Flow-Augmented Call Graph: A New Foundation for Taming API Complexity

Qirun Zhang, Wujie Zheng, and Michael R. Lyu

Computer Science and Engineering
The Chinese University of Hong Kong, China
{qrzhang,wjzheng,lyu}@cse.cuhk.edu.hk

Abstract. Software systems often undergo significant changes in their life cycle, exposing increasingly complex API to their developers. Without methodical guidances, it is easy to become bogged down in a morass of complex API even for the professional software developers. This paper presents the Flow-Augmented Call Graph (FACG) for taming API complexity. Augmenting the call graph with control flow analysis brings us a new insight to capture the significance of the caller-callee linkages in the call graph. We apply the proposed FACG in API recommendation and compare our approach with the state-of-the-art approaches in the same domain. The evaluation result indicates that our approach is more effective in retrieving the relevant APIs with regard to the original API documentation.

Keywords: API Recommendation, Static Analysis, Control Flow Analysis.

1 Introduction

Software systems often undergo significant changes during the in-service phrase of their life cycle [7]. Most of contemporary software systems are becoming larger with exposing increasingly complex API to developers [8]. A recent survey conducted in Microsoft Research reveals that software developers usually become lost working on projects with complex API, unsure of how to make progress by selecting the proper API for a certain task [15]. Previous literature [6,20] points out that working with complex APIs in large scale software systems presents many barriers: understanding how the APIs are structured, selecting the appropriate APIs, figuring out how to use the selected APIs and coordinating the use of different APIs together all pose significant difficulties. Facing with these difficulties along, developers spend an enormous amount of time navigating the complex API landscape at the expense of other value-producing tasks [16].

A methodical investigation of the API usage in large software systems is more effective than an opportunistic approach [2]. The API relevance is usually considered to tame the API complexity. Two APIs are relevant if they are often used together or they share the similar functionality. According to previous literature [19], the original API documentation which groups the APIs into modules is the best resource to indicate the API relevance. However, few projects provide the insight to capture the relevant APIs in their documentation [8]. With the need exposed, recommendation systems specific to software engineering are emerging to assist developers [13]. Recommending relevant

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 386–400, 2011.

APIs (namely, API recommendation) in software systems is also a long-standing problem that has attracted a lot of attention [4,8,9,14,19,22]. Generally, there are two fundamental approaches in API recommendation concerned with data mining techniques and the other with structural dependency. The mining approaches emphasize on "How API is used" and extract frequent usage patterns from *client code*. The structural approaches, on the other hand, focus on "How API is implemented" and recommend relevant APIs according to structural dependencies in *library code*.

Previous work on API recommendation heavily relies on the call graph. The call graph is a fundamental data representation for software systems, which provides the first-hand evidence of interprocedural communications [17]. Especially for the work concerned with API usage, the impact of the call graph is critical. The call graph itself does not imply the significance of callees to the same caller. However, the callees are commonly invoked by a caller under various constrains. For example, the callees enclosed with no conditional statement will be definitely invoked by the caller, whereas the callees preceded by one or more conditional statements are not necessarily invoked. Previous approaches relying on the call graph do not distinguish the differences between callees.

Our approach takes a different path from the previous work by extending the very foundation in API recommendation—the call graph. We introduce the Flow-Augmented Call Graph (FACG) that assigns weights to each callee with respect to the control flow analysis of the caller. The key insight of the FACG is that a caller is more likely to invoke a callee preceded by less conditional statements. Thus the bond between them is stronger comparing to the others preceded by more conditional statements. Therefore, the significance of caller-callee linkages can be inferred in the FACG. The insignificant callees in the FACG can be eliminated, so that the API complexity can be reasonably reduced with respect to specific software tasks. In this work, we employ the FACG in API recommendation and conduct our evaluation on several well-known software systems with three state-of-the-art API recommendation tools.

This paper makes the following contributions:

- We propose the Flow-Augmented Call Graph (FACG) as a new foundation for taming API complexity. The FACG extends the call graph by presenting the significance of caller-callee linkages in the call graph.
- We apply the FACG in API recommendation and evaluate our approach on several well-documented software projects. We employ their module documentation as a yardstick to judge the correctness of the recommendation. The evaluation indicates our approach is more effective than the state-of-the-art API recommendation tools in retrieving the relevant APIs.
- We implement our API recommendation approach as a scalable tool built on GCC. Our tool copes with C projects compliable with GCC-4.3. All the supplemental resources are available online[1].

The rest of the paper is organized as follows. Section 2 describes the motivating example. Section 3 presents the approach to build FACG and recommend relevant APIs.

[1] `http://www.cse.cuhk.edu.hk/~qrzhang/facg.html`

```
1  APR_DECLARE(apr_status_t) apr_pool_create_ex(...)
2  {
3      ...
4      if ((node = allocator_alloc(allocator, MIN_ALLOC - APR_MEMNODE_T_SIZE)) == NULL)
5      {
6          ...
7      }
8      ...
9  #ifdef NETWARE
10     pool->owner_proc = (apr_os_proc_t)getnlmhandle();
11 #endif /* defined(NETWARE) */
12     ...
13     if ((pool->parent = parent) != NULL)
14     {
15 #if APR_HAS_THREADS
16         ...
17         if ((mutex = apr_allocator_mutex_get(parent->allocator)) != NULL)
18             apr_thread_mutex_lock(mutex);
19 #endif /* APR_HAS_THREADS */
20         ...
21 #if APR_HAS_THREADS
22         if (mutex)
23             apr_thread_mutex_unlock(mutex);
24 #endif /* APR_HAS_THREADS */
25     }
26     ...
27     return APR_SUCCESS;
28 }
```

Fig. 1. `apr_pool_create_ex()` in Apache

Section 4 compares our approach with three state-of-the-art tools. Section 5 summaries the previous work. Section 6 conducts the conclusion.

2 Motivating Example

We motivate our approach by selecting a real world API `apr_pool_create_ex()`[2] from the latest Apache HTTP server-2.2.16. Consider the code snippet shown in Fig. 1, if we investigate the control flow of this API, we may obtain two interesting observations. First, the call-site of API `getnlmhandle()` at line 10 is subject to the macro `NETWARE`[3], and this API will never be called by `apr_pool_create_ex()` on the platforms other than Netware®. Second, the call-site of `allocator_alloc()` at line 4 is unconditional, whereas `apr_thread_mutex_lock()` at line 18 is subject to two conditions. As a result, `allocator_alloc()` is much more likely to be called by `apr_pool_create_ex()` than `apr_thread_mutex_lock()`. However, in the conventional call graph shown in Fig. 2(a), the callees are identical to the caller. These observations reveal that the conventional call graph is blind to the significance among different callees, and we are likely to miss some critical information if we treat every callee as the same in a call graph.

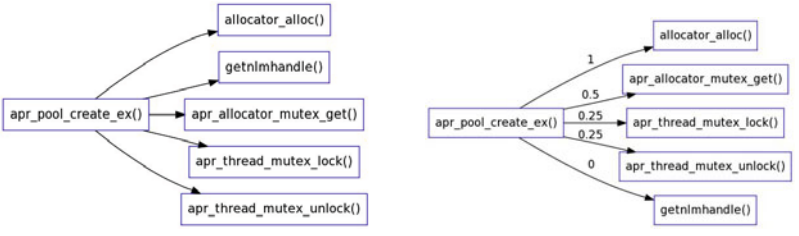
In this paper, we propose the Flow-Augmented Call Graph (FACG), which aims to address the limitations of the conventional call graph. We build the FACG with respect

[2] The irrelevant code is omitted.

[3] Netware is a network operating system developed by Novell, Inc. Find more on `http://httpd.apache.org/docs/2.0/platform/netware.html`

Table 1. Flow distributions of each API in the previous example

API name	Description	Flow
apr_pool_create_ex	Creating a new memory pool.	
allocator_alloc	Allocate a block of mem from the allocator	1
apr_allocator_mutex_get	Get the mutex currently set for the allocator	0.5
apr_thread_mutex_lock	Acquire the lock for the given mutex.	0.25
apr_thread_mutex_unlock	Release the lock for the given mutex.	0.25
getnlmhandle	Returns the handle of the NLM owning the calling thread.	0



(a) Call Graph (b) Flow-Augmented Call Graph

Fig. 2. Call Graph and Flow-Augmented Call Graph

to the fact that some of the call-sites are unconditional while others are conditional; some are concerned with more conditions while the others are with fewer conditions. By applying the control flow analysis, we observe that the unconditional call-sites occupy the caller's major control flow, whereas the call-sites under more conditions tend to reside in a less important sub branch. It is more possible for the caller to invoke the callee with fewer or without conditions. In order to cope with individual call-sites, we split the control flow equally for every branch in the control flow graph. For the motivating example in Fig. 1, we initialize the inflow by 1. The final distribution of the control flow for each callee is shown in Table 1. Especially, `getnlmhandle()` is eliminated since our analyzer is built on GCC Gimple IR, where all of the macros have been preprocessed by the compiler. Finally, we extract the description of each API from Apache Http Documentation[4] to interpret the insight beyond our FACG. As shown in Table 1, among the five callees, API `allocator_alloc()` is more likely to accomplish "creating a new memory pool" than other less important APIs (e.g., `apr_thread_mutex_lock()`) in this case. The FACG shown in Fig. 2(b) indicates that the linkage of `apr_pool_create_ex()` and `allocator_alloc()` is the most significant one among all potential caller-callee pairs.

3 Approach

3.1 Augmenting the Call Graph with Control Flow

Parsing Source Code. The GCC compiler is chosen as the backbone parser. Our static analyzer takes the advantage of Gimple [10] Intermediate Representation and is able to

[4] `http://apr.apache.org/docs/apr/1.4/modules.html`

capture essential information (e.g., basic block, API call-site and structure accessing) from the source code. The current implementation only works for C; however, it can be easily extended to other languages through different GCC front ends.

Reducing CFG. We adopt the definition of CFG from those presented by Podgurski and Clarke [11].

Definition 1. *A* Control Flow Graph *(CFG)* $G = (N, E)$ *for procedure* P *is a directed graph in which* N *is a set of nodes that represent basic blocks in procedure* P*.* N *contains two distinguished nodes,* n_e *and* n_x*, representing ENTRY and EXIT node, where* n_e *has no predecessors and* n_x *has no successors. The set of N is partitioned into two subsets,* N^S *and* N^P*, where* N^S *are statement nodes with each* $n_s \in N^S$ *having exactly one successor, where* N^P *are predicate nodes representing predicate statements with each* $n_p \in N^P$ *has two successors*[5]*.* E *is a set of directed edges with each* $e_{i,j}$ *representing the control flow from* n_i *to* n_j *in procedure* P*. All nodes in* N *are reachable from ENTRY node* n_e*.*

In the common case that the CFG is reducible [21], eliminating loop back-edges results in a DAG and this can be done in linear time[1]. For the irreducible CFG, we adapt the conservative approximation from [12] and unroll every loop exactly once. This is done at early stage so that the DAG instead of CFG is considered in building the FACG.

Calculating the Flow of Callees. For a callee Q, if a path with flow x contains Q, we say Q has flow x along the path, otherwise Q has flow 0 along the path. The flow of Q in the caller is the sum of the flow of Q along all the paths. To calculate the flow of Q, a naive strategy is to employ an exhausted graph walking strategy to collect the flow of callees in each path. However, this is infeasible as there are an exponential number of paths [12]. We propose an approach to calculate the flow of Q incrementally. We define the inflow and outflow of each basic block n_i as the following:

Definition 2. *The inflow of a basic block is defined as:*

$$IN(n_i) = \begin{cases} \sum_{e_{j,i} \in E} OUT(n_j) & if(n_i \neq n_e) \\ n_0 & if(n_i = n_e) \end{cases} \tag{1}$$

Definition 3. *The outflow of a basic block is defined as:*

$$OUT(n_i) = \begin{cases} IN(n_i) & if(n_i \in N^S) \\ \dfrac{IN(n_i)}{2} & if(n_i \in N^P) \end{cases} \tag{2}$$

The inflow $IN(n_i)$ denotes the flow of all the paths arriving the basic block n_i (n_i is counted in the paths), and the outflow $OUT(n_i)$ denotes the flow of all the paths

[5] As indicated in [11], the outedge of each n_i in CFG is at most two. This restriction is made for simplicity only.

Algorithm 1. Algorithm to determine the flow of each callee in the DAG

Input : $G(V, E)$,directed, acyclic CFG of procedure P;
V is topologically sorted;
the set of Q, where Q is the callee of P;
Output: $Q.Flow = IN(n_x)_Q$ for each callee Q;
foreach *callee* Q *of procedure* P **do**
  foreach $n_i \in V$ **do**
    foreach $n_j \in PRED(n_i)$ **do**
      $IN(n_i) \leftarrow IN(n_i) + OUT(n_j)$;
      $IN(n_i)_Q \leftarrow IN(n_i)_Q + OUT(n_j)_Q$;
    end
    if n_i *has call-site of* Q **then**
      $IN(n_i)_Q \leftarrow IN(n_i)$;
    end
  end
  $Q.Flow \leftarrow IN(n_x)_Q$
end

arriving a successor of n_i through n_i. We also use $IN(n_i)_Q$ and $OUT(n_i)_Q$ to denote the inflow and outflow of n_i associated with a callee Q. $IN(n_i)_Q$ denotes the flow of Q in all the paths arriving n_i, and $OUT(n_i)_Q$ denotes the flow of Q in all the paths arriving a successor of n_i through n_i.

Definition 4. *The inflow of a basic block associated with Q is defined as:*

$$IN(n_i)_Q = \begin{cases} \sum_{e_{j,i} \in E} OUT(n_j)_Q & if(n_i \neq n_e \text{ and } n_i \text{ does not contain } Q) \\ 0 & if(n_i = n_e \text{ and } n_i \text{ does not contain } Q) \\ IN(n_i) & if(n_i \text{ contains } Q) \end{cases} \tag{3}$$

Definition 5. *The outflow of a basic block associated with Q is defined as:*

$$OUT(n_i)_Q = \begin{cases} IN(n_i)_Q & if(n_i \in N^S) \\ \dfrac{IN(n_i)_Q}{2} & if(n_i \in N^P) \end{cases} \tag{4}$$

In order to calculate the flow information effectively, we first rank the basic blocks using topological sorting. For n_e, we calculate $IN(n_e)_Q$ according to the definition. We then calculate the inflow of the basic blocks associated with Q in the order of topological sorting. If n_i contains Q, $IN(n_i)_Q$ is determined by $IN(n_i)$. Otherwise, $IN(n_i)_Q$ is the sum of the inflow of the predecessors of n_i associated with Q, which has been calculated. Finally, the flow of Q is equal to the inflow of n_x associated with Q, i.e., $IN(n_x)_Q$. The overall algorithm is shown in Algorithm 1.

Augmenting the Call Graph. We initialize the inflow of each procedure P by 1 (i.e., $n_0 = 1$), and propagate the flow in the CFG . Upon the completion of Algorithm 1, we

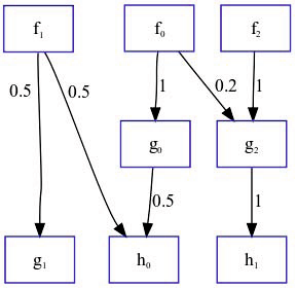


Fig. 3. The FACG Example

augment every caller-callee edge according to $Q.Flow$ for each callee Q to build the FACG.

In the FACG, the significance of each caller-callee linkage can be indicated on each edge in the call graph. We revise the definition of the call graph [3] to give a formal definition of FACG.

Definition 6. *A* Flow-Augmented Call Graph *(FACG)* $G = (N, E)$ *for procedure* P *is a directed multigraph in which each node* $n \in N$ *corresponds to either a caller* P *or a callee* Q*, and each weighted edge* $e \in E$ *represents a call-site augmented with control flow.*

3.2 Recommending the Relevant APIs

The relevant APIs have similar functionalities, and thus they may access some program elements (i.e., the callee APIs and the structures) in common. Therefore, we adopt the set of program elements accessed by the APIs to recommend relevant APIs. However, an API may access many program elements (along with its callees), most of which are irrelevant to the main functionality of the API. These irrelevant elements can easily dominate the relevance calculation of APIs, and introduce noises to recommendation results[6]. To reduce the impact of the irrelevant elements, one may consider only the elements directly accessed by the APIs in the conventional call graph (For example, Saul *et al.* [19] consider only the neighboring functions of an API as the candidates for recommendation). But many relevant APIs for a API query may be far from the query in the call graph, and are missed by such kind of approaches.

Despite the difficulty of selecting a representative set of program elements accessed by an API from the conventional call graph, the task is feasible using the FACG. Given the FACG, the *significant* callees of a caller API can be found with regard to the flow-augmented edge. The representative set of program elements of an API is determined along with its *significant* callees. Basically, if a callee is called with a large flow in the FACG, it is considered to be the *significant* callee. Note that the flow can be propagated along the FACG, and a callee that is called indirectly by an API can be significant to the API as well. For example, in Fig. 3, f_0 calls g_0 with flow 1, and g_0 calls h_0 with

[6] We further discuss this in section 4.4.

Table 2. Subject Project

Software	Version	KLOC	#C files	#Functions
Httpd	2.2.16	299.7	571	2188
D-Bus	1.1.3	99.2	108	1608
Tcl	8.5.9	227.1	207	1880
Tk	8.5.9	260.1	201	2303

flow 0.5, therefore, the flow of f_0 to h_0 is $1 * 0.5 = 0.5$. If we use 0.5 as the threshold of flow for determining *significant* callees, then the *significant* callees of f_0, f_1, and f_2 are: g_0, h_0 of f_0; g_1, h_0 of f_1; and g_2, h_1 for f_2.

We then calculate the relation of two APIs as the cosine similarity of their representing vectors. Cosine similarity can capture the similarity of two vectors without biasing to vectors with large norm, and it is widely used in text retrieval. The cosine similarity of two vectors is

$$cosine(f, g) = \frac{f \cdot g}{||f|| \cdot ||g||} \tag{5}$$

For the example shown in Fig. 3, the relation of f_0 and f_1 is 0.5, and the relation of f_0 and f_2 is 0 (for simplicity of presentation, we omit the structures accessed by the APIs). While it is difficult to distinguish the relevance of f_1 and f_2 to f_0 in the conventional call graph, it is clear that f_1 is more relevant to f_0 than f_2 is in the FACG, since f_1 and f_0 have some main functionalities in common. We thus recommend f_1 as a highly relevant API of f_0.

4 Evaluation

We compare the proposed approach with three state-of-the-art API recommendation tools: Suade [14], Fran [19] and Altair [8]. Suade recommends a set of API by analyzing the specificity and reinforcement, Fran performs a random walk algorithm in the call graph to find relevant APIs. Altair suggests the recommendation based on API's internal structural overlap. Our evaluation over the four tools is conducted with regard to the specific task suggested by Fran [19]: *Given a query API, retrieve other APIs in the same module*. The subject projects in our evolution section are Apache HTTP Server, Tcl/Tk library, and D-Bus message bus system. These subject projects are chosen because they are documented well, and the original API documentation which groups the APIs into modules is the best resource to tell the API relevance. Table 2 gives the basic description on the subject projects.

4.1 Experimental Setup

The experiments are conducted on an Intel Core 2 Duo 2.80GHz machine with 3GB memory and Linux 2.6.28 system. Suade, Altair, and Fran are freely available online. As mentioned in Altair [8], Suade is not initially designed for API recommendation; we use a re-implementation from Fran [19]. Fran proposes two algorithms,namely, FRAN

Table 3. A Comparison of API Recommendation Tools. Curly underline indicates a matched recommendation.

	Suade	Fran	Altair	our approach
apr_os_file_get	N/A	N/A	apr_file_writev pipeblock pipenonblock proc_mutex_posix_cleanup proc_mutex_posix_acquire proc_mutex_posix_release proc_mutex_sysv_cleanup proc_mutex_sysv_acquire proc_mutex_sysv_release proc_mutex_fcntl_cleanup	apr_os_pipe_put apr_os_pipe_put_ex apr_file_pool_get apr_os_file_put apr_file_buffer_size_get apr_file_close apr_file_ungetc apr_unix_child_file_cleanup apr_file_name_get apr_file_open_stderr
apr_fnmatch	tolower rangematch make_autoindex_entry	apr_palloc apr_pstrdup strlen apr_pstrcat ap_make_full_path ap_make_dirstr_parent find_item toupper memset ignore_entry	N/A	rangematch pcre_maketables apr_uri_unparse apr_fnmatch_test ap_str_tolower atoq strip_paren_comments ap_filter_protocol apr_match_glob is_token
Tcl_SetVar2	Tcl_SetVar Tcl_SetVar2Ex EnvTraceProc TclpSetVariables Tcl_NewStringObj Tcl_GetString TclFreeObj	Tcl_SetVar TclpSetVariables EnvTraceProc Tcl_ExternalToUtfDString getuid uname __ctype_b_loc Tcl_DStringInit TclpGetPwUid Tcl_DStringFree	ObjFindNamespaceVar Tcl_FindNamespaceVar TclLookupSimpleVar TclObjLookupVarEx TclObjLookupVar TclLookupVar Tcl_ObjSetVar2 Tcl_SetVar2Ex Tcl_SetVar Tcl_ObjGetVar2	Tcl_SetVar2Ex Tcl_UnsetVar2 Tcl_GetVar2Ex Tcl_GetVar2 Tcl_UpVar2 TclVarErrMsg TclLookupVar Tcl_SetVar TclObjLookupVar Tcl_FindNamespaceVar

and FRIAR. They concludes that a combination of the two algorithms can achieve better performance. We use their implementation of the combined algorithm in our experiment, denoted as Fran to avoid the naming confusion. Suade and Fran need to be initialized with a call graph. We feed them with the call graph extracted by our implementation based on Gcc Gimple IR. Altair is built on LLVM, which can gracefully handle the source code. Moreover, Fran implements the top-k precision/recall measurement, we adopt the result to calculate the F1 score for Fran and Suade in our evaluation.

4.2 Case Studies

Case study is *de rigueur* in evaluating the result obtained by API recommenders [19]. Suade, Fran and Altair used human examination [14,19] and API naming [8] (concerned with the prefix `apr_` and `ap_` in Apache HTTP server). However, without convincing ground truth to support the judgement, these case studies have several limitations, which have been well-discussed in Fran [19] (which conducted an additional quantitative study as a supplement).

We sought to judge the result objectively and bring about fair comparisons among four tools. As suggested in Fran [19], the original project documentation which groups the APIs into modules is the best resource to judge the API relevance. Therefore, we take the module content as a yardstick to avoid subjective judgement on the correctness of relevant APIs and underline the relevant APIs that appear in the module in Table 3. Table 3 shows the recommendation set obtained by the four tools with respect to the queries list below. The cases are chosen in order to indicate the typical situations in the related projects.

Case 1: `apr_os_file_get()` is a function in the Apache Portable Runtime (APR) which the Apache HTTP server is built on top of. The APR documentation indicates that this API belongs to the Portability Routines module[7] and its functionality is to "convert the file from apr type to os specific type". This API is not directly called by Apache HTTP server, yet it exports the interface for developers to extend Apache HTTP server. In this case, Suade and Fran return no result with regard to this query because it is not in the call graph. Among the top 10 results returned by Altair, there is no relevant APIs according to the documentation. We investigate the source code and find that `apr_os_file_get()` is a simple function with two lines of code and accesses only one data structure `apr_status_t`. Altair computes the structural overlap among APIs; however, the useful information available for this query is limited and there are many APIs, which only access `apr_status_t`. Altair cannot distinguish the difference among them and returns the irrelevant results. Our approach investigates the structral information with the help of FACG and records how data structures are accessed by the APIs explicitly. Take `pipeblock()` in Altair's result for example, this API accesses `apr_status_t` in different branches with regard to the control flow, where our approach is capable of distinguishing this case from a single access as in `apr_os_file_get()`. Among the top 10 results obtained by our approach, three APIs can be found in the documentation which are identified to be relevant to the query shown by a curly underline.

Case 2: `apr_fnmatch()` is a member of the Filename Matching Module in Apache HTTP server, which is described in the documentation as "to match the string to the given pattern"[8]. This query is a *self-contained* API, which simply manipulates the strings without accessing any data structures. As discussed in Altair [8], Altair may not return any result for these *self-contained* API. However, `apr_fnmatch()` is called by many other APIs in Apache HTTP server, and the documentation indicates that there are two APIs `apr_finatch_test()` and `apr_match_glob()`, that are relevant to it. Suade and Fran attempt to answer the query by searching the call graph. Since the call graph is not able to tell the significance of each callees to the caller. The result returned by Suade and Fran implies that those approaches rely on the conventional call graph may "get lost" in the API jungle because all the neighbour nodes in the call graph appear to be "the same". Our approach, on the other hand, only considers the callee APIs on the major flow of the caller rather than those less important ones. Within our FACG, such explorations in the API jungle can be directed to the caller/callee more relevant to the query. In the end, our approach finds both of the other two APIs in the module according to the top 10 result.

Case 3: `Tcl_SetVar2()` in Tcl library belongs to the group of APIs that manipulate Tcl variables[9]. All of the four tools return meaningful results with regard to the query. `Tcl_SetVar2()` is widely used in Tcl to create/modify the variable, consequently, it has large neighbour sets (i.e., parent, child, sibling and spouse set defined by Fran). Both of Suade and Fran return `Tcl_SetVar()` which is a wrapper function of the query.

[7] `http://apr.apache.org/docs/apr/1.4/group__apr__portabile.html`
[8] `http://apr.apache.org/docs/apr/1.4/group__apr__fnmatch.html`
[9] `http://www.tcl.tk/man/tcl8.5/TclLib/SetVar.htm`

Table 4. The Precision and Recall Performance

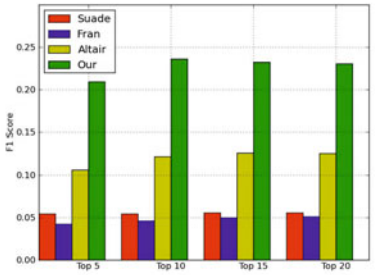
		Suade	Fran	Altair	Our approach
Precison	Top-5	0.111	0.155	0.266	0.384
	Top-10	0.104	0.134	0.236	0.319
	Top-15	0.109	0.135	0.222	0.278
	Top-20	0.109	0.135	0.212	0.252
Recall	Top-5	0.025	0.072	0.099	0.173
	Top-10	0.027	0.094	0.136	0.242
	Top-15	0.028	0.103	0.154	0.283
	Top-20	0.029	0.109	0.163	0.314
F1 Score	Top-5	0.033	0.070	0.114	0.181
	Top-10	0.034	0.075	0.132	0.209
	Top-15	0.035	0.080	0.133	0.213
	Top-20	0.035	0.081	0.132	0.213

Suade also returns `Tcl_SetVar2Ex()` which is called by the query. Most of the rest APIs in their results has nothing to do with variable manipulation. The situation is quite the same as in case 2; the conventional call graph does not distinguish the differences between callees. The top 10 results obtained by Altair and our approach are all related to variable manipulation in Tcl. This can be confirmed by taking look at the naming of these APIs. However, the relevant APIs (which are listed in the documentation and are the ground truth of F1 comparison) rank higher in our result. The reason behind is our approach supported by FACG is likely to consider the most important neighbours in the call graph. For example, query `Tcl_SetVar2` calls four APIs. Among them, `Tcl_SetVar2Ex` occupies the main flow of the query, so that it ranks high in the result. The result precisely illustrates the main advantage of the FACG. Moreover, in our top 10 results, we retrieve six out of the nine APIs that are documented in the same module, which is the best result from all four tools.

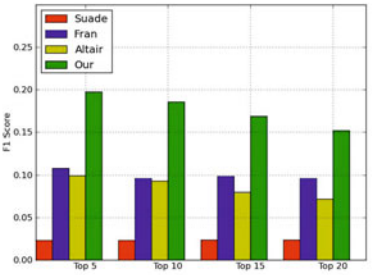
4.3 Quantitative Study

To perform the quantitative study, we compared the effectiveness of the four tools in retrieving relevant APIs at four recommendation-set size cutoffs, top-5, top-10, top-15 and top-20. Three measures ($precision$, $recall$ and the F1-measure) of performance in information retrieval are adopted in our evaluation. All are defined by the recommendation set retrieved by the four tools. Let A be the recommended set obtained by each tools, and B be the set of relevant APIs which appear in the module. The $precision$ and $recall$ is defined as follows: $precision = |A \cap B|/|A|$ and $recall = |A \cap B|/|B|$. Precision measures the accuracy of obtaining the relevant APIs while recall measures the ability to obtain the relevant APIs. The F1-measure is the equally-weighted harmonic mean of the precision and recall measures, defined as $F = 2 * presion * recall/(presion + recall)$. It is usually engaged as the combined measure of both precision and recall.

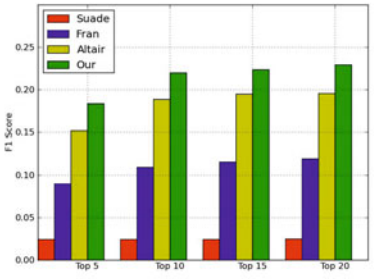
The summary of precision and recall performance is shown in Table 4. It can be seen that our approach achieves the highest precision. Moreover, our approach improves the precision rate over Suade, Fran and Altair by 184.8%, 120.6% and 31.7% respectively,



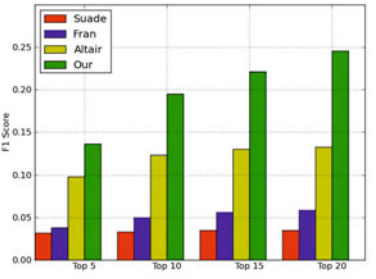
(a) F1 Comparison on Tcl-8.5.9 Library



(b) F1 Comparison on Tk-8.5.9 Library



(c) F1 Comparison on Apache-2.2.16



(d) F1 Comparison on D-Bus-1.1.3

Fig. 4. Overall F1 Score Comparison

which indicates that our approach is able to suggest the most precise recommend set among the four. In addition, our approach achieves recall improvement over Suade, Fran and Altair by 828.4% , 167.7% and 83.3% respectively. Finally, the overall performance measurement is determined by F1-Score, where our approach achieves the highest F1 score with a large improvement of 495.6%, 166.7% and 59.7% over Suade, Fran and Altair respectively. The performance measurement indicates that our approach is able to recommend relevant APIs much more effectively than all other tools. Fig. 4 shows the F1 score comparisons over all the subject projects in our experiment. It is clearly seen that our approach dominates the performance in all recommendation-set size cutoffs.

4.4 Discussion on the Impact of FACG

The main insight in our work is to deploy the FACG to address the significance of caller-callee linkages. We conduct this subsection investigating our recommendation algorithm on the conventional call graph to further illustrate the impact of the FACG.

We apply our algorithm on the conventional call graph which treats every caller-callee linkage identically. Table 5 shows the F1 score of top-20 recommendation sets compared with our FACG approach on the four subject projects. On average, the performance of recommendation using the FACG is 41.7% higher than using the conventional call graph. More specifically, Table 6 lists the top 10 results of the case study in

Table 5. F1 Score of Top-20 Result

Subject Project	Httpd	D-bus	Tcl	Tk
Approach relying on the call graph	0.16	0.17	0.17	0.10
Approach relying on the FACG	0.23	0.24	0.23	0.15

Table 6. Top-10 results of our recommendation algorithm on the call graph

Query	apr_os_file_get	apr_fnmatch	Tcl_SetVar2
Top-10 Result	apr_file_pool_get apr_file_buffer_size_get apr_file_ungetc apr_file_name_get apr_file_buffer_set apr_file_flush_locked apr_unix_child_file_cleanup database_cleanup apr_file_unlock apr_os_pipe_put_ex	rangematch find_desc make_parent_entry apr_match_glob ap_file_walk ap_location_walk ap_process_request_internal ap_process_resource_config include_config dummy_connection	Tcl_SetVar Tcl_SetVar2Ex Tcl_TraceVar2 Tcl_ResetResult Tcl_ObjSetVar2 TclGetNamespaceForQualName Tcl_TraceVar TclObjLookupVarEx EstablishErrorInfoTraces Tcl_DeleteNamespace

section 4.2. The relevant APIs supported by the module documentation are underlined with a curly line as well. As mentioned before, the recommendation algorithm based on the conventional call graph is blind to the difference between callees; therefore, it searches more candidates than our FACG approach, without distinguishing the significance of each candidate. Take `Tcl_SetVar2` for example, the first two results are directly linked with the query in call graph, thus they rank on the top in Table 6. However, although other relevant APIs (e.g., `Tcl_GetVar2`) appear in the candidate set, their significance is not clear enough in the call graph to be distinguished from other insignificant ones in the top 10 result. Moreover, the approach based on the call graph introduces some APIs (e.g., `TclGetNamespaceForQualName`) irrelevant to variable manipulation into the top 10 result, whereas the FACG approach can properly filter them as shown in Table 3. The evaluation demostrates the benefit of the FACG in capturing the essence concerned with API usage and the impact of applying the FACG in API recommendation.

5 Related Work

There are mainly two categories of API recommendation approaches. The approaches which recommend APIs by using mining techniques belong to the first category. These approaches usually mine certain patterns or code snippets from sample code repositories. Prospector [9] developed by Mandelin *et al.* synthesizes the Jungloid graph to answer a query providing the input and output types. Prospector traverses the possible paths from input type to output type and recommends certain code snippets according to API signatures and a corpus of the client code. XSnippet [18] developed by Sahavechaphan *et al.* extends Prospector by adding more queries and ranking heuristics to mine code snippets from a sample repository. Context-sensitive is introduced to enhance the queries in XSnippet and produce more relevant results. Strathcona [4] developed by Holmes *et al.* is dedicated to recommending code examples matching the structural context. Six heuristics are applied to obtain the structural context description in the stored repository. MAPO [24] developed by Zhong *et al.* takes the advantage of mining frequent usage patterns of an API with the help of code search engines.

PARSEWeb [22] developed by Thummalapenta *et al.* mines open source repositories by using code search engines as well. Different from MAPO, PARSEWeb accepts the queries of the form "Source type⇒Destination type" and suggests the relevant methods that yield the object with the destination type.

The approaches in the second category aim at recommending APIs with respect to structural dependency. Zhang *et al.* propose a random-walk approach [23] based on PageRank algorithm to rank "popular" and "significant" program elements in Java programs. Inoue *et al.* proposed another approach [5] inspired by PageRank algorithm, which can be employed to rank valuable components in software systems based on the use relations. Suade [14] developed by Robillard is focused on providing suggestions for aiding program investigation. It accomplishes the suggestion by ranking the desired program elements concerned with the topological properties of structural dependency in software systems. Fran [19] developed by Saul *et al.* extends the topological properties by considering neighbouring relationships in the call graph. Altair [8] developed by Long *et al.* recommends the relevant APIs according to the overlap of commonly accessed variable information.

To the best of our knowledge, all of the previous API recommendation approaches rely on the conventional call graph. By distinguishing the significance of caller-callee linkages, the proposed FACG improves the accuracy of recommending relevant APIs.

6 Conclusion

This paper presents the Flow-Augmented Call Graph (FACG) to tame the API complexity. Augmenting the call graph by control flow analysis brings us a new foundation to capture the significance of caller-callee linkages. We employed API recommendation as a client application and engaged the FACG to retrieve the relevant APIs. We further conduct the experiment on four large projects with original documentation as ground truth to judge the performance, and compared our approach with three other state-of-the-art API recommendation tools. The case studies and quantitative evaluation results indicate our approach is more effective in retrieving the relevant APIs.

Acknowledgments. The authors would like to thank the anonymous reviewers for their insightful feedback. We would also like to thank Jie Zhang and Jianke Zhu with the writing. This research was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/10E).

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Princiles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. DeMarco, T., Lister, T.: Programmer performance and the effects of the workplace. In: ICSE, pp. 268–272 (1985)
3. Hall, M., Hall, M.W., Kennedy, K., Kennedy, K.: Efficient call graph analysis. ACM Letters on Programming Languages and Systems 1, 227–242 (1992)

4. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 117–125. Springer, Heidelberg (2006)
5. Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S.: Component rank: Relative significance rank for software component search. In: ICSE, pp. 14–24 (2003)
6. Ko, A.J., Myers, B.A., Aung, H.H.: Six learning barriers in end-user programming systems. In: VL/HCC, pp. 199–206 (2004)
7. Lehman, M.M., Parr, F.N.: Program evolution and its impact on software engineering. In: ICSE, pp. 350–357 (1976)
8. Long, F., Wang, X., Cai, Y.: API hperlinking via structural overlap. In: ESEC/SIGSOFT FSE, pp. 203–212 (2009)
9. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: PLDI, pp. 48–61 (2005)
10. Merrill, J.: Generic and Gimple: A new tree representation for entire functions. In: Proceedings of the 2003 GCC Developers Summit, Citeseer, pp. 171–179 (2003)
11. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering 16, 965–979 (1990)
12. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: ICSE, pp. 240–250 (2007)
13. Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software 27(4), 80–86 (2010)
14. Robillard, M.P.: Automatic generation of suggestions for program investigation. In: ESEC/SIGSOFT FSE, pp. 11–20 (2005)
15. Robillard, M.P.: What makes APIs hard to learn? answers from developers. IEEE Software 26(6), 27–34 (2009)
16. Robillard, M.P., Coelho, W., Murphy, G.C.: How effective developers investigate source code: An exploratory study. IEEE Trans. Software Eng. 30(12), 889–903 (2004)
17. Ryder, B.G.: Constructing the call graph of a program. IEEE Trans. Software Eng. 5(3), 216–226 (1979)
18. Sahavechaphan, N., Claypool, K.T.: XSnippet: mining for sample code. In: OOPSLA, pp. 413–430 (2006)
19. Saul, Z.M., Filkov, V., Devanbu, P.T., Bird, C.: Recommending random walks. In: ESEC/SIGSOFT FSE, pp. 15–24 (2007)
20. Stylos, J., Myers, B.A.: Mica: A web-search tool for finding API components and examples. In: VL/HCC, pp. 195–202 (2006)
21. Tarjan, R.E.: Testing flow graph reducibility. In: STOC, pp. 96–107 (1973)
22. Thummalapenta, S., Xie, T.: PARSEWeb: a programmer assistant for reusing open source code on the web. In: ASE, pp. 204–213 (2007)
23. Zhang, C., Jacobsen, H.A.: Efficiently mining crosscutting concerns through random walks. In: AOSD, pp. 226–238 (2007)
24. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: mining and recommending api usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009)