Generating Customised Control Flow Graphs for Legacy Languages with Semi-Parsing

Céline Deknop[1,2], Johan Fabry[2], Kim Mens[1], Vadim Zaytsev[3]
[1]ICTEAM institute, UCLouvain, Belgium
[2]Raincode Labs, Brussels, Belgium
[3]Formal Methods & Tools, UTwente, The Netherlands
{celine.deknop, kim.mens}@uclouvain.be, johan@raincode.com, vadim@grammarware.net

***Abstract*—We propose a tool and underlying technique that uses semi-parsing to extract control flow graphs from legacy source code (i.e., COBOL). Obtaining such control flow graphs is relevant in the industrial setting of legacy modernisation, to quickly demonstrate to code owners that modernisation engineers did not break their business logic. They need to be convinced that a migration did not affect the flow around critical parts of their code such as database accesses. Focusing on the control flow around embedded SQL queries and confirming that the code logic has been preserved improves customers' trust and satisfaction in the modernisation. Our proposed algorithm and approach uses fuzzy parsing as opposed to full parsing to parse mainly the control flow constructs, while delegating the full parsing of embedded languages like SQL to an external parser, and produces a control flow graph directly while skipping over most of the input in linear time. Such a fuzzy parser is easier to construct and adapt to particular languages and needs than a full parser with a visitor to elicit control flow. Comparisons are made of the fuzzy parser to an industrial-strength full parser.**

***Index Terms*—Semi-parsing, fuzzy parsing, legacy modernisation, control flow graph, COBOL, SQL, industrial use case**

I. Introduction

Control Flow Graphs (CFGs) have many uses for testing, analysing, understanding or comparing programs [2]. However, obtaining them is computationally heavy, requiring significant development effort when the target language or its specifications are unusual, as is often the case for legacy languages. Specialised CFG generation tools are available only for some languages, and tend to impose such strict structural conditions on their output that they are not suitable for every scenario. Another option is to generate CFGs with a bespoke tool from parse trees produced by full parsers. In this case, the produced CFGs can match exactly one's analysis needs. Unfortunately, full parsers are not always easily obtainable, and sometimes non-existent. The effort necessary to create a full parser from scratch is rather large (a famous expert quote puts it at 2–3 years [28]). As an alternative, semi-parsing techniques have been proposed [37] and could suffice for creating CFGs, as we will explain in the following sections. Semi-parsers are simpler to write as they only need to parse those language fragments relevant for the CFG, and make it easier to adapt the generated CFG to a project's needs.

In an industrial context of legacy code modernisation, CFGs could be used to prove to code owners that the behaviour of their programs before and after migration is unchanged, especially around critical parts of the code like database reads and writes. For this, we use a semi-parsing technique based on fuzzy parsing [25], [37]. We evaluate the feasibility of this approach by creating a semi-parser for COBOL that is easily configurable while leveraging an existing full parser for embedded SQL code. This allows our parser to create CFGs containing only control flow structures of relevance, and to obtain a full parse subtree of the SQL embedded in the COBOL code, in order to be able to compare that the behaviour of both are preserved by a migration.

This paper is structured as follows: Section II presents the idea of semi-parsing in general and introduces our chosen technique of fuzzy parsing; Section III presents our industrial use case, and discusses how CFGs could be leveraged to handle it; Section IV introduces CFGs, detailing how they are usually generated as well as presenting some related work about them; Section V presents the implementation details of our fuzzy parser for COBOL with embedded SQL, going over the parser configuration, its preprocessing step, the semi-parsing algorithm itself and a few concrete examples of the CFGs it generates; Section VI validates the correctness and performance of our fuzzy parser by comparing it with a full industrial parser for COBOL[1]; Section VII discusses several metrics to take into account when deciding which approach would be better suited for different uses cases; Section VIII concludes the paper and presents directions for future work.

II. Parsing vs. Semi-parsing

Parsing as the analysis of syntax of computer programs has been an integral part of automated compilers since their beginning. In a broader sense, it can be understood as the process of eliciting the structure that was expected to be found in a software artefact, and making this structure explicit as preparation for further processing [41]. The syntactic commitments of an artefact can be as strict or as loose as the situation dictates, and the expectations are usually expressed in some explicit form: a grammar, a regular expression, a type, a schema, etc, sometimes collectively being referred

[1] This paper is accompanied by an artefact that can be found here: https://zenodo.org/deposit/6806075

to as "grammars in a broad sense" [22]. Grammars can have vocabularies ranging from several distinct symbols to over 5000 of them [38]. When the syntactic expectations are expressed in a form close to that of a context-free grammar, many parsing algorithms are readily available [15]. Most of them expect the program either to comply to these expectations fully, or be declared invalid.

Parsers are metaprograms that typically take a program in textual form as input and produce a more structured representation of the program in the form of a tree, a graph, a table, and so on. We distinguish two groups of such metaprograms: full parsers and semi-parsers.

Full parsers [15] have a complete overview of the syntactic expectations of a language, and expect the input to fully conform to all of them. In exchange, they produce a fully structured ("parsed") version of the input. It is not uncommon for abstract syntax trees to contain information about variable types, name scopes and other complicated matters useful for figuring out the semantics of the parsed program afterwards.

Semi-parsers [6], [37] admit to having only a partial overview and embrace it by handling uncertain portions of the input tolerantly or not at all, and yield only an estimation of the full structured representation. Depending on particularities, their output can be a normal full tree/graph constructed under a set of assumptions (e.g., "a semi-colon is expected, but let us assume it" or "a variable is undefined, but it seems like its type could be string"), or a partial tree with explicit gaps (e.g., one does not need to fully understand HTML to be able to extract all CSS snippets from it). Fact extractors can be seen as extreme forms of semi-parsers that only collect imported module names to construct a dependence graph, or only handle comments and line continuations to count the logical lines of code. In the industrial practice of language processing one often comes across situations where a full definition of a legacy language is unobtainable or unreliable, or when the investment in the development of a full parser cannot be justified. Semi-parsing offers an interesting alternative.

As wonderful and omnipresent as full parsers can be, they have at least three disadvantages in the context of software modernisation. Firstly, they are very complex to write, and literally take years to complete [28]. Many legacy codebases contain a language cocktail, and it would have been a major investment to develop a full high quality parser each time a new fourth generation language (4GL) enters the scene, and even handling hard-to-parse exotic 3GLs like REXX or RPG can in many contexts be inexpedient. It has been estimated by one of the experts on the Y2K problem that in 1998 there were at least 200 proprietary 4GLs in active use [20], and many of them are still active today. Secondly, in a world where just handling COBOL (the most used legacy language, and one of the 700 that can be found on mainframes) means being ready to encounter up to 300 different COBOL dialects [27] and to deal with various versions of competing compilers that accept diverging syntax, strict adherence to the syntactic expectations is not only undesirable, it is highly unrealistic. There are always "uninteresting" syntactic deviations in spelling keywords or using the preprocessor to handle non-existing statements as extensions. On top of that, modernisation engineers often need to deal with obfuscated code (sometimes up to the point of it not being executable) or partial code (e.g., `COPY` books in COBOL serve the same function as libraries in other languages, but with lexical parameterisation — such a `COPY` book is never a fully parsable compilation unit). Thirdly, language embedding is still an open problem, especially in technology-defined grammar classes (e.g., a composition of two context-free grammars is still context-free, but a composition of two LALR grammars or two ALL($*$) grammars can be anything). Legacy languages like COBOL, PL/I, FORTRAN or 4GLs, often contain fragments of embedded SQL queries, JCL requests or CICS commands. Each such language boundary must be manually programmed and thoroughly tested, and even then without a full parser of the embedded language, the `EXEC` block containing the embedded queries will stay unparsed. Interestingly, this is a well-known phenomenon in semi-parsing, which refers to such unparsable regions as "lakes" [37].

Semi-parsers have a number of disadvantages as well, such as overapproximating the input language — we will address them concretely in the following sections. The industrial state-of-the-art attitude is to accept semi-parsers in program analysis, but not in program transformation [3]. However, they have three advantages that are worth mentioning up front: (1) since their development effort is much lower, it is possible to create them in an agile way, catering fully to customers' needs and using them in early stages of the project during joint workshops and feasibility studies; (2) due to their built-in tolerance, semi-parsers are often naturally applicable to a range of language dialects, and have the capacity to skip over embedded fragments just as easily as they skip over other "uninteresting" input parts; (3) it is often possible to overstep the formal limitations of language composition and grammar modularity, and develop a bespoke "grammar" (in a broad sense) for each scenario. Semi-parsers are uniformly accepted also in interaction situations where responsiveness is more important than precision — such as syntax highlighting or contextual code completion in an IDE.

The core of the particular algorithm we use in this paper, is based on the idea of *fuzzy parsing* [25] with anchor tokens. In short, the semi-parser scans the input in search of one of the anchors it knows (e.g., `EXEC`, `IF`, etc) and skips all other tokens. Once the anchor matches, the actual parsing begins. Formally speaking, this is a form of event-based parsing [40] where parse actions are executed reactively after witnessing something specific in the input, and not dictated by the grammar. For our approach, which will be described in full in Section V, the "actual parsing" can mean one of two things, depending on the anchor: it could lead to an invocation of an external full parser (for embedding SQL and incorporating full parse trees of each query into the result); or to a modification to a control flow graph that we are constructing instead of the parse tree. Thus, it is a bespoke setup combining semi-parsing [37] ideas of fuzzy parsing [25] with parsing in a broad sense [41].

III. Industrial Use Case

Raincode Labs (https://www.raincodelabs.com) is an independent compiler company that, as part of its portfolio, provides services for legacy system modernisation. One such service is *PACBASE migration* [32]. PACBASE is a fourth generation language [42] for which vendor support has been phased out [17]. Following the declared termination of support, owners of PACBASE codebases naturally declared it unwanted, and seek ways to retire their PACBASE code as well. In theory, companies could maintain the COBOL code generated by PACBASE, but this code is not friendly for the eyes nor for maintenance activities by human developers (unformatted, dead code fragments, partly obfuscated names, overly deeply nested control constructs, etc). Rewriting it from scratch is also not feasible in practice, due to the amount of effort it would require [34]. Raincode's PACBASE migration service provides a viable alternative: it automatically refactors PACBASE-generated COBOL code to human-readable COBOL, in a manner that is configurable by the customer [8], [9]. This solution exists for twenty years and has successfully refactored over 350 million lines of COBOL code.

For some customers the very fact that such a large amount of code has already been processed by the PACBASE migration kit and successfully deployed afterwards, serves as a sufficiently convincing argument for applying it to their own code. Other customers desire a more solid assurance that the solution works on their specific codebase, which they usually consider as "different" in some way from all the other, already processed, ones. Hence, part of the process before setting up a migration is showing and convincing the customer that the solution works on their codebase. While applying the refactoring itself is not a problem, there are currently no tools that, given (a part of) the customer's code, illustrate that the migrated COBOL code has the same behaviour as the original code, i.e. that the migration is indeed truly a refactoring.

One way to assure a customer that the code changes are behaviour-preserving, is to illustrate that database accesses remain unchanged, since they are at the heart of their business logic. For database access, COBOL allows the inclusion of embedded SQL through the `EXEC SQL` ... `END-EXEC` statement. This statement is treated by the mainframe COBOL preprocessor, so it is technically not part of the COBOL standard, but of the Db2 standard, along with ways to embed SQL queries in FORTRAN, REXX, C and C++ [19]. By showing that the control flow that leads to each database access, is unchanged after the refactoring, the customer becomes more confident that the core business logic of his application remains unchanged. Some analysis tool that processes the two versions of a COBOL program (before and after refactoring) and shows both CFGs, with a focus on execution of SQL queries, would hence be an important asset in conversations with the customer.

In such an analysis tool, performance is crucial. Its intended use is within an interactive session with the customer as part of a proof-of-concept (POC) evaluation. Such a POC consists of firstly migrating a representative subset of the customer's codebase, and secondly evaluating, together with the customer, whether the refactored result matches the customer's requirements. The migration for a POC is a multi-hour process, possibly on the hardware of the customer. This makes it hard to argue for adding an extra post-processing phase that would perform the control flow analysis over the entire codebase. Instead, in an interactive session a customer would pick a few representative programs and the tool should produce the graphs on the fly. Thus, tool performance is crucial.

In fact, if such a proposed tool for comparing CFGs existed, we could also envision it being used during the verification phase after the full migration. It can flag any programs with significant differences in the control flow, for the list to be reviewed by the migration engineers, and possibly discussed back with the customer. In such an application scenario performance still remains crucial, since a full migration typically concerns millions of lines of COBOL code. It is not uncommon for servers to be provisioned such that the migration of the codebase becomes an overnight process. An extra verification phase should not add an overhead that would require a significant increase in processing time or in server provisioning, otherwise it simply would not be used.

IV. Control Flow Graphs

Control Flow Graphs (CFGs) are a representation of all the paths that could be followed during the execution of a program [2]. Each node of a CFG represents a code statement, and edges connecting nodes represent steps in the program's control flow. Such graphs can be useful when one needs to understand a program at a glance, or when the language that was used to create the program is not well known to the user. In the context of our industrial use case, the focus is less on understanding the entirety of a legacy program, but more on allowing the user to zoom in on and compare the control flow around database read and writes, a critical part of the business logic. This can be achieved more easily using a graph than through highlighted code since when generating that graph, we can pick and choose what to show or not, i.e. keeping only the most important parts of the control flow to shorten the output. This can be seen as a kind of intentional slicing [36].

The most generic way of creating a CFG is by using a full parser: given the parse tree of a piece of code, generating its corresponding control flow graph is a relatively straightforward traversal of the tree, creating nodes and connecting them as needed. Another way to generate CFGs is to use one of the many existing tools. The issue with that approach is that those tools are only available for some languages, mostly widely used ones. Popular CFG-handling tools include GrammaTech's CodeSonar (previously known as CodeSurfer) [14] and Ghidra (and the entire GTIRB-based ecosystem) [30], UCLA's Aurora [35] and UniLeiden's JShowFlow [5], Eclipse plugins EclipseCFG [1] and Atlas SDK [12], Python modules StatiCFG [4] and PyCFG [13], etc. Using a ready-made tool presents the advantage of being a very fast way to obtain a

CFG, but may limit the ability to tune the tool's output to one's particular needs. Also, when dealing with less widely known languages, and legacy languages with multiple dialects, it is possible that no quality tools exist that are freely available. This is the case for COBOL. For some more confidential languages, it even happens that no parser is available at all.

Yet another way to generate CFGs is to use a one-stop-shop language workbench with a meta-programming language like Spoofax [21] or Rascal [23]. They come equipped with their own way of analysing the control flow of code, for these examples called FlowSpec [33] and DCFlow [18]. While these are generally of higher quality and abstraction level, all language workbenches require a good deal of background knowledge and often a steep learning curve from their users, which in the context of this project was not sufficiently available within Raincode Labs.

While each of the above techniques are interesting, we wanted to explore an approach that did not require the user to know any meta-programming or DSL, and chose to go with a semi-parsing technique since it is both lightweight and accessible, while providing the advantage of allowing to generate CFGs of desired structure.

V. Our approach

In this section we describe in detail the approach we used to create our fuzzy parser. To allow it to be very flexible, we divided it into two parts: the parsing logic and a configuration file, with the idea being that in most cases users of the parser should only edit the configuration file to be able to adapt the parser to their needs. First we present the parser configuration, followed by a preprocessing phase, then the intuition of how the parsing algorithm works, and finally we explore some results we obtained by applying it on a few small handcrafted examples.

A. Parser configuration

The configuration is a Python file that contains a set of arrays, each defining a type of language structure we are looking for in the code. Each element of those arrays is a specific definition of that structure, e.g. an `IF` statement is a specific definition of a condition as illustrated by Listing 1:

Listing 1: Extract of our parser configuration file for COBOL.

```
conditions = [
    {"start": r"\sIF(\s)+",
     "condition_delimiter": r"\sTHEN(\s)+",
     "mandatory_delimiter": False,
     "single_branch": r"ELSE\s", "end": r"\sEND-IF"},
    ...
    ]
```

Using this file, users can both pick exactly what structures they are interested to see in the output control flow graphs, and define to the parser what is the syntax of those structures. The content of this configuration file would therefore change mostly in function of the language that is being parsed, but one could also omit a certain type of structure if they do not want to have them in the output. Whereas such omissions can result in CFGs that do not fully reflect the language semantics yet, it does help in building the fuzzy parser incrementally, by adding one language construct at the time, until reaching the full language semantics that one wants to cover.

For our industrial use case in COBOL, we included the following structures:

Condition The COBOL statements `IF/ELSE` and `EVALUATE ... WHEN`.

Loop The COBOL statements `GO TO`, `PERFORM`, `PERFORM THRU`, `PERFORM VARYING` and `NEXT SENTENCE`. Note that in COBOL, most of these are not conventional loops, but behave more like jumps through the code that either return to where they started (PERFORMs) or do not return at all (GO TO and NEXT SENTENCE). The only exception is `PERFORM VARYING`, which is equivalent to a standard Java for-loop. It is present only in the post-migration files, but its inclusion was requested by Raincode engineers.

Parsable This structure is used to define program fragments to be analysed by an external parser, in our case `EXEC SQL` since we want to delegate the parsing of embedded SQL fragments to a separate parser.

Ignorable structures that we do not want to see in the generated CFGs, but that are sometimes needed to disambiguate the rest. Usually these are strings and comments.

Special This structure groups special language-specific cases that need to be dealt with. In the case of COBOL, for example, we need to handle the special DOT statement `(.)` which closes all open statements. For our industrial case we parse only the procedure division of COBOL programs, with a special anchor to skip to that division.

Every element of the arrays found in the configuration file can be edited or removed, and new ones can be added. In case of adding a new element, some edits to the actual parsing code may be necessary to implement the new structure's logic.

A concrete example of such a configuration file for COBOL can be found on GitHub [10]. To define the syntax of the statements we are looking for, we decided to use Python's regular expression module `re` [31]. This choice was motivated both by its ease of use and by COBOL's inner complexity. Indeed, some statements can have multiple correct syntaxes, like the well-known `GO TO` which can also be written as just `GO` while having exactly the same semantics.

B. Preprocessing

Before being able to parse our COBOL files, we need to preprocess them. (Not in the sense of using the COBOL preprocessor, but rather altering the input of our parser). Since some of the files used in our use case are generated by PACBASE, they contain line numbers and program identifiers, which are a part of the line-by-line structure, but not a part of the core syntax. An example of what a PACBASE-generated file looks like prior to such cleaning can be seen in Figure 1. Technically columns 1–7 and 73–80 can be ignored by the fuzzy parser itself reacting to pseudotokens (similar to how most Python parsers work with indent/dedent pseudotokens),

```
012900 F9520-C. MOVE DAT73C TO DATCTY.                                ALCB018
012910          MOVE DAT71C TO DAT71.                                 ALCB018
012930          MOVE DAT74C TO DAT73.                                 ALCB018
012940          MOVE '00111' TO TT-DAT GO TO F9520-T.                 ALCB018
```

Fig. 1: A PACBASE-generated program fragment: only columns 12–72, also known as "Area B", should be parsed as COBOL, with columns 8–11 ("Area A") defining the structure (divisions, sections and paragraphs).

but that would bias our parser too strongly towards position-based languages which often require drastically different parsing techniques anyway [39]. Thus, we opt for a gentle preprocessor that replaces everything outside Areas A and B with spaces and keeps the COBOL code syntactically correct and semantically equivalent. This also allows us to deal with line continuations and comment markers at column 7.

C. *Parsing*

Since we are relying on semi-parsing (see Section II), the core of our algorithm is relatively simple. First, when our parser is instantiated, it goes trough the configuration file to define the anchors. To simplify the parser's code, the different regexes are *compiled* and passed to another object that will keep track of which ones are active (i.e. still present in the rest of the input). The parser only handles the actual matches of the regex, keeping both a list of the next match for each anchor, and one with iterators that allow us to retrieve the next matches when needed.

The parser moves through the input as a long string with an index, starting at zero and jumping from anchor to anchor. At each step, our parser finds the anchor that has the earliest match in the input, and passes it to a handler that processes them differently according to their type (Condition, Loop, Parsable, Ignorable or Special, see subsection V-A) and according to their exact match. This process creates Node objects that will form the final CFG, and analyses any needed additional information like an `IF`'s condition.

It is important to note that our Ignorable (comments and strings) matches are used only when they are needed to disambiguate. When looking for the next anchor, if we find a match for an `IF`, but find that an Ignorable anchor overlaps it, we pick the string or comment instead of the if, and don't add any node to the graph.

The only other special case worth mentioning regarding the computation of additional details about a node is the case of Parsable nodes. To parse the embedded SQL statements it encounters, our parser simply calls the SQL parser generated from an open source ANTLR grammar [24]. Our Parsable node objects have access to the SQL grammar and a function allowing them to parse a string using the `antlr4` Python module. After retrieving the SQL code, conveniently contained between the `EXEC SQL` and `END-EXEC` statements, they pass it to ANTLR to create a parse tree that is stored inside the Node object for later use.

The parser's iterative processing of the input, anchor by anchor, continues until either the end of file is reached or no more match is found for any of the anchors.

To finish fully constructing the final CFG, we make a pass through our loop nodes to make sure that they point towards the right children. Indeed, given the jumpy nature of COBOL code, it is common to have `GO` or `PERFORM` statements pointing towards a label that is further down the code, and that we did not have knowledge about yet at the moment of parsing the loop itself. For simplicity, we resolve all of these after the first parsing phase, when we are guaranteed to have found all correct labels to point to.

At the end, we obtain an object representation of a CFG corresponding to the given file. It is serialised into a file to be visualised later. For this, we use the GraphViz [11] library, which has the advantage of being directly connected to Python and easy to use to produce a visualisation of such graphs. In later stages of our work, rather than just visualising it, we will use the CFG object representation to compare it with other CFGs (for example, the CFG of a COBOL program before and after a migration step).

D. *Example*

We now illustrate concretely how the parser works on the small handcrafted COBOL example of Listing 2.

Listing 2: A small handcrafted COBOL code example

```
 1      IDENTIFICATION DIVISION.
 2      PROGRAM-ID. Example1.
 3      ENVIRONMENT DIVISION.
 4      DATA DIVISION.
 5      PROCEDURE DIVISION.
 6          IF A > 0
 7            NEXT SENTENCE
 8          ELSE
 9            DISPLAY "First if branch"
10          END-IF.
11          IF B = 0
12            DISPLAY "Second if"
13          END-IF.
```

After loading all anchors and searching for all their first occurrences, the fuzzy parser will be left with only the following: the *special* `PROCEDURE DIVISION` anchor on line 5 to start the parsing, the *condition* anchors `IF`, `ELSE` and `END-IF`, the *loop* `NEXT SENTENCE` anchor and finally the *ignorable* string anchor.

Let us analyse the parser operations line by line:

- Even though we start at line 1, there is nothing of interest to parse until line 5, where we find the first `PROCEDURE DIVISION` anchor. This does not produce any node, but advances the parser's index up to the start of line 6.
- On line 6, we find a match for one of the *condition* anchors. A node denoting the start of an `IF` is created, and the parser also takes note of its condition `A > 0`. To

allow us to keep track of how the nodes are embedded, we also increment a *depth* variable. All nodes created after this point will be inside the `IF`, at a depth of 1.

- On line 7, a match is found for a *loop* anchor. A node is created, and a new anchor is added to the list: the parser now looks for a DOT (`.`), which will indicate the end point of the loop. (In COBOL, `NEXT SENTENCE` redirects control to after the next dot in the input).
- On line 8, there is again a match for one of the *condition* anchors, this time for a branch. No new node is created in the graph, we simply mark the "True" branch of the first condition node as closed (new nodes that we parse next will no longer be added as its children).
- Line 9 does contain a string, but it does not overlap with another anchor, so it is ignored by the parser.
- On line 10, there are two separate entities to parse: first, the `END-IF` which is a match for one of the *condition* anchors and will result in the closing of the first condition node. The internal *depth* variable is decreased by one and this closure is matched with the nearest open condition of the same depth. The second match concerns the DOT, which is processed to act as the end point of the loop node, adding a temporary node in the graph to mark its position.
- Finally, another `IF` node with the condition `B = 0` is created for line 11, line 12 is ignored and line 13 closes the second condition node.

As described above, after this first parsing phase is done, we make sure all loop nodes present in the graph point to the correct place. In our example, we suppress the previously created DOT node, pointing the `NEXT SENTENCE` to where that DOT node was, and the graph is complete.

The CFG produced by parsing the program of Listing 2 is shown on the left side of Figure 2. We represent the start and end point with a diamond shape, and show the COBOL statements in ellipses, with some edge annotations.

Note that due to the semantics of COBOL, very slight changes to a COBOL program can change its CFG greatly. For example, if we transfer the DOT seen on line 10 to the end of line 13 (i.e. after the second `END-IF` and not the first), we change where the `NEXT SENTENCE` jumps to, and therefore the CFG itself. The CFG output corresponding to this variation of code Listing 2 is shown on the right side of Figure 2. The parsing itself is almost the same, except for the step at which the DOT is parsed, which is now the last one.

Listing 3: A small COBOL program with embedded SQL code

```
1  IDENTIFICATION DIVISION.
2  PROGRAM−ID. Example2.
3  ENVIRONMENT DIVISION.
4  DATA DIVISION.
5  PROCEDURE DIVISION.
6      IF A > 0
7        EXEC SQL SELECT ∗ FROM TABLE END−EXEC
8      END−IF.
```

The last feature in our fuzzy parser is its ability to make use of an external parser to analyse an embedded language, in our case the SQL code present in between an `EXEC SQL` and `END-EXEC` statement in a COBOL program. Consider the code of Listing 3.

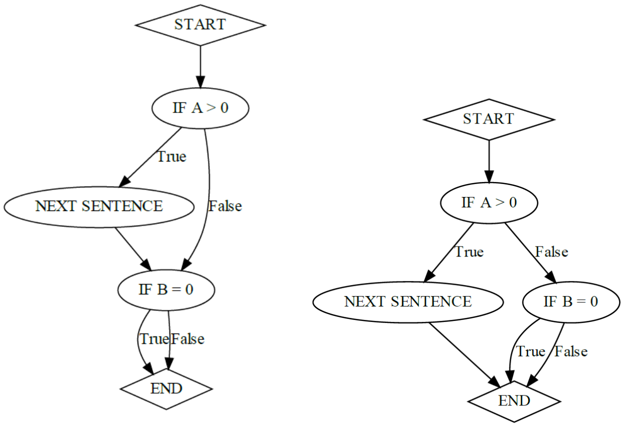

Fig. 2: Generated CFGs for code example from Listing 2 and its variation with the DOT only on line 13

The parsing happens in a very similar way as described earlier with the addition of a *Parsable* anchor in the anchors found present in the program (on line 7, instead of the loop anchor found previously on that line in listing 2). As before, the parsing starts with the `PROCEDURE DIVISION` and the `IF`. When arriving at line 7, the handler for the Parsable anchor creates a node and extracts the SQL code between the delimiting COBOL statements. This SQL is stored in the node as plain text for ease of representation with Graphviz, but it is also passed to an ANTLR parser, which creates a parse tree of the code that we will be able to use when we need to compare the SQL code pre- and post migration of a COBOL program. The parsing is finished with the processing of the `END-IF` at line 8, and for this program we do not have any loops to clean-up after the first phase. Figure 3 shows the resulting CFG for the code of Listing 3, with the embedded SQL shown in boxes to visually differentiate it from COBOL.

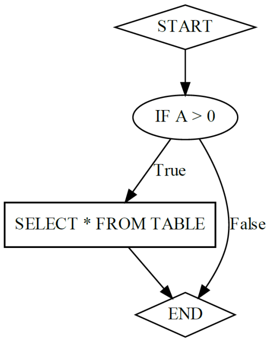

Fig. 3: Generated CFG for the code example from Listing 3

VI. Evaluation

In this section, we present how we evaluated our approach both from a qualitative and a quantitative point of view. We first go over the methods we used to ascertain that the produced CFGs are correct, and then compare how the performance of our fuzzy parsing approach fares against using a full parser instead.

A. Qualitative Results

The first step we took to ensure that our control flow graphs were correct was to manually write small pieces of COBOL code to act as test files. We wrote some containing basic forms of the structures we wanted to analyse, along with some interesting edge cases, or cases that generated bugs during development. Using those as a base, we wrote a test suite to examine and verify the details of what was found during the parsing phase: which anchor was triggered when, what nodes did it create, and whether or not the content of those nodes was what we expected. We also took care to test the produced CFGs, i.e. contained nodes and links between them. Listing 2 was part of this test suite, as well as its variation in Listing 3. One of our tests makes sure that both variants indeed produce the two different outputs shown in Figure 2. In total, we wrote 95 tests over 32 different handcrafted files, considering nested conditions and loops, some `EXEC SQL` to test our integration with ANTLR, the inherent case insensitiveness of COBOL as well as difficult cases where string or comments contain fragments that could be mistaken for parsable code.

While this test suite was very useful during the incremental development phases of our fuzzy parser, it was still limited in a number of ways. Since our collaboration with Raincode Labs gave us access to their industrial-strength full parser for COBOL, we took advantage of that. We wrote a script to transform the parse trees created by their full parser to CFGs and compared these CFGs to our own. The comparison itself was done manually by probing diverse programs. (An automated CFG differ script is currently under construction). We found that the CFGs were semantically equivalent in all cases, even if the syntax used was not always exactly the same. COBOL contains a lot of synonyms, which the Raincode parser normalises (e.g., characters like < and = in the input appear as `GREATER` and `EQUAL` in the tree). The resulting CFGs produced from the Raincode parser outputs for the code in Listing 2 and its variation can be seen on Figure 4. Apart from the small syntactic differences mentioned above, they are equivalent to the CFGs in Figure 2.

B. Quantitative Results

To compare their performance, we applied both our fuzzy parser and the full parser of Raincode to four sets of files: our test files, NIST COBOL test suite [29] (a well-known open-source test suite for COBOL85), and all COBOL files both pre- and post-migration of one of Raincode Labs' recent PACBASE migration projects. In total that gave us 32 small handcrafted files, one open-source project of 410 files, and two sets of 3014 files each. Whereas the first two sets are available to reproduce our experiment, the files of the PACBASE migration use case are confidential.

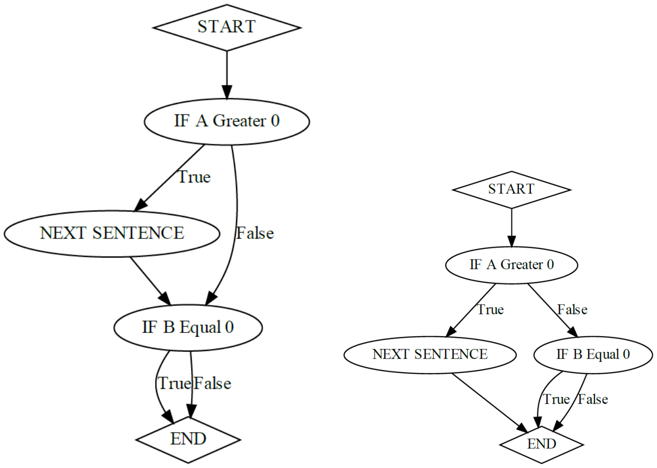

Fig. 4: Outputted CFGs for code example 2 and its variation using Raincode's parser

The set-up of our experiment is as follows: for each COBOL file we run the first phase of our fuzzy parser (creating the nodes, but not linking them) and Raincode's full parser. This is the fairest comparison we could come up with: at that point our fuzzy parser has not fully generated the CFG yet, but from Raincode side, we are left with a full parse tree that still needs to be transformed to a CFG as well. We chose not to measure the time it would take to obtain a CFG for both methods because in the case of Raincode parser, getting the CFG requires the parse tree to be written to disk in XML form and then be read from disk again. The alternative would be to modify their parser so that it creates CFGs immediately, but this would have required substantial help from someone at Raincode Labs. Also note that our fuzzy parser uses ANTLR to parse the embedded SQL code found inside the COBOL files, while the full parser leaves `EXEC` blocks untouched. We performed our comparisons both with and without parsing the embedded SQL, and found that running the ANTLR parser does not pose too much of an overhead, taking roughly 20 seconds for all 3014 files of our industrial uses cases (only those files actually contain `EXEC SQL` statements with embedded SQL code). The execution times reported in this paper were computed with ANTLR parsing turned on, presumably giving our fuzzy parser a slight unfair disadvantage in the comparison with the Raincode parser. There are limits in comparing tools in practice, and we hope our explanation helps the readers to interpret the results.

The Raincode parser reports its parsing times in a database while our fuzzy parser uses Python's `time.time()` to obtain the execution time. For each file, we run each parser 20 times and then calculate its average execution time. All of our tests were performed on a 12-core, 24-threads 3.70 GHz processor, with 32 Gb of 3500 MHz RAM. We did a few runs with a warm-up phase of 20 runs which were not included in the

	Av. file size (lines)	Fuzzy total time (s)	Full total time (s)	Speedup fuzzy w.r.t. full
Test files (32)	15	0.03	9.35	312
NIST (410)	781	37,45	127.87	3.4
Use case post-migration (3014)	1715	273.01	959.12	3.5
Use case pre-migration (3014)	2675	1272.29	947.53	0.7

TABLE I: Execution times on all file sets

average, but saw no significant changes in our numbers: neither for the fuzzy nor for the full parser. We therefore do not include warm-ups in our final reported numbers. The summary of our findings can be found in Table I. The average file size for each codebase is included, as it turned out to be relevant for the discussion to come. We then give the sum of the average execution time over the entire codebase for both the fuzzy and the full parser, and finally the speedup of using our fuzzy parser versus the Raincode full parser, for each codebase.

When analysing these results, we observe that the fuzzy parser offers a significant speedup when the files are small, but loses its advantage as the average file size grows larger. A likely explanation for this phenomenon is that the Raincode full parser is I/O bound, meaning that reading a file of any size is a fairly costly operation for them, while the parsing task itself has become very optimised over their parser's 20 year of existence. On the other hand, our fuzzy parser, implemented in Python, scans the input files faster, but sees its execution time grow as the files become longer and more complicated due to the amount of regex matches to find, and the complexity of keeping track of embedded condition and loops to generate the CFG. This explains why the full parser actually outperforms the fuzzy parser in the last line of Table I. The files pre-migration are quite large and contain lots of embedded structures (their inappropriate nestedness levels is one of the reasons for refactoring), providing a worst case for our fuzzy parser, which then becomes slower than the full parser. Both parsers seem to have near-linear performance.

To confirm this phenomenon, we created a synthetic test set. We took a base file of 1000 lines and grew it in size by copying parts of is contents up to 30K lines. We then ran both parsers on these manually-generated files to observe their behaviour. The resulting graphs are shown in Figure 5. On the top graph, we can observe that the fuzzy parser is faster when the files remain under 10K lines, but that execution time keeps growing with the file size, while the full parser's execution time grows more slowly, beating the fuzzy parser's execution time for files larger than 10K lines. The bottom graph plots each parser's execution time normalised by the file size, i.e. the "cost" of running a parser. As expected, running the full parser on small files has a very high cost, but this cost decreases sharply as the file size grows to 5000 lines, and keeps decreasing slowly afterwards. In a contrary fashion, the fuzzy parser is less costly to run on files smaller than 5000 lines, then roughly equivalent to the full parser's cost up to files of 10K, but then gets slightly costlier as file size increases. This aligns with expectations of a semi-parser as something that works fast on small inputs.

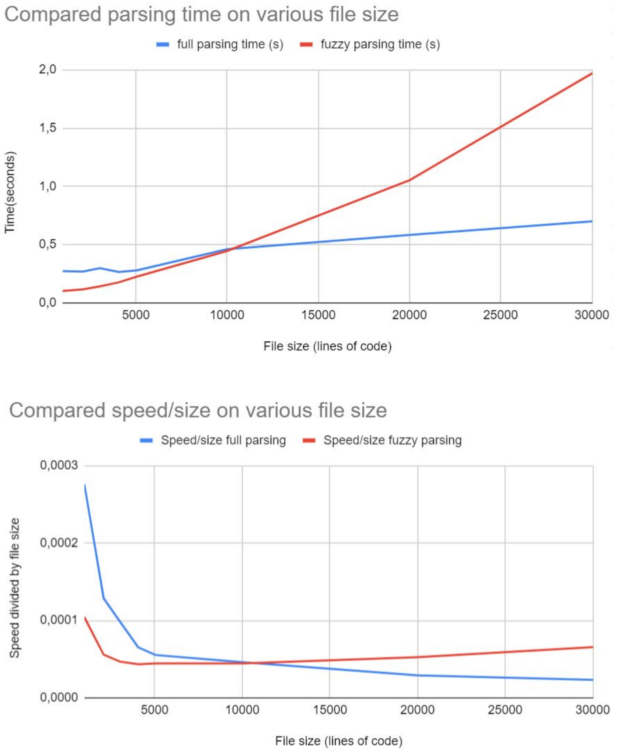

Fig. 5: Graphs comparing both parsers' performance on various file sizes

To obtain these numbers, we did some initial optimisations on our fuzzy parsing code, and pre-compiled it using the Nuitka Python compiler [16], which eventually allowed us to get a performance roughly equivalent to (or better than) that of the full parser for most files in our experiments. We would like to emphasise here that we are comparing the current implementation of our fuzzy parser prototype to an established highly-optimised industrial-strength parser. Further promising optimisations of the fuzzy parser exist, in particular to get rid of the slow regex module in favour of something faster.

While the execution times are significant (20 minutes for our longest run), this is not how our parser would typically be used in the industrial use case. In our experiment, we ran every file one by one in order to be able to compute the average execution time for all of them. As mentioned in Section III, the fuzzy parser would be used either while in a meeting with a customer, or ran on servers overnight after a full migration task. In the first case, the files would still be computed one-by-one, but only two at a time during discussions, requiring 0.42s on average for a pre-migration file, and 0.09s for its post-migration equivalent, which is more than acceptable and lower than the time that would be needed by the full parser (respectively 0.31s and 0.32s on average for pre- and post-migration files). In the second case, on a server, the process would be run in parallel to a very computationally heavy rewriting system for COBOL refactoring. The execution of our fuzzy parser is fairly lightweight and thus multi-thread-friendly, requiring only 5% of the processing power and 40Mb of RAM on the computer used for the experiment. This means that, putting the refactoring processes aside, up to 20 threads could be run in parallel on that specific machine, resulting in a total execution time of 77.2s to compute both sets (threads are fully independent), or just over a minute. For an overnight process, this overhead is again more than acceptable.

VII. Discussion

In this section, we discuss the viability of using semi-parsing techniques like fuzzy parsing, versus using full parsers, to generate CFGs; criteria of correctness, efficiency and modularity; as well as the semi-parser's ability to be dialect agnostic and its applicability to partial or non-compilable code.

A. Efficiency and Correctness

Both these criteria were discussed at length in the previous section. While our CFGs generated with a semi-parser were found to be correct when we analysed them in detail, using the fuzzy parser will not always result in faster execution times as compared to using the industrial-strength full parser we had at our disposal, especially not for larger programs. We already proposed a few solutions to this issue, the easiest one being to parallelise the execution of the CFG creation process when dealing with many files of larger size. However, execution time is not the only factor we should consider when talking about efficiency. The time resources that need to be invested before getting results is also of importance. Of course, if a full parser is already available for a given language, writing a script that extracts CFGs from its parse trees may indeed not be more complicated than writing a semi-parser. However, whereas we did have access to such a full parser for the case of COBOL (and welcomed it because it allowed us to compare the correctness and efficiency of our fuzzy parser with that full parser), especially when working with legacy languages we do not always have access to such a full parser. When no parsers are readily available, the time and expertise required to write a semi-parser will be far less than what is needed for writing a full parser. In our case, it took us a few months to obtain satisfying results for our fuzzy parser versus the estimated 2-3 years it could have taken for writing a full parser [28].

B. Modularity

We can distinguish two important modularity concerns: the ability to cherry-pick what we want to be contained (or not) in the produced CFG, and the ability to parse other embedded languages. When creating CFGs from parse trees, picking what will be included in the final graphs is fairly easy, although less flexible than the configuration file that our semi-parsing solution proposes. In regard to parsing embedded languages, combining two grammars into a single one is often difficult to impossible as explained in Section II. As we experienced, at least for the case of parsing embedding SQL in COBOL, semi-parsing makes this possible relatively straightforwardly.

C. Dialect agnosticism

It is well-known that many languages, and legacy languages in particular, have multiple dialects or versions, COBOL being a particularly impressive case with up to 300 language dialects [27]. Using a fuzzy parser, as long as the syntax of the structures we are interested in, remains roughly the same, the dialect used would not interfere much with the CFG generation. In case of a diverging syntax for one of the control flow structures considered, the only change needed would likely be to modify the regex expression describing it in the configuration file, making the adaptability of our fuzzy parser to language dialects very high.

D. Ability to handle partial or non-compilable code

When using a full parser, the code needs to be complete and syntactically correct in order to be processed. With fuzzy parsing, this is not required (although severe syntax errors within chosen structures will of course yield incorrect CFGs). In our industrial use case, this proves very useful in cases where the code is confidential, and the customer wishes to give away as little as possible of their codebase. Imagine a contract by which the customer agrees to give only the procedure divisions to Raincode Labs, but not the data divisions, which can be seen more critical to the customer because they contain all variable definitions and values, while the former "only" describes the execution. In this case, a full parser would need to be partly rewritten to analyse this code, while with a fuzzy parser we would be able to do so without any modification.

VIII. Conclusion

In this paper, we presented an alternative way of generating control flow graphs using the semi-parsing technique of fuzzy parsing. We explained how such graphs could be useful in the industrial use case of a migration process that refactors PACBASE-generated COBOL into human-readable and maintainable plain COBOL. The goal is to help improve the customer's trust in the migration process by using CFGs to show that the control flow of their codebase pre and post migration remains equivalent around critical database accesses.

Our choice of using semi-parsing allowed us to create a tool to generate such graphs that is easy to configure and adapt, can be run on partial or non-compilable code if required by the customer, can leverage existing full parsers to analyse embedded languages, and has execution times comparable to what can be obtained from an industrial-grade full parser, at least for the file sizes considered in the industrial use case.

In the future, we want to keep our focus on this industrial use case. Still left to do is to compare the generated CFGs before and after migration automatically, to highlight the differences in them (or lack thereof) around the database reads and writes. We will also work on a way to visualise these differences in a compact way that would be suited to be shown during live meetings with customers at Raincode Labs.

We also wish to create tool support to flag compared files automatically if they have enough divergences to require some manual analysis from engineers.

In parallel, working on an automated script to further validate that our generated CFGs are correct is desirable, as well as doing further optimisation work to address the scalability issue of the fuzzy parsers' performance for larger file sizes. The avenues of dropping the *re* module or moving away from implementing it in Python altogether, are the first that come to mind. Finally, we wish to apply this CFG generation using semi-parsing to other languages such as FORTRAN [7] or Julia [26].

Acknowledgments

We thank Raincode Labs for its collaboration and its support in providing the use case. We also thank the Innoviris research funding agency of the Brussels-Capital Region, for providing the funds necessary to conduct our CodeDiffNG Applied PhD research project.

References

[1] A. Alimucaj, "Eclipse Control Flow Graph Generator," SourceForge, http://eclipsefcg.sf.net, 2009.

[2] F. E. Allen, "Control Flow Analysis," in *Proceedings of a Symposium on Compiler Optimization*. ACM, 1970, p. 1–19.

[3] D. Blasband, *Rise and Fall of Software Recipes*. Reality Bites Publishing, 2016.

[4] A. Coet, L. Truong, V. Kumar, A. Naku, and R. Golosynsky, "StatiCFG," https://github.com/coetaur0/staticfg, 2018.

[5] D. de Muinck Keizer, "A Control Flow Graph Generator for Java Code," Ph.D. dissertation, Leiden University, The Netherlands, 2009. [Online]. Available: https://theses.liacs.nl/25

[6] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile parsing in TXL," *Automated Software Engineering*, vol. 10, no. 4, pp. 311–336, 2003.

[7] DEC-10-AFDO-D, *FORTRAN IV Programmer's Reference Manual*. digital, 1967.

[8] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev, "Improving Software Modernisation Process by Differencing Migration Logs," in *Proceedings of the 21st International Conference on Product-Focused Software Process Improvement (PROFES)*. Springer, 2020, pp. 270–286, DOI: 10.1007/978-3-030-64148-1_17.

[9] C. Deknop, K. Mens, A. Bergel, J. Fabry, and V. Zaytsev, "A Scalable Log Differencing Visualisation Applied to COBOL Refactoring," in *Proceedings of the Ninth Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 1–11.

[10] C. Deknop, "FuzzyParser GitHub repository," https://github.com/CelineDknp/SemiParsingCFG, 2022.

[11] J. Ellson, E. Gansner, Y. Hu, S. North *et al.*, "Graphviz," https://graphviz.org/, 2022.

[12] EnSoft Corp, "Atlas for Java and C — Understand Code Someone Else Wrote!" https://www.ensoftcorp.com/atlas/sdk/, 2022.

[13] R. Gopinath, "PyCFG for Python MCI," https://github.com/vrthra/pycfg, 2017.

[14] GrammaTech, "CodeSonar C/C++: SAST when Safety and Security Matter," https://www.grammatech.com/codesonar-cc, 2020.

[15] D. Grune and C. J. H. Jacobs, *Parsing Techniques — A Practical Guide*, 2nd ed. Addison-Wesley, 2008. [Online]. Available: https://dickgrune.com/Books/PTAPG_2nd_Edition/

[16] K. Hayen *et al.*, "Nuitka the Python Compiler," https://nuitka.net/index.html, 2022.

[17] Hewlett-Packard Development Company, "Survival Guide to PACBASE™ end-of-life," https://www8.hp.com/uk/en/pdf/Survival_guide_tcm_183_1316432.pdf, Oct. 2012.

[18] M. Hills, "Streamlining Control Flow Graph Construction with DCFlow," in *Proceedings of the Seventh International Conference on Software Language Engineering*, ser. LNCS, B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds., vol. 8706. Springer, 2014, pp. 322–341.

[19] IBM Corporation, "Db2 / 11.5 / Introduction to Embedded SQL," https://www.ibm.com/docs/en/db2/11.5?topic=da-embedded-sql, Dec. 2021.

[20] C. Jones, *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences*. Addison-Wesley-Longman, 1998.

[21] L. C. L. Kats and E. Visser, "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs," in *Proceedings of the 25th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 444–463.

[22] P. Klint, R. Lämmel, and C. Verhoef, "Toward an Engineering Discipline for Grammarware," *ACM Transactions on Software Engineering Methodology (ToSEM)*, vol. 14, no. 3, pp. 331–380, 2005.

[23] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 168–177.

[24] I. Kochurkin and I. Khudyashev, "MySQL (Positive Technologies) grammar," https://github.com/antlr/grammars-v4/tree/master/sql/mysql/Positive-Technologies, 2015–2017, licensed under the MIT License.

[25] R. Koppler, "A Systematic Approach to Fuzzy Parsing," *Software—Practice & Experience*, vol. 27, no. 6, pp. 637–649, 1997.

[26] T. J. Lab, "The Julia Programming Language," https://julialang.org, 2022.

[27] R. Lämmel and C. Verhoef, "Cracking the 500-Language Problem," *IEEE Software*, vol. 18, no. 6, pp. 78–88, 2001. [Online]. Available: https://doi.org/10.1109/52.965809

[28] V. Maslov, "Re: An Odd Grammar Question," http://compilers.iecc.com/comparch/article/98-05-108, May 1998.

[29] National Institute of Standards and Technology, "NIST Cobol Available Test Suites," https://www.itl.nist.gov/div897/ctg/cobol_form.htm, 2022.

[30] T. Neale, "GTIRB Ghidra Plugin," GitHub, https://github.com/GrammaTech/gtirb-ghidra-plugin, 2019.

[31] Python Software Foundation, "`re` — Regular expression operations," https://docs.python.org/3/library/re.html, 2022.

[32] Raincode Labs, "PACBASE Migration: Flexible Process," https://www.raincodelabs.com/pacbase/, 2021.

[33] J. Smits and E. Visser, "FlowSpec: Declarative Dataflow Analysis Specification," in *Proceedings of the 10th International Conference on Software Language Engineering (SLE)*. ACM, 2017, pp. 221–231.

[34] A. A. Terekhov and C. Verhoef, "The Realities of Language Conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000. [Online]. Available: https://doi.org/10.1109/52.895180

[35] UCLA Compilers Group, "Avrora: The AVR Simulation and Analysis Framework," http://compilers.cs.ucla.edu/avrora/cfg.html, 2003–2005.

[36] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. [Online]. Available: https://doi.org/10.1109/TSE.1984.5010248

[37] V. Zaytsev, "Formal Foundations for Semi-parsing," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014 ERA)*, S. Demeyer, D. Binkley, and F. Ricca, Eds. IEEE, Feb. 2014, pp. 313–317. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747184

[38] ——, "Grammar Zoo: A Corpus of Experimental Grammarware," *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)*, vol. 98, pp. 28–51, Feb. 2015. [Online]. Available: https://doi.org/10.1016/j.scico.2014.07.010

[39] ——, "Parser Generation by Example for Legacy Pattern Languages," in *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, M. Flatt and S. Erdweg, Eds. ACM, 2017, pp. 212–218. [Online]. Available: https://10.1145/3136040.3136058

[40] ——, "Event-Based Parsing," in *Proceedings of the Sixth Workshop on Reactive and Event-based Languages and Systems (REBLS)*, T. Kamina and H. Masuhara, Eds., 2019. [Online]. Available: https://doi.org/10.1145/3358503.3361275

[41] V. Zaytsev and A. H. Bagge, "Parsing in a Broad Sense," in *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, ser. LNCS, vol. 8767. Springer, Oct. 2014, pp. 50–67. [Online]. Available: https://doi.org/10.1007/978-3-319-11653-2_4

[42] V. Zaytsev and J. Fabry, "Fourth Generation Languages are Technical Debt," International Conference on Technical Debt, Tools Track (TD-TD), 2019. [Online]. Available: http://grammarware.net/text/2019/4gl-techdebt.pdf