Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 23–29
doi: 10.14794/ICAI.10.2017.23

Static Analysis Toolset with Clang

Bence Babati, Gábor Horváth, Viktor Májer, Norbert Pataki

Dept. of Programming Languages and Compilers,
Fac. of Informatics, Eötvös Loránd University, Budapest
babati@caesar.elte.hu,xazax.hun@gmail.com,viktor.majer1@gmail.com,
patakino@elte.hu

Abstract

Static analysis is an emerging method to analyse programs without execute them only based on their source code. Static analysis can be used for various tasks, e.g. searching bugs. Clang is an open source compiler for C/C++/Objective-C. It is built on the top of the LLVM compiler infrastructure. It is a rapidly evolving project which is supported by Apple, Google, and Microsoft. Nowadays Clang is a popular tool to develop new static analyzers. One of the advantages of Clang is its modular architecture. One can use the parser as a library to build tools. The Static Analyzer is the part of the Clang compiler. It is utilizing the same AST that is generated by the compiler.

In this paper we present our tools that are based on Clang. The tools cover wide spectrum of static analysis: detecting portability issues in include dependencies of C++ Standard Template Library, refactoring for parallelized constructors and code comprehension. The C++ Standard does not define which standard header files include an other standard header file. It is easy to take advantage of an implemention-specific dependency because the code compiles. It can result in portability issue when an other STL implementation works in different way. One of our tool detects these problems. A tool of ours is able to make copying operations concurrent in specific cases. The aim of our code comprehension tool is finding specific C++ code snippets based on queries.

Keywords: C++, static analysis, Clang

MSC: 68N15 Programming languages

1. Introduction

Static analysis is a kind of software analysis. Its idea is analysing only the source code, so program execution is not required for the analysis. The main advantage of

this method is, that the program is not be executed, and a lot of problems can be detected by this technique in the early phase of development. The purpose of static analysis to find bugs in the source code. The tools are not perfect, they could not find all bugs and sometimes go wrong. However, these tools are beneficial ones [5].

In general, finding bugs in an earlier stage of software development highly could reduce the cost of the software [10]. So, using this kind of tools during the development can be effective in many perspective, for example these tools can provide a kind of automatic checking of the source code.

Of course, the range of the kind of detected problems can be different, it depends on the used static analysis technique. There are many analysis methods to detect issues from simple ones, like regular expression based searches to very complex algorithms. Nowadays, the market leading static analysis tools use very advanced algorithms to detect issues, for example symbolic execution[6]. The complexity of the analysed language can influence the algorithms, as well.

One can use static analysis for many reasons: finding bugs [3], checking coding conventions [12], detecting compatibility issues [2], code comprehension, counting code metrics, refactoring or obfuscating code, etc.

Clang is an open source compiler for C/C++/Objective-C. It is built on the top of the LLVM compiler infrastructure. It is a rapidly evolving project which is supported by Apple and Google. Clang is getting more and more popular because of its modular architecture. One can use the parser as a library to build tools [4].

Clang has many related tools that can be used for improved code quality. For instance:

- Clang Static Analyzer: finds bugs in C, C++, Objective-C programs with the help of symbolic execution
- Clang LeakSanitizer: a runtime memory leak detector
- Clang-Tidy: C++ linter that finds typical programming errors, like style violations, interface misuse and bugs.
- Clang-Rename: is a C++ refactoring tool. Its purpose is to perform efficient renaming actions in large-scale projects such as renaming classes, functions, variables, arguments, namespaces etc.

2. Our tools

2.1. Code Comprehension tool

We have developed a code comprehension tool that can find specific locations in large code bases. The users can query the source code with a user-friendly graphical user interface. Querying a precise AST is a superior approach to text search due to the following reasons:

- The overload resolution is already done

- Macros already expanded
- Templates already instantiated
- Type aliases are resolved
- Implicit language constructs (e.g., implicit casts) can be matched
- Context sensitive queries can be formulated (more expressive than regex)

The query language is based on the ASTMatcher library [14]. This query language is a declarative one. Let us consider the following examples:

- where is a function call that is used in variable definition:

```
varDecl(isDefinition(),
        hasInitializer(callExpr().bind("Call")))
```

- Where are the identifiers of types that are subtypes of `Foo`:

```
recordDecl( isDerivedFrom(
  hasName( "Foo" ))).bind( "id" )
```

This language is not straightforward for the first time. In order to not require the user to have deep knowledge about the Clang syntax tree and the query language the tool provides the user with semantic autocomplete and suggestions. There is also a catalog for frequently used queries. This can also be a good starting point and documentation for new users. The GUI will highlight the matches of the query for the user - see 1.

One major limitation of the ASTMatcher library is that it does not support cross translation unit queries.

The tool can work with any project that can be compiled using Clang and can provide a JSON compilation database.

2.2. Refactoring tool

We have developed a tool that refactors the code for optimizing concurrent copy operations.

The heart of our approach is to copying the data members of an object from a given class in different execution threads. To achieve this, we arrange the data members into tuples which tuples are disjunct groups of the fields to be copied. The key question is how to form these groups to provide more efficient copy-constructor than its single-threaded version.

It is hard to programmatically determine the costs of copying an object. For instance, the compiler does know how many elements are stored in a vector when it is copied. Thus we provide "marker interfaces" to extend: `ExpensiveObject` to indicate that the copy of a type is presumably expensive, and `C4Type` to claim a concurrent copy constructor for it.

Our approach is

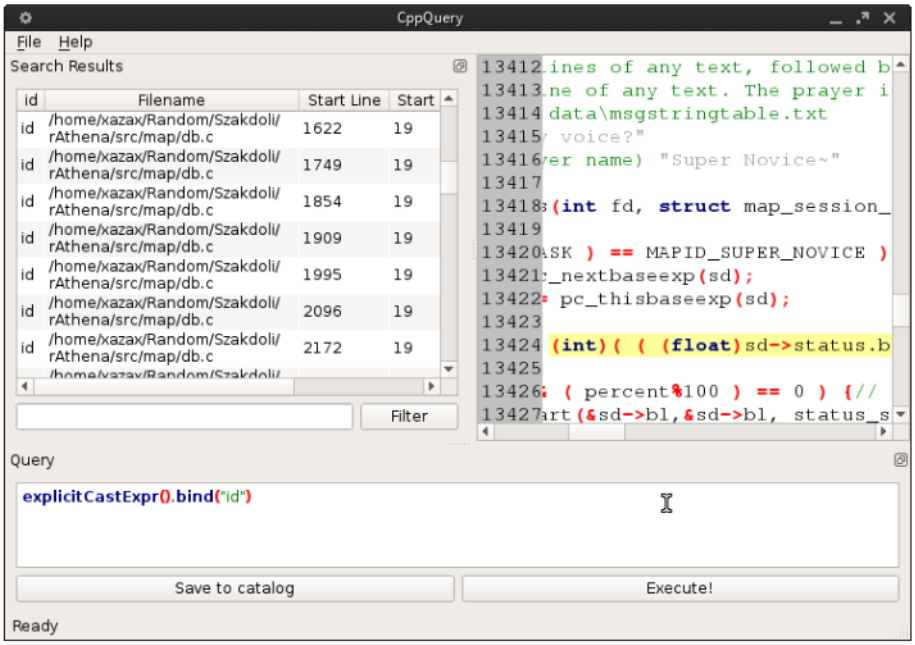

Figure 1: The GUI of the code comprehension tool

1. look up every class in the compilation units those do not have user-defined copy constructor and marked as `C4Type`

2. generate copy constructor for each of these classes by creating separate copy-groups according to the mentioned grouping rules, let these groups execute on separate threads and finally join these threads

If more than 1 group is created, injector generates simultaneous copy statements for the groups by utilizing `std::thread` and *lambda* expressions. The following code snippet is responsible for generating the concurrent copy constructor:

```
// copy ctr's body
ccopy::CopyGroups cg;
cg.divideBy(recordDecl->field_begin(), recordDecl->field_end());

for(auto cit = copygroups.begin(); cit != copygroups.end(); ++cit) {
    std::string groupName = cit->first;
    ccopy::CopyGroups::group_type group = cit->second;
    insertCopyStmt(rewriter, groupName, group.getFields());
}
```

A parallel copy statement looks as follows:

```
std::thread default_group([&](){
    i = other.i;
    y = other.y;
});
```

We have measured the runtime speed-up and the results are promising [8].

2.3. Portability issue detection

C++ STL is handy, well-designed standard library that is based on the generic programming paradigm. The usage of STL highly decreases the number of classical C/C++ mistakes (e.g. memory leak). On the other hand, STL introduces many new problems. For instance, one can write code that compiles with an implementation but fails with another one.

The root cause of this portability problem is that the C++ Standard does not define which standard headers include other standard headers. It depends on the developers of current STL library implementation because they create the hierarchy of the library. The hierarchy of STL implementations could be different. It is not a problem, when we use correctly the standard headers, but the compilers do not check it comprehensively. Therefore it can be the root cause of portability problems. These unportable codes should result in compilation error with every STL implementation, but the compiler does not know that is wrong. The compiler finds the used definitions and does not check how they reached. For instance:

```
#include <algorithm>

int main() {
  std::vector<int> v;
}
```

This code compiles and works perfectly when g++ 5.3 with libstdc++ STL implementation is in-use. However, if we upgrade our compiler to g++ 6.1, the STL implementation is also updated. The new compiler fails to compile the code because it does not find vector in std namespace.

```
error: 'vector' is not a member of 'std'
```

How this situation could be happen? The hierarchy of libstdc++ STL implementation has been changed. Till this version $< algorithm >$ included $< random >$, that included $< vector >$, so if one includes *algorithm*, the *vector* header also can be used.

During the preprocessing a dependency graph is building up that can be mapped to the original include hierarchy. The nodes of the graph are the files and the directed edges mean include relation between the two endpoints, the source node includes the destination node. Also the nodes contains additional information, that may be needed later, for example user-defined header or not, part of Standard

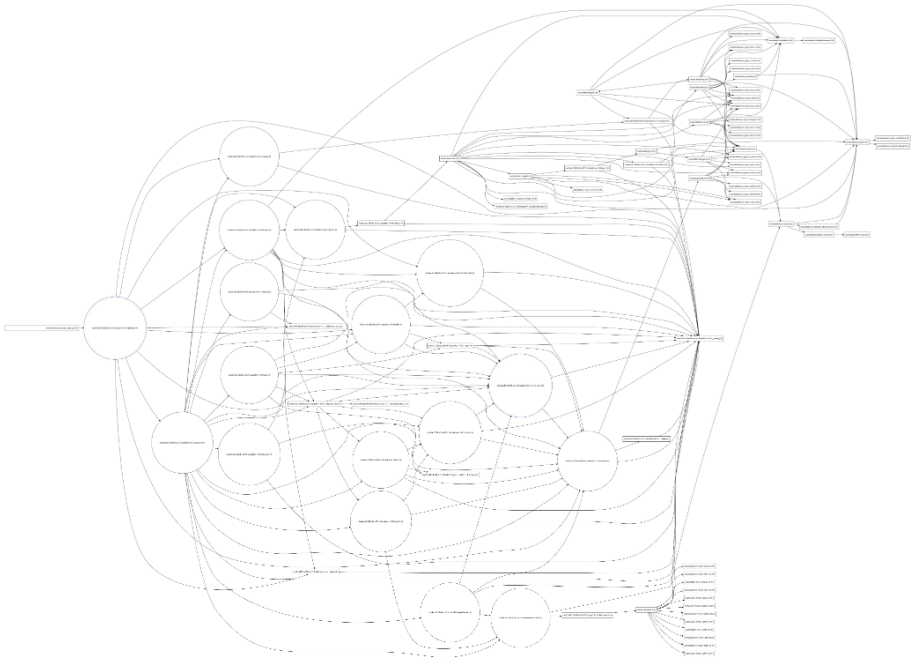

Figure 2: Dependency graph

Template Library or not, etc. In this graph, the relations between headers are clear, we know for each file which other files includes and which other files are included it. A dependency graph can be seen on the figure 2.

Another important data is collected in this stage, macro usage only can be gathered right here. With the macro usage information the proper nodes are extended in the graph. At the end of the preprocessing, this graph can be considered as definitive, no one will extend the graph with any new node or directed edge, but extra information can be added to nodes in the next stage.

The last stage is the graph analysis. The dependency graph is complete right now, the different analysers can start their work to walk over the graph. The graph analyser does not depend on each other, just visiting the given graph. Every previously described issues are implemented as a graph analyser because they just seek after erroneous parts in the graph independently from each other.

3. Conclusion

Static analysis becomes more and more popular because we can take advantage of this approach in many ways: we can find subtle bugs inexpensively, we can transform the source code for to be maintainable or to be faster. Software complexity metrics can be evaluated by this method or aiming programmers with code

comprehension. Clang is an evolving infrastructure for creating new tools for C or C++ source code.

In this paper we present our standalone tools that have been developed with Clang. We have created a code comprehension tool that can query the source code with a declarative query language. The tool supports the compilation of queries. We have also developed a refactoring tool that transforms the sequential copy operations to parallelized ones. Finally, we have created a tool that searches for a subtle portability problem that the compilers cannot find. This portability issue is related to the standard library.

References

[1] Babati, B., Pataki, N.: *Analysis of Include Dependencies in C++ Source Code*, Annals of Computer Science and Information Systems **13**, pp. 149–156 (2017).

[2] Brunner, T., Pataki, N., Porkoláb, Z.: *Backward Compatibility Violations and their Detection in C++*, Acta Electrotechnica Et Informatica **16(2)**, pp. 12-19 (2016).

[3] Horváth, G., Pataki, N.: *Clang matchers for verified usage of the C++ Standard Template Library*, Annales Mathematicae et Informaticae **44**, pp. 99–109 (2015).

[4] Horváth, G., Pataki, N.: *Source Language Representation of Function Summaries in Static Analysis*, In Proc. of the 1th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS 2016), Paper Nr. 6.

[5] Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: *Why don't software developers use static analysis tools to find bugs?* in Proc. of the 2013 International Conference on Software Engineering, ICSE '13. (2013), pp. 672–681.

[6] King, C.: *Symbolic execution and program testing*, Communications of the ACM, Vol. **19**, pp. 385–394 (1976).

[7] Lopes, B. C., Auler, R.: "Getting Started with LLVM Core Libraries", Packt Publishing (2014).

[8] Májer, V., Pataki, N.: *Concurrent object construction in modern object-oriented programming languages*, in Proc. of the 9th International Conference on Applied Informatics (ICAI), Vol. 2, pp. 293–300.

[9] Meyers, S.: "Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library", Addison-Wesley (2001).

[10] Nagappan, N., Ball, T.: *Static analysis tools as early indicators of pre-release defect density*, in Proc. of the 27th International Conference on Software Engineering, ICSE '05. (2005), pp. 580–586.

[11] Pandey, M., Sarda S.: "LLVM Cookbook", Packt Publishing (2015).

[12] Pataki, N., Cséri, T., Szűgyi, Z.: *Task-specific style verification*, AIP Conf. Proc. Vol. **1479**, ICNAAM 2011: International Conference on Numerical Analysis and Applied Mathematics, pp. 490–493.

[13] Stroustrup, B.: "The C++ Programming Language", Addison-Wesley (1999).

[14] Clang AST matcher library, `http://clang.llvm.org/docs/LibASTMatchers.html`