTK_TREE_MODEL (priv->store),
            &iter, ..., ..., ...);
        ...
    }
    ...
}
```

**Figure 9: Skeleton of a code smell in gimp-2.6.6.**

beginning of the function. However, the variable is used in the head of the `for`-loop, whereas the expression is used in its body. Such code is difficult to understand and to maintain. Moreover, it crucially depends on `GTK_TREE_MODEL` being a macro, not a function call, because otherwise the two occurrences of the expression could yield different values.

Figure 10 shows the skeleton of a code smell found in httrack-3.43-4. The anomaly here is that `MD5Init`, as occurring in hundreds of projects, normally takes only one argument; in httrack-3.43-4, though, it takes two. A closer look reveals that the project uses *its own implementation* of MD5; the additional argument determines whether `MD5Init` should try to correct broken endian. This code is an obvious

```
int domd5mem (...)
{
    int endian = 1;
    ...
    MD5_CTX ctx;
    ...
    MD5Init (&ctx, * ( (char*) &endian));
    MD5Update (&ctx, ..., ...);
    MD5Final (..., &ctx);
    ...
}
```

**Figure 10: Skeleton of a code smell in httrack-3.43-4.**

case of bad design, because callers of `MD5Init` are required to do the job that should be done by the callee. Interestingly, there are two calls to `MD5Init` in httrack-3.43-4, and both use different second arguments. One of them is the call shown in Figure 10, which obviously does some kind of check; another one simply passes 0. It is highly probable that the other call is defective.

As can be seen from the results, using the lightweight parser incurs an effectiveness penalty (JADET, which uses the same anomaly detection mechanism, is more effective), but allows us to perform cross-project anomaly detection in reasonable time and in a language-independent fashion. Also, in spite of its simplicity it is accurate enough to allow detecting real defects. As can be seen from the examples given in this section and in the introduction, some issues can only be found by a cross-project analysis. This is true for the bad design of `MD5Init` shown in this section, and for the wrong call to `setsockopt` shown in the introduction.

*Some issues found by cross-project anomaly detection cannot be found by single-project approaches.*

## 5.7 Anomalies in Gentoo Linux distribution

We have run our anomaly detection on all 6097 projects to find out how many anomalies our approach can detect in the Gentoo Linux distribution. Our approach needed 3 days and 18 hours to analyze these; this gives an average analysis time per project of about *53 seconds*. During the process, there were 21,562 anomalies found in total, ranging from 0 for most projects, to 765 for open64-4.2.1-0.src, with an average of less than 4 anomalies per project.

## 5.8 Threats to Validity

The results of our experiments are subject to the following threats of validity:

**Threats to *external validity*** concern our ability to generalize the results of our study. We have focused our evaluation on projects written in C and being part of the Gentoo Linux distribution. This might influence the effectiveness of both the parser and the whole approach. It might be that our results would be worse if we focused on a different language or used a different codebase. However, our earlier experiments with the parser have shown it to be effective for other programming languages too [6]. On the other hand, Linux distributions contain as large a variety of programs, as one can think of, so we think this threat is negligible.

**Threats to *internal validity*** concern our ability to draw conclusions about the connections between our independent and dependent variables. The results of the categorization process performed on anomalies depend on the expertise of the human applying the approach. However, if anything, this would *improve* our results—because we have marked anomalies as true positives only if we were completely sure that they are indeed defects or code smells. An experienced developer may spot potential problems where we see false positives.

Our sample of 20 projects can be biased and allow us to achieve better results than could be achieved for the whole Gentoo Linux distribution. This threat is real, and unfortunately we cannot mitigate it. For this, one would need to do cross-validation, but in practice this

means classifying all or almost all anomalies found for the whole distribution, which is practically infeasible.

**Threats to *construct validity*** concern the appropriateness of our measures for capturing our dependent variables. The greatest threat is that the tools we have used or implemented could be defective. To counter this threat, we have run several manual assessments and counter-checks; so we believe that any defects left affect only a small number of anomalies, and thus do not spoil the results overall.

# 6. RELATED WORK

There are approaches that use large code bases to find correct API usages—these are closely related to our work. Zhong et al. created MAPO [21], which provides sample sequences of method calls. MAPO provides information about the order in which the methods are typically called and also provides a recommender that helps the programmer decide if he is following a correct pattern or not. In contrast to MAPO, our approach is able to express how objects flow through calls, is language independent, and includes automatic anomaly detection. Thummalapenta and Xie [16] describe Alattin: an approach for finding neglected conditions by comparing the code with sample code obtained using a code search engine. This is similar to our approach in that it aims for cross-project anomaly detection. The main advantage of Alattin, and one that could theoretically be integrated into our approach, is that it recognizes so-called "infrequent patterns". The main drawback is that it is limited to recognizing only neglected conditions, and it is not language independent. Also, one major drawback of all approaches that use code-search engines is that they cannot control the quality of code that is used for learning, because they do not have influence on the code that is being returned by code-search engines. In contrast, when using knowledge bases, users can create knowledge bases that are particularly well-suited to their needs.

Our work builds on the anomaly detection mechanism of JADET by Wasylkowski, Zeller, and Lindig [19]. We improve on that technique by not only adding cross-project capability and language independence, but also by making temporal properties more expressive: JADET does not include arguments' numbers in its temporal properties, and therefore cannot detect patterns where it is not only ordering of calls that matters, but also the flow of values between arguments. Also, we have improved the ranking system.

Some existing anomaly detection approaches could theoretically be adapted to our infrastructure and then used for the purposes of cross-project anomaly detection while still preserving scalability. PR-Miner [9] by Li and Zhou with its patterns being essentially sets of events from our function models is an obvious candidate. Tikanga by Wasylkowski and Zeller [18] is another, improving on JADET by, among others, using the concept of operational preconditions and operating on much more expressive CTL formulas instead of temporal properties. These improvements seem possible to incorporate in our approach. Ramanathan et al. [14] use a form of temporal properties extended with axiomatic preconditions, resulting in fairly expressive specifications—which could perhaps be incorporated into our approach. All of these are based on static analysis, and it is not clear how any of the dynamic analysis approaches— [4, 20, 8], etc.—could be used for cross-project anomaly detection. The same problem occurs for approaches that mine structurally complicated specifications— [13, 15], etc.

Several approaches that extract information from code in a lightweight fashion exist. Murphy and Notkin [12] developed a lexical lightweight approach for source model extraction and Moonen [11] used island grammars to create robust parsers. Both these approaches provide a general framework for extracting information from source code that needs to be customized by the user. Our parser is specialized for the extraction of code structure and function calls. Collard et al. [1] developed an XML-based lightweight fact extractor for C++. Dagenais and Hendren [2] developed partial program analysis for Java. These tools, unlike our parser, are bound to a specific language. Ctags[9] is a tool that can recognize language objects and supports many different programming languages. However, Ctags is only capable of creating an index file with such objects, and this makes it inapplicable for program analysis purposes.

# 7. CONCLUSIONS AND CONSEQUENCES

In modern software systems, correct interface usage is more than just signatures and types; you need to do the right things in the right order and at the right time. In this paper, we have shown that (a) a lightweight parser is efficient, yet effective enough to mine usage rules from large bodies of almost arbitrary source code; that (b) the resulting properties efficiently determine anomalies in new projects; that (c) a significant amount of these cross-project anomalies indicates defects or code smells; and that (d) despite its maturity, the Linux distribution still contains thousands of such issues.

Despite the advances, there is still much room for improvement. Our future work will focus on the following topics:

**Using CTL specifications.** Our Tikanga tool [18] mines CTL formulae rather than simple temporal properties. This dramatically increases the number of true positives (42% rather than 25% for JADET). Combining light-weight parsing with CTL formula mining should improve precision for the present approach as well.

**Explore API evolution.** Usage rules may change over time. We are working on identifying these changes—to keep our recommendations up to date, and to identify obsolete usages that must be updated as well.

**Early programmer support.** Once mined, usage rules can be easily integrated into the programming environment, warning the user about potential problems before they become serious ("Most Linux programmers think otherwise") Likewise, temporal properties can become part of the documentation or the specification. The mined properties can also be checked against given ones, giving additional opportunities for validation.

**User feedback.** To put the approach to full use, we are currently building a web site *checkmycode.org* where interested programmers can have their code checked against "the wisdom of Linux code" (Figure 11). We expect the feedback from this service to further guide and shape our future research.

[9]http://ctags.sourceforge.net


**Figure 11: On our web site www.checkmycode.org, programmers can have their code checked against our knowledge base, or "the wisdom of Linux code".**

For future and related work on anomaly detection, see

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

**Acknowledgements.** Sascha Just provided us with the packages of the Gentoo Linux distribution. Valentin Dallmeier, Gordon Fraser, Kim Herzig, Yana Mileva, David Schuler, and anonymous reviewers provided helpful comments on earlier revisions of this paper.

## 8. REFERENCES

[1] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *IWPC 2003: Proc. 11th IEEE International Workshop on Program Comprehension*, 2003.

[2] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *OOPSLA 2008: Proc. 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 313–328, 2008.

[3] M. Fowler. *Refactoring. Improving the design of existing code.* 1999.

[4] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT 2008/FSE-16: Proc. 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349, 2008.

[5] B. Ganter and R. Wille. *Formal concept analysis: mathematical foundations.* 1999.

[6] N. Gruska. Language-independent sequential constraint mining. Bachelor thesis, Saarland University, 2009.

[7] D. N. Götzmann. Formale Begriffsanalyse in Java: Entwurf und Implementierung effizienter Algorithmen. Bachelor thesis, Saarland University, 2007. Available from http://code.google.com/p/colibri-java/.

[8] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS 2009: Proc. 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 5505, pages 292–306. 2009.

[9] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE 2005: Proc. 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, 2005.

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

[11] L. Moonen. Generating robust parsers using island grammars. In *WCRE 2001: Proc. Eighth Working Conference on Reverse Engineering*, 2001.

[12] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[13] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE 2009: Proc. 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392, 2009.

[14] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *PLDI 2007: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 123–134, 2007.

[15] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, Sept. 2008.

[16] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE 2009: Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[17] A. Wasylkowski and A. Zeller. Mining operational preconditions. Technical report, Saarland University, Software Engineering Chair, 2008. Avaliable from http://www.st.cs.uni-saarland.de/publications/.

[18] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE 2009: Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[19] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC/FSE 2007: Proc. 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44, 2007.

[20] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE 2006: Proc. 28th international conference on Software engineering*, pages 282–291, 2006.

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009: Proc. 23rd European conference on object-oriented programming*, Lecture Notes in Computer Science 5653, pages 318–343. 2009.