API Hyperlinking via Structural Overlap

Fan Long
Tsinghua University
longfancn@gmail.com

Xi Wang
MIT
xi@csail.mit.edu

Yang Cai
MIT
ycai@csail.mit.edu

ABSTRACT

This paper presents a tool Altair that automatically generates API function cross-references, which emphasizes *reliable* structural measures and does not depend on specific client code. Altair ranks related API functions for a given query according to pair-wise overlap, i.e., how they share state, and clusters tightly related ones into meaningful modules.

Experiments against several popular C software packages show that Altair recommends related API functions for a given query with remarkably more precise and complete results than previous tools, that it can extract modules from moderate-sized software (e.g., Apache with 1000+ functions) at high precision and recall rates (e.g., both exceeding 70% for two modules in Apache), and that the computation can finish within a few seconds.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*Navigation*

General Terms

Algorithms, Documentation, Experimentation

Keywords

API recommendation, overlap rank, module clustering

1. INTRODUCTION

Contemporary software systems provide an increasingly complex API to developers, consisting of hundreds or even thousands of functions. Looking up API usages in documentations such as the Unix man pages, the Java API specification, and the MSDN library, is a daily exercise for many developers. Those documentations come up with comprehensive and detailed *cross-references* for a large number of API functions, e.g., SEE ALSO sections, providing hyperlinks to related functions that accomplish the same or a relevant task. Such informative layout is an effective way to organize the knowledge and help avoid getting lost in the jungle.

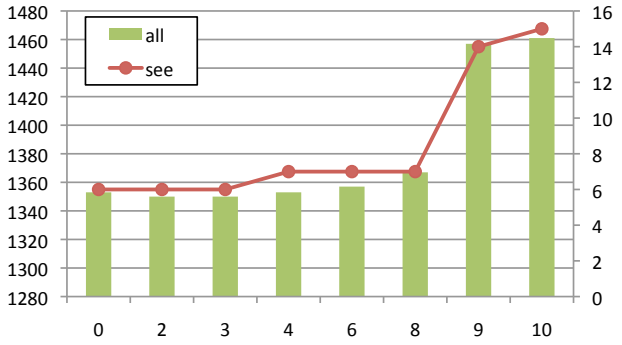


Figure 1: Numbers of all functions (left Y-axis) and those commented with cross-referencing tag `@see` (right Y-axis) in Apache 2.2.x (X-axis) source code. The version numbers are not continuous.

However, maintaining these separate documentations is painful and labor-intensive. Many documentation tools such as doxygen [3], javadoc, and the C# compiler allow to manually fill in cross-references. Developers can annotate related functions in code comments using specific grammars, e.g., leading by `@see` or enclosed by `<seealso>` tags, so as to generate cross-references in documentations. However, it is generally difficult for developers to track all related API for each function. In addition, with the API evolving and growing, it becomes tedious to maintain the knowledge and keep up to date by hands.

Take Apache HTTP server [1] as an example, the source code of which is fairly well documented. As shown in Figure 1, from year 2005 to 2008 the number of documented API functions grows from 1353 to 1461, while the number of those commented with `@see` grows from 6 to 15, accounting for only 0.4%∼1.0% of all API functions — only a small portion of API functions have cross-referencing annotations. This paper will explore automatic techniques to improve the coverage by generating function cross-references from source code.

One way to discover related API functions is to mine frequent usage patterns in *client* code [30, 22, 47, 36, 48]. The basic idea is based on association rules or co-citation [39], say, given a function f, find all functions g that are often

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.

`int BZ2_bzCompressEnd ( bz_stream *strm )`

Releases all memory associated with a compression stream.

See also

- `BZ2_bzCompressInit` and `BZ2_bzCompress`
- `BZ2_bzDecompressEnd`, `BZ2_bzDecompressInit` and `BZ2_bzDecompress`

Figure 2: An example output of Altair. It hyperlinks the bzip2 API function `BZ2_bzCompressEnd` to five related functions and organizes them into two modules, i.e., compress and decompress.

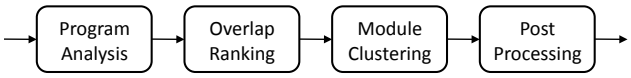


Figure 3: Stages of Altair.

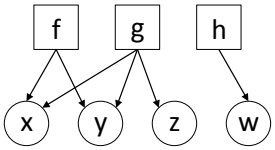


Figure 4: Example access graph. f, g, h represent functions, while x, y, z, w represent data. A directed edge from f to x means that function f may access some data x.

called together with f. However, the approach might be sensitive to how f is used in specific client code. If the client code neither calls f much nor calls f and g simultaneously, it could not find these patterns, which leads to missing and unreliable results. Particularly, it could not extract much useful information from library code, where no client code is actually invoking these API functions. We will illustrate the problem in Section 7.1.

Our tool Altair takes a different approach — it emphasizes how API functions are implemented rather than how they are used, which is insensitive to specific client code. It is based on the observation that in general API functions are related because they share state, e.g., the data they access *overlap* (see Section 2 for a formal definition). Altair performs static analysis to extract structural information from source code, i.e., how API functions access data, and then computes pair-wise overlap between them. Given query f, Altair finds the related functions that overlap with f on the most data that f accesses. In doing so, it can produce more *complete* yet *precise* results. In addition, Altair introduces a novel technique to cluster API functions into modules.

Figure 2 illustrates an example output of Altair. When given a query `BZ2_bzCompressEnd`, which is an API function from a popular data compressor bzip2 [2], Altair is able to recommend five related functions and tries to arrange them into two modules (compress and decompress). We believe that this is a better organization for API understanding and navigation.

We implement Altair for C programs. It performs analysis, ranking, clustering, and post processing successively, as shown Figure 3, and then answers user queries for API navigation. Experiments show that Altair notably outperforms previous API recommendations tools, and that it can further help to cluster API functions into meaningful modules.

The main contributions of this paper are: 1) adoption of effective ranking and clustering measures, 2) program analysis to compute the measures, and 3) an evaluation against several popular C software packages. Altair, along with demos and source code, is publicly available at

`http://pdos.csail.mit.edu/~xi/altair/`

The rest of the paper is organized as follows. Section 2 defines the overlap measure for ranking. Section 3 presents the static analysis algorithm for computing the measure from source code. Section 4 defines the modularity measure and presents the clustering algorithm. Section 5 discusses alternative measures and limitations. Section 6 describes implementation detail. Section 7 shows experimental results. We survey related work in Section 8 and then conclude in Section 9.

2. OVERLAP RANK

Altair hinges on the hypothesis that developers tend to *encapsulate* state or data structures using a set of well-defined API functions; these functions are the preferred or the only means to access the data structures [23]. Therefore, these API functions are related since the data that they access overlap.

Consider a simple case. A program is represented as a bipartite *access graph*, as shown in Figure 4. There are two kinds of vertices: functions (f, g, and h) and data (x, y, z, and w). A directed edge from f to x means that f may access x. In the graph f and g share access to both x and y. More challenging issues will be further discussed in Section 3.

Let $\mathcal{N}(f)$ denote the set of data that f may access. Given query f, how much it overlaps with g relates to the shared data that they both access.

Definition 1. Given a function f, the overlap with function g is defined as

$$\pi(g|f) = \frac{|\mathcal{N}(f) \cap \mathcal{N}(g)|}{|\mathcal{N}(f)|}. \tag{2.1}$$

$\pi(g|f) \in [0, 1]$ represents the proportion of f's data that is shared with function g. For example, as in Figure 4 we can learn $\pi(g|f) = 1$ and $\pi(f|g) = 2/3$. For each function f, we may compute an overlap value with other functions and rank these values for recommendation.

The measure is asymmetric, which is consistent with our intuition: while g links to some "hot" function f, it does not mean that f should necessarily link back to g. A symmetric variant will be used in Section 4 for clustering. See Section 5 for discussion.

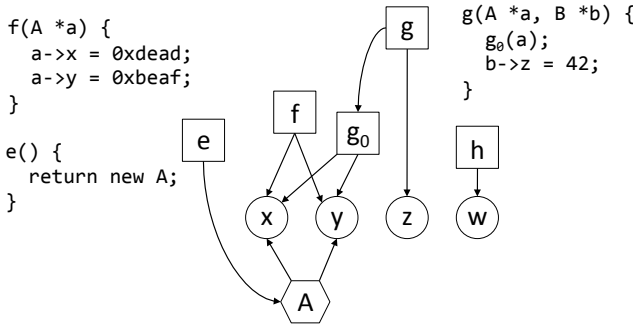


Figure 5: Augmented access graph. e, f, g, g_0, h represent functions, x, y, z, w represent data, and A represents a composite type.

3. ANALYSIS

This section describes the detail of computing a bipartite access graph via program analysis.

We will use the code in Figure 5 as a running example. In this example, function e returns a newly allocated object of a composite type A; function f accesses two fields x and y of a, which is of type A as well; function g accesses field z of b, which is of type B, and calls an internal (private) function g_0, which behaves similarly to f. We omit the detail of functions g_0, h, and type B for brevity.

3.1 Challenges and Heuristics

There are several challenges arising from the example in Figure 5. We describe them below, as well as heuristics that Altair employs to address them.

Calls

When g calls g_0, should we merge g_0's effect (e.g., the set of data it accesses) into g or not? In practice it is common to organize a big function's implementation into several internal, smaller functions. If we do not merge their effect, the big function's overlap with others would be inaccurate. On the contrary, API functions in a higher-level module should be isolated from lower-level modules; merging their effect in this case is inappropriate since they belong to separate modules.

We observe that public functions are usually a natural boundary when developers are designing and implementing API functions. Therefore, Altair merges the effect of *private* functions into their callers. If their callers are also private, it does so recursively. If their callers are public functions, it stops merging. A simple heuristic rule to determine private/public for a C function is to see whether it is declared as `static` or `extern`.

Open Programs

The analysis should conclude that both f and g access two fields x and y. To do so, it should identify that the first parameters a in f and g may alias, i.e., they may refer to the same data. It is straightforward to perform a pointer analysis over a whole program to compute such may-alias results. However, the results would be missing for an open program with library code only, e.g., *no* client code is present to call f and g.

To this end, Altair conservatively assumes that all variables of the same composite type (e.g., structure/record) may represent the same data. That is, we represent local data using type fields; all instances of the same composite type are projected into a single one. In doing so Altair is able to connect library API functions at a price of possible precision loss, e.g., it cannot distinguish two instances of the same type. Generally this is not a big problem, though it may lead to some conservative results.

Edge	Type	Description
access(f, x)	$V_F \times V_D$	f loads from/stores to x
call(f, g)	$V_F \times V_F$	f calls g
allocate(f, t)	$V_F \times V_T$	f (de)allocates t
composite(t, x)	$V_T \times V_D$	t has a field x

Figure 6: Edges of augmented access graph. The sets V_F, V_D, and V_T represent functions, data fields, and composite types, respectively.

Allocation

Allocation functions such as `malloc` and `free` in C do not explicitly touch any field of a given object, e.g., function e. However, behaving as constructors and destructors, they do affect all fields. We should not miss their effect.

Altair recognizes standard C allocation functions and concludes that invoking them will access all fields of corresponding types. It asks users to annotate customized allocation functions. In doing so, Altair is able to identify them and calculate correct effect for functions that invoke allocations. In our experiments (Section 7.4) we see that such annotation effort is negligible.

3.2 Augmented Access Graph

Altair augments the access graph to incorporate with the heuristics discussed above. As depicted in Figure 5, there are three kinds of vertices in the augmented access graph:

- functions V_F, e.g., e, f, g, g_0, and h;
- data (composite fields[1]) V_D, e.g., x, y, z, and w;
- composite types V_T, e.g., A.

Meanwhile, there are four kinds of edges between vertices, as listed in Figure 6. They are used to indicate that a function may access some data directly, or indirectly via a callee, or via a composite type due to allocation, or a combination of the above.

To construct an augmented access graph from source code, the analysis works as follows. For each load/store statement, e.g., one in function f that accesses data x of composite type A, Altair adds a corresponding access edge from f to x and a composite edge from A to x. For each call statement, e.g., g calls g_0, if the callee g_0 is a private function, Altair adds a corresponding call edge from g to g_0. If function e invokes an allocation function for type A, Altair adds a corresponding allocation edge from e to A. Note that there may be type conversions. For example, `malloc` returns a void pointer, which will often be converted to a specialized type. Altair tracks such conversions to deduce the real type.

[1] Our implementation also represents each global variable (e.g., a Boolean flag) as a data vertex.

To convert an augmented access graph into a bipartite graph, i.e., to find out what data set each function may access following all paths, one can compute graph reachability (transitive closure) [32] for each $f \in V_F$. For example, in Figure 5 both functions g and e will connect to x and y after computing transitive closure. The resulting bipartite access graph only contains V_F, V_D, and edges between them. The algorithm has a time complexity of $O(|V||E|)$, where V and E are vertices and edges, respectively.

If the augmented access graph is huge, one could compute a limited number of iterations for an approximation. In our experiments Altair uses the transitive closure approach and is able to finish in a short time.

3.3 Ranking

It is straightforward to compute overlap between functions from a bipartite access graph, based on (2.1), as well as normalize (the largest to 1) and sort them in descend order. If $\pi(g|f)$ and $\pi(h|f)$ have the same overlap rank, Altair further compares $\pi(f|g)$ with $\pi(f|h)$.

If a function has too many related candidates, Altair shows top l results ($l = 10$ by default). If still too many equal ones, those with normalized overlap values less than a threshold (default 0.85) will be discarded.

Note that a wrapper function that accesses no data but simply calls other functions does not overlap with any functions. In this case Altair separately considers its callees as related functions.

4. CLUSTERING

This section presents a module definition and an algorithm to cluster API functions into modules.

A straightforward clustering method would be grouping functions by the data set they access [38, 45]. However, it may not work in general. Real-world API implementations may *not* be perfectly modularized; two functions in different modules may still share a small portion of data (see Section 7.3 for a case study).

4.1 Modularity

Based on the overlap measure (2.1), we define a joint measure called *overlap coefficient* [43] as follows.

$$\pi(f,g) = \frac{|\mathcal{N}(f) \cap \mathcal{N}(g)|}{\min(|\mathcal{N}(f)|, |\mathcal{N}(g)|)} \tag{4.1}$$

$$= \max(\pi(f|g), \pi(g|f)) \tag{4.2}$$

$\pi(f,g)$ measures how much f and g share in common, which is symmetric and can be easily derived from overlap rank.

Our intuition used to define modularity is that the interconnection between modules should be relatively loose. Consider the simplest two-module case. Assume that a set of functions F can be partitioned into two modules, a subset $S \subseteq F$ and the complement $\overline{S} = F - S$. According to the intuition, compared to the overlap of all vertices in S, the overlap between S and $\overline{S}$ should be small. We denote them as $\operatorname{vol} S$ and $\operatorname{vol} \partial S$, respectively, given as follows.

$$\operatorname{vol} S = \sum_{f \in S, g \in F} \pi(f,g) \tag{4.3}$$

$$\operatorname{vol} \partial S = \sum_{f \in S, g \in \overline{S}} \pi(f,g) \tag{4.4}$$

1: $D_{ff} \leftarrow \sum_g \Pi_{fg}$, given overlap matrix Π.
2: $L \leftarrow D^{-\frac{1}{2}} \Pi D^{-\frac{1}{2}}$.
3: Compute eigenvectors μ_k of L.
4: $X \leftarrow (\mu_1, \ldots, \mu_k)$.
5: Normalize X to Y, where $Y_i = X_i / ||X_i||_2$.
6: Apply k-means to cluster Y.
7: Cluster Π_f into k modules accordingly.

Figure 7: Algorithm for k-module partitioning.

In other words, a good module S should be able to minimize $\operatorname{vol} \partial S$ with respect to $\operatorname{vol} S$ (similarly for $\operatorname{vol} \overline{S}$). Formally, we define modularity based on the concept of graph conductance [12].

Definition 2. Given a set of API functions F, the conductance of a module S ($S \neq \emptyset$ and $S \subsetneq F$) is defined as

$$\Phi(S) = \frac{\operatorname{vol} \partial S}{\min(\operatorname{vol} S, \operatorname{vol} \overline{S})} \tag{4.5}$$

and the modularity of F is defined as

$$\Phi(F) = \min_S \Phi(S). \tag{4.6}$$

$\Phi(F) \in [0,1]$ describes how modularized F is. A small value indicates a loose module, while a large value indicates a tighter one.

4.2 Spectral Clustering

However, finding the optimal partition for S and $\overline{S}$ is known to be NP-hard [37]. A popular technique is to relax the combinatorial problem in continuous real values, which is briefly described as follows.

The *overlap matrix* of $\pi(f,g)$ over all functions F is denoted as Π, given by $\Pi_{fg} = \pi(f,g)$ and $\Pi_{ff} = 0$ (all diagonal elements are zero). Let D be the diagonal matrix of Π and I be an identity matrix. We define L and $\mathcal{L}$ as follows.

$$L = D^{-\frac{1}{2}} \Pi D^{-\frac{1}{2}} \tag{4.7}$$

$$\mathcal{L} = I - L \tag{4.8}$$

Let λ_k be the k-th smallest eigenvalue of $\mathcal{L}$ and μ_k be the corresponding eigenvector. An interesting result in spectral graph theory is about the second smallest eigenvalue λ_2, as follows [12].

$$\lambda_2 / 2 \leq \Phi(F) \tag{4.9}$$

We can then use $\lambda_2/2$ to approximate modularity $\Phi(F)$. The corresponding eigenvector μ_2 is a real-valued approximation of the two-module bi-partitioning. We omit the proof detail due to space limitation.

In a general k-module case, Altair adopts the widely-used Ng-Jordan-Weiss spectral clustering algorithm [31] to partition n API functions in a set F into k modules, as shown in Figure 7. The algorithm first computes an $n \times k$ matrix $X = (\mu_1, \ldots, \mu_k)$ formed by the first k eigenvectors ($\mathcal{L}$ and L have the same eigenvectors so we use L), and normalizes each row of X for a new matrix Y. So Y contains n normalized k-dimensional vectors, where $||Y_i|| = 1$. Then Altair applies a classical k-means algorithm [24] to group the n vectors into k clusters, and accordingly cluster the n functions into k modules. See Section 7.3 for a case study.

The matrix computation in the algorithm has a time complexity of $O(n^3)$. Altair exploits highly-optimized libraries BLAS and LAPACK for fast linear algebra computation, which are available on most platforms.

We choose the spectral method rather than other clustering algorithms due to the following reasons.

First, the definition of modularity based on spectral graph theory matches our intuition well, as explained in Section 4.1.

Besides, hierarchical clustering algorithms used in previous work [34, 28] generally require the *triangle inequality* hold for a dissimilarity measure d over functions [13], i.e., $d(f,g) + d(g,h) \geq d(f,h)$ for any functions f, g, and h, which is not true in our case [10], given $d(f,g) = 1 - \pi(f,g)$. Spectral clustering does not impose such a requirement.

Moreover, spectral clustering has been proven to outperform other clustering algorithms such as k-means in many cases [31].

4.3 Practical Issues

In the clustering algorithm we assume that the number of modules k is priorly known and used as input to Altair. If k is unknown, which is often true in practice, a commonly-used workaround is to recursively bi-partition functions and stop when meeting some criterion, e.g., $\lambda_2/2$ exceeds a given threshold $\phi \in [0,1]$. Though the workaround may lead to less precise results, it is acceptable in our experiments.

One may tune ϕ to bias module sizes — a larger ϕ generally results in smaller modules. Particularly, when setting $\phi = 0$, Altair simply groups functions with non-zero overlap values as modules; on the other extreme, when setting $\phi = 1$, each individual function may be regarded as a separate module.

5. DISCUSSION

This section discusses measures and limitations for generating API cross-references.

5.1 Alternative Measures

Below are other possible measures we have investigated.

Similarity

In fact, our initial implementation of Altair was based on SimRank [19], a powerful measure used to compute similarity between two items. However, there are several serious issues when using it to generate API cross-references.

First, it focuses on how two functions are *similar* rather than how they are *related*. Consider a "big" function f that contains the core functionality and another "smaller" function g that only touches a very small portion of state. The two are *not* similar because they differ vastly on sizes of data they access, though the smaller one g is clearly related to f and it would be missed using SimRank. Other similarity measures have the similar problem.

Besides, SimRank has some counter-intuitive properties that is not suitable here. For example, when two functions both access two data fields rather than one, their similarity value computed by SimRank would become even lower [7].

Association

As mentioned before, association measures reflect how often two functions are used together in client code. As we will see in Section 7.1, an association-based API recommendation algorithm FRIAR [36] will miss related functions in many cases, especially when client code is limited.

Importance

Given a set of functions, an importance ranking algorithm computes the "hottest" ones, such as HITS [20]. Note that such an algorithm itself does *not* accept a given query as input at all; it ranks all functions rather than finds most relevant ones to a query.

To adapt an importance ranking algorithm for finding related functions, one may first find a "neighbor" set of the given query and rank the set according to importance. For example, FRAN [36] finds neighbors in a call graph and ranks them using HITS. However, the most important functions may not be relevant to the query. See Section 7.1 for a comparison between Altair and two importance-based algorithms, Suade [35] and FRAN [36].

5.2 Limitations and Extensions

Altair computes pair-wise relevance based on overlap. One may introduce other heuristic measures, such as naming convention [47] and source organization [40], e.g., two API functions with a common prefix or declared in the same header file may be related. Besides, Altair does not understand semantic relevance, e.g., recommending a SHA-1 hashing function when given a query of MD5.

Since Altair requires to analyze API implementations, it does not work for low-level system calls, the code of which is usually unavailable. Altair may incorporate with association-based approaches in this case.

Currently Altair does not consider overlap between two wrapper functions. From a wrapper function users can only navigate to the functions it wraps.

It remains unclear how to weight edges in an access graph. Altair simply treats them uniformly. It is straightforward to generalize overlap rank to a weighted access graph.

Altair accepts a query of one function. It is easy to extend Altair to accept a query that consists of a set of functions. It could first merge the data that these functions access and compute the overlap with others.

6. IMPLEMENTATION

Altair's backbone is written in C++. The analysis part is implemented in the LLVM compiler infrastructure [21], which includes a GCC front-end.

Our current implementation to deal with C language features is unsound in several ways. It does not support pointer arithmetic. It resolves function pointers intra-procedurally and ignores unresolved ones.

Altair provides XML and JSON back-ends. Combined with code annotations (e.g., a brief description of each API function) extracted by doxygen, Altair provides a friendly, interactive user interface for searching and browsing API functions via the Exhibit web application framework [17]. Readers may refer to Altair's web site for online demos.

7. EXPERIMENTS

In this section we evaluate Altair's ranking and clustering algorithms for API hyperlinking and present quantitative performance results. All experiments were conducted on a laptop with a 2.4 GHz Intel Core 2 Duo CPU and 4 GB memory, running Mac OS X 10.5.6.

7.1 Ranking Comparison

We first compare Altair (ranking only) with three recent algorithms: Suade [35], FRAN, and FRIAR [36]. As discussed in Section 5, Suade and FRAN are based on importance while FRIAR is based on association.

Note that Suade is not initially designed for API recommendation; we use a re-implementation from the FRAN and FRIAR paper [36]. We also adopt all the four test cases from the paper, plus two additional ones, all based on the snapshot of Apache HTTP server 2.0.x source code on October 1st, 2003. They are from a separate, low-level API called Apache Portable Runtime (APR) that the Apache HTTP server is built on top of. Note that most APR function names are with prefix "`apr_`".

Since none of the tools use function names as recommendation heuristics, it is reasonable to compare results by inspecting whether they are prefixed with "`apr_`". Note that many functions inside the Apache server (rather than APR) are with prefix "`ap_`", which should generally be considered irrelevant to APR functions.

Altair uses default ranking settings described in Section 3, along with two annotations for APR allocation functions, i.e., `apr_palloc` and `apr_pcalloc`.

Figure 8 lists the queries and the corresponding results produced by each tool. FRIAR answers none for first four cases, so we ignore it during those discussions.

First, `apr_file_eof` is a file function that tests the end-of-file indicator. Suade returns an irrelevant function, which is also included in FRAN's result. FRAN further returns `apr_file_read`, a related function with the common prefix "`apr_file_`"; its second result `ap_rputs` is not part of APR but a server function that sends a string to clients, which has nothing to do with file operations. Altair retrieves nine file operation functions (see names `apr_file_*`) and all of them are from the same module of the query.

Similarly, Altair returns more precise and complete results from the network (socket) module for the second query `apr_socket_listen`, a networking function that listens on a socket. Suade returns two non-APR functions. FRAN returns three related functions, while others are not. For example, `ap_log_perror` is used to append an entry in a server's error log, which is not related to networking.

The third query is `apr_collapse_spaces`, a self-contained function that simply strips all spaces in a string. It is not necessarily to be used with any other functions. In this case, Suade and FRAN returns some irrelevant results; Altair returns nothing, as we expected.

The fourth query `apr_pool_terminate` is *not* an API function and should have not be used for comparison. We put it here for completeness purpose only because it appeared in the previous paper. The function is documented as "programs do NOT need to call this directly". It is called internally by APR to clean some *global* state (e.g., a global pool) when APR terminates. Note that the function's name is confusing — actually it does *not* even have any parameter, meaning it will not be used together with pool routines such as `apr_pool_destroy` that destroys a given pool object (see next query). Altair returns three results. The first one, `apr_pool_initialize`, is a similar internal function that initializes global state when APR starts, corresponding to the query `apr_pool_terminate`. The second result cleans up global state, too; it has no parameter either and does a similar job to the query function. The third function allocates a new pool; Altair gets it because the function may access the global pool as the query function does.

The next query `apr_pool_destroy` is a "real" pool API function. Altair returns precisely pool operations. Suade returns an unrelated thread function in the second. The results returned by FRAN and FRIAR include a standard C string function `strlen`, which is not a part of APR.

The last query `apr_hash_get` is a hash table lookup function. In this case Altair returns `apr_hash_*`, clearly from the same module of the query. Other tools return some irrelevant memory operations and miss the expected hash table functions.

Most test cases were adopted from previous work rather than chosen by ourselves, with the hope that the comparison could be more fair. We added the last two cases because otherwise FRIAR would give nothing for all queries. We have further tried many other APR functions. The results show that Altair consistently outperforms previous recommendation tools. Suade and FRAN sometimes recommend irrelevant functions, while FRIAR may simply answer none, e.g., for the first four cases. The overlap-based ranking criterion used in Altair is more reliable; even if some API functions are never called by any code, Altair can still produce remarkably precise and complete results.

We did not do comparison against additional APIs, because the other tools only provided test data for the Apache snapshot. We also recognize that ideally we should compare these tools based on some prior, quantitative standard (if existed), though it is difficult to do so.

7.2 Modular Precision and Recall

In addition to ranking we evaluate whether Altair is able to extract meaningful modules with respect to given queries; other API recommendation tools do not support doing so.

We reuse Apache 2.0.x from the previous subsection, which contains 1,051 public functions. Due to space limitation, we show results of the first two queries: `apr_file_eof` in the "file I/O handling" module and `apr_socket_listen` in the "network routines" module. According the documentation, the two modules have 51 and 54 functions, respectively.

Since the number of module k is unknown, Altair uses the bi-partitioning workaround by tuning threshold ϕ, as discussed in Section 4.3, and finds the module that the given query belongs to.

Let A and B be sets of functions extracted by Altair and documented by Apache for the same module, respectively. We measure precision and recall rates [43] as the threshold ϕ varies, defined as follows.

$$\text{precision} = \frac{|A \cap B|}{|A|} \qquad \text{recall} = \frac{|A \cap B|}{|B|} \tag{7.1}$$

Roughly speaking, precision measures the percentage of extracted functions while recall measures that of documented functions.

As shown in Figure 9, Altair achieves both high precision and recall rates, i.e., 82.2% and 72.5% for the file module, and 93.0% and 74.1% for the network module, given $\phi = 0.2$. The threshold ϕ should be small because the documented modules by Apache are coarse-grained (more than 50 functions in one module). As ϕ gets larger, Altair extracts finedgrained modules that are more related to the query, thus the precision increases and the recall decreases. For example, for query `apr_file_eof` with $\phi = 0.5$, `apr_file_seek` and

	Suade	FRAN	FRIAR	Altair

`apr_file_eof(`
`  apr_file_t *file)`
	`do_emit_plain`	`apr_file_read`
		`ap_rputs`
		`do_emit_plain`
			N/A	`apr_file_seek`
				`apr_file_read`
				`apr_file_dup`
				`apr_file_dup2`
				(... 5 more)

`apr_socket_listen(`
`  apr_socket_t **new_sock,`
`  apr_socket_t *sock,`
`  apr_pool_t *pool)`
	`listen`
	`make_sock`
		`apr_socket_opt_set`
		`apr_socket_close`
		`apr_socket_bind`
		`ap_log_perror`
		`ap_sock_disable_nagle`
		`make_sock`
			N/A	`apr_socket_shutdown`
				`apr_os_sock_get`
				`apr_socket_atmark`
				`apr_socket_send`
				`apr_socket_recv`
				`apr_socket_sendto`
				(... 14 more)

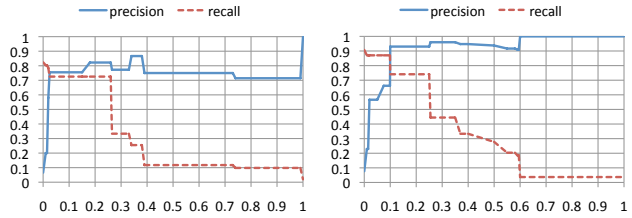
`apr_collapse_spaces(`
`  char *dest,`
`  const char *src)`
	`__ctype_b_loc`	`ap_cfg_getline`
		`read_quoted`
		`get_x_coord`
		`get_y_coord`
		(... 41 more)
			N/A	N/A

`apr_pool_terminate(void)`	`apr_pool_destroy`	`apr_pool_destroy`
		`apr_pool_clear`
		`destroy_and_exit_process`
		`start_connect`
		(... 15 more)
			N/A	`apr_pool_initialize`
				`apr_pool_cleanup_for_exec`
				`apr_pool_create_ex`

`apr_pool_destroy(`
`  apr_pool_t *pool)`
	`apr_allocator_owner_get`
	`apr_thread_exit`
	`apr_pool_terminate`
	(... 24 more)
		`apr_pool_create_ex`
		`strlen`
		`apr_palloc`
		(... 138 more)
			`apr_pool_create_ex`
			`apr_pool_clear`
			`strlen`
			(... 29 more)
				`apr_pool_clear`
				`apr_pool_create_ex`
				`apr_palloc`
				(.. 6 more)

`apr_hash_get(`
`  apr_hash_t *ht,`
`  const void *key,`
`  apr_ssize_t klen)`
	`find_entry`
	`find_filter_def`
	`dav_xmlns...`
	`dav_xmlns...`
	`dav_get...`
	(... 25 more)
		`apr_palloc`
		`apr_hash_set`
		`memcpy`
		`strlen`
		`apr_pstrdup`
		(... 95 more)
			`apr_hash_set`
			`apr_palloc`
			`apr_hash_make`
			`strlen`
			`apr_pstrdup`
			(... 18 more)
				`apr_hash_copy`
				`apr_hash_merge`
				`apr_hash_set`
				`apr_hash_make`
				`apr_hash_this`
				(... 3 more)

Figure 8: A comparison of API recommendation tools.



(a) `apr_file_eof` (b) `apr_socket_listen`

Figure 9: Clustering threshold ϕ (X-axis) with corresponding precision and recall rates (Y-axis) for extracting modules of two APR functions.

Compress	`BZ2_bzCompressInit`
	`BZ2_bzCompress`
	`BZ2_bzCompressEnd`

Decompress	`BZ2_bzDecompressInit`
	`BZ2_bzDecompress`
	`BZ2_bzDecompressEnd`

File operations	`BZ2_bzReadOpen`
	`BZ2_bzRead`
	`BZ2_bzReadClose`
	... (8 in total)

Utility	`BZ2_bzBuffToBuffCompress` `BZ2_bzBuffToBuffDecompress`

Figure 10: The bzip2 API with four modules.

`apr_file_read` were retained in the results since they are more related to the query, while less related functions like `apr_file_dup` were excluded.

Note that the recall rates may be underestimated, because some documented API functions are not available on our testing platform (e.g., `apr_socket_sendfile`). In other words, they are not compiled and analyzed by Altair. Testing on another platform or changing Apache compile configurations may further improve the results.

Besides, with $\phi = 0$ the recall rates may not reach 100%, which means that Altair misses some functions in the modules even when it does not do clustering. One example is `apr_file_remove` in the file module, which simply delegates calls to the system call `unlink` and does not overlap or share state with others. The result poses an interesting issue that one cannot extract the file module given `apr_file_remove`, because it is a *corner* function that does not connect to other file operations in data. On the other hand, our experience shows that starting with a query of either `apr_file_eof` or `apr_file_read` will result in almost the same module, because they are both "core" functions and share common data with other operations in file module.

It is worth noting that simply using function parameter type `apr_file_t` of `apr_file_eof` to group the APR file module does not work. Even another software package Subversion [4] has a number of API functions with `apr_file_t` as parameter types; they are clearly neither part of the file module nor APR.

The clustering results show that Apache (especially APR) is fairly modularized and that Altair can extract modules with acceptable precision and recall rates from a moderate-sized program (1,000+ functions).

7.3 Case Study

This case study uses bzip2 to illustrate how k- and bi-partitioning work for clustering, and to justify our argument that naive grouping of functions is not sufficient.

According to the manual, the bzip2 API contains 16 official functions, which can be split into four modules, as listed in Figure 10. Altair first extracts them from source code and computes pair-wise overlap. It detects one wrapper function, `BZ2_WriteClose`, which simply delegates calls to another API function `BZ2_WriteClose64`. Since the wrapper `BZ2_WriteClose` can be considered to belong to the same module of `BZ2_WriteClose64`, Altair removes the wrapper and continues with the rest 15 functions.

To visualize clustering results we represent each function as a k-dimensional vector, using Y_i generated in the fifth step of the k-partitioning algorithm (see Figure 7).

If the number of modules $k = 4$ is priorly known as input, Altair outputs exactly the same module partitions as in Figure 10. Here we omit the figure since it is difficult to visualize 4-dimensional vectors.

If k is unknown, Altair performs recursive bi-partitioning ($k = 2$) unless $\lambda_2/2$ exceeds some threshold ϕ. The results are shown in Figure 11. The first bi-partition (a) successfully distinguishes seven file operations, the second (b) identifies three decompress functions, and (c) separates three compress functions, leaving two utility functions. The results are consistent with Figure 10, indicating that these API functions are well separated by Altair — our tool even precisely identifies fine-grained compress and decompress functions, which operate on the same structure. Simply grouping function can hardly identify the two modules since they still share a small amount of data.

In addition, we measure $\lambda_2/2$ for bi-partitions (a), (b), (c), and further partitions of the four resulting modules, shown in Figure 11(d). We can see that the values of $\lambda_2/2$ for (a), (b), and (c) are relatively small, indicating that bzip2 API is fairly modularized so that it can be easily partitioned. Interestingly, if we continue partitioning the four resulting modules, $\lambda_2/2$ will increase noticeably. Hence $\phi = 0.4$ is a good stop criterion in this case; modular precision and recall can both reach 100%.

Once having module results, Altair can generate documentations with clustered cross-references, as in Figure 2. The module information can be further used as prerequisite input (e.g., the initial set of functions) to other software engineering tools, such as specification mining tools [6]. For instance, the finite-state automaton of the bzip2 compress API should be as follows.

$$(\texttt{BZ2\_bzCompressInit}; \texttt{BZ2\_bzCompress}^*; \texttt{BZ2\_bzCompressEnd})^*$$

A specification mining tool is expected to learn it from running trace. However, if the trace is *noisy*, e.g., containing some unrelated function `BZ2_bzDecompress`, the resulting automaton would be imprecise [25]. As we have shown in the case study, Altair is able to extract precise modules from structural information and help these tools.

7.4 Quantitative Results

Figure 12 lists benchmarks for quantitative results, consisting of five popular software packages. The line of code (KLOC) is measured over LLVM intermediate representations (IRs), which will not be affected by spaces or comments. V and E are numbers of vertices and edges in augmented access graphs during analysis, respectively. F is the number of functions Altair retrieves for ranking and clustering. Time and memory consumption are measured using the `time` command [5]. We also list allocation functions we manually annotated, at most two for each package.

We do not report analysis and ranking time, because both are less than one second. The clustering time measures the first bi-partition of all functions, only a few seconds for all benchmarks. Moreover, even if we set $\phi = 1$, i.e., to force Altair to exhaustively bi-partition every function into a separate module, clustering of openssl (the most time-consuming benchmark) can finish within 90 seconds. The time can be further reduced when Altair runs on multi-core processors. The first and exhaustive bi-partitions of openssl can finish within two and eight seconds, respectively, on a Linux SMP server with four 2.0 GHz quad-core processors.

Since httpd-2.2.10 has been manually annotated with `@see` in source code, we estimate how much Altair may help it. Of 1,461 documented functions in httpd-2.2.10, 15 (1.0%) are commented with `@see`, while Altair generates cross-references for 895 functions (55.1%). The percentage may be further improved, because Altair reuses the makefile of httpd for analysis and not all Apache modules are compiled. We believe that the coverage rate is acceptable.

In addition, for the 15 functions that are manually annotated with `@see`, Altair's results cover 10 of them. Since by default Altair only relates wrappers to functions they wrap, changing Altair's configuration for wrappers would increase the number to 14, missing only one `@see` that appears to be semantic relevant.

8. RELATED WORK

This section relates Altair to previous work.

API Recommendation

There is a large body of work of API tools that help developers to learn API usages in a convenient way. Prospector [27] is a notable tool that answers queries of how to create certain Java types. It mines code snippets from both API type signatures and client code for reuse. ROSE [49] finds the relation of functions by mining co-changes in version histories of software. Strathcona [14] and XSnippet [41] find relevant code examples from a repository, which focus on how to accomplish a specific task composing several API functions from existing examples. Altair emphasizes related API implementations in the same modules. They could complement and leverage each other.

PR-Miner [22] extracts frequent API patterns from source code. If there is any violation, i.e., an API pair is mostly used together but not at some program point, it indicates a potential bug. As shown in Section 7.1 and in previous work [36], though effective for bug detection, such algorithms might be missing for extracting API cross-references because they depend on specific client code.

The FRAN algorithm [36] extends Suade [35] to recommend API functions using random walks, which shares part of the goal of Altair. As discussed before, both algorithms rank functions based on importance out of call graphs. Altair constructs an access graph to exploit overlap between functions, which can return more precise and complete results in general.

Module Clustering

There is a rich research literature on partitioning software for comprehension and clustering modules to recover the design [8, 16]. Similar techniques have been developed to identify objects in legacy code written in procedural languages

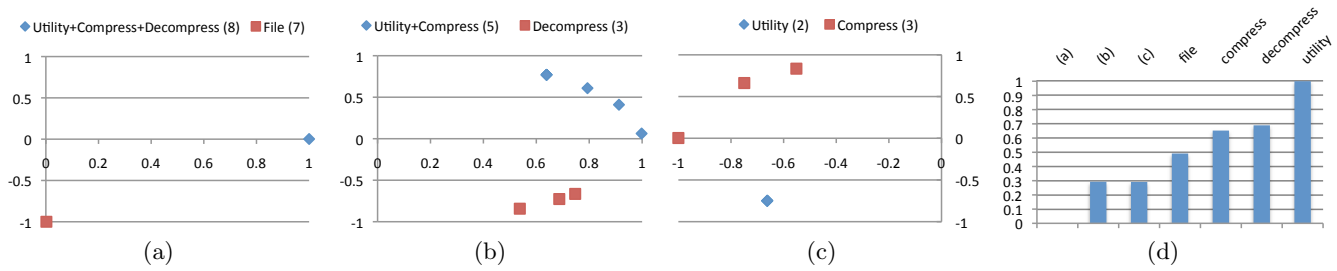


Figure 11: Recursive bi-partitioning of bzip2 API modules, where (a) separates file operations, (b) separates decompress, (c) separates compress from utility functions, and (d) measures $\lambda_2/2$ for each bi-partition. Points (some may overlap) represent functions, the numbers of which are indicated in the parentheses along with their module names.

Package	KLOC	V	E	F	Clustering (sec)	Memory (MB)	Allocation
bzip2-1.0.5	30.0	175	458	16	< 1	4.6	–
sqlite-3.6.5	163.8	2,391	7,873	751	1	55.8	`sqlite3_{malloc, free}`
httpd-2.2.10	256.6	4,061	10,069	1,188	1	109.9	`apr_{palloc, pcalloc}`
subversion-1.5.6	438.8	6,620	17,391	1,632	9	205.1	`apr_{palloc, pcalloc}`
openssl-0.9.8i	553.8	7,197	12,999	3,599	28	374.5	`CRYPTO_{malloc, free}`

Figure 12: Benchmarks. KLOC is measured over LLVM IRs; V and E are the numbers of vertices and edges in augmented access graphs (before computing transitive closure), respectively; F is the number of API functions; we list clustering time (ranking and analysis time is negligible), memory consumption, and allocation functions we manually annotated.

(see [9] for a survey). Even in object-oriented software systems, one may inspect the use relations among classes [18]and cluster them into larger components for visualization. Clustering call graphs [46] is another option.

It remains unclear what criteria may work best for clustering general systems. As for API functions written in C, our analysis seems to work well, since Altair captures pair-wise overlap of both global and local data between functions. In addition, the overlap and modularity measures are easy to interpret.

Concept analysis [38] is also an effective approach to identify modules. There have been debates on whether concept analysis [42] or clustering [34] works better. The main problem of concept analysis is that it may produce meaningless, "tangled" modules [38], especially when two modules cannot be perfectly decoupled, e.g., they share minor state. Spectral clustering used by Altair is more flexible and does not have this limitation.

Spectral clustering was first introduced in the 1970s [12]. As discussed in Section 4.2, it matches our intuition of modules well, imposes no further requirements over measures, and usually outperforms other methods. Our experiments show that it can be computed efficiently using linear algebra libraries. Sometimes it is referred as normalized cut [37] in the area of computer vision and machine learning.

Program Analysis

Altair conservatively assumes that two variables with the same type may alias and takes the advantage for a faster computation. It may adopt other more precise and expensive pointer analysis [33].

Altair computes data that each function f may access, The information can be used to extract data fields that are often used together, e.g., a buffer and its length, which is useful for discovering consistency and concurrency bugs [26].

Structural overlap is related to program slicing [44, 15], though the formulation is different. To compute overlap between functions, one should track shared data fields and ignore values that simply flow from one function to another via assignment. Besides, Altair computes overlap in an open program rather than in a whole program.

Another interesting technique is automatic program partitioning. Swift [11] partitions a web application into client and server under some security constraints, and performs a minimum cut to reduce network communication between them. Flowcheck [29] also computes a minimum cut to estimate a program's potential privacy leaks via dynamic analysis. Altair instead uses a normalized cut for clustering, because using minimum cut would result in small, meaningless leaf sets.

9. CONCLUSION

To automate generation of API function cross-references, Altair computes pair-wise overlap and rank related functions accordingly. It outperforms previous API recommendation tools by exploiting reliable structural information, which is not sensitive to specific client code. Altair also helps developers understand software structures by decomposing them into meaningful modules using spectral clustering. It would be interesting to collect user behavior for guiding cross-reference generation in the future.

10. ACKNOWLEDGMENT

We would like to thank Alvin Cheung, Frans Kaashoek, Xu Liu, Xuezheng Liu, Yiming Liu, David Sontag, Fan Sun, Linchun Sun, Zhilei Xu, Chenke Yang, Junfeng Yang, Nickolai Zeldovich, and Wujie Zheng for their useful feedback and helpful discussions, and the anonymous reviewers for their insightful comments.

11. REFERENCES

[1] Apache HTTP server. `http://httpd.apache.org/`.

[2] bzip2. `http://www.bzip.org/`.

[3] doxygen. `http://www.doxygen.org/`.

[4] Subversion. `http://subversion.tigris.org/`.

[5] `time` – time command execution. `http://www.manpages.info/freebsd/time.1.html`.

[6] G. Ammons, R. Bodìk, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[7] I. Antonellis, H. Garcia-Molina, and C.-C. Chang. Simrank++: Query rewriting through link analysis of the click graph. In *VLDB*, 2008.

[8] L. A. Belady and C. J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 2(1):23–29, 1981.

[9] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software — Practice & Experience*, 26(1):25–48, 1996.

[10] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[11] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[12] F. R. K. Chung. *Spectral Graph Theory*. AMS, 1997.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[14] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, 2005.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.

[16] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, 1985.

[17] D. F. Huynh, D. R. Karger, and R. C. Miller. Exhibit: Lightweight structured data publishing. In *WWW*, 2007.

[18] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *ICSE*, 2003.

[19] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. In *KDD*, 2002.

[20] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *SODA*, 1997.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[22] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, 2005.

[23] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM SIGPLAN Symposium on Very High Level Languages*, 1974.

[24] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[25] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE*, 2006.

[26] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[27] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI*, 2005.

[28] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[29] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, 2008.

[30] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE*, 2000.

[31] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2001.

[32] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Helsinki University of Technology, 1995.

[33] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. In *PASTE*, 2004.

[34] S. Phattarsukol and P. Muenchaisri. Identifying candidate objects using hierarchical clustering analysis. In *APSEC*, 2001.

[35] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE*, 2005.

[36] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *ESEC/FSE*, 2007.

[37] J. Shi and J. Malik. Normalized cuts and image segmentation. In *CVPR*, 1997.

[38] M. Siff and T. Reps. Identifying modules via concept analysis. In *ICSM*, 1997.

[39] H. Small. Co-citation in the scientific literature: A new measure of the relationship between two documents. *Journal of the American Society for Information Science*, 24(4):265–269, 1973.

[40] S. Srivastava, M. Hicks, and J. S. Foster. Modular information hiding and type-safe linking for C. In *TLDI*, 2007.

[41] N. Tansalarak and K. Claypool. XSnippet: Mining for sample code. In *OOPSLA*, 2006.

[42] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, 1999.

[43] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 2nd edition, 1979.

[44] M. Weiser. Program slicing. In *ICSE*, 1981.

[45] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002.

[46] S. Xanthos. Clustering object-oriented software systems using spectral graph partitioning. ACM Student Research Competition, 2005.

[47] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[48] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage pattern. In *ECOOP*, 2009.

[49] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, 2004.