Analysis of Include Dependencies in C++ Source Code

Bence Babati, Norbert Pataki
Department of Programming Languages and Compilers,
Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
Email: {babati, patakino}@caesar.elte.hu

***Abstract*—The C++ Standard Template Library (STL) is the flagship example for libraries based on the generic programming paradigm. The usage of this library is intended to minimize classical C/C++ errors, but does not warrant bug-free programs. Furthermore, many new kinds of errors may arise from the inaccurate use of the generic programming paradigm, like dereferencing invalid iterators or misunderstanding remove-like algorithms.**

Unfortunately, the C++ Standard does not define which standard header includes another standard headers. It is easy to write code that works perfectly on an implementation but fails to compile with another implementation of STL. These unportable codes should be result in compilation error with every STL implementation. However, in this case the compiler does not warn us that this code is erroneous.

In this paper we present our tool that is based on the Clang. This tool is able to detect the missing include directives that are patched by the STL implementation's internal structure. It also reports the unnecessary include directives to avoid extra compilation time. The background of our tool is discovered and we briefly present the underlying data structures and algorithms. We analyse how these problems occur in open source libraries and programs. Which environment proves oneself to be lazy or strict? How the developers take advantage of this portability issue?

I. Introduction

NOWADAYS, the C++ language is very popular in educational and industrial environments as well. It provides a wide range of programming language elements, from the low level bit manipulation and pointer usage, which mainly come from C language to the high level, modern programming paradigms, such as function overloading, exceptions and templates. Runtime efficiency is important in C++, with the guiding principle being "do not pay for what you do not use". Efficiency can be very important in industrial environment.

On the other hand, the C++ language is continuously evolving, new standards come with a bunch of new features, for instance the variadic templates which appeared in C++11. These new features keep C++ popular. Actually, there are millions of lines C++ code and new lines are created day by day, that someone has to maintain later. Thus, the detection of bugs has become very important. The earlier the bug is found, the lesser the cost. There are analysis tools on the market that can catch bugs in the source code. Some of them find problematic code snippets during the compilation stage, others can do it at runtime.

Supported by the NKP-17-4 New National Excellence Program of the Ministry of Human Capacities

In this paper we describe an issue related to C++ Standard Template Library (STL). This issue may cause portability problems because of the underlying STL implementation's undefined include dependency. We present our tool that can analyse the source code and find these kind of bugs. The tool is based on Clang compiler infrastructure [1].

This paper is structured as follows. The paper begins with an introduction to STL and static analysis in section II. We present the related problems in details in section III. After we describe our tool that is based on Clang in section IV. Our tool has been evaluated on open source projects, so the collected result can be seen in section V. Finally, this paper concludes.

II. Prequisites

A. Standard Template Library

The *Standard Template Library (STL)*, is a standard C++ library of container classes, algorithms, iterators, and functors [2]. STL provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parametrized: almost every component in the STL is a template that cannot be compiled in advance. Therefore all definitions of the STL components should be written in the header files [3].

The STL includes *container* classes: classes whose purpose is to contain other objects. The library includes the classes `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`. Each of these structures is a template, and can be instantiated to contain many types of object.

The STL also includes a large collection of *algorithms* that manipulate the data stored in containers (for instance, `for_each`, `copy`, `find_if`, etc.). Algorithms are global function templates, not member functions. Every algorithms operate on a range of elements, rather than on a container. Algorithms are decoupled from the STL container classes. This means that algorithms can be used with vectors, but also work with lists, and even with elements of C arrays, etc.

Iterators make connection between the algorithms and containers, which are generalization of pointers. Their public interface originates from pointer-arithmetic. Iterator is the mechanism that makes it possible to decouple algorithms from containers: algorithms are templates, and are parametrized by the type of iterator, therefore they are not restricted to a single type of container.

Finally, the STL includes a large collection of *function objects*, also known as *functors*. Just as iterators are generalization of pointers, function objects are generalization of functions: a function object is anything that you can call using the ordinary function call syntax supported by *operator()*. Function objects are an important part of generic programming because they allow abstraction not only over the types of objects, but also over the operations that are being performed.

The complexity of the library is greatly reduced because of this layout. As a result of this layout we can extend the library with new containers and algorithms *simultaneously*. This is a very important feature because object-oriented libraries do not support this kind of extension [4].

The STL that is specified in the C++ standard does not belong to a specific implementation. Many STL implementations are available and these are not the same. Extensions and different approaches appear in the STL implementations.

The usage of C++ STL highly reduces the classical programming errors, like memory leaks and invalid pointer dereferences but misunderstanding generic programming paradigm can lead to new kinds of errors, for example, iterator invalidation or improper use of algorithms (`unique`, `remove`) [5]. The include dependencies in the STL are not well-defined and many dependencies belong to the implementation of STL that can be a main reason of a subtly portability issue [6].

B. Static analysis

Static analysis is a kind of software analysis. Its idea is analyzing the software without execution, it can target the source code or byte code as well. The main advantage of this method is that the code is not executed. In general, finding bugs in an earlier stage of software development highly could reduce the cost of the software [7]. The tools are not perfect, they could not find all bugs and sometimes go wrong. However, these tools are beneficial ones [8]. Using these kind of tools during the development can be effective in many respects, for example these tools can provide a kind of automatic checking of the source code.

Of course, the static analysis technique used to affects what kinds of problems can be detected. However, the language also affects the efficiency of the analysis. 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[9]. Symbolic execution have become more and more important in the modern software development [10]. The complexity of the analysed language also can influence the algorithms, for example the analysis of a language which has pointers could be harder than others which do not have [11].

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.

One of the advantages of Clang is its modular architecture. One can use the parser as a library to build tools. The popularity of this compiler also implies that it is well tested. Clang has been applied in many different static analysis tasks: finding semantic mistakes in the usage of STL [5], searching for semantic differences in source code according to different C++ standards [12], improving static analysis with function summaries [13], finding move semantics related bugs [14], detecting uses of uninitialized memory in C and C++ [15], customizable style validation [16], speed-up special operations [17] etc.

III. Theoretical issues

The following theoretical issues will be addressed in this paper. To get a deeper view on these, the next subsections will summarize the problems. All of them are related to include hierarchy and the usage of header files, but each of them is different.

A. Internal STL hierarchy

1) Motivation: C++ is a standardized language [3]. The first official standard released in 1998. Three improved C++ standards was released later and the next one comes in 2017. The language is continuously evolving to meet the expectations. The C++ Standard is a very long and formal paper and defines the element of the C++ language and the Standard Template Library provided features, but there are subtle opportunities for the compiler and library implementations.

Also, there are some weaker points in the Standard. In C++, to modularize our program, the header inclusion can be used which comes from the C language. Header inclusion happens during the preprocessing stage of the compilation. The include preprocessor directive provides header inclusion. When the preprocessor reaches out an include directive, it opens the given file and copies its content to the place of directive. Also, this process can be recursive. At the end of preprocessing of a translation unit, the preprocessed source file is ready that contains all data which is necessary for the compiler. For example, if there is an include for *vector* header, preprocessor will copy the content of *vector* header to the place of include [18]. This mechanism increases the build time intensively [19].

Standard Template Library uses header inclusion to separate headers into different files. The Standard defines for each STL header file what it has to provide, for example *vector* standard header has to provide a data structure called `vector`. The developers can include these files, if they would like to use their features. Some standard headers need to use a symbol from another standard header file. It is possible, but this mechanism results that, the features of included header file are available by including the other one.

It would be okay, the different modules have to be connected in some cases, for example map uses features of utility. However, it could lead issues later.

The main problem is that the C++ Standard does not define exactly which standard headers must include the other ones. It depends on the developers of current STL library implementation because they create the hierarchy of the library. Thus, the hierarchy of STL implementations could be different. It is not a problem when the standard headers are used correctly, but the compilers do not check it comprehensively. Therefore it can be the root cause of portability problems.

Easy to write code that compiles with an implementation but fails with another. These unportable codes should result in compilation error with every STL implementation, but the compiler does not know that is wrong. It finds the used definitions and does not check how they can be reached.

2) Examples: This problem appears when the developers try to take advantages of hierarchy, that is not well defined. In the next examples, the problem will be highlighted more obviously and how easy to commit this kind of mistake.

In this case, the compiler have to be upgraded, that requires to upgrade the used STL implementation also. We used g++ 5.3 with libstdc++ STL implementation. Here is a simple program that compiles and works fine. For example, see Listing 1, there is an include directive for algorithm header, but in the main function a variable with type `std::vector<int>` is constructed.

```
#include <algorithm>

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

Listing 1: Vector from algorithm

Next step is upgrading 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'
```

Listing 2: Compiler error

How this situation could happens? 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.

In the next example, compiler is switched with another and this also results in switching one STL implementation with another. At the beginning, Clang compiler is used with libc++ STL implementation. Another interesting example can be seen in Listing 3, which is very simple, it just includes $algorithm$ header and uses `std::shared_ptr` through this header.

```
#include <algorithm>
#include "myclass.h"

int main() {
        std::shared_ptr<MyClass> x;
}
```

Listing 3: Header memory from header algorithm

It compiles and works with Clang using libc++ STL, but we change the compiler to g++ 6.1 with libstdc++ STL. It results a compilation error, because `std::shared_ptr` class template cannot be found.

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

Listing 4: Compiler error

The root cause of this error has been that we have changed the used STL implementation to another one. In libc++, $algorithm$ header includes $memory$ because some implemented algorithms use smart pointers from $memory$ header. In the libstdc++ implementation, the algorithms do not use smart pointers so $algorithm$ does not include $memory$ header necessarily.

3) Clarification: As a short summary for the problem, the include hierarchy in STL implementations could be different and it can be changed by time. It could be annoying, if the code that compiles with a specific compiler, does not compile with a different compiler that uses another STL implementation. To avoid this problem no one should take advantage of the include dependencies of the used STL implementation.

B. Legacy style C header include

For backward compatibility with C language, the C++ has to provide features of C standard library. It means, almost every C standard header file is available through their appropriate C++ version, for example, features of *stdlib.h* is available by *cstdlib*. The main difference between these header files is that the C++ version brings functions into the `std` namespace.

On the other hand, the original C header files can be included, but the C++ standard formulates them as deprecated. It does not require, that these headers have to be available by definition at all times. The C headers are accessible only, if the build environment contains a C compiler beside the C++ compiler, for example gcc and g++. It is possible, the C++ compiler is only a C++ compiler and does not care about C, that implies the C standard header files are not available certainly. Since the C++ only requires the C++ version of C standard header files, a short example can be seen in Listing 5.

```
#include <stdlib.h>

int main() {
    abort();
}
```

Listing 5: C header usage

C. Unused header files

The unused header files are not really an issue because they do not cause any compilation error. By definition a needless include does not contain any declaration or definition, that is used somewhere in the source code. That means, these kind of includes safely can be removed without affecting the build procedure [20].

However, the deletion of unnecessary header files is a build process optimization because unused include files can highly increase the build time [21]. Just for clarification, in the next sample in Listing 6 depicts how an unused header looks like.

```
a.h:
void foobar();

main.cpp:
#include "a.h"
int main() {
}
```

Listing 6: Unused header

These problems often occur in big C/C++ projects. Meyers devoted an item for the STL-related header files but no tool was mentioned to detect the misusage [6]. We have analysed many open-source projects and all of them contains at least one of mentioned problems, so we have developed a tool.

IV. The tool

A. Technical background

The previously described problems can appear from time to time, and it often results in a compiler error that must be fixed. To detect these kind of problems, a new tool was created to statically analyse the source code. The targeted problems are related to each translation unit and its include hierarchy, so translation units can be processed separately from each other.

The tool is based on Clang library and takes advantage of its libraries, for example. use its abstract syntax tree, which is created from the analyzed source. The main advantage of using Clang is the proper C++ parser that follows the C++ standard.

B. Conceptual problems

We have faced some interesting problems, which require to solve them in order to create a usable, handy tool.

a) Mandatory includes: Almost every standard header uses features from another ones, but it is not well documented. Among the used includes, there are many which are required to be included. That means, it must include them to implement functionality defined by the standard. This mechanism can help us to filter out a lot of false positive results because a header file cannot be implemented without include another one. This kind of required relation is transitive such as the header inclusion. For example, *map* has to include *utility*, because `std::map`'s `insert` member function returns a pair of iterator and bool, that means, the insertion succeeded if *first* member of `std::pair` is an iterator to the newly inserted element and the *second* member's value is true. If it is not, the *first* contains an iterator to an element with equal key and the *second* is false.

```
pair<iterator,bool> insert (const value&);
```

Listing 7: std::map::pair

b) Private headers: To determine the exact location in terms of header of a given class or function template definition may be very hard. In theory, the expectation is that the standard header files contain the definition for data structures, functions and other programming units but in practice they do not. The C++ Standard specifies that the *vector* header has to provide definition for `vector` type, but it does not specify that the definition of vector has to be placed in *vector* header. As previous examples depict that the header inclusion mechanism is transitive, so if *a.h* includes *b.h* and *b.h* includes *c.h*, then the *a.h* also includes *c.h*.

However, the definitions may be separated into different header files and these headers are included in the standard header. This mechanism fulfills the requirements of C++ Standard, because if one includes *vector* in the source file and *vector* includes *vector.h* that contains the definition for *vector* class template, then `vector` class can be accessed through *vector* header.

In some STL implementations header inclusion is used. That makes hard to determine the symbols real place because during the analyses the tool searches for a symbol, that it comes from *a.h*, but we need that information which Standard header functionality is implemented in this header. For example, in libstdc++ the implementations mainly are in other "internal" header files, those are included by standard headers, e.g: *algorithm* includes *stl_algo.h* and *stl_algobase.h*, which contain the implementation of algorithms. So, need to track these includes, to find their belonging standard header.

The tool can catch these internal headers, because they are not defined by user, so they come from the system and are not standard header files. In the dependency graph, the belonging relation between the header file is marked with a different kind of edge. For more precise result the belonging relation in the graph should be corrected in analysing stage that requires deep knowledge from the C++ Standard. This mechanism can be used to determine which standard header file's functionality implemented in a system header. For example, *vector* header has to provide `std::vector`, but it is implemented in *vector_impl.h* file, then the tool has to know that the *vector_impl.h* file implements a part of functionality of *vector* header, so it belongs to the *vector* header.

c) Unused header indirect usage: The deletion of needless includes can cause problems indirectly, because they do not contain any used declaration, but it can include other header file that contains a declaration, which is used. In this case, the tool has to detect dependencies between include files and before a header file is suggested to be removed, it should check this header file's include dependencies and add the used ones, if is not included by another header file.

d) Knowledge from C++ standard: The tool requires information from the C++ Standard. Some of them are not must have dependencies, but they can help to improve the process. Many kinds of knowledge are required from standard, to make analysis more precise.

- Header files: the tool has to determine whether a header file is C or C++ standard file or it is not. The system_header pragma usually helps to determine, that is user defined or not, but more precise decision is necessary. To be able to select standard header files, some knowledge is required from the C++ Standard. Those nodes which are standard header files should be marked in the graph.

- Mandatory includes, this is a list of headers which have to be included for each standard header file.
- Symbols: Symbols that are provided by standard header files. This is a very large data collection and it has collisions if we just check for names, for example in case of function overloading. These symbols are used during the visiting of AST to correct edges of these internal headers in the graph. This is not mandatory but it results in a more precise analysis.

C. High-level overview

In this part, a high-level overview will be depicted, how the tool works and solves any issues. The tool analyses the source code, thus execution of the code is not required. Also, it needs to known the exact compiler arguments for the analysis, for instance macro definitions. The build procedure usually has many different parts, such as compilation, linking etc. To follow this convention, the analysis can be separated into three stages.

a) Preprocessing: First of them is preprocessing, when the tool collects information about the include hierarchy. The preprocessor is used for this task. In Clang, the compilation process is very customizable, so someone can attach to the preprocessor with callbacks. When the preprocessor catches an include directive, the tool receives a callback to this event. The header inclusion uses deep first search, so all information about the include hierarchy of the processed translation unit can be collected.

During the preprocessing a dependency graph is building up that can be mapped to the origininal 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 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.

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.

b) Analyzing: The second stage is the analyses of AST and extract the necessary information from. After the preprocessing, the dependency graph is done for a while, but it may change in this stage. Right now, the preprocessed source code has to be analyzed. The Clang will do every parsing related task, for example tokenization, parsing, etc and it builds up the abstract syntax tree from the source code. Through Clang libraries, we just take advantage of its architecture by creating visitor for abstract syntax tree. The visitors support the walking on AST. Thus, using visitors, all kind of declaration, expression or etc can be catched in the code. After AST visiting, the graph is done and is not be changed later. An example graph is depicted in Figure 1. For reference, the circle nodes are STL related headers and the rectangles are user defined or other system headers in the graph.

c) Visitors: Two different visitors were used for different purposes. One of them is for catching every usage place in the user files of every possible function, class, typedef, macro etc. A few number of properties is used from used symbols, but the most important one is the location to know where the symbol is and where it comes from. For each usage, the corresponding node in the graph will be extended with what symbol is used and where. Catching all symbols is necessary to produce feasible output.

Another visitor is capturing definitions only from system headers. This functionality is necessary to properly detect private headers in the STL implementation. It watches definitions of function, class, typedef etc. and if they are not in standard header file, but their file is included by standard headers, then corrections should be made in the graph. In case private files will behave as standard header in the graph, because they really implements features of standard header. This kind of private header mapping and validation can be important because popular STL implementations use private headers to modularize their structure.

d) Graph analyzers: 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 analysers no 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. The Figure 2 depicts how the architecture looks like.

Different analyzers use different graph walking algorithms to detect the targeted issue. STL internal hierarchy analyzer walks over nodes which belongs to user defined files, checks the used symbols and searches for routes between their headers and the usage place. If there is no valid route, it creates a report for that header file where the definition is placed. At the end, it aggregates the results and removes duplicates.

In details, during the walking on nodes, the nodes are marked as valid or invalid flag, this flag indicates that using symbols from this file would be okay or not. The user-defined files always are valid, because we try to detect issues related to Standard Template Library. We can determine the routes from usage location to the location of definition of the symbol. If only routes are available which do not contain non standard header file, then the usage of this symbol could be marked an invalid.

For example: in *main.cpp*, `std::vector` is used in the

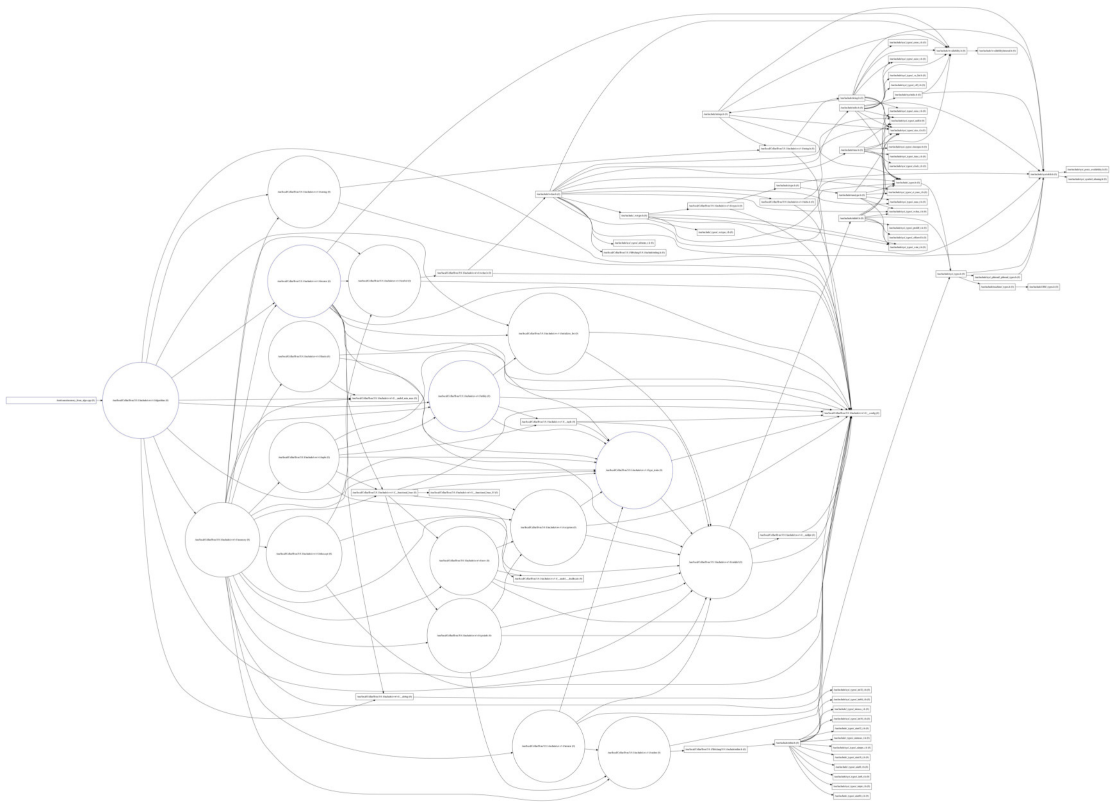

Fig. 1: Dependency graph

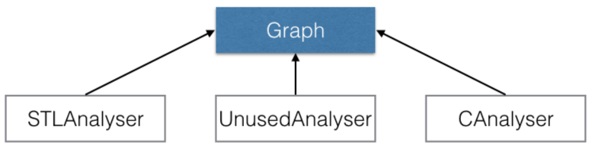

Fig. 2: Graph analyzers

declaration of function `foo` and the `std::vector` definition comes from *vector* header, so the routes from *main.cpp* to *vector* header are searched from in the include hierarchy graph. This seems to be reasonable because *main.cpp* includes directly *vector* header.

```
main.cpp:
#include <vector>

void foo(const std::vector<int>& v);
```

Listing 8: Vector usage sample

There may be multiple include directive paths to a file. In this case, we have to aggregate them to set the flag correctly. The used rule for aggregation is that, if there is a flow from a file that is included by user-defined file or it is a user-defined file then this inclusion is valid, otherwise it is not. In the next example in Listing 9, the problem is explained.

```
main.cpp:
#include <map>
#include "myheader.h"
...
std::pair<int, int> p;
...
myheader.h:
#include <utility>
...
```

Listing 9: Multiple path inclusion

In this example, there are two include directives for map and *myheader.h* in *main.cpp* file, but somewhere in the file the `std::pair` is used as type of variable. In *myheader.h* file, there is only one include directive for utility. Just for reference `std::pair` is defined in *utility*. However, map and *myheader.h* file include *utility* too. Two paths exist in the graph from *main.cpp* to *utility* that contain the `std::pair` definition. In this example the node of *utility* should be marked as valid because it is included through an user header too.

D. Output format

The combination of functionalities results in a powerful tool that can be used to analyse source codes for detecting portability issues and unnecessary includes, which increase the compilation time. The tool at the end of the analysis need to aggregate the results of three kind of analysis. The program has many filenames with used symbol names and usage place, header files that are not used. The result of the analysis is three lists, which contain header files:

- header files to be added as include and a file for each, where you should be add
- header files which proposed to be removed and a file for each, where they are right now
- C style standard header files with their current location, e.g. stdlib.h

For example, check Listing 10 for sample code and Listing 11 for the output.

```
main.cpp:
#include <stdlib.h>
#include <algorithm>

int main() {
// works with g++ 5.3
std::vector<int> v;
abort();
}
```

Listing 10: Sample

```
Add these include files:
vector (main.cpp)
Remove these include files:
algorithm (main.cpp)
Swap these C include files with
   their C++ version:
stdlib.h (main.cpp)
```

Listing 11: Output of analysis

V. Results

We have developed the described tool and the comprehensive testing has been started. The tool finds all mistakes that are presented in this paper as examples. All mentioned features are implemented and the tool is able to detect portability issues and needless includes.

During the development the tool was tested on open-source projects to verify the prototype of software. Quite popular open source projects were used for this testing. Different characteristics projects were selected for testing purposes, some of them are library and some of them are standalone application that provides solution for a given problem. Their programming style differs from each other, that makes the testing more complete. However, the size of projects is not so large, about many ten or hundred thousand lines of code, but enough to analyse them with our tool. Every found bug required to check manually to verify if it is really a bug that process takes much time. The complete list of the analysed projects:

- Tinyxml2, https://github.com/leethomason/tinyxml2
- Json for Modern C++, https://github.com/nlohmann/json
- Bloaty, https://github.com/google/bloaty
- Guetzli, https://github.com/google/guetzli
- Yaml-cpp, https://github.com/jbeder/yaml-cpp
- Flatbuffers, https://github.com/google/flatbuffers
- Orc, https://github.com/apache/orc

To analyze projects, the tool requires exact compiler invocations for each translation unit. These are stored in a json formatted file called compilation database[1]. Compilation database contains a section for each translation unit that describes how it was originally compiled. The sections have three fields:

- directory: directory of file
- command: the exact build command
- file: name of the compiled file

For instance, see Listing 12.

```
[
{
"directory": "/tmp",
"command": "clang++ /tmp/main.cpp",
"file": "/tmp/main.cpp"
}
]
```

Listing 12: Compilation database

These can be produced by capturing the build execution of project or can be generated by some build systems. There are a few software products that can capture build flow and some build systems can generate it by default, e.g. CMake can generate compilation database if you pass `CMAKE_EXPORT_COMPILE_COMMANDS` parameter.

For testing purposes, to make this process easier, an executor script has been developed that parses compilation database json files and analyses every source file that with our analyzer tool. It analyses the source files with given compiler options and writes the output of tool into files on the disk. It makes analysing of projects and evaluating the results easier.

The previously described issues were searched for, portability problem caused by taking advantage of internal STL hierarchy, unused header files, legacy C header file usage. In this the test phase, several open-source projects were analysed by our tool. The libc++[2] STL implementation was used to analyse projects. The results of analyses are depicted on Table I. The first column means the STL internal hierarchy usages, second means unused header count, and the third one means the C header usage in legacy style.

TABLE I: Results

	STL	Unused	C
Tinyxml2	0	0	1
Json for Modern C++	0	3	0
Bloaty	3	9	14
Guetzli	2	4	34
Yaml-cpp	1	16	2
Flatbuffers	0	3	7
Orc	4	8	20

[1]https://clang.llvm.org/docs/JSONCompilationDatabase.html
[2]https://libcxx.llvm.org/

As it can be seen, the hits by different issues are not same. The STL internal hierarchy usage issue were found in a few number of cases. Unused headers and C header usage have been found in a large number. The distribution of hits is not equal for each projects, some of them has many unused header and some of them has any other issues. To conclude this section, real issues were found in real open source projects, that proves that the tool can catch them. However, there were some false positive reports, but they have been fixed in the software and not counted in results.

VI. Related work

The analysis of C++ programs is quite popular these days, thus this is not the first tool which target to analyse the include dependency in any kind of way.

One of them is Include-what-you-use (IWYU) [22] tool from Google that is an open source project [3]. It analyses the include hierarchy as well, but it is different in regards of purpose of the tool compared to our. It generally suggests to include header file which provides the symbols that one uses in the source files. It aims to detect places where one does not include the header file which provides the given symbol. The similarity with our tool is both of them can detect unused headers but it comes from behaviour of the tools.

However, our tool tries to find several kind of portability issues along with unused headers in the include dependency.

VII. Conclusion

In this paper we have introduced subtle issues in the C++ Standard Template Library that cause portability issues. To overcome this kind of portability problem, we have created a tool to detect portability issues in the include hierarchy of Standard Template Library. Deep knowledge is required from the C++ Standard about the STL header files to detect this kind of mistakes, mainly how they include each other and what they need to provide by definition.

This tool is based on Clang library that is the most appropriate technique to create new kind of analysis tools for C and C++ nowadays. It is a command line tool, so it analyzes the source code and prints out the result, where one can fix and improve the implementation. Also, it is able to determine whether a header file is not used in a translation unit. It is implemented as an extra feature with minimal cost, but it is very useful feature of this tool. The tool has some impressive results, it is tested on open source projects.

References

[1] C. Lattner, “LLVM and Clang: Next generation compiler technology,” 2008, lecture at BSD Conference 2008.

[2] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.

[3] B. Stroustrup, *The C++ Programming Language (special edition)*. Addison-Wesley, 2000.

[4] N. Pataki, “C++ Standard Template Library by safe functors,” in *Proc. of 8th Joint Conference on Mathematics and Computer Science, MaCS 2010, Selected Paper*, H. F. Pop and A. Bege, Eds. Komárno: Novadat Ltd., 2011, pp. 363–374.

[5] G. Horváth and N. Pataki, “Clang matchers for verified usage of the C++ Standard Template Library,” *Annales Mathematicae et Informaticae*, vol. 44, pp. 99–109, 2015. [Online]. Available: http://ami.ektf.hu/uploads/papers/finalpdf/AMI_44_from99to109.pdf

[6] S. Meyers, *Effective STL*. Addison-Wesley, 2001.

[7] N. Nagappan and T. Ball, “Static analysis tools as early indicators of pre-release defect density,” in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE ’05. New York, NY, USA: ACM, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062558

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2013.6606613

[9] C. King., “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, pp. 385–394, 1976. [Online]. Available: http://dl.acm.org/citation.cfm?id=360252

[10] C. Szabó, M. Kotul, and R. Petruš, “A closer look at software refactoring using symbolic execution,” in *The 9th International Conference on Applied Informatics - Volume 2*, 2014, pp. 309–316. [Online]. Available: http://dx.doi.org/10.14794/ICAI.9.2014.2.309

[11] Z. Xu, T. Kremenek, and J. Zhang, “A memory model for static analysis of C programs,” in *ISoLA’10 Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I*. Heidelberg: Springer-Verlag Berlin, 2010, pp. 535–548.

[12] T. Brunner, N. Pataki, and Z. Porkoláb, “Backward compatibility violations and their detection in C++ legacy code using static analysis,” *Acta Electrotechnica et Informatica*, vol. 16, no. 2, pp. 12–19, 2016. [Online]. Available: http://dx.doi.org/10.15546/aeei-2016-0009

[13] G. Horváth and N. Pataki, “Source language representation of function summaries in static analysis,” in *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS ’16. New York, NY, USA: ACM, 2016, pp. 6:1–6:9. [Online]. Available: http://doi.acm.org/10.1145/3012408.3012414

[14] A. Baráth and Z. Porkoláb, “Automatic checking of the usage of the C++11 move semantics,” *Acta Cybernetica*, vol. 22, no. 1, pp. 5–20, 2015. [Online]. Available: http://dx.doi.org/10.14232/actacyb.22.1.2015.2

[15] E. Stepanov and K. Serebryany, “Memorysanitizer: Fast detector of uninitialized memory use in C++,” in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 46–55. [Online]. Available: http://dx.doi.org/10.1109/CGO.2015.7054186

[16] N. Pataki, T. Cséri, and Z. Szűgyi, “Task-specific style verification,” *AIP Conference Proceedings*, vol. 1479, no. 1, pp. 490–493, 2012. [Online]. Available: http://dx.doi.org/10.1063/1.4756173

[17] V. Májer and N. Pataki, “Concurrent object construction in modern object-oriented programming languages,” in *The 9th International Conference on Applied Informatics - Volume 2*, 2014, pp. 293–300. [Online]. Available: http://dx.doi.org/10.14794/ICAI.9.2014.2.293

[18] B. Babati, N. Pataki, and Z. Porkoláb, “C/C++ preprocessing with modern data storage devices,” in *2015 IEEE 13th International Scientific Conference on Informatics*, Nov 2015, pp. 36–40. [Online]. Available: http://dx.doi.org/10.1109/Informatics.2015.7377804

[19] J. Mihalicza, “Compile C++ systems in quarter time,” in *Proceedings of 10th International Scientific Conference on Informatics*, 2009, pp. 136–141. [Online]. Available: http://dx.doi.org/10.14794/ICAI.9.2014.2.309

[20] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, “Removing false code dependencies to speedup software build processes,” in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’03. IBM Press, 2003, pp. 343–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=961322.961375

[21] J. Mihalicza, “How #includes affect build time in large systems,” in *Proceedings of the 8th international conference on applied informatics (ICAI 2010)*, 2010, pp. 343–350.

[22] C. Silverstein, “Implementing Include-what-you-use using clang,” 2010, 2010 LLVM Developers’ Meeting Talk.

[3]https://github.com/include-what-you-use/include-what-you-use