Analyzing Multilingual Software

Removed for Review

Removed for Review

***Abstract*— Developer preferences, language capabilities and the persistence of older languages contribute to the trend that large software codebases are often *multilingual* – that is, written in more than one computer language. While developers can leverage monolingual software development tools to build software components, companies are faced with the problem of managing the resultant large, multilingual codebases to address issues with security, efficiency, and quality metrics. The key challenge is to address the opaque nature of the language interoperability interface: one language calling procedures in a second (which may call a third, or even back to the first), resulting in a potentially tangled, inefficient and insecure codebase. An architecture is proposed for efficient static analysis of large multilingual codebases – the MLSA architecture. Its modular and table-oriented structure addresses the open-ended nature of multiple languages and language interoperability APIs. We focus here on the construction of a call graph capturing both inter-language and intra-language. The algorithm for extracting multilingual call graphs from C/Python codebases is presented in Datalog rules, and several examples of multilingual software engineering analysis are presented. The state of the implementation and testing of MLSA is presented, and the implications for future work are discussed.**

Keywords— Software Engineering, Software Systems and Testing, Software and Systems Quality, Static program Analysis.

I. INTRODUCTION

Large software projects may typically have components written in different languages. Companies that have a large software codebase may face the issue of applying security, efficiency and quality metrics for a product spanning many languages [1] [2]. A developer or developer organization may choose one language for numerical computation and another for user interface implementation, or they may have inherited or be mandated to work with legacy code in one language while extending functionality with another. While there are many such drivers promoting multilingual codebases, they comes with significant software engineering challenges. Although a software development environment might support multiple languages (e.g., Eclipse IDEs) it may leave the language boundaries, language interoperability [3], opaque. While it may be possible to automatically inspect individual language components of the codebase for software engineering metrics, it may be difficult or impossible to do this on a single accurate description of the complete multilingual codebase.

Our objective is to develop software engineering tools that address large multilingual codebases in a lightweight, open[i] and extensible fashion. One of the key tools and prerequisites for several kinds of software analysis is the call graph. The call graph is also where language boundaries directly meet. In this paper we focus on the issues of generating multilingual call graphs using C/C++, Python and Javascript interoperability examples.

The MLSA (*MultiLingual Software Analysis*) architecture – a lightweight architecture concept for static analysis of multilingual software – is presented. We motivate our use of the *Datalog* language [4] for writing the filter programs that process language boundaries and we present examples of rules for some of the most common forms of interoperability APIs. Three implementation examples are presented showing the benefit of multilingual call graph analysis from the perspectives of efficiency and security. Our results are summarized and future directions discussed in the final section of the paper.

II. PRIOR WORK

A multilingual codebase can arise for historical reasons. As languages rise and fall in popularity, their varying use results in a codebase that reflects developer or commercial historical preferences [5]. Libraries for numerical computation may have been constructed in FORTRAN, C and C++ for example, and front-end libraries may have been built in JavaScript. But there are also good software engineering reasons for choosing to develop different parts of a project in different languages: [6] argues for increased developer productivity among other benefits.

Nonetheless, a multilingual codebase also poses significant software engineering challenges. Although multilingual code is common, development tools tend to be language specific, with some cross-platform functionality. As one example among many, *Checkmarx*[ii] offers static analysis [7] for a wide range of languages, but individually. While each language codebase can be analyzed, the transitive closure of calls across language boundaries is harder to capture. This can give rise to several issues:

- Redundancy, e.g., procedures in several different language libraries for the same functionality, necessitating refactoring [8].
- Debugging complexity as languages interact with each other in unexpected ways [9].
- Security issues relating to what information is exposed when one language procedure is called from another [10].

One approach to handling the issues of multilingual systems is to use a versatile monolingual environment [11] [12], but of course this is not too useful an approach for existing multilingual codebases. Another approach leverages a metalanguage, e.g., Rascal [2], which provides tools with

which program analysis algorithms can be written for different languages. Of course, this does not specifically address the problems that arise due to the language boundaries.

Our approach here is more directed and more 'bare bones' – targeted primarily at the language interface and using little extra infrastructure. In the next section, we will describe the architecture for *MLSA*, a set of lightweight open source tools for multilingual software analysis. There are many important software metrics and analyses for large software architectures [13]. In the subsequent section, we delve into one specific analysis: call-graph generation for multilingual codebases,

III. MLSA Architecture

It is very likely that new computer languages will continue to be developed and used, while existing languages will also tend to persist [5] leading to an increasingly crowded and complex landscape of multilingual software development. While there are certainly many cross-language interoperability APIs already, new ones are also continually being developed for new languages or language mixes. Responding to this, we propose a lightweight and modular architecture concept for *MultiLingual Software Analysis* - the MLSA Architecture.

User directed static analysis of a multilingual codebase is carried out in MLSA by the application of pipelines of small filter programs. Each filter program produces and consumes CSV (comma separated value) table files. This architecture is illustrated in Figure 1. The initial filters consume a monolingual AST (abstract syntax tree) generated by an appropriate monolingual parser in the *monolingual layer* of the architecture. Because each language AST differs, the programs that consume the monolingual ASTs to process interoperability APIs must be also language specific; this happens in the *interoperability layer*. A small set of interoperability APIs is currently handled, but the addition of more is relatively modular and contained within the interoperability layer. In the final, *multilingual layer*, all the program data has been transformed to multilingual, and procedures in different languages can be related to one another for static analysis.

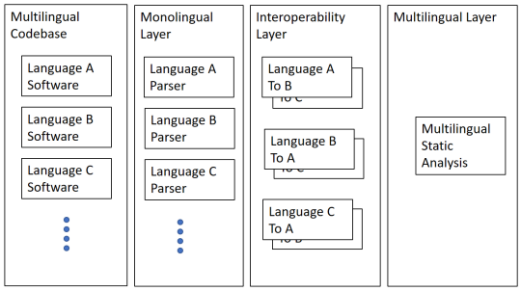

Figure 1: MLSA Architecture (data flows right to left)

The current implementation covers C, Python and JavaScript languages. At the monolingual layer, the AST for C programs is generated by *Clang-Check*, those for Javascript using *SpiderMonkey*, and a Python library function generates the Python AST. At the interoperability layer, filter programs for *Python.h*, *PyV8* and *JQuery* support limited interoperability between the three languages.

MLSA filter programs for static analysis include the generation of the forward and reverse control flow, the identification of variable use and of variable assignment, the allocation of heap memory, the identification of procedure calls and so forth. Consider an illustrative example of such a pipeline: A Reaching Definitions Analysis [14] (RDA) filter.

A monolingual AST filter `CASN` inputs the (C) AST and generates a CSV file of variable assignments locations. Another filter `CRCF` generates the reverse control flow as a CSV file. A multilingual filter `RDA` takes both as input and calculates reaching definition for each assignment.

The MLSA architecture promotes modularity. In the reaching definitions example of the previous section:

1. Adding extra languages just requires adding new monolingual filters for the language. For Python these are the `PASN` and `PRCF` filters.
2. Modifying analyses just requires reconfiguration of the analysis pipeline: for example, substituting the CSV output of `CCON` containing condition locations for that of `CASN` would perform reaching definitions analysis for the condition statements.

The architecture also supports open source interactions. It is relatively easy to specify and build new filters, or replace old filters with more efficient ones.

IV. Creating a Multilingual Call Graph with MLSA

This section details a specific analysis implemented on the MLSA architecture: creation of a multilingual call graph. The implementation relies on monolingual AST filters for C/C++, Python and Javascript to generate CSV files containing monolingual call graph information. These are in turn processed by language boundary filters that characterize specific cross-language interoperability APIs, generating modified CSV files which can then be integrated into a single, multilingual call graph.

The linkage of an interoperability API call (in the call graph) to the cross-language procedure called is based directly on the specification of that API. Consider an example from the *python.h* C/Python interoperability API [iii]: A python function *pfun* defined in the file *pfun.py* can be called from a C program by first loading the module using *PyImport_Import()* with argument *pfun.py*, by getting a pointer to the function *pfun* using *PyObject_GetAttrString()*, and finally by calling the function using *PyObject_CallObject()* with the pointer as argument. It is attractive to implement interoperability processing with a declarative rule for two reasons:

1. It becomes easy to relate it to the specification of the interoperability API.
2. More rules for this and other interoperability APIs (as more are built or achieve widespread use) can be added in a modular fashion.

Datalog, a database query language that is a subset of Prolog, has been adopted for program analysis by some researchers [15] [4]. Advantages to its use include the ease and

conciseness of expressing program analysis rules in Datalog as opposed to, for example, C++ or Java, as well as the ease with which the results of separate analysis can be combined, due to the uniform representation. MLSA already stores the data exchanged between filter programs in a tabular (CSV) form that can be readily transformed to Datalog predicates to build the extensional databases that are the input for the filters.

A. Call Graph Terminology

The following relations and functions represent the call graph and provide functionality for processing interoperability APIs. Relations start with a capital, and variables will be lower case, prefixed by a question mark. Functions are indicated by use of square brackets,

- CALLS(*?x, ?y, ?arglist, ?line)*: *?x* calls *?y* at line number *?line* with the argument list *?arglist*. The CSV files generated by the monolingual AST filters generate this extensional database.
- FIRST[*?arglist*] = *?expr:* The first argument in *?arglis*t is *?expr*.
- REST[*?arglist*] = *?restargs*: with the first argument removed from *?arglist* the result is *?restargs*.
- EVAL[*?expr, ?line*]=*?val:* Static analysis of the value of *?expr* at *?line* yields *?val* as one possible value (and *?val* could be unknown). (We use Reaching Definitions to implement this [14]).
- STRING(*?expr*)*:* The value of *?expr* is a string.
- UNKNOWN(*?expr*)*:* the value of *?expr* cannot be determined statically.

B. Language Boundary Processing

There are many different language interoperability APIs for executing code or calling procedures from one language in another. And more are likely being added all the time. Based on our study of the existing interoperability APIs, it's possible to cluster how these support procedure calls into three groups: *Anonymous, File-based* and *Procedure-based*. Each of these will be addressed in turn, showing how they can be leveraged to generate a multilingual call graph from separate monolingual call graphs and rules for processing the interoperability API. These rules are expressed in Datalog.

1) Anonymous

The most typical form of interoperability API allows a string containing code in one language to be executed in a second. Examples of this include:

- *Python.h*: *PyRun_SimpleString()* - Will execute a string of Python commands in C/C++
- *Stdlib.h*: *system()* - Will execute a Linux shell command from C/C++. Variants exist in many languages, such as Python's *os.system()*.
- PyV8: *PyV8.JSContext.eval()* – executes a Javascript string from Python.
- Emscripten: *emscripten_run_script()* – executes a Javascript string from C/C++.

Because these APIs allow an unlabeled block of code to be executed from a single procedure call in the program, we refer to them as *anonymous multilingual procedures*. They pose many issues for software engineering because the code string may be opaque to code analysis: the code may be created at run time, making it difficult to determine a-priori if the code abides by software engineering criteria design to promote correct, maintainable, efficient and safe code.

The approach proposed here is to add an edge to the call graph for each possible value of the argument string that can be statically identified (e.g., a string literal). The edge leads to a special node labeled *Anonymous*. In the case that at least one possible value of the argument string cannot be identified, that node is labeled *Anonymous-Dynamic*. Although such multilingual code may in fact be correct and safe, it is certainly worthwhile from a software engineering perspective to label it clearly for more thorough inspection.

The Datalog rules to handle *PyRun_SimpleString* are as follows (Datalog syntax *rule-result :- predicate, predicate, ..)* :

```
CALLS( ?x, "Anonymous", ?actargs, ?line ) : -
    CALLS( ?x, "PyRun_SimpleString", ?actargs, ?line ),
    FIRST[ ?pactargs ]  = ?expr,
    EVAL[ ?expr, ?line ] = ?y,
    STRING( ?y ).

CALLS( ?x, "Anonymous-Dynamic", ?actargs, ?line ) : -
    CALLS( ?x, "PyRun_SimpleString", ?actargs, ?line ),
    FIRST[ ?pactargs ]  = ?expr,
    EVAL[ ?expr, ?line ] = ?y,
    UNKNOWN( ?y ).
```

2) File-Based

Another common form of interoperability API allows a filename for a program written in one language to be executed from a second language. The program will at least contain some statements and it may call other procedures. The objective is to add a node to the call graph that represents calling the main program in the file as a procedure, and to add the procedure calls by this node as edges in the call graph.

Examples of this form of cross-language API include:

- *Python.h*: *PyRun_SimpleFile(Filename)* – will execute the python code in Filename from a C/C++ program.
- PyV8: *PyV8.JSContext.eval(jsfile.read)* – executes the contents of a Javascript file *jsfile* from Python.
- JQuery: *JQuery.ajax(url: "pyfile.py")* – executes a Python file from Javascript.

If the possible values for the filename can be statically determined, then the code to be executed is directly visible to code analysis. It is reasonable to assume that a call graph can be generated for the file, and the issue is just to stitch this in to the calling language's call graph. However, if the filename is unknown, then we have the same opacity issues as we had for anonymous multilingual procedures and thus this needs to be flagged for inspection.

The approach proposed here is to add an edge to the call graph for each possible value of the filename, and to attach the call subgraph for the file at that node. The Datalog code to handle *PyRun_SimpleFile* is as follows:

```
CALLS( ?x, ?y, ?arglist, ?line ) : -
    CALLS( ?x, "PyRun_SimpleFile", ?actargs, ?line ),
    FIRST[ ?pactargs ]  = ?expr,
    REST[ ?actargs ]    = ?arglist,
    EVAL[ ?expr, ?line ] = ?y,
    STRING( ?y ).

CALLS( ?x, "FileBased-Dynamic", ?actargs, ?line ) : -
    CALLS( ?x,"PyRun_SimpleString", ?actargs, ?line ),
    FIRST[ ?pactargs ]  = ?expr,
    EVAL[ ?expr, ?line ] = ?y,
    UNKNOWN( ?y ).
```

3) Procedure-based

The most sophisticated form of interoperability API call is where the procedures from a file written in one language are read into a context object in a second language, and then can each be called from the context object with procedure parameters. Examples of this include the follow:

- *Python.h*: *PyObject_CallObject(pFunc, pArgs)* – when *PyImport_Import()* is used to load a python module, individual functions from the module can then be called with *pFunc*.
- JSAPI: *JS_CallFunctionName(cx,gl, fn, args, &ret)* – calls Javascript function *fn* from a C program with arguments *args* and return *ret*.
- Python-Bond: *js.call(fname, args)* – calls Javascript function *fname* with arguments *args* in Python.

This multiple step interoperability API means that the filename with source code is presented first to (one or more) procedure call(s) to create the multilingual context. If that argument can be statically evaluated, then a call graph can be generated for all the procedures that can be called. If at least one of the possible values of the filename is unknown, then any call to a procedure from this context is labeled in the call graph as a Procedure-Based Dynamic call to flag it for more detailed code review. The Datalog rules for *PyObject_CallObject* are:

```
CALLS( ?x, ?y, ?arglist, ?line ) : -
    CALLS( ?x, "PyObject_CallObject", ?pactargs, ?line ),
    FIRST[ ?pactargs ]  = ?expr,
    REST[ ?actargs ]    = ?arglist
    EVAL[ ?expr, ?line ] = ?y,
    STRING( ?y ).

CALLS( ?x, "ProcedureBased-Dynamic", ?arglist, ?line ) : -
    CALLS( ?x, "PyObject_CallObject", ?pactargs, ?line ),
    FIRST[ ?pactargs ]  = ?expr,
    REST[ ?actargs ]    = ?arglist
    EVAL[ ?expr, ?line ] = ?y,
    UNKNOWN( ?y ).
```

These are simplified rules in that:

1. The first argument to *PyObject_CallObject*, ?*expr*, is a function pointer whose value is set by a call to *PyObject_GetAttrString()* with arguments the imported context and the name of the function to call.
2. The argument list is composed by *PyTuple_SetItem* and not immediately in the same form as C/C++ procedure argument lists.

V. EXAMPLES

In this section, we consider several example-analyses of multilingual call graph using MLSA.

A. Multilingual Example

The graph in Figure 2 depicts a multilingual program in which a C++ program calls a Python program, and that Python program then calls a JavaScript Program. The C/C++ programs are represented by oval nodes, the Python programs by rectangular nodes, and the JavaScript programs by hexagonal nodes.

In our implementation, the MLSA programs can determine cross-language calls based on a small set of interoperability APIs, mainly *Python.h*'s *PyRun_SimpleFile* for calling a Python program from a C/C++ program, *PyV8*'s *eval*()

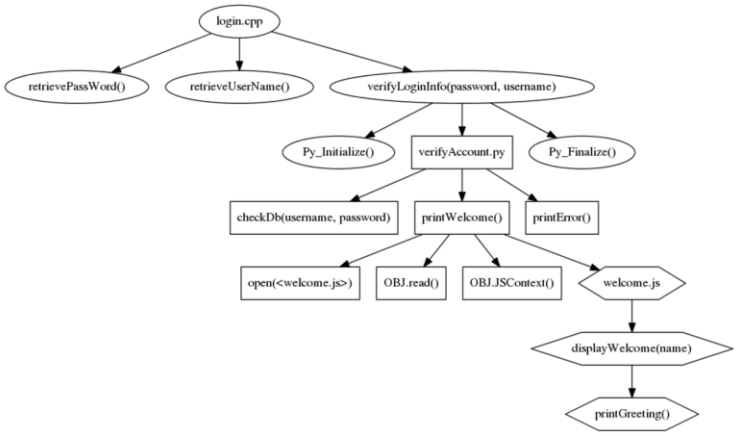

Figure 2: Multilingual Call Graph

function for calling a JavaScript program from a Python program, and *JQuery's* ajax function for calling a Python program from a JavaScript program The MLSA pipeline contains separate programs for analyzing the AST of each program based on the language in which the program was written. These programs parse the AST to recover all the function calls, their arguments, and their scope (where they are found in the program – inside of a function or the main body of the program). The end product of these computations is a CSV file for each program. The CSV files are processed by filters for the interoperability APIs. Currently these are not written in Datalog but in Python.

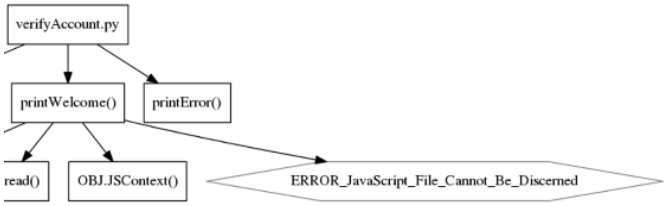

Figure 3: Statically unresolved call from Python to Javascript

Those resulting CSV files are combined into one, multilingual CSV file, which is then used to create a tree of all the function calls in the system. The graph in Figure 2 is a rendering of a DOT file generated from the multilingual CSV by *GraphViz*.

B. Dynamic Call Example

Any multilingual calls that are not resolvable before run-time pose software engineering risks and should be flagged for review. Figure 3 depicts a potential security issue in which a Python program *verifyAccount.py* calls a JavaScript program, but the identity of the JavaScript program cannot be determined before run time.

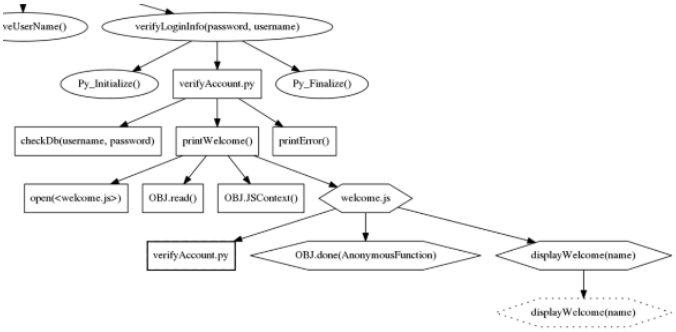

Figure 4: Multilingual Call Graph with Circularity

This occurs in this case because the argument to the *js.call()* procedure is an expression. The value of the expression is computed as unknown after a Reaching Definitions data flow analysis. The call is still placed into the call graph but with an ERROR flag.

C. Hidden Circularity Example

One of the main challenges we identified with a multilingual codebase is hidden circularity. For example, the C/C++ code may be validated free of unsafe C/C++ calls. But there may be interoperability API calls, which on their own seem safe, but when their call graphs are inspected they result in unsafe calls back to the original C/C++ code.

The graph in Figure 4 is an example of such a circular system, in which the Python program *veriftyAccount.py* calls the JavaScript program *welcome.js*, and then the same JavaScript program *welcome.js* calls the Python program *verifyAccount.py*. The circularity of the system is represented visually on the graph in Figure 4 by the node outlined in a thick perimeter. The call may be fine, but because it is hidden from regular software engineering inspection behind the interoperability API, it requires another look.

The graph also shows a standard recursive call cycle in *displayWelcome()*. On the bottom right, the graph shows this recursive call as a node outlined in black dots. Because such recursion is not problematic, it's not flagged for more inspection.

VI. Implementation and Testing

We have implemented the language boundary filters and call graph construction methods of Section 4 with some restrictions. They are not implemented in Datalog. For each function call, the parsing programs retrieve the name of the function called, the scope of the function call (whether the function was called inside a function definition, the main function, or even another function call), and the arguments of the call. The arguments can be literals (such as a character, string, integer, double, or Boolean) or variables. For now, the variable's name can be retrieved, but the value of the variable is not available unless it is specifically initialized or assigned to a strong. That is, the static evaluation of the RDA results is limited to assignment to a constant. The output of the call graph filter is a CSV file which has a series of rows, one per call, including: Parent procedure name; Called procedure name; and, Argument strings for the call.

To test the programs in the pipeline, a small codebase of 35 Python and 30 C/C++ programs was built to measure robustness and. The programs were collected by Internet browser search. The first 30 C/C++ and Python programs that were returned from search that were shorter than 100 lines of code were selected. (In fact, 35 Python programs were added due to the calling relations between some of the programs.)

Of the 35 Python programs collected, 8 were successfully processed and did not encounter any errors when creating the CSV file. The pressing problem with the monolingual filter for the Python files is that it cannot handle keyword arguments or lambda arguments. Other less pressing issues with the software developed are as follows: cannot handle dictionary structures; only works for calls and arguments that are expressed using binary operations; cannot handle dynamic dispatch; cannot handle function calls with attributes that have arguments; cannot handle list operations as arguments.

Of the 30 C/C++ programs collected, 22 did not encounter any errors while the CSV file was being created. The main issues that the C/C++ monolingual filter program cannot handle include: python calls other than *PyRun_SimpleFile*; redefinitions as functions (functions with the same name as functions in standard libraries); definitions in external C files.

In addition to the C/C++ and Python programs used for testing, 5 C/C++ programs that call Python programs, along with those Python programs, were also collected to test combining CSV call graph files and to create a multilingual call graph. All 5 C/C++/Python combinations were successful.

VII. Conclusion

This paper has addressed the issues faced by companies that must manage software architectures and libraries in different languages to enforce security, efficiency, and quality metrics. Because many existing software engineering tools are monolingual, even though multilingual code is widespread, issues relating to the language boundaries may get overlooked.

We proposed an architecture comprised of monolingual filter programs that analyze single language AST and identify the cross-language boundary. The filters generate language independent information in tabular CSV format. Additional multilingual filters operate on the CSV files in pipelines. This architecture has advantages of modularity and efficiency and is open-source friendly. We noted that the open-ended nature of the challenge of dealing with multiple languages and language interoperability APIs, as well as the tabular form data on the MLSA pipeline, motivated our use of Datalog's declarative, rule-based approach to writing multilingual filters.

We focused on one specific problem in this paper, the generation of multilingual call graphs. The challenge identified is to build a single call graph from a set of monolingual call graphs. Datalog rules were designed to handle three common interoperability forms. Examples of the software engineering use of multilingual call graphs were presented as well as preliminary performance results.

Ongoing work involves extending the performance testing to a larger codebase and including timing and comparisons with other call graph tools. MSLA does not use Datalog for multilingual processing of interoperability APIs or other analyses such as RDA; these are written in Python. We will Datalog so that the addition of more languages and filters becomes an open and modular process. Because our goal is to process large codebases, the efficiency of Datalog could be raise problems. We plan to leverage current literature in highly-efficient static analysis with Datalog [16] [4].

Acknowledgements

Removed for review.

VIII. References

[1] Z. Mushtak and G. Rasool, "Multilingual source code analysis: State of the art and challenges," in *Int. Conf. Open Source Sys. & Tech.*, Lahore, Pakistan, 2015.

[2] T. van der Storm and J. Vinju, "Toward Multilingual Programming Environments," *Science of Comp. Prog.; Special issue on New Ideas and Emerging Results in Understanding Software*, vol. 97, pp. 143-149, 2015.

[3] D. Barrett, A. Kaplan and J. Wileden, "Automated support for seamless interoperability in polylingual software systems," in *4th ACM SIGSOFT symposium on Foundations of software engineering*, New York, 1996.

[4] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *Int. Conf. on Datalog*, Oxford UK, 2010.

[5] L. Meyerovich and A. Rabkin, "Empirical Analysis of Programming Language Adoption," in *ACM OOPSLA*, Indianapolis, IN, 2013.

[6] M. Toele, "Multilanguage Software Development with DLX," Dept. of Computer Science, University of Amsterdam, Amsterdam Netherlands, 2007.

[7] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, 2016.

[8] D. Strien, H. Kratz and W. Lowe, "Cross-Language Program Analysis and Refactoring," in *6th Int. Workshop on Source Code Analysis and Manipulation* , 2006.

[9] S. Hong and e. al, "Mutation-Based Fault Localization for Real-World Multilingual Programs," in *30th IEEE/ACM International Conference on Automated Software Eng.*, 2015.

[10] S. Lee, J. Doby and S. Ryu, "HybriDroid: static analysis framework for Android hybrid applications," in *31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, 2016.

[11] J. Heering and P. Klint, "Towards monolingual programming environments," *ACM Trans. on Prog. Languages & Systems LNCS*, vol. 174, 1985.

[12] B. Seeber and M. Takatsuka, "Toward multilingual component-based software respositories: The intermediate Graph Format," Report #662 School of Info. Tech., University of Sydney, Syndey Australia, 2010.

[13] J. Lakos, Large-Scale C++ Software Design, Addison-Wesley; 1st Edition, 1996.

[14] F. Nielson, H. Nielson and C. Hankin, Principles of Program Analysis, Springer, 2005.

[15] J. Whaley, D. Avots, M. Carbin and M. Lam, "Using Datalog with Binary Decision Diagrams for program Analysis," in *3rd APLAS*, Tsukuba Japan, 2005.

[16] J. H., S. B. and S. P., "Soufflé: On Synthesis of Program Analyzers.," in *Computer Aided Verification (Springer Lecture Notes Comp. Sc., 780.)*, Toronto, Canada, 2016.

[i] Repository reference removed for review.

[ii] http://www.checkmarx.com

[iii] https://docs.python.org/2/c-api/index.html