

Fex: Assisted Identification of Domain Features from C Programs

Patrick Müller*, Krishna Narasimhan*, and Mira Mezini*

mueller, kri.nara and mezini (@cs.tu-darmstadt.de)

* TU Darmstadt, Germany

Abstract—Modern software typically performs more than one functionality. These functionalities or features are not always organized in a way for modules representing these features to be used individually. Many software engineering approaches like programming language constructs, or product line visualization techniques have been proposed to organize projects as modules. Unfortunately, much legacy software suffer from years or decades of improper coding practices that leave the modules in the code almost undetectable. In such scenarios, a desirable requirement is to identify modules representing different features to be extracted. In this paper, we propose a novel approach that combines information retrieval and program analysis approaches to allow domain experts to identify slices of the program that represent modules using natural language search terms. We evaluate our approach by building a proof of concept tool in C, and extract modules from open source projects.

I. INTRODUCTION

Large-scale software projects typically consist of a composition of domain features, which often also come in different variants. For instance, an embedded software system that manages a modern automobile is typically composed of features such as the multimedia system, tyre pressure monitors, etc. To manage the complexity of development and evolution of such software, it is desirable that domain features are modularized in well-defined code modules and their dependencies and variants are systematically modelled in product-lines [1].

Unfortunately, when the need to manage or restructure existing software into a product line arises, it is often very challenging to do so, due to one of the following reasons: (a) the system has been developed without a product line in mind, or (b) inappropriate language constructs are used to structure the program into modules. In case (a), we need to identify code that pertains to the implementation of individual domain features. Code that implements a single feature can be spread across multiple files inside the project.¹ Berger et al. [2] conducted an industrial study on the significance and difficulties of identifying features in large-scale projects. In case (b), we refer to preprocessor directives like `#ifdef` that the C programming language offers to modularize software along the domain features. The C preprocessor has been found to be notoriously harmful owing to obtrusive syntax and hindrance to comprehension and maintainability [3]. Some approaches have been proposed to improve comprehension of projects organized using preprocessors, most notably the work

from Le et al. [4] or `PEoPL` [5]. But, they require the code to be well-structured using `ifdefs` in the first place. Moreover, given that most modern languages have done away with the preprocessor pragmas, we need a better solution that avoids preprocessors altogether.

In this paper, we propose a novel approach to identifying code that pertains to the implementation of a domain feature of interest, as a prerequisite to re-engineering legacy software into first-class feature modules and compositions thereof. Program slicing is a common static program analysis technique to extract sections of code that depend on a particular piece of data (variables in code). But, domain experts may not be able to determine variables to serve as starting points for extracting a module. To bridge this gap, we develop a program analysis technique that synergistically combines information retrieval (IR) with control- and data-flow analyses, thus yielding what we call a *natural code analysis* approach. Based on this analysis infrastructure, we implement a slicing technique that takes natural language terms as input. We use the term *slicing criteria* to represent the input provided by the user, which in our approach is comprised of the feature anchor and a similarity threshold. The former serves as input search criteria which the user provides as a natural language term. The latter is a value between 0 and 1 that determines how close the terms in the program should be to the input term. One can think of the similarity threshold as a slider that the domain expert can control to arrive at the desired module.

More specifically, our information retrieval creates a program *Corpus* - an index that maps terms to locations in the program - and internally maintains a map of the corpus to the control flow graph (CFG) of the program. We slice the corpus based on the slicing criteria and use the map to identify nodes in the CFG that correspond to this corpus slice. Given the identified CFG nodes, we apply data-flow analysis to gather their control and data dependencies. The result is a module that contains the code that corresponds to the input term - representing a domain feature. Our approach is not intended to replace manual effort, i.e., to be fully automatic. On the contrary, we believe that iterative input from domain experts is a crucial aspect to the module extraction process. The goal is to assist the process by relieving the human expert from the heavy lifting.

Program analysis and slicing techniques are generally driven by data or program variables. To the best of our knowl-

¹In the rest of the paper, we will sometimes use the word `module` to refer to code scattered throughout the project that represent a feature.

edge, this is the first approach that performs feature-anchored program analysis by embedding information retrieval into domain-driven program slicing.

To understand better where such a feature identification approach could be useful as a first step to re-engineer modules, consider the following scenarios:

- **Grbl:**² is a high-performance controller for CNC milling, a cutting tool that is mounted on a rotating spindle to selectively remove material from a dedicated workpiece. The axis module, that is concerned with the control of motors inside the mill are currently deeply tied into an ocean of C-code making it hard to re-use or analyse independently. Our running example in this paper is a simplified snippet of the grbl code illustrating how our approach can identify the subset of the controller concerned with axis-related tasks.
- **Language transpiling:** is the task of transforming sections of a software from one language into another. Such transpilation is useful with the advent of safe languages like Rust which serve as an alternative to unsafe languages like C. Companies like Mozilla are currently transpiling their C-based projects into Rust. But they do this incrementally on a feature-level to avoid a big-bang transpiling that could consume too much time and human resources. An approach such as ours could help alleviate the pain of identifying the pieces of code concerned with the feature that needs to be transpiled. As part of our evaluation, we have extracted a module from a popular grep alternative called Silver-Searcher, which is equivalent to a module transpiled into Rust and available as a pull request in the project upstream.
- **Linux GKI:** or the Generic Kernel Image is a proposal from the Linux core team to create a single kernel image that will be used in all devices and providers to encourage re-use. Currently, there exist multiple provider-specific kernels. In order for such a project to manifest, providers will have to identify and extract code specific to their unique features that have been now merged into their provider-specific kernels. An approach such as ours could alleviate the pain involved in such laborious identification of features.

We evaluated our approach on six open source C projects: **Parson**, **inotify**, **fping**, **silver-searcher**, **redis** and **memcached**, ranging from 3k to 165k lines of code, from which we extracted eight different modules in total. We apply the tool that implements our approach, **Fex** to these benchmarks to extract modules that represent significant features, e.g., the module that parses json files in the Parson project. We compare the results against a set of modules that we extracted manually based on detailed analysis and understanding of the code-base to serve as a ground truth.

We put several measures in place to make sure that manually extracted modules represent meaningful program features.

To start with, the extracted modules do not break existing functionality – existing test-cases pass.

Second,

two of the authors with three respectively four years of experience independently agreed that the manually extracted modules represent program features. Third, one of our extracted modules is externally validated because it corresponds to a module for which there is an ongoing pull request for transpiling it from C into Rust. We compare the results of **Fex** with an extraction using *grep*.

Our results show that the extracted modules

are close to the ground truth modules in terms of matching lines of code.

For deviating lines of code, we perform a thorough manual analysis to understand the root causes and to provide insights on how the gap could be closed. As the application of our tool on **redis** – the largest benchmark in our set (165k LoC) – shows, the approach scales well to code written for real-world software.

To sum up, the contributions in this paper are as follows:

- A novel information retrieval (IR) approach to work on C files like text documents so that they can be queried using natural language. Given a C program (a set of C files) one can use the IR technique to create a corpus of the project, which is persisted for subsequently performing queries based on terms for re-engineering purposes.
- A novel program slicing technique for C that makes use of natural language terms as slicing criteria and uses the corpus built by the IR technique to identify the code elements pertaining to a slice.
- A tool that implements our approach, **Fex** which we use to extract modules from open-source C projects with very encouraging results.

The remainder of the paper is organized as follows.

We present our methodology in Section II and our evaluation experiments and results in Section III. We discuss threats to validity and outline future work in Section IV, related work in Section V and conclude in Section VI.

II. APPROACH

An overview of the workflow supported by our approach is depicted in Figure 1. Given some legacy software, the extractor creates the project corpus (cf. Section II-A), which is stored in a database to ensure that the extractor is not required to be re-run for every new query. Once the corpus is ready, the user is prompted for a keyword representing the desired feature in domain terms, which is used to filter the corpus (Section II-B). Finally, the feature extractor maps the filtered slice of the corpus to syntactically correct code that represents the implementation of the desired feature (Section II-C).

A. Creating the corpus

The creation of the corpus is the responsibility of the **Corpus Extractor**. Its input is a C project with a set of source files - the output is the `corpus` for the input project.

²<https://github.com/gnea/grbl>

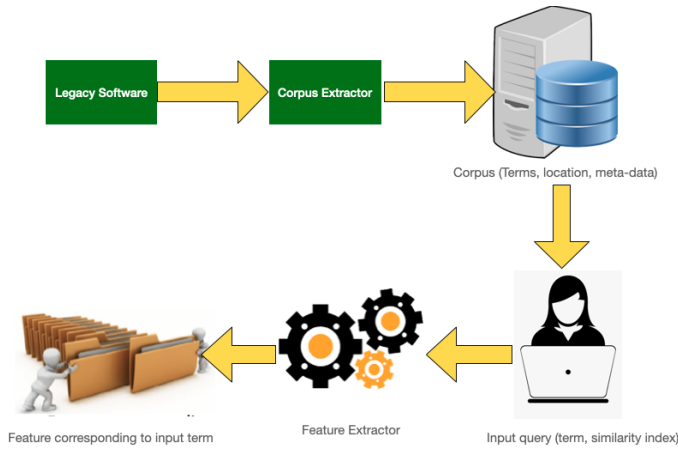


Fig. 1: Approach Overview

The corpus is a matrix that maps terms occurring in code to the following attributes of them extracted from the code:

- **Locations** where the term occurs (line, column)
- **Program context** in which the term occurs (comment, preprocessor macro, identifier, function definition)
- **Weight** - determines how important a term is within each function in code. There is one weight entry for each pair of function and term. Hence, weight is in fact a matrix itself - the Term Document Matrix (TDM). There are different strategies to determine the weight of a term in a function - the most simple one is to count the number of occurrences a given term has in a function.

A **Term document matrix (TDM)** [6] is a matrix that represents how frequently a term occurs in a document. The rows of the matrix represent individual documents and the columns represent the terms. There are various methods to decide what values are represented in the matrix, the most obvious one being to encode the number of occurrences of each term. Consider the two documents:

- 1) D1 = "Time heals everything"
- 2) D2 = "Time cures everything"

The TDM for the above document would be:

	Time	heals	cures	everything
D1	1	1	0	1
D2	1	0	1	1

In natural language processing, a popular alternative to the "number of occurrences" to describe the frequency is to use the **Term frequency-inverse document frequency**³ or in short **TFIDF**, which we use in our corpus as **weight**. This is a statistic that represents the importance of a term in a document. This value is directly proportional to the number of times a term occurs in a document, and the number of documents that contain this term. In order to query for a term you have to create a query vector, which has 1 at the index

³<https://nlp.stanford.edu/IR-book/html/htmledition/queries-as-vectors-1.html>

that corresponds to the input term and 0 in all other indices, e.g., $(\dots 0 1 0 \dots)$.

As one can imagine, the size of the TDM can get quite large for a big project with lots of terms. For this purpose, **Latent Semantic Indexing (LSI)** [7] is an indexing and retrieval technique that groups terms that are semantically equivalent into concepts, and finds the connections between these terms and concepts. LSI then uses a matrix decomposition technique called **Single value decomposition (SVD)** [8] to reduce the size of the matrix based on the relationship between the terms and the concepts.

To create the corpus for the given project, the extractor performs the following steps:

- 1) **Tokenize the source code:** We use a lexer to tokenize the source files. This will allow us to manage the individual words in our project.
- 2) **Produce the documents:** We remind the reader that our information retrieval process requires the source files to be available as documents. To enable retrieving information at a fine granularity level, we map source files into several documents, rather than having one document per file. Documents represent one of several different constructs in the original source code, i.e., a single function is a document, and all declarations in one source file are a document on their own.
- 3) **Normalize the tokens:** We perform a post-processing on the tokens to normalize them, which results in *terms*. During the normalization we transform all tokens to lower case and split them for different naming styles, e.g., snake case or camel case, while also retaining the original version. For example, the token (*parse_axisCommand*) gets normalized to the terms (parse, axis, command, axisCommand, and parse_axiscommand). In addition, we retain the program context for all term occurrence locations, i.e., whether the term is an identifier, is guarded by a preprocessor macro, or occurs within a comment.
- 4) **Shorten the corpus:** We then reduce the size of the resulting corpus by filtering out C keywords.
- 5) **Create the TDM:** As already mentioned, the TDM maps pairs of documents and terms, (d, t) to a number n . What n represents depends on the chosen weighting scheme. In our default case of TFIDF, n corresponds to the relative importance of t inside the document d . For the sake of simplicity, in Table I, we show the TDM for a single function (i.e., the TDM is reduced to a single column *Weight*)
- 6) **Reducing the matrix:** For big projects with many functions, the corpus can become quite large. Latent Semantic indexing⁴ uses a matrix factorization technique called Single value decomposition (SVD) to reduce the size of a matrix. The resulting matrix exists in what is called the SVD space.

⁴<https://nlp.stanford.edu/IR-book/html/htmledition/latent-semantic-indexing-1.html>

```

1 void parse_command(char* input) {
2   axis_command = NULL;
3   if (input[0] == 'G') {
4     // mm or inches
5     unit = parse_unit(input);
6     axis_command =
7       parse_axis_command(input);
8     mode = 0;
9     move_x(axis_command);
10    move_y(axis_command);
11  } else if (input[0] == 'H') {
12    coolant();
13  } else { FAIL(UNSUPPORTED_COMMAND); }
14  if (axis_command) {
15    do_command(mode);
16  }
17  return;
18 }

```

Fig. 2: Example Code

TABLE I: Corpus for running example, Locations are tuples of (Line \times Column \times Context), where Context is i for Identifier, c for Comment and m for preprocessor Macro

Term	Locations	Weight <small>parse_command</small>
axis_command	(2,3,i), (6,5,i), (7, 7,i), (9,12,i), (10,12,i), (14,7,i)	0.78
move_y	(10,5,i)	0.30
parse	(1,6,i), (5,12,i), (7,7,i),	0.60
unsupported_command	(13,17,m)	0.30
move_x	(9,5,i)	0.30
input	(1,26,i), (3,7,i), (5,23,i), (7,26,i), (11,14,i)	0.78
fail	(13,12,m)	0.30
coolant	(12,5,i)	0.30
axis	(2,3,i), (6,5,i), (7, 7,i), (9,12,i), (10,12,i), (14,7,i)	0.85
command	(1,12,i), (2,3,i), (6,5,i), (7,7,i), (9,12,i), (10,12,i), (13,17,i), (14,7,i), (15,5,i)	1.00
parse_axis_command	(7,7,i)	0.30
null	(2,18,i)	0.30
do_command	(15,5,i)	0.30
unsupported	(13,17,i)	0.30
parse_unit	(5,12,i)	0.30
move	(9,5,i), (10,5,i)	0.48
mm	(4,8,c)	0.30
inches	(4,14,c)	0.30
unit	(5,5,i), (5,12,i)	0.48
mode	(8,5,i), (15,16,i)	0.48
parse_command	(1,6,i)	0.30

LSI also takes care of mapping the input query on the full TDM to the SVD space.

For illustration, consider the code in Figure 2 and its corresponding corpus in Table I. In contrast to a TDM, this shows the corpus information corresponding to a term in a single row, i.e., a transposed TDM, and occupies only one column, because our example code contains only one function (i.e., there is only one document). In a typical case, the TDM column will have multiple sub-columns, one for each document, indicating whether it contains the term. The most frequently occurring term in the example is **command**, which has the highest TDM entry value and the least occurring terms have a TDM entry value of **0.30**.

TABLE II: Relevant TDM entries for query "axis"

axis_command	(2,3), (4,49), (6,12), (7,12), (11,7)
axis	(2,3), (4,49), (4,64), (6,12), (7,12), (11,7)
parse_axis_command	(4,64)

The process of creating the corpus can be quite demanding for a large project. Computing the SVD for a corpus is time consuming, too. To avoid reoccurrence of these overheads, we persist the Corpus in a serialized Java object, so that we compute the corpus and its SVD only once for a program and can re-use it for every new query.

B. Extracting a corpus slice

To query the corpus for slices representing some feature, the user provides a tuple (*term*, *similarity-threshold*).

The term here corresponds to the string that likely represents the feature of interest.

We envision that the domain expert will have to explore different permutations of input terms and similarity thresholds before arriving at his intended result. For every input query, a `similarity score` is computed for each document. This score measures how similar the input term is to the document; how this value is calculated depends on the IR model in use.

We

filter out all documents with a similarity score less than the threshold. In the resulting documents, we gather all terms related to the input term, i.e., the terms where the input term is a substring.

For illustration, assume that we query the corpus of the example in Figure 2 for the term *axis*. We construct the initial query vector, which has 21 entries - one for each term in the corpus Table I, with a one in each position involving the query term (*axis*) and zeros for the rest. This query vector is then mapped to the SVD space of the corpus in this concrete example, using LSI. The similarity score for *axis* in our case for the only document (*function parse_command*) is 1.0, which means that the document makes the cut for the next step. Next, we gather all related terms, ending up with the slice of the corpus in Table II.

Overall, the corpus extraction⁵ works in two phases. In the first phase, we create the corpus as described in II-A. In the second phase, we use a given term to retrieve the corpus slice as described in II-B. This phase can be repeated several times for one program, e.g., because several terms can be used to describe a single feature. The way we construct and query the corpus is modular, so that we can use different IR methods, like LSI or VSM. We have implemented our corpus extractor in a fashion that the same format can be extracted out of grep or any other text search tool, so that results from other tools can be used as input for feature extraction, as well.

C. Extracting the feature

The final step is to use the corpus slice from the previous step to reconstruct a syntactically correct code slice comprising

⁵It is implemented in Kotlin.

Input: CS = Corpus Slice as List,
CFG = Control flow graph of program (LLVM representation)
Internal Variables: RELEVANT_STMTS

Procedure Main

1. for entry<TERM, LSTLOCATION> ∈ CS
2. for LOCATION ∈ LSTLOCATION
3. NODE_TERM ← node corresponding to TERM in LOCATION
4. STMT_TERM ← Statement surrounding NODE_TERM
5. RELEVANT_STMTS.add(STMT_TERM)
6. for STMT ∈ RELEVANT_STMTS
7. ProcessStatement(STMT)
8. IFDS
9. RETURN Program consisting of all Statements in RELEVANT_STMTS

Procedure ProcessStatement(STMT)

10. for VARREFEXPR ∈ STMT
11. DECLARATION ← Definition of VARREFEXPR in CFG
12. RELEVANT_STMTS.add(DECLARATION)
13. for FUNCCALLEXP ∈ STMT
14. DEFINITION ← Definition of FUNCCALLEXP in CFG
15. RELEVANT_STMTS.add(DEFINITION)
16. if STMT is a return statement
17. LSTASSIGNEDSTATEMENTS ← Statements that consume the return value of STMT
18. RELEVANT_STMTS.add(DEFINITION)
19. if STMT ∈ BLOCK
20. RELEVANT_STMTS.add(CFG.start(BLOCK))
21. RELEVANT_STMTS.add(CFG.end(BLOCK))
22. STMT.processed ← true
23. for STMT_UNPROCESSED ∈ RELEVANT_STMTS
24. where STMT_UNPROCESSED.processed != true
25. ProcessStatement(STMT_UNPROCESSED)

Fig. 3: Outline of the code slicing algorithm

the implementation of the desired feature. The feature extractor is responsible for the following:

- 1) Use locations from the terms in the filtered corpus as initial input
- 2) Extract a slice of the program that contains the statements in the locations from step 1 along with their control and data-dependencies

The feature extractor algorithm is presented in Figure 3.

- 1) In Lines 1-5, we mark all statements that correspond to retrieved locations as relevant.
- 2) In Line 7, we process each statement to find its dependencies
- 3) In Lines 10-12, we mark definitions of every variable reference inside the processed statement as relevant.
- 4) In Lines 13-18, we perform similar markings to handle flows between function call sites and definitions.
- 5) In lines 19-21, we mark the start and end of blocks to ensure syntactic completion of the sliced CFG.
- 6) Finally, we mark the statement as processed and continue till all relevant statements are processed.

We illustrate this process on the code of Figure 2, with the search term *Axis*.

- 1) Mark the statements in lines 2, 6, 7, 9, 10, 14 since they are contained in our locations retrieved from the corpus. These line numbers are the first entry of each tuple of the column **Locations** corresponding to the terms from the sliced corpus.

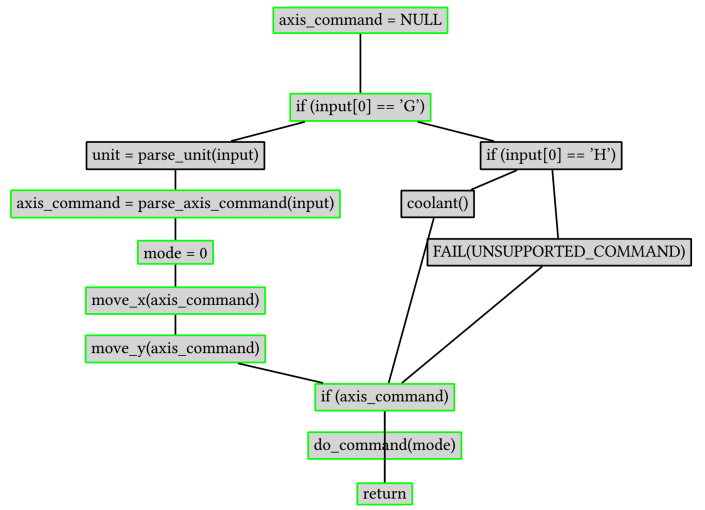


Fig. 4: CFG of LLVM IR of parse_command

```

1 void parse_command(char* input) {
2   axis_command = NULL;
3   if (input[0] == 'G') {
4     axis_command =
5     parse_axis_command(input);
6     int mode = 0;
7     move_x(axis_command);
8     move_y(axis_command);
9   } else if (input[0] == 'H') {
10    coolant();
11  } else { FAIL(UNSUPPORTED_COMMAND) }
12  if (axis_command) {
13    do_command(mode);
14  }
15 }

```

Fig. 5: Example Code- Sliced

- 2) We add line 15 since it is in the body of the if in line 14 and the if including its condition in line 8
- 3) We add line 3 since we add statements that are controlled by the if.
- 4) We collect all block endings, where a statement is marked, i.e., the closing braces in 11,16 and 18. In addition add the function head i.e. Line 1.

The CFG of the function parse_comand in Figure 2 is illustrated in Figure 4, where the relevant nodes are highlighted in green. The feature extractor results in the code in Fig. 5.

The feature extractor is based on Phasar [9] and is implemented in C++. Phasar implements the IFDS framework to solve inter-procedural, finite, distributive subset (IFDS) problems introduced by Reps, Horwitz and Sagiv [10]. Given the IFDS framework, designers of data-flow program analyses need only define a set of flow functions, which the framework solves. In order to define a program analysis using Phasar, we need to implement the following four functions ⁶.

- 1) The analysis developer uses the **getNormalFlowFunction** function to express what happens during intra-

⁶<https://github.com/secure-software-engineering/phasar/wiki/Writing-an-IFDS-analysis>

procedural data flows. For our implementation, we mark every intra-procedural data flow dependency of a relevant node as relevant.

- 2) The analysis developer uses the **getCallFlowFunction** function to express what happens when a call-site is encountered, i.e., how to handle the relationship between the actual parameters and the formal parameters of a function. For our implementation, we mark every formal parameter of an actual relevant parameter as relevant
- 3) The analysis designer uses the **getRetFlowFunction** function to indicate how to handle the flow from a return statement to the variable the return value is assigned to. For our implementation, we mark the variable where the value of a relevant return statement is assigned, as relevant.
- 4) The analysis designer uses the **getCallToRetFlowFunction** to indicate how facts flow around a call site, e.g. facts that are not modified by the callsite itself but still have to be propagated. For our implementation, we accordingly propagate all facts that still are relevant after the call statement.

Our feature creator compiles the original source code to an SSA-compliant LLVM-IR including debug information. Single static assignment (SSA) [11] is a popular attribute of any intermediate representation of code, that emphasizes that each variable be assigned only once. The advantage is that optimizations are simplified and analyzing properties of variables are facilitated. Phasar provides an SSA representation of the LLVM IR, which we use to make our analysis more efficient. We mark the initial locations returned by our corpus extractor as relevant and initiate the IFDS solver, which uses the flow function to mark the relevant control and data dependencies. We augment this with other necessary control dependencies that are not included by our slicing algorithm a.k.a unconditional jump instructions. We then use the LLVM debug information to retrieve the original source code of all marked instructions to produce the sliced version of the original source code.

III. EXPERIMENTS

We validated **Fex** by applying it to extract modules from a set of real C programs ranging between 5k and 165k lines of code. We present the setup of the experiment and its results in the following.

A. Methodology

We queried Github for C-projects. We consider a C-project any project with a significant portion of the code written in C (>80% lines of C-code).

We sorted all C-projects returned by the above query by the number of stars and obtained a list of candidate projects. We read descriptions from project websites searching candidates with interesting modules and identified six such projects. For each project, one of the authors with three years of C experience manually identified modules; another author with four years of C experience independently validated them. In

the case of a disagreement, both authors agreed on a solution. The manually identified modules serve as our ground truth. For each of them we defined representative search terms that capture the intention. The projects and the extracted modules are:

- 1) *parson*⁷: json library, 5439 lines of non-header code. We identified two modules to extract from *parson*, one module concerned with JSON parsing and one module concerned with JSON serialization.
- 2) *inotify-tools*⁸: Collection of tools to watch for filesystem events, 4741 lines of non-header code. We defined a module concerned with handling of internal stats, e.g., the number of file access events.
- 3) *fping*⁹: A variant of ping, 2261 lines of non-header code. For *fping* we identified a module concerned with handling of internal stats, e.g., response times respectively.
- 4) *silver searcher*¹⁰: Code search, 5085 lines of non-header code. The module we extracted is responsible for filtering filenames using regular expressions.
- 5) *memcached*¹¹: Key/value store mainly used for caching, 24262 lines of non-header code. For *memcached* we defined a stats handling module, similar to *fping* and *inotify-tools*, as well as a module for the handling of commands that the webserver accepts.
- 6) *redis*¹²: In memory database with a web-api, 165159 lines of non-header code. In the case of *redis* we defined a module that is concerned with the handling of the cluster-manager mode of redis' commandline interface.

We extracted each ground truth module into its own file and added corresponding header files. Hence, in some cases we had to remove or add the *static* modifier to reduce or increase the visibility of functions and variables. The manually extracted modules are "able to be merged", i.e., no conflicts with the base repository and that the same unit tests (including any CI checks from the repository) pass, as they did before our refactoring.

The external validity of our ground truth for **Silver searcher** is implicitly ensured, as it was defined equivalent to a code module on which a transpilation to Rust was performed; there has been a pull request for the transpiled Rust version of the module (cf Section D)¹³.

To facilitate the extraction of code from larger modules, we introduced the notion of an inter-procedural distance limit to constraint how far through call edges our tool looks up. We limited all of our experiments to two call edges, except for *redis* where we deactivated the inter procedural analysis.

We applied **Fex** using the search terms (see Table III) of the ground truth modules and compared automatically retrieved modules with the ground truth ones. For comparison, we

⁷<https://github.com/kgabis/parson>

⁸<https://github.com/inotify-tools/inotify-tools>

⁹<https://github.com/schweikert/fping>

¹⁰https://github.com/ggreer/the_silver_searcher

¹¹<https://github.com/memcached/memcached>

¹²<https://github.com/redis/redis>

¹³https://github.com/ggreer/the_silver_searcher/pull/1418

performed a set level difference between the lines from the tool extracted output and from the ground truth. Intersecting lines are considered to be correctly extracted; those in the ground truth but not in the extracted modules are considered missing lines; those in the extracted modules but not in the ground truth are considered additional. We do not keep track of comments and exclude those from our comparison.

We perform manual analysis on each of these differing sets to gather insights on what sort of manual intervention, or future support could be necessary. In addition we use *grep* to generate a baseline extraction to which we can compare.

Table III summarizes the results.

Each line lists the project, the extracted module, and the used search term in the first three columns.¹⁴ We used a similarity threshold of 0.85 for all of the extractions. The column **lines** shows the overall number of LoCs in the module and - in parentheses - unique LoCs after excluding empty ones and comments. The meaning of **extracted correctly**, **missing**, and **additional** is self explanatory, we display these values for both **Fex** and the tool *grep*. All extractions complete in less than 3 minutes except for one outlier which took 34 minutes. Our validation consists in (a) a comparison with *grep*, (b) an analysis of lines that are missing from the extracted modules in comparison to the ground truth, (c) an analysis of lines that get extracted, but are not part of the ground truth, i.e. additional lines, and (d) an discussion of the impact of chosen search terms.

B. Comparison with *grep*

We compare the results of our slicing approach with the regular expression search tool *grep*. To the best of our knowledge, *grep* is the only other well known tool, with which we can apply natural language search for sub-sets of C code. Both *INFOX* [23] and *CLUSTERCHANGES* [12] are not comparable, since they focus and rely on changes/changesets e.g. from forks, whereas **Fex** takes only the current state of a project/repository as input. For all the projects and modules we used the exact same search terms, as with **Fex**. The results can be seen in Figure III. In order to make the results comparable we removed all comments from the *grep* results, since we are mainly concerned with the executable/compilable source code.

While **Fex** achieved a recall for extracted module between 62.67% and 99.37%, the alternative of using *grep* as a starting point to identify modules from C-programs achieved only a recall between 1.4% and 18.3%

C. Analysis of missing lines of code

Figure 7 elaborates on the missing lines for each of the expected modules relative to the number of lines that were correctly extracted and on the respective root causes.

First, we highlight that all but one (**fping-statistics**) extracted modules are close to the ground truth. The portion

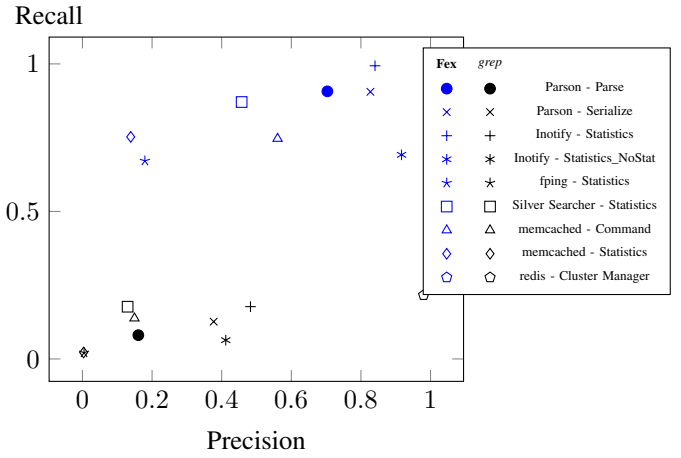


Fig. 6: Comparison of Precision & Recall for **Fex**(in blue) and *grep* (in black)

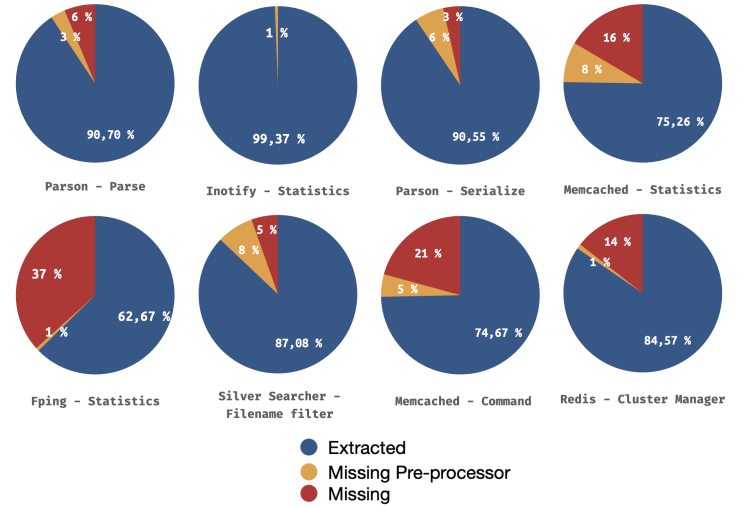


Fig. 7: Missing lines relative to properly extracted ones

of the correctly extracted lines (blue) constitutes roughly 75% – 99% in the respective pies. Moreover, a lot of missing lines are accounted for by preprocessor macros, which have limited support with Phasar and LLVM as they are optimization frameworks that do not intend to preserve behavior during transformations (orange parts of the pies). The **fping-statistics** module is an exception with 38% actually missing lines.

Next, we discuss reasons for actually missing lines (i.e., not comments or preprocessor directives) and for additional lines.

In **fping**,

all missing lines (56) result from lines that are semantically related to the module under investigation, but do not have a data-dependence on any relevant variables. Consider e.g., line 1 in the snippet in Figure 8. It prints a new line and does not refer to relevant variables (`num_alive`, `num_hosts`), referred to Lines 2 and 3; thus is not included in the extracted module, even if it is part of an overall code block that prints statistics.

¹⁴As it can be seen, the query is almost similar to the expected module.

TABLE III: Modules and Extraction Results. †: We used different constraints for *redis*, see III-A

Project	Module	Search Term	Lines	Extracted correctly		Missing		Additional		Runtime
				Fex	<i>grep</i>	Fex	<i>grep</i>	Fex	<i>grep</i>	
Parson	Parse	<i>parse</i>	400(301)	273	22	28	289	115	18	9 seconds
Parson	Serialize	<i>serialize</i>	360(201)	182	23	19	183	38	4	34 minutes
inotify	Statistics	<i>stats,stat</i>	270(159)	158	28	1	162	30	88	<1 second
inotify	Statistics	<i>stats</i>	270(159)	110	7	49	183	10	7	<1 second
fping	Statistics	<i>stats</i>	240(140)	94	2	56	166	430	14	19 seconds
silver searcher	File Filter	<i>ignore</i>	260(240)	209	37	31	215	248	56	24 seconds
memcached	Command	<i>command,cmd</i>	3150(1370)	1023	141	347	1481	802	204	13 seconds
memcached	Statistics	<i>stats</i>	710(485)	365	8	120	512	2267	337	130 seconds
redis†	Cluster Manager	<i>clustermanager</i>	4490(2689)	2274	492	415	2501	10	39	11 seconds

Lines 5, 6 are not included in the extracted module either, because the print statement in line 7 is also not included.

For **silver searcher**, our extraction misses 8 lines in the filename filter - this is because our tool does not readily include all lines in multi-line initializers, a feature that merely

needs a bit more engineering. For illustration, consider Figure 10, where lines 2-6 are not included in the initialization.

For **memcached-command**, we miss 347 lines, which are in a way or another due to our limited macro handling.

The root cause for the 120 missing lines in the **memcached-stats** module is the same, since 40 lines contain macro invocations.

For **redis-cluster manager**, we miss 415 lines.

Of those, 50 are declarations of functions where we only extract the definitions, 26 are macro usages, and the rest are due

to the wrong handling of multiline lines and struct/array declarations.

Overall, most missing lines are due to a few limitations of the current prototype: (a) handling of function/struct and array declarations and (b) handling of simple pre-processor directives, e.g. `#include`. With the exception of the *fping* extraction, all of the other modules are affected by this. All of these categories are solvable with engineering effort and should not prevent the principal applicability of **Fex**.

D. Analysis of additional lines of code

The extractor produces 140 additional lines for the parse module of **Parson**, out of which 103 pertain to the inclusion of methods that represent the API for JSON object modification, e.g., `json_value_init_object`, `json_object_resize`, etc.

We did not include these methods in our ground truth since to our opinion they represent a separate module. However, during the parsing there is extensive use of functions that create and modify JSON objects and thus our extraction process includes these methods.

With code-bases containing multiple inter-dependent modules (for parsing and modification of JSON objects), the domain expert could apply **Fex** repeatedly on the extracted broader parsing module to arrive at further splits, one for modification of the objects in our example.

The extracted statistics module of **fping** has 430 additional lines compared to the ground truth, 380 of which result from the fact that the stat module is controlled by a flag called `stats_flag`. Since the search term *stats* occurs in this flag, our tool marks several occurrences of this flag as relevant code, e.g., Figure 9 lines 2,5 and 6. The largest chunk of erroneously extracted lines results from the snippet in lines 1-3 of Figure 9. Since these lines belong to the part of the code that does the main commandline parsing, **Fex** transitively includes most of this commandline parsing and its dependencies.

In the **redis-cluster manager** module all 10 additional lines stem from the command line parsing that activates the extracted module. A large source of additional lines in our *memcached* stats module extraction stems from the way stats are collected throughout the project. Figure 11 shows such an example. Our stats module

writes the collected stats directly into one of two global structs. They are called `stats` or `stats_state` respectively. This leads to a large number of relevant starting locations that do not represent our module, but rather usages of the module. This leads to the inclusion of a lot of lines that we do not consider relevant for the module.

In scenarios where the code is organized with multiple variables with similar names to the search terms, a domain expert with a little bit of knowledge about the code could rename one of the occurrences such as `stats_flag` to break the chain and obtain a more representative module.

E. Impact of the search terms

Our approach relies on carefully selected search terms - thus, we opted for some empirical observations about the impact that the term selection may have on the resulting code. To this end, we experimented with two different sets of search terms in the case of *inotify*. They are:


```

1 fprintf(stderr, "\n");
2 fprintf(stderr, "%7d targets\n", num_hosts);
3 fprintf(stderr, "%7d alive\n", num_alive);
4
5 update_current_time();
6 curr_tm = localtime((time_t*)&current_time.tv_sec);
7 fprintf(stderr, "[%2.2d:%2.2d:%2.2d]\n", curr_tm->
    tm_hour, curr_tm->tm_min, curr_tm->tm_sec);

```

Fig. 8: Example Code from *fping*, that shows constructs that lead to non extracted code

```

1 ...
2 case 's':
3 stats_flag = 1;
4 break;
5 // Other commandline cases
6 ...
7 if (stats_flag)
8 print_global_stats();

```

Fig. 9: Additional lines in the stats module of *fping*

- 1) (*stats*, *stat*)
- 2) (*stats*)

In the context of files in a system, there are two concepts that may be referred to by the term *stat*: The singular of the word "stats" and the linux function `stat` that returns file information. Removing the singular version (*stat*) reduced the number of additional lines by 20 removing all calls to the linux function `stat`. But, it also increased the missing lines by 48, since there is a function called `stat_it` which is relevant for our module inside the ground truth.

In summary, the selection of the search term impacts the results. This was also hinted at when we discussed additional lines. But, in a sense this is a feature and not a bug. Our tool is modular enough to accommodate for input from more sophisticated forms of search terms, like declarative queries or more sophisticated expressions as proposed in literature [13][14][15].

The goal of **Fex** is not to replace the human in the extraction process, rather to serve as a prequel step to do much of the heavy lifting of identifying pieces of code potentially representing a feature. It provides the engineers with means to explore features in code. Project code can at the end be cut into modules by different criteria.

IV. THREATS TO VALIDITY AND FUTURE WORK

In this section, we discuss some threats to validity and planned future work to address those concerns.

a) Scope of informational retrieval: Currently, we consider each function to be a document, which could be problematic in poorly modularized code-bases with large functions with many lines of code. This risk can potentially be mitigated by choosing finer-grained definitions of documents. Another aspect of informational retrieval that could be improved is

```

1 const char *ignore_pattern_files[] = {
2     ".ignore",
3     ".gitignore",
4     ".git/info/exclude",
5     ".hgignore",
6     NULL
7 };

```

Fig. 10: Array Declaration in Silver Searcher

```

1 STATS_LOCK();
2 stats_state.hash_power_level = hashpower;
3 stats_state.hash_bytes = hashsize(hashpower) * sizeof(
4     void *);
5 STATS_UNLOCK();

```

Fig. 11: Usage of stats struct in memcached

to apply other indexing techniques than LSI. Such a future experiment could also serve to help us understand if additional lines resulting out of similarly sounding variables can be avoided, if more fine-grained weighting schemes to terms are considered.

b) Choice of ground truth: Since our ground truth modules were defined by one of the authors, there could be an element of bias involved in the selection. An ideal scenario would have been to conduct a thorough survey using domain experts, which is left for future work. However, we have tried to mitigate this issue by having another author oversee the defined ground truths and submitting pull requests to the maintainers. The status of the pull requests as of writing this paper is as follows. **Inotify-tools** accepted the request with positive feedback.¹⁵ **Parson** closed the request without explanation.¹⁶ This is presumably because our retrieved modules constitute a new file each, while they explicitly want the code to be within two files; this is stated in their "About page"¹⁷ as "Lightweight (only 2 files)". The rest are still not processed as of writing this paper.

c) Generality: Since our experiments required a lot of manual effort, an evaluation with a representative set of large projects was infeasible. We have attempted to mitigate this concern by conducting a partial analysis on two large code-bases. In addition we want to investigate the effects of the introduced inter-procedural distance limit further.

d) Preprocessor macros: At the moment **Fex** does not support the handling preprocessor macros, which leads to imprecision in features that use those macros. Although currently, our missing line numbers are not vastly affected by this, we are aware that preprocessor macros support is vital to our approach and We plan to add support for the handling of macros to **Fex**, by extending *SPL^{LIFT}* [16].

V. RELATED WORK

To the best of our knowledge our work is the first to combine IR and slicing to extract features based on natural language

¹⁵<https://github.com/inotify-tools/inotify-tools/pull/129>

¹⁶<https://github.com/kgabis/parson/pull/152>

¹⁷<https://github.com/kgabis/parson>

input.

A. Identifying and extracting features

The desire to identifying features from legacy code is nothing new. CodeCarbonCopy [17] allows developers to identify functionality to be transferred into a another part of the project. Automated Software Transplantation [18] is another technique that identifies functionalities from a *host* based on an input code location. Although both these approaches perform code analysis to identify dependencies of the functionality, the input is a code location, which would expect the user of the system to be well-versed with the code. In contrast, our approach allows developers to begin with a term in natural language and only expects the user to fix potential missing and additional lines later.

Another perspective to the same problem is to look at software as a product line and manage the individual features. Approaches have been proposed to extract product lines from Software. ArgoUML-SPL [19] is an open source tool that extracts Software product lines (SPL) from UML diagrams. This may be useful for identifying potential features in a large application, but in order to manifest the SPL in code, one would have to use an approach such as ours to aid with extraction of these features. Another approach that helps to organize requirements as product line requirements is CoreReq [20].

The above mentioned approaches expect that developers to think about feature-based software development prior to the coding phase, but we have established in Section I to be not always the case in reality.

Perhaps the closest to our work from the perspective of extraction of modules are the ones that attempt to reverse engineer SPL from existing software. But4Reuse [21] is a generic approach to extract software product lines from artifacts ranging from images to source code. Although in theory their approach should work with problems such as ours especially since they have adapters for C and Java programs, source level feature extraction requires careful implementation of adapters, and these adapters cannot capture the specific semantics of a particular software, but rather are written using language-specific hints such as preprocessors. Shantnawi et al. [22] recover SPL from already existing product variants in object-oriented code, which also relies on language-specific features to separate variants in code.

B. Managing features

Another school of thought to handling features in software is to manage them without modifying the code. Zhou et al. [23] have proposed a technique to use opportunities present in forks of branches in repository to identify features. Their approach also uses information retrieval to label commits into clusters, which provides technical leads with a visual overview of in-development features for a project. This approach and ours complement each other.

The C-preprocessor is often used to identify features in a code. Peopl [5] allows

to identify features, visualize and edit them. Typechef [24] relies on existing variant information in preprocessor pragmas to check for type errors in variants. Malaqueis et al. [3] harness the information present in preprocessor based C-code to provide visualization of existing variants. *Leviathan* [25] and the approach by Stănculescu et al. [26] provide projections for single configurations and thus enable the evolution of only one configuration. *CIDE* [27], [28] provides the basis for the analysis of software product line features, for Java.

All of these approaches are helpful for abstracting, visualizing and managing existing features in source, but do not help when users want to have a starting point to even identify, where these features are inside the code-base, especially without necessarily relying on preprocessor directives.

Copy-paste-redeemed [29] abstracts clones into modules from C source code. Linked Editing [30] does not abstract modules, but provides an opportunity to link code clones and edit them together. These approaches are complementary to ours.

C. Program slicing

Weiser [31] introduced a framework and a process to extract program slices back in 1981, applications of which are endless [32] including ours. The second phase of our approach draws a lot of inspiration from the techniques proposed there. Infoslicer [33] is one such closely related work that proposes an algorithm to extract inter-procedural program slices. Since we use the SSA form of the LLVM's IR, it is envisionable that we could employ state of the art pointer analysis provided by SVF [34]. srcSlice [35] is a highly scalable program slicing technique that is able to gather program slices for every variable in a large code-base such as the linux kernel within 15 minutes. Although such a technique would be useless for people who want to identify codes linked to a natural language term, such techniques can be plugged into our program slicing part for larger examples to avoid resource-overhead.

VI. CONCLUSION

Large software today are a collection of multiple functionalities and organizing them as individual functionalities would greatly benefit maintenance costs. Unfortunately, because many large scale software are not built with this intention prior to development phase, individual features or functionalities are largely scattered throughout the code-base. Current techniques to identify and extract such inter-dependent code bases rely on program slicing techniques that require code locations as input, something a non-expert in the code-base may not possess. To this end, we propose a novel approach that takes a natural language term as input and combines information retrieval and static analysis techniques to extract modules representing these terms. We built a prototype tool to implement this approach, evaluate it on five fairly large open source code bases and report insights based on this experiment. Our experience with the tool reveals that even though such a subjective module extraction cannot be fully automated, it can go a long way in reducing the manual effort required for such an endeavour.

ACKNOWLEDGEMENTS

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. This work has been co-funded by the Crossing SFB 119 and through the support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

- [1] P. C. Clements and L. Northrop, "Software product lines: Practices and patterns," ser. SEI Series in Software Engineering. Addison-Wesley, August 2001.
- [2] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarniecki, "What is a feature? a qualitative study of features in industrial software product lines," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 16–25. [Online]. Available: <https://doi.org/10.1145/2791060.2791108>
- [3] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The discipline of preprocessor-based annotations - does ifdef tag n't endif matter," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 297–307.
- [4] D. Le, E. Walkingshaw, and M. Erwig, "ifdef confirmed harmful: Promoting understandable software variation," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 143–150.
- [5] B. Behringer, J. Palz, and T. Berger, "Peopl: Projectional editing of product lines," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 563–574.
- [6] Y. Zhao, "Chapter 10 - text mining," in *R and Data Mining*, Y. Zhao, Ed. Academic Press, 2013, pp. 105 – 122. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/B9780123969637000106>
- [7] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis." 1990, pp. 391–407.
- [8] L. N. Trefethen and D. Bau, "Numerical linear algebra." SIAM, 1997.
- [9] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An interprocedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.
- [10] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61. [Online]. Available: <https://doi.org/10.1145/199448.199462>
- [11] M. Braun, S. Buchwald, S. Hack, R. Leiða, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," in *Compiler Construction*, R. Jhala and K. De Bosschere, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 102–122.
- [12] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 134–144.
- [13] Z. Shang, E. Zraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, and T. Kraska, "Democratizing data science through interactive curation of ml pipelines," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1171–1188. [Online]. Available: <https://doi.org/10.1145/3299869.3319863>
- [14] Z. Zhao, L. De Stefani, E. Zraggen, C. Binnig, E. Upfal, and T. Kraska, "Controlling false discoveries during interactive data exploration," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 527–540. [Online]. Available: <https://doi.org/10.1145/3035918.3064019>
- [15] C. Reichenbach, Y. Smaragdakis, and N. Immerman, "Pql: A purely-declarative java extension for parallel programming," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 53–78. [Online]. Available: https://doi.org/10.1007/978-3-642-31057-7_4
- [16] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "Spl^{lift}: Statically analyzing software product lines in minutes instead of years," *SIGPLAN Not.*, vol. 48, no. 6, p. 355–364, Jun. 2013. [Online]. Available: <https://doi.org/10.1145/2499370.2491976>
- [17] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 95–105. [Online]. Available: <https://doi.org/10.1145/3106237.3106269>
- [18] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 257–269. [Online]. Available: <https://doi.org/10.1145/2771783.2771796>
- [19] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 191–200.
- [20] I. Reinhartz-Berger and M. Kemelman, "Extracting core requirements for software product lines," *Requirements Engineering*, vol. 25, no. 1, pp. 47–65, Mar 2020. [Online]. Available: <https://doi.org/10.1007/s00766-018-0307-0>
- [21] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 101–110. [Online]. Available: <https://doi.org/10.1145/2791060.2791086>
- [22] A. Shatnawi, A.-D. Seriai, and H. Sahrouri, "Recovering software product line architecture of a family of object-oriented product variants," *Journal of Systems and Software*, vol. 131, pp. 325 – 346, 2017. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121216301327>
- [23] S. Zhou, S. Stănculescu, O. Leßenich, Y. Xiong, A. Wąsowski, and C. Kästner, "Identifying features in forks," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 105–116. [Online]. Available: <https://doi.org/10.1145/3180155.3180205>
- [24] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: Toward type checking ifdef variability in c," in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, ser. FOSD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 25–32. [Online]. Available: <https://doi.org/10.1145/1868688.1868693>
- [25] W. Hofer, C. Elsner, F. Blendingner, W. Schröder-Preikschat, and D. Lohmann, "Toolchain-independent variant management with the leviathan filesystem," in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, ser. FOSD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 18–24. [Online]. Available: <https://doi.org/10.1145/1868688.1868692>
- [26] S. Stănculescu, T. Berger, E. Walkingshaw, and A. Wąsowski, "Concepts, operations, and feasibility of a projection-based variation control system," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 323–333.
- [27] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 3, Jul. 2012. [Online]. Available: <https://doi.org/10.1145/2211616.2211617>
- [28] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachsel, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the# ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.
- [29] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 630–640.
- [30] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 2004, pp. 173–180.
- [31] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, p. 439–449.
- [32] L. Du and P. Cai, "A survey on applications of program slicing," in *Soft Computing in Information Communication Technology*, J. Luo, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 215–220.
- [33] Y. Sun, Y. Zhang, and J. Qian, "Program slicing method of llvm ir based on information-flow analysis," in *2019 International Conference*

on *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2019, pp. 383–390.

- [34] Y. Sui and J. Xue, “Svf: Interprocedural static value-flow analysis in llvm,” in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: <https://doi.org/10.1145/2892208.2892235>
- [35] C. D. Newman, T. Sage, M. L. Collard, H. W. Alomari, and J. I. Maletic, “srcslice: A tool for efficient static forward slicing,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 621–624.