MLSA Reference Documentation

Version 1.0.2 July 2017

1.0 Background

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. 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 come 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 - 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.

Heterogeneous or multilingual codebases 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.
- Debugging complexity as languages interact with each other in unexpected ways.
- Security issues relating to what information is exposed when one language procedure is called from another.

The objective of the MLSA (*MultiLingual Software Analysis*) Research Group is to develop software engineering tools that address large multilingual codebases in a lightweight, open 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. We have chosen to focus on the issues of generating multilingual call graphs using C/C++, Python and Javascript interoperability examples. The MLSA architecture is a lightweight architecture concept for static analysis of multilingual software.

2.0 Overview of the MLSA Architecture

Lightweight programs (which we call *filters*) operate on program source files or/and on data files and produce data files. The filters can be stacked in pipelines, where each filter in the pipeline reads data files generated by prior filters and in turn generates new data files. The design motivation behind this structure is to allow pipelines of filter programs to be constructed to implement program analysis. This modular design is important to isolate the language-specific first pipeline stages from later language-independent modules and in this way support sophisticated analysis fo multilingual codebases.

2.1 Example

An example is shown in Figure 1 below, where the rectangles are MLSA filter programs and the source code files and data files are shown as ovals.

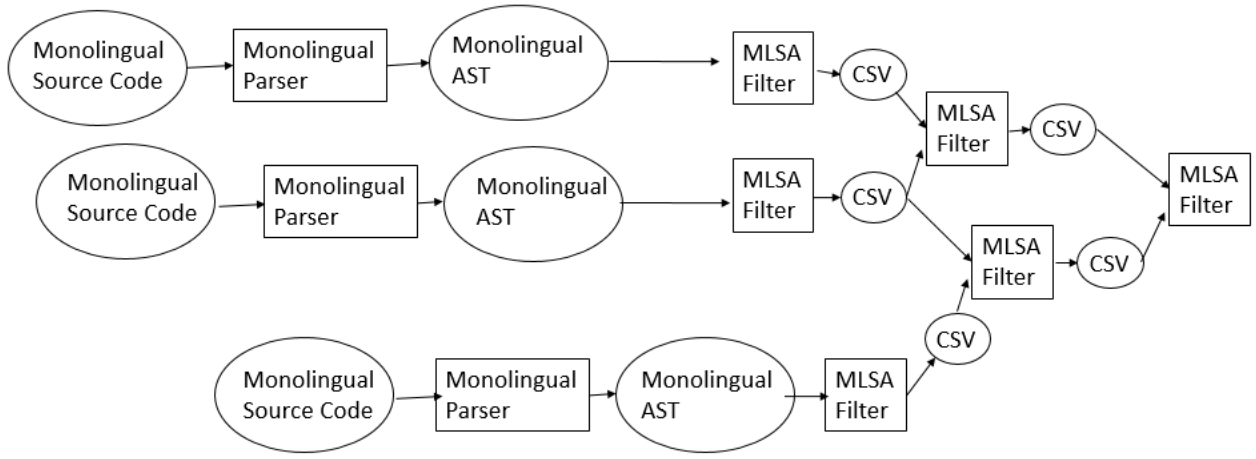

Figure 1: MLSA Example Pipeline

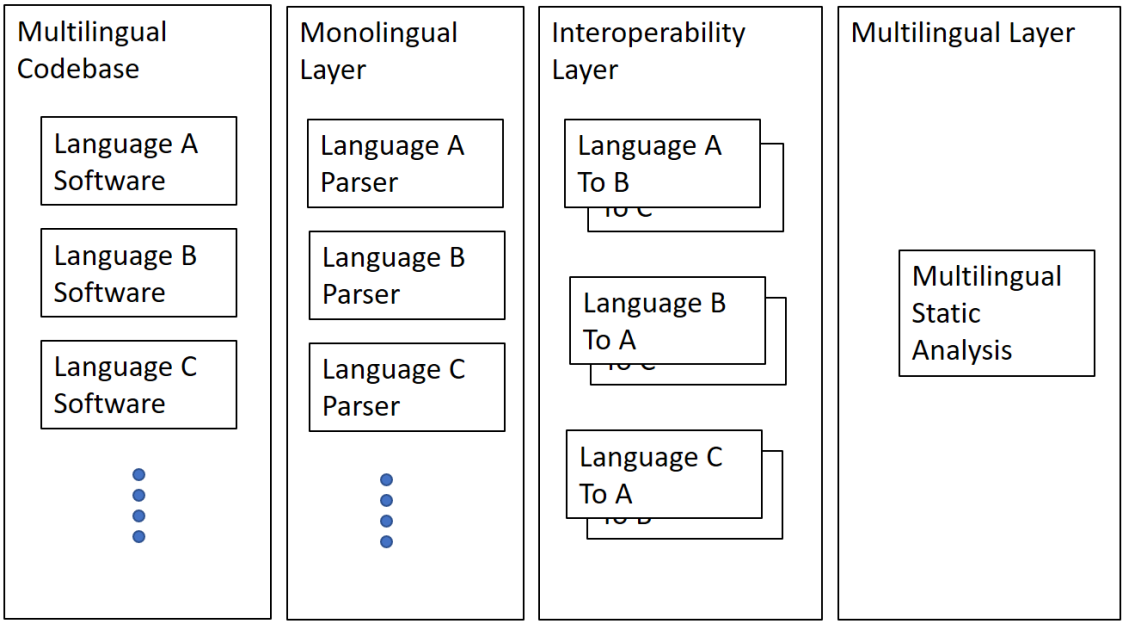

Figure 2: MLSA Architecture (data flows right to left)

Figure 2 shows the general software architecture of a static analysis in MLSA: Source files in multiple languages are first processed to extract monolingual information to data files. Next a set of MLSA interoperability filters (programs) run on these data files to produce single, multilingual data files for the source code base being processed. Static analysis filters then operate on these multilingual data files.

3.0 Installing and testing MLSA

When the MLSA project is cloned it will produce a folder with the following subfolders.

- Bin – contains python code implementing MLSA filter programs and MLSA pipelines
- Doc – contains this document and other MLSA documentation

- Test – contains the program testcode.py and test folders test0 through test5 that can be used to determine a correct installation
- Examples – contains C/C++, Python and JavaScript subfolders with various programs downloaded from the web to evaluate MLSA.

To run MLSA you will need to have installed the following prerequisites

1. Clang 3.8
2. Python 2.7
3. SpiderMonkey 2.4
4. DOT (Graphviz) 2.38
5. ps2pdf
6. Evince or other PDF viewer
7. Bash shell

All MLSA testing was done using Ubuntu. Clang, Python, SpiderMonkey (js24) and Graphviz can be installed on Ubuntu using “sudo apt-get install”.

Once you have installed these, you can test whether your MLSA installation works by cd-ing to the test folder and invoking the testcode.py program. As a first step, cd to the top level mlsa directory. In that directory, do “source mlsapath.bash” to add yourinstallation/mlsa/bin to your executable path.

Next, cd to the test folder. All calls to testcode.py will automatically diff the results generated with the correct results and report the differences in a text file called testN_stats.txt where N is the argument given to the testcode.py program:

1. ./testcode.py 0 – will test the C function call generator.
2. ./testcode.py 1 – will test the C control flow, assignment collector and RDA pipeline.
3. ./testcode.py 2 – will test the Python function call generator.
4. ./testcode.py 3 – will test the multilingual Python and C functional call pipeline.
5. ./testcode.py 4 – will test the Javascript function generator.
6. ./testcode.py 5 – will test the multilingual Python, C and Javascript pipeline

If you get no differences reported for all these, then your MLSA installation is operating correctly and you can go ahead and use MLSA!

A good place to start is with the **mlcg.py** program in bin. This program will invoke the multilingual call graph pipeline on the programs and/or folders you give it as argument. For example, in the MLSA mlsa/test folder if you type:

```
mlcg.py test0
```

Then the multlingual pipeline is called for all the (C/C++, Python or Javascript) programs in test0 and a single call graph generated. Procedure calling between files in the same and different languages will be identified (for the limited set of

interoperability calls that have been implemented) and the call graph will reflect this, but programs with no procedure calls in common are fine too. The resulting call graph is a forest of trees. Recursion is flagged after one full cycle and several other kind of interlanguage calls are flagged also.

To get deeper into MLSA you will need to know more about the individual filter programs and pipelines that have been implemented. You can also build new pipelines or add new filters.

4.0 MLSA Filters and Pipelines

This section describes the set of filter programs that have been constructed to date.

AST file generation

- Clang-check is used to generate AST files for C and CPP programs
- The Python AST library is used to generate AST for Python programs
- SpiderMonkey is used to generate the AST for JavaScript programs

Monolingual procedure call filters

- cFuncCall.py reads a NAME.c_ast.txt (or cpp) file and generates a NAME.c_call.csv file containing the function call information in the file. It also adds to a file containing all the functions in the system, SystemName_funcs.csv, all of the functions defined in the program NAME.c.
- pyFunCall.py reads a NAME.py_ast.json file and generates a NAME.py_call.csv file containing the function call information in the file. It also adds to a file containing all the functions in the system, SystemName_funcs.csv, all of the functions defined in the program NAME.py.
- jsFunCall.py reads a NAME.js_ast.json file and generates a NAME.js_call.csv file containing the function call information in the file. It also adds to a file containing all the functions in the system, SystemName_funcs.csv, all of the functions defined in the program NAME.js.

Interoperability filters

- pyViaC.py reads a C function call file NAME.c_call.csv and scans for Python interoperability. Currently it only implements the Python.h PyRunSimpleFile API. It outputs a revised NAME.c_finalcall.csv.
- jsViaPy.py reads a Python function call file NAME.py_funcall.csv and scans for Javascript interoperability. Currently it only implements the PyV8 eval API. It outputs a revised NAME.py_finalcall.csv.
- pyViaJs.py reads a Javascript function call file NAME.js_call.csv and scans for Python interoperability. Currently it only implements the JQuerry ajax API. It outputs a revised NAME.js_finalcall.csv.

Multilingual combination and graphing filters

- mergeFunCall.py merges the function calls in the XX_finalcall.csv files into a single functional call file. When called from mlcg.py, this output file is given the name of the first argument to mlcg.py, e.g. if the argument was test0, then the file is called test0_callgraph.csv
- generateDOT.py produces a PDF file from a call graph CSV file displaying the call graph.

Flow Control filters

- cFlowControl.py reads an AST file NAME.C_ast.txt (or CPP) and generates a CSV file containing the forward flow control information NAME.C_fcfg.csv, and reverse control flow information NAME.C_rcfg.csv
- pyFlowControl – does not exist yet
- jsFlowControl – does not existyet

Assignment collectors

- cAssignmentCollector.py reads the C AST file and locates all variable assignments and their line numbers. This provides an input that can be used in various kinds of assignment analysis. It is currently only used in the RDA analysis.
  It currently also implements two simple static evaluation functions:
  - It can report the assignment of a literal to a variable, and
  - it can detect the use of strcpy in a C program to set a character array to a literal.
  - Anything else it marks either as an expression or a variable.

  It generates the file NAME.c_vars.csv with the information about assignments for each variable.

- pyAssignmentCollector – does not currently exist
- jsAssignmentCollecto – does not currently exist

Reaching Definitions

- RDA (Killgen/exit-entry) implements a reaching definitions analysis for each variable in the program. It reads the _vars.csv file (where the file name and language appears before the underscore) for the program to identify all variable assignments, and it reads the _rfcg.csv file to get the reverse control flow for the program. It generates the file _rda.csv with the solutions for the line entry sets. (It doesn't record the exit set solutions but it does derive them).

- cRDAGroup.py sets up the RDA pipeline of cAssignmentCollector, cFlowControl and then RDA.

Pipelines

- cSA.py sets up the call graph and RDA pipeline for C/CPP sources.
- pySA.py sets up the call graph pipeline for Python sources
- jsSA.py sets up the call graph pipeline for Javascript sources.
- Mlcg.py processes its argument list of files and folders, calling cSA, pySA and jsSA as necessary to produce a combined multilingual call graph.

5.0 Data files

There are three kinds of data files in MLSA: Data files that contain a monolingual Abstract Syntax Tree (AST) in text format, data files that contain a monolingual AST in JSON format, or data files in comma separated values format (CSV) that contain the results of various kinds of static analysis.

If a source code file is called NAME.X where NAME is the root file name and X is the language suffix (e.g., test.cpp or analyze.py, etc.) , then data files are named using this root file name as follows:

- AST files: NAME.X_ast.txt or NAME.X_ast.json
- Monolingual procedure call graph files: NAME.X_call.csv
- Monolingual procedure call graph files with API integration: NAME.X_finalcall.csv
- Combined multilingual call graph file: NAME_callgraph.csv
- Forward flow control file: NAME.X_fcfg.csv
- Reverse flow control file: NAME.X_rcfg.csv
- Monolingual variable assignments: NAME.X_vars.csv
- Monolingual Reaching definitions analysis: NAME_X_rda.csv

6.0 Detailed Status per Module

JavaScript: jsFuncCall.py

- Walks through JSON object to find:
    - Scope (function or main body)
    - Call name
    - Arguments
        - Literal and variable
        - Array, dictionary, subscripts
        - Member variables
        - Function calls
        - Binary/Unary Operations

        - Anonymous Functions (similar to lambda)
    - Line number (just for variables)
    - DOES NOT get class
    - DOES NOT get the function calls inside an Anonymous Functions
- Writes all the functions defined in the program to the system's function csv file (information including name, class, and number of parameters)

Python: pyFuncCall.py

- Walks through python AST object to find:
    - Scope (function or main body)
    - Class
    - Call name
    - Arguments
        - Literal and variable
        - list, tuple, dictionary, subscripts
        - Member variables
        - Lambda
        - Function calls
    - Also gets kwargs and starargs for calls, and defaults and varargs for lambdas
    - Line number (just for variables)
    - DOES NOT get the function calls inside lambdas
- Writes all the functions defined in the program to the system's function csv file (information including name, class, and number of parameters)

**member function calls are written as OBJ.functionName

**function calls inside function calls are given the same scope as the outer function But, they are also shown as arguments in that call, and have an id attached that corresponds with the line where the call is found in the csv file

**all attribute functions (e.g., function1.function2.function3()) are given the same scope

**all member functions are given the prefix OBJ (e.g., OBJ.func())

pyViaC.py

- Searches for “PyRun_SimpleFile”
    - If found, looks for string with “.py”
        - if not found, searches through the RDA CSV file for that particular program
            - looks for corresponding line number
            - returns all strings with “.py”

    - for each '.py' string found, adds new line to csv with same scope, class, and id, but with call name of python file
    - if '.py' is not found, adds new line to csv with same scope, class, and id, but with call name "ERROR_Python_File_Cannot_Be_Discerned"
- If no "PyRun_SimpleFile" in line, will just append line to new csv file

pyViaJs.py

- Searches for "OBJ.ajax"
    - If found, looks for keyword url
        - Looks for string after url with '.py'
    - If string is found, adds new line to csv with same scope, class, and id, but with call name of python file
    - If string not found, adds new line to csv with same scope, class, and id, but with call name "ERROR_Python_File_Cannot_Be_Discerned"
- If no "PyRun_SimpleFile" in line, will just append line to new csv file

jsViaPy.py

- Searches for "OBJ.eval"
    - If found, looks ahead for OBJ.JSContext
        - If found, looks for argument read() and gets id number
            - Finds open() in csv file (line before read) and gets ".py" string
    - If string is found, adds new line to csv with same scope, class, and id, but with call name of python file
    - If string not found, adds new line to csv with same scope, class, and id, but with call name "ERROR_Python_File_Cannot_Be_Discerned"
- If no "PyRun_SimpleFile" in line, will just append line to new csv file
- NOTE: This assumes that all functions are on the same line
    - PyV8.JSContext().eval(open("file.py").read()))

generateDOT.py

- Reads in multilingual csv file
- Creates an object of class Program for each program's csv
- Programs consist of a name, a program type, a boolean hasMain that determines whether the Program can stand on its own or must be called by another program, and a copy of the program's csv to parse through
- Programs parse through their csv and create Function objects in the global Function list for all the functions defined in the Program
- A Function has a name, the name of the program it is found in, its class name (if it is a member function), and a list of all the function calls found in the function

- Once all Programs have been processed and Functions have been created, the program looks through the Functions list to determine all the Programs that can stand alone and are not called by another Program
    - These Programs are added to the global Call list as a Call object.
- A Call object consists of a function name, an ID, a function name and ID of the function where the function call was located, the program name and class name of the function, the program type, a list of the call history of the call (a function calling another function that calls another function...), a list of the program history (a program calling another program which then calls another...), a boolean circular to determine whether the function call is in a circular system (determined through the program history), a boolean recursive to determine whether the function call is recursive (determined through the call history), a boolean used which functions as a stopping point for iterative purposes, and a boolean independent to determine if the call can stand on its own
- The program iterates through the Call list, comparing the Call objects with their equivalent Function objects, and adding more Call objects based on the list of calls in the Function object
    - if the Call object is recursive or circular, no more Call objects will be added for that particular Call object
- Once all Call objects have been added (all the Call objects' used boolean is true), the program iterates through the Call list and creates nodes (edges) and flows (vectors) in the call graph
    - Each Call object is given a node with the label as the call ID, program name, function name, and the arguments in the call
        - Node0x884993735 [shape=oval, label="1. (ex1.cpp) func(arg)"];
    - The node shapes for each language are as follows:
        - oval for C/C++
        - rectangle for Python
        - hexagon for JavaScript
    - If a Call object is recursive, the node will be dashed
    - If a Call is circular or there is an error to be flagged (for example, if the name of a program could not be determined in an API), the node will have two dashed lines
    - If the Call object is not independent, it is also given a flow, which consists of the node ID for the call and for the parent function
        - Node0x224912150 -> Node0x884993735;
- Extra:
    - Python programs have a rectangular node, C/C++ programs have an oval node, and JavaScript programs have a hexagonal node
    - A recursive call stops after the second loop, and the node is indicated by a dashed outline
    - A circular call stops after the first call in the second loop, and the node is indicated by a double-dashed outline
    - An unidentifiable program call is indicated by a double-dashed outlined node of the shape of the type of program

C/C++: CControlFlow.py

- Reads in AST .txt file
- Parses the AST to:
  - Extract entries that have information on line numbers
  - Convert those entries into a tree for processing
- Assigns to each line number in the program:
  - A set of line numbers which could be entered from the given line number
  - This represents the forward control flow
- Uses the generated forward control flow to derive the reverse control flow:
  - Each line number of the program contains the set of lines that could enter the given line number
- Outputs
  - The forward control flow to NAME.X_fcfg.csv
  - The reverse control flow to NAME.X_rcfg.csv
- DOES NOT SUPPORT the following cases (returns inaccurate results):
  - For loops
  - Do-While loops
  - Return and break statements
  - Switch statements
  - Multiple statements on one line:
    - Because we treat line numbers as labels for analysis, multiple statements should appear on different lines
    - For example: if (x > 10) x = 100; is not supported.
    - The working convention would be:
      If (x > 10)
      X = 100;

C/C++: CAssignmentCollector:

- Reads in AST .txt file
- Parses the AST to extract information on variable assignments, including:
  - Line number of assignment
  - Variable name
  - Variable scope
  - Value assigned (R-Values)
    - If a literal, the literal will be added
    - If a variable, ‘?v’ will be added
    - If an expression, ‘?e’ will be added
    - If unknown, ‘?’ will be added
- To account for character array assignments in C, the program looks for the function call ‘strcpy’
- In C++, string assignments are supported
- Output:
  - The variable assignments to NAME.X_vars.csv
- DOES NOT SUPPORT:
  - R-Values returned by a function
  - Array initializations (other than type char)

    - Elements of arrays (assignments to or from)
    - Increments within for loops
    - Objects and pointers (assignments to or from)
    - Data members of objects
    - Multiple assignments on one line
- Note: the above cases may still appear in the NAME.X_vars.csv file, but the information will be inaccurate and therefore analysis of these cases should not be trusted.

RDA:

- getControlFlow.py:
    - Reads in NAME.X_rcfg.csv
    - Extracts information from the control flow CSV file into a Python list and returns that list
- RDAKillGen.py:
    - Reads in NAME.X_vars.csv
    - Extracts information from the variable assignment CSV into a Python List
    - For each line that an assignment takes place, a kill/gen list for that line is created
    - The definitions for kill/gen in this case are as follows:
        - Kill is given by the name of the variable (being assigned) along with every value that the variable is assigned in the scope that the variable belongs to
        - Gen is given by the name of the variable (being assigned) along with the value that it is assigned at the given line
- RDAEntryExitList.py:
    - Reads in control flow and kill/gen Python lists
    - Solves Entry and Exit sets for each line in the program with the following definitions:
        - Entry set is given by:
            - If the line of interest is not arrived at by any other line, the names of the variables that have the scope of the line number along with a question mark indicating their unknown values
            - Else, it is composed of the exit sets for all lines that could have entered the line
        - Exit set is given by:
            - If the line of interest has an assignment, then the list is composed of the entry list for that line, the kill list for the line, and the gen list for the line (in that order)
            - This order tells us that we union the entry list, difference the kill list, then union the gen list
    - Output:
        - Solved Entry sets in NAME.X_rda.csv

- cRDAGroup.py:
  - Top-Level RDA program
  - Contains conventions for naming control flow and variable assignment CSVs
  - Calls CControlFlow.py to generate the control flow graph CSV
  - Calls CAssignmentCollector to generate the variable assignment CSV
  - Calls cRDA to read and interpret the CSVs
- cRDA.py:
  - Calls getControlFlow.py to generate the Python list representing the control flow graph
  - Calls RDAKillGen.py to generate the kill/gen Python list
  - Calls RDAEntryExitList.py to solve the entry/exit sets

MLSA Research Group Description

The MLSA (*MultiLingual Software Analysis*) Research group aims to develop software engineering tools that address large multilingual codebases in a lightweight, open and extensible fashion. Their publicly available open source MLSA software architecture is their initial step to this goal. The MLSA research Group is based in the Computer and Information Science Department of Fordham University in New York City, USA.

MLSA Logo