Lightweight Multilingual Software Analysis

Damian M. Lyons[1], Anne Marie Bogar[1] and David Baird[2]
[1]*Department of Computer and Information Science, Fordham University, New York NY, U.S.A.*
[2]*Bloomberg L.P., New York NY, U.S.A.*

Keywords: Software Engineering, Programming Languages, Software Systems and Testing, Software and Systems Quality.

Abstract: Large software systems can often be multilingual – that is, software systems are written in more than one language. However, many popular software engineering tools are monolingual by nature. Nonetheless, companies are faced with the need to manage their large, multilingual codebases to address issues with security, efficiency, and quality metrics. This paper presents a novel lightweight approach to multilingual software analysis – MLSA. The approach is modular and focused on efficient static analysis computation for large codebases. One topic is addressed in detail – the generation of multilingual call graphs to identify language boundary problems in multilingual code. The algorithm for extracting multilingual call graphs from C/Python codebases is described, and an example is presented. Finally, the state of current testing on a database of programs downloaded from the internet is detailed and the implications for future work are discussed.

1 INTRODUCTION

Companies with a large software base often face the challenge of having to manage software architectures and libraries in different languages to enforce security, efficiency, and quality metrics (Mushtak and Rasool, 2015) (van der Storm and Vinju, 2015) (Lakos, 1996). Software development environments such as Eclipse tend to be language specific – a multiple language project would be developed in a set of Eclipse IDEs for each language and a common project. However, to address questions such as refactoring for efficiency, it is necessary to be able to analyse the entire existing code base, and existing software tools are weaker in this cross-platform aspect of multilingual systems (Strien, Kratz, and Lowe, 2006) (Hong and al, 2015).

Although automatic software analysis tools can be of great value in software engineering, their widespread use is limited by many factors (Christakis and Bird, 2016). Rather than proposing a common language model or metalanguage and complex IDE for cross-platform software engineering – a top-down solution - we take the approach of developing a set of simple, open source tools to support static analysis of a multilingual code base from the bottom-up. We present an overview of our toolset, which we will call *MLSA* (*M*ulti*L*ingual *S*oftware *A*nalysis: pronounced Melissa) in this paper, and also present our solution to one key issue in making multlingual call graphs.

In the next section we briefly overview the current literature and our motivation. Section 3 introduces the *MLSA* approach and architecture, a bottom-up, lightweight perspective on static analysis tools. Section 4 motivates and delves into detail on a specific topic of importance, generating multilingual call graphs. Our results are summarized and future directions charted in the final section.

2 PRIOR LITERATURE

Heterogeneous or multilingual code bases arise in many cases because software has been developed over a long period by both in-house and external software developers. 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.

A multilingual codebase gives rise to many software engineering issues, including

- Redundancy, e.g., procedures in several different language libraries for the same functionality, necessitating refactoring (Strien, Kratz, and Lowe, 2006).

- Debugging complexity as languages interact in unexpected ways (Hong and al, 2015).
- Security issues relating to what information is exposed when one language procedure is called from another (Lee, Doby, and Ryu, 2016).

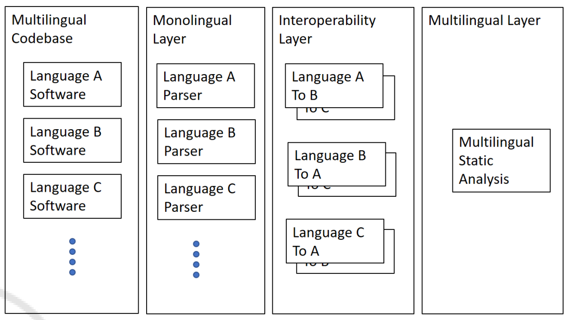
Although multilingual code is common, development tools tend to be language specific, with some cross-platform functionality. As one example among many, *Checkmarx*[1] offers static analysis (Christakis and Bird, 2016) for a wide range of languages individually. One approach to handling the issues of multilingual systems is to instead use a versatile monolingual environment (Heering and Klint, 1985), but of course this is not too useful an approach for existing multilingual codebases. A more ‘reverse engineering’ friendly approach is to leverage a metalanguage, e.g., Rascal (van der Storm and Vinju, 2015), 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 (Lakos, 1996). In the subsequent section, we delve into one specific analysis: call-graph generation for multilingual code bases using C/Python programs as an example.

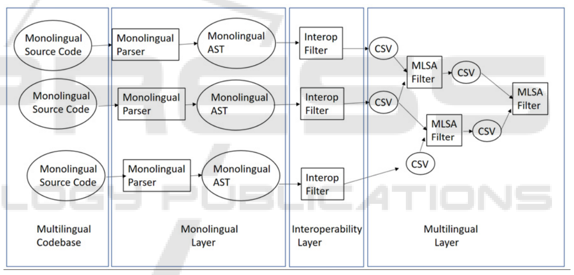
3 THE MLSA ARCHITECTURE

User directed static analysis of a multilingual code base is carried out in MLSA by the application of pipelines of small filter programs, producing and consuming CSV (comma separated value) table files. The initial filters consume a monolingual AST (abstract syntax tree) generated by the appropriate monolingual parser. This lightweight, open source architecture is shown in Figure 1; The MLSA Software Architecture[2] is shown in Fig. 1(a), and an example of a dataflow for an MLSA analysis is shown in Fig. 1(b).

Static analysis in MLSA begins with the monolingual layer in Fig. 1(a), where a language specific parser generates the AST for each monolingual program component in the multilingual codebase. Currently our implementation covers C, Python and JavaScript. The AST for the C programs is generated by Clang-Check, those for Javascript using SpiderMonkey, and a Python library function generates the Python AST.



(a) MLSA Software Architecture



(b) Example MLSA Dataflow

Figure 1: MLSA 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 layer, all the program data has been transformed to multilingual, and procedures in different languages can be related to one another. An example of this processing is presented in the next section.

[1] http://www.checkmarx.com.
[2] http://goo.gl/5tFQ7t.

3.1 Modularity

Example multilingual layer MLSA filter programs 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 (Nielson, Nielson, and Hankin, 2005) (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 RDA 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 RDA for the condition statements.

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

3.2 Computational Efficiency

The MLSA architecture was also designed with computational efficiency in mind. The policy of dividing the analysis into pipelines was chosen with the objective of making parallelism and dependency explicit. For relatively small multilingual codebases, a static analysis network (as in Figure 1(b)) could be distributed among multiple cores. The parallelism and dependency can be derived directly from the pattern of AST and CSV file use.

For a large codebase, in a realistic large company scenario with a widely-distributed set of code developers and contractors, a static analysis network might need to function continually (a daily basis for example) on cloud computing, regenerating and updating CSV file components across the network. In this scenario, the architecture also promotes a 'just in time' efficiency where CSV files are only recalculated when needed by code changes, with dependency information from the CSV files.

3.3 Multilingual Analysis

Call graph analysis (CGA) is a useful software engineering tool (Ali and Lhotak, 2012). In particular, for multilingual code, the call graph can be used to investigate the boundary line between languages, a boundary that is opaque in many tools. For example, a C program may call a Python procedure in addition to many C procedures. Consider that one such C procedure `OpenPort` exposes a security risk and needs to have its invocations pass a security review. Just looking at the call graph of the C procedures, some of which may invoke the Python procedure, can give the false security that it shows all the call sequences for the program. However, the Python code may itself call `OpenPort` or may call other procedures in that same C program that in turn call `OpenPort`.

In the next section, we present one specific problem we are addressing with MLSA – the construction of multilingual call graphs – with the objective of eliminating issues with the opacity of language boundaries.

4 MULTILINGUAL CALL GRAPH ANALYSIS

Call Graph for the program S is defined $CG(S_c)=(V_c, E_c)$ consisting of a set of nodes:

- $V_c=\{(pname,parglist)\}$ where $pname \in Procs(S_c)$ is a procedure name within the program S_c, and *parglist* is the argument list in the procedure call;
- $E_c \subseteq V_c{}^2$ links a node v to node u *iff* some execution of *v.pname* calls *u.pname* with arguments *u.parglist.*

For imperative language without first class functions constructing a CG is not challenging. For functional languages and OO languages, the issue of dynamic dispatch complicates the construction, and a technique such as Control-Flow Analysis (CFA) (Nielson, Nielson, and Hankin, 2005) must be used. Calling a cross-language procedure may be almost trivial (in the C/C++ case) or may involve a cross-language API as in the case of JNI (C/Java) or `Python.h` (C/Python) and others. A monolingual call graph analysis will yield leaf nodes that are the cross-language API calls. We restrict the call graph CG to be a tree for ease of display, with recursive calls as leaf nodes.

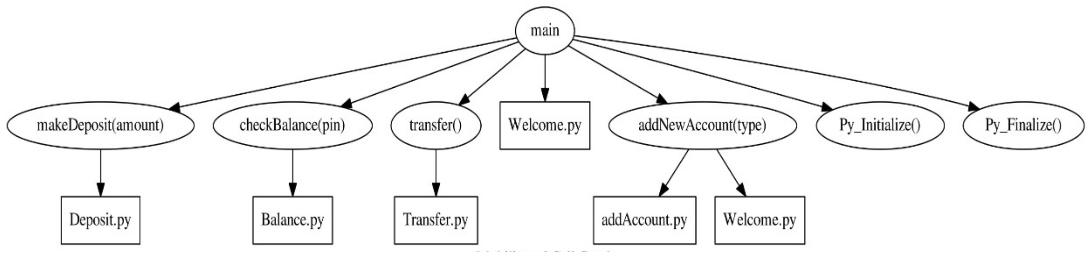


Figure 2: Example C Call Graph.

4.1 MLSA CG Construction

The *MLSA* approach is to develop a set of filters, one per language, that 'disambiguates' the cross-language API so that the name of called cross-language procedure and its arguments are directly visible. If this can be done, then the construction of the multilingual call graph is not difficult.

For example, consider a C program *S.c* that calls some Python procedures defined in *S.py*. The Python call graph filter generates a call graph CG_p from its source input *S.py*, $Py\text{-}CG(S.py)= CG_p$ and the C call graph filter similarly generates $C\text{-}CG(S.c)= CG_c$. The multilingual call graph CG is constructed as follows:

```
1. $CG=CG_C$
2. For each leaf $v\in V_C$
     if $v.pname\in\{u.pname \mid u\in V_P\}$
       copy the subtree with root $u$
       to $CG$ with root $v$.
```

The more difficult step is the disambiguation of the cross-language API to generate the monolingual call graphs above.

Consider the C/Python boundary API (`Python.h`[3]): Python code can be called from C non-interactively in the following ways (each of which have several variants and may require setup code):

- `PyRun_SimpleString(pyCodeString)`
- `PyRun_SimpleFile(filePtr, fileName)`
- `PyObject_CallObject(pFunc,pArgs)`

The first just executes whatever code is in `pyCodeString`. The second executes whatever code is in `fileName`. The last executes the python function `pFunc` with arguments `pArgs` (where the python module needs to have been loaded a-priori using `PyImport_Import()`).

While the first can be treated simply as an unnamed python procedure call, the other two are more challenging because the name of the python procedure to be called is given by an argument value. If the argument is a variable or expression, then this is a constrained version of the dynamic dispatch problem. Our approach is to use a Reaching Definitions Analysis (RDA) to determine the set of possible values (Nielson, Nielson, and Hankin, 2005) for the arguments of the cross-language API call.

4.2 MLSA Language Boundary Filters

Let us consider the program S_c to be a set of *(ℓ,b)* basic block *b* (elementary statement) with line number *ℓ*. *MLSA* extracts this information from the language AST file. The set *B* of elementary statements includes a procedure call statement, and for *(ℓ,b), b∈B* procedure call, we define:

- *target(b)*: name of the called procedure
- $arg(b)=a_0,\dots,a_n$: arguments of the call

Finally, we define $RDA(p,X,\ell) = \{(x,\ell')|x\in X\}$ to be the line number *ℓ'* of the last assignment in procedure *p* for each variable *x*. The API call (and its variants) `PyRun_SimpleFile` is processed as:

```
If $(\ell,b),\ b\in B,\ target(b)=$PyRun_SimpleFile
 For $(x,\ell')\in RDA(p,\{a_0\},\ell)$, with $arg(b)=a_0,\dots,a_n$
   Calculate $y=Eval(x,\ell')$, and if $y\neq\varnothing$,
     Add $(y,\varnothing)$ to $V_C$ and $((p,\alpha),(y,\varnothing))$ to $E_C$
```

The RDA analysis determines the line *ℓ'* that the first argument to the API call was last assigned. The *Eval* function determines if the value can be statically evaluated. Not all values can, of course. So, in the case that it is not possible, this is marked using ∅. *In fact*, finding that a C program is calling a Python program whose name can *only* be determined by run time calculations is itselfa software engineering concern, and should be flagged for review.

If the value can be statically determined, then the name of the Python procedure is added to the call graph. The subtree for the called procedure will be

[3] https://docs.python.org/2/extending/embedding.html.

added to the multilingual call graph when the C and Python call graphs are merged.

The `PyObject_CallObject` API call (and variants) is processed as follows:

```
If $(\ell,b)$, $b\in B$, $target(b)$=PyObject_CallObject
 For $(x_i,\ell')\in RDA(p,\{a_0\},\ell)$, with $arg(b)=a_i$ $i=0..n$
  Calculate $y_i=Eval(x_i, \ell')$, and if all $y_i\neq\varnothing$,
  Add $(y_0,y_1,\ldots,y_n)$ to $V_C$ , $((p,\alpha),(y_0,y_1,\ldots,y_n))$ to $E_C$
```

This is an extension of the processing for the file API call to include an RDA analysis of all the arguments for the cross-language procedure call. In the strictest implementation, the call graph can be completed only if the procedure name and all the arguments' values can be statically determined. However, a more reasonable approach might be to insist only that the cross-procedure name be statically determined, since it is reasonable that the values of the arguments to the procedure might only be determined at run time.

5 IMPLEMENTATION

As an example, consider a C program with a call graph as shown in Figure 2. The C program uses the Python.h interface to call some Python scripts (using `PyRun_Simplefile`) for the user interactions. In a monolingual analysis, these are the leaves of the call graph (recall we have restricted this to a tree).

We have implemented the language boundary filters and call graph construction methods in section 4.2 with some restrictions. 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 stated in the function call. That is, the RDA analysis has not been added. The output of the call graph filter is a CSV file which has a series of rows, one per call, containing:

- Parent procedure name
- Called procedure name
- Argument strings for the call

The C filter also replaces the `PyRun_Simplefile` call with the name of the python file being called, treating the file name as a function – the first step in eliminating the opaque boundary.

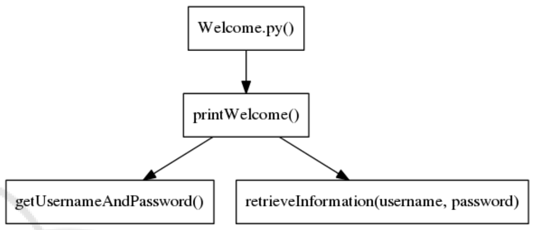


Figure 3: Example Python Call Graph.

The Python call graph in Figure 3 shows a root node called `Deposit.py`: The monolingual python filter creates this as a 'main' procedure for the python file, and it consists of any executable code not encapsulated in procedures. Using the cross-language file name as the cross-language procedure call name simplifies the final stage of processing, matching the leaves and roots in monolingual call graph CSV files and producing a combined CSV file showing both C and Python calls.The resulting CSV file is then processed to create a dot file that, through *Graphviz*, will generate a call graph diagram like the ones in Figures 2-4. The program represents function calls in C programs by an oval node and function calls in Python programs by a rectangular node, thereby making the multilingual aspect of the

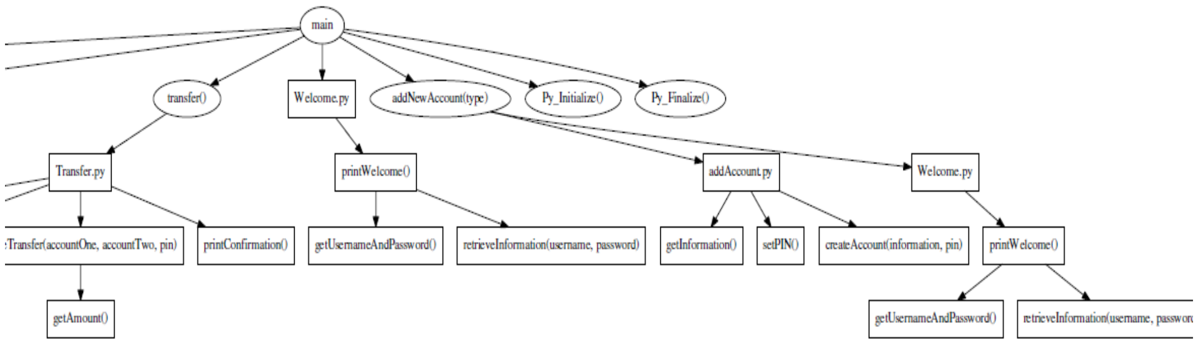


Figure 4: Multilingual Call Graph (Cropped for size).

call graph visually apparent.

The call graph, as seen in Figure 4 (cropped for size), depicts procedures in both the C and Python languages to visually represent their mutual call relationships. Note that the same Python procedure (`Welcome.py`) called from different points in the C program produces a different subtree. However, because the RDA module was not implemented in the filter that generated Figure 4, the argument values are not visible.

To test the programs in the pipeline, a small codebase of 35 Python and 30 C/C++ programs was built to ensure that the MSDA software could handle code of potentially unfamiliar style (to us). 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++ and Python combinations were successful.

6 CONCLUSIONS

This paper has introduced a lightweight approach to multilingual software analysis – MLSA. This work addresses 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 that relate to the language boundaries may get overlooked.

We propose an architecture comprised of monolingual filter programs that analyse single language AST and identify the cross-language boundary. The filters generate language independent information in 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 (to add additional language or analysis filters for example).

We focus on one specific problem, the generation of multilingual call graphs and develop a detailed approach for this. The C/Python interface is used as an example throughout. Finally, we present an example multilingual call graph analysis in overview, and describe the current status of the work based on a database of 75 C and Python programs downloaded from the Internet.

Two areas of work on this project concern the monolingual filters and the completion of the RDA analysis. The current monolingual filters directly parse AST text files and many of the trivial errors recorded in testing relate to this parsing. One solution is to move to a JSON AST format and leverage existing libraries to parse the files. While the completion of the RDA analysis allows argument values to be variables and expressions, determining the value of these expressions is a separate concern limited by the scope of static techniques.

While the argument of computational efficiency from design is argued here, current work includes collecting performance statistics to support this as well as to expand the small codebase used. The call graphs generated by MLSA are similar in representation to those generated by the Eclipse IDE, and future work will include a more detailed comparison of the MLSA call graph filters with other available tools.

ACKNOWLEDGEMENTS

The authors acknowledge the contributions of Bruno Vieira, Sunand Raghapathi and Nicholas Estelami in developing MLSA tools. The authors are partially supported by grant DL-47359-15016 from Bloomberg L.P.

REFERENCES

Ali, K., and Lhotak, O. (2012). Application-only Call Graph Construction. *Ecoop'12 Proceedings of the 26th European conference on Object-oriented Programming*. Beijing.

Christakis, M., and Bird, C. (2016). What developers want and need from program analysis: an empirical study. *31st IEEE/ACM Int. Conference on Automated Software Engineering*. Singapore.

Heering, J., and Klint, P. (1985). Towards monolingual programming environments. *ACM Trans. on Prog. Languages and Systems LNCS, 174*.

Hong, S., and al, e. (2015). Mutation-Based Fault Localization for Real-World Multilingual Programs. *30th IEEE/ACM Int. Conference on Automated Software Eng.*

Lakos, J. (July 20, 1996). *Large-Scale C++ Software Design*. Addison-Wesley; 1 edition .

Lee, S., Doby, J., and Ryu, S. (2016). HybriDroid: static analysis framework for Android hybrid applications. *31st IEEE/ACM Int. Conference on Automated Software Engineering*. Singapore.

Mushtak, Z., and Rasool, G. (2015). Multilingual source code analysis: State of the art and challenges. *Int. Conf. Open Source Sys. and Tech.*

Nielson, F., Nielson, H., and Hankin, C. (2005). *Principles of Program Analysis*. Springer.

Strien, D., Kratz, H., and Lowe, W. (2006). Cross-Language Program Analysis and Refactoring. *6th Int. Workshop on Src Code Analy. and Manipulation* .

van der Storm, T., and Vinju, J. (2015). Toward Multilingual Programming Environments. *Science of Comp. Prog.; Special issue, New Ideas and Emerging Results in Understanding Software, 97*, 143-149.