STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.04

INSTRUMENTATION OF C++ PROGRAMS USING AUTOMATIC SOURCE CODE TRANSFORMATIONS

ZSOLT PARRAGI AND ZOLTÁN PORKOLÁB

Abstract. The main tool for programmers is always the compiler, but there are also many other tools to help the development process. Some of these tools work on the source code of the program, analyzing, measuring or transforming it. Implementing a source based tool is a complex task, especially for complex languages such as C++. In recent years the C++ language received an easy-to-use library for developing such software, in the form of clang tooling. However, this library only focuses on processing a single translational unit of the program, independently to the other parts of the build process. Tools which ignore this big picture could result in failures when used on larger projects, or incorrect runtime behavior. In this paper, we describe some of these challenges encountered in real-world C++ projects and propose possible solutions for future tools to fix or mitigate the issues.

1. Introduction

There are tools which work on an already built binary, by intercepting calls (such as strace[13]), running the code on a virtual machine (such as valgrind[14]), or by transforming the binary before (such as syzygy[6]) or during (such as orbit profiler[12]) the execution of the program. There are tools which work within the compiler, using transformations on the intermediate language in it - for example, sanitizers[10] in the compilers are usually implemented this way. There are also tools which work by analyzing, and possibly modifying the source code. For example, static analyzers[11] work by performing more detailed checks on the source code, even providing automatic correction options for some cases.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: PROGRAMMING LANGUAGES – *Languages Constructs and Features.*

Key words and phrases. C++ programming language, source code transformation, instrumentation.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

All of these have their disadvantages.

When a tool works on an already existing binary, it lacks information. Debug symbols can be generated for any type of builds, but optimizations, such as inlining limit the options available for tools even then. This can be countered by running the tool on binaries compiled with special flags and providing an API that programs can use to share information.

When a tool is integrated into the compiler, it is only available with that compiler – and as these tools are often under active development, possibly even limited to recent compiler versions, limits their use especially for software targeting several compilers, operating systems or platforms.

When a tool works on the source code, it is limited to the capabilities of the language, and it is subject to the differences between the compilers and the complexity of interpreting the source code. These disadvantages are especially crucial in the case of C++: While the C++ standard[7] is detailed, it leaves choices to the compiler, and there are several examples for the most used compilers providing different results for simple looking C++ programs - sometimes even diverging from the standard. Interpreting the language is also challenging because of the preprocessor: as C/C++ programs tend to use many different configuration options[1], there is not a single AST to be analyzed and modified.

Most tools could be implemented using different techniques, and there are examples for implementing the same tool in different ways: Code coverage can be measured with in-compiler instrumentation (the `-fcoverage` option of clang), with transforming the source code (Coco[**Bullseye**]), or with a tool working on a special binary (gcov[5]). On the other hand, it is entirely possible that a tool can not be implemented in all three ways: Uninitialized memory reads can be detected by a binary tool (valgrind) or an in-compiler instrumentation tool (memory sanitizer in clang), but implementing it with a source transformation is not possible within the limits of the language.

In this article, we focus on the problems and possible solutions when implementing AST level source code instrumentation tools based on clang tooling. Source-based tools were chosen because of their generality: tools working on the binary or as part of the compiler are limited to the platforms where the tool runtime or the compiler is supported. This is often a limiting factor even on desktop systems - several tools, such as valgrind, or the clang sanitizers only work on Linux-like systems. On other, especially embedded systems, the problem is even more significant: these targets often have custom compilers, making compiler based techniques unusable, and possibly limited or no support for running external runtime tools along the main program.

Our focus is how these transformations can be integrated into and performed on large-scale projects. We discuss questions like how a source code transformation tool can be included in the build process, or how the amount of available configurations increases the complexity or possible problems[16][15].

It is also important to mention that source code transformations could easily result in behavior changes of the program[4], and ensuring that these are not happening is a similarly important aspect of tool development. In some cases, this is impossible. In this situation, it is important to minimize and document these - as in the case of the previous Coco example, which results in behavior changes with specific operators. While we mention that this is an issue, further analysis of the question is out of scope for of article.

2. Transformation overview

A C++ program is built by transforming every C++ source file separately into an object file, then linking those object files, and dependencies together into a library or executable. This process is also layered: the dependencies used by the linking step are built similarly, but often provided only in the final, binary form. Larger projects usually consist of multiple components, each built this way, depending on each other. Based on this, we can split dependencies into two categories: internal, which are built by the project, and external, which are expected to be found in a compatible binary format.

The first issue with program instrumentation is handling the dependencies: when the instrumentation changes the build process – as in the case of in-compiler or source based instrumentations –, it is possible that changes have to be made in the dependencies. An extreme example for this is the memory sanitizer[10]: it requires every dependency, including the C++ standard library, to be built using the memory sanitizer.

This is, even more, an issue when using source transformation tools: in this case, the task is not only the addition of some compiler flags into the build process of the dependency, but the actual execution of another tool during its build. It is also important to note that some dependencies are only provided in a binary form, making transformations in them impossible.

While the tool itself can not make the task of building everything in the necessary way more manageable, the problem can be mitigated by limiting what parts of the software transform. In case of the memory sanitizer example, there are no better choices because the way it is designed, but most software should be implemented in a way that would allow at least limited usage without rebuilding everything. To achieve this, tools either need a way to decide which files they can safely change, or they should not rely on any change that would change the "interface" of a file.

Clang tools generally use a compilation database for executing the tool: a JSON file containing every compilation command with all of their parameters. This file can be generated by commonly used C++ build systems, and then the tool can look up the specific compilation parameters from it. This process is executed as follows:

(1) Configure the project. For some tools, like CMake[8], this step also generates the compilation database.
(2) Build the project. For some tools, like bear[9] with make, this is the step that generates the compilation database. If it was generated by the compilation step, and the project uses no generated source files, this step could be skipped.
(3) Run the tool on some or all of the source files.

This process works perfectly with read-only tools, that do not change the source code. It is also suitable for some transformation tools, by repeating the second step once more after the transformation and compiling the modified program. Unfortunately, this approach leads to issues in some special cases.

Focusing on a single component of the build process, C++ sources include other files, handled by the preprocessor. A header file can be, and often will be included by more than one source file. While this usually does not result in any issues, it is possible that different source files include the headers in the context of varying preprocessor definitions. It is also possible that a source file includes a header multiple times with a different preprocessor definition context, or simply a build can include a source file multiple time with different compilation flags, providing a different name to the resulting object file.

These all could cause problems when changing the source code:

```
#ifdef SOMETIHNG_DEFINED
int foo(SOMETHING_DEFINED a);
#else
void foo();
#endif
```

With a simple transformation approach, it is possible that this file is transformed twice:

- first, when included with the definition set, only the first part is transformed
- after that, when included with the definition not set, only the second part is transformed

With the transformation process implemented the previously described way, depending on the exact sources, this could cause compile or runtime errors: As the process performs in-place transformations, the execution of the second compilation would work on the source file already transformed by the first execution of the tool. This is often the desired behavior: if the tools result

is permanent – such as when using automatic refactoring tools –, in the ideal outcome the transformation should include every sub-transformation required by any used configuration, and further runs of the tools should not result in additional changes. In this case, if the results conflict, the developer could be expected to look at them and fix the remaining issues manually.

With automatic, temporary transformations, however, user interactions during the build should be avoided, but keeping the same number of source files as initially is not a requirement: different translation units could use different versions of the sources, as long as these provide the same result a correctly implemented single file would. As the changes presented by the preprocessor definitions are limited to the current unit, this statement will hold. Based on this, the previous process can be generalized as follows:

(1) Configure the project.
(2) For every compiler invocation in the source code:
    (a) Invoke the transformation tool with the same parameters as the compiler, providing an out of place transformation in a unique temporary directory: every input file used by the compilation process should be written to a different location
    (b) Invoke the original compiler command, on the modified files
    (c) (Optional) Remove the temporary files

This change in the execution of the tool solves most of the mentioned issues: by doing an out-of-place transformation, always based on the original source codes, different transformation processes will not be based on the previous outputs – the tool will not accidentally transform the same source location multiple times. And by invoking the tool just before the original compilation program, we prevent accidental overrides: each compilation will be executed immediately after the required source codes are transformed. Finally, by requiring a unique temporary directory, we guarantee that parallel builds will not cause issues when the same file is used by multiple translational units transformed at the same time.

While these transformations increase the IO bandwidth required by the compilation commands, a memory file system could be used to avoid actual disk writes.

This approach also gives the advantage that it can be implemented as a wrapper around the compilation command. While the previous version required the generation of a compilation database and a separate run for the tool based on that database, the modified version does not need any change in the build script, except for a change in the compiler executable. This approach is used for example by the Coco coverage tool, which merely changes the system PATH seen by the build process.

The disadvantage is that this approach assumes that at least the transformation tool and the compiler can be run on the same platform. As our goal is constructing tools with clang tooling, which supports most Unix-like systems and windows, this is likely achievable.

We also have to note that with this simple modification, we did not solve the issue when a file is included multiple times, but differently within a single translational unit. Related issues are addressed later.

3. A note on compiler specific preprocessor definitions

The process we described is limited when the source code contains compiler-specific conditional blocks. While these are not common in high-level code, as the transformation process works on the entire source code, it will encounter these: they are commonly used in standard library implementations, and also in several widely used C++ libraries, such as boost[2].

In our experience, most transformations do not require changes in these parts of the code. If for some reason this is required, an AST transformation based tool can not be used. While it is possible to modify the predefined definitions for a clang tool, these conditions are not there without reasons in the source code. While it is possible that the only reason behind them is a compiler specific optimization or a compile-time optimization, the more likely reason is that other compilers can not understand the code within the condition.

An excellent example for this is the boost preprocessor library, which has numerous preprocessor conditions because of the slight differences between the preprocessors in different compilers. Trying to parse a different branch of that library other than what is designed for that compiler will likely result in errors during the early stages of compilation, and the inability of the compiler to produce a valid AST.

If such a macro is in the code base of the project the tool has to modify, and it is an uncommon case, the tool could get away be reporting a diagnostic, and provide developers the ability to manually resolve it. For transformations in third-party libraries and common occurrences, and when the tool has to guarantee that it will not miss any instrumentation, this is not an option.

As an example, the Coco[3] code coverage tool falls into this category: missing covered code would not be acceptable in a code coverage tool. On the other hand, coverage analysis also falls into the category where AST information is not required. While the tool is based on source transformations, it does so based on the token stream.

As it does not have to be able to construct a complete AST from the tokens, only to find the blocks and conditions in the code, it could change the transformation process to the following:

(1) Run the original compiler with an additional flag, which instructs it only to preprocess the source code, and output the result
(2) Instrument this preprocessed source code
(3) Run the original compiler using the modified sources

This process defers the preprocessing of the sources to the original compiler, to avoid any possibility of interpreting preprocessors definitions differently. The result should be a C++ source code, possibly referencing builtins specific to the used compilers. While these builtins will likely prevent another compiler from completely parsing and validating the source code, any compiler should be able to tokenize the result.

4. Mixing C and C++ code

Another issue is presented when C and C++ code is intermixed. While our goal is the instrumentation of C++ programs, C++ projects sometimes also contain C source files, which can not be built by a compiler in C++ mode.

If a C++ program also contains C source files, it adds additional questions when developing a tool. The first is, should the tool also have C support?

Some tools instrument code fragments that are only valid in C++ – in which case, they may safely ignore the question, as they will never have to modify a source code fragment which fails be parsed by a C compiler. Tools however often do not fall into this category.

Some tools could instrument C sources too, but the transformations done by the tool require a C++ compiler – it is possible that instrumenting C code in a similar manner is impossible, or will not be completely reliable. For example, RAII is unavailable in C sources, but it is possible for exceptions to pass through C code if it calls a C++ function.

Tools also have to be aware that header files may be shared between C and C++ source files. For this, the header file has to be compatible with both C and C++ compilation: Every C++ specific language has to be behind a conditional preprocessor directive. While the standard way to do this is the `__cplusplus` definition, some project uses their specific definitions provided by the build system.

The file also has to contain at least one global variable, or one function with extern C linkage when compiled with a C++ compiler, and these have to be in conditional sections which do not contain any not conditionally defined C++ symbols. Otherwise, even if the file is used by both a C and a C++ compiler,

the name mangling in C++ would ensure that the different language object files use different symbol names, preventing any possible issues.

The process described in the previous section ensures that the tool can not break the compilation of a header when it is used by a C or a C++ compiler: if a header is included in multiple translation units, it will be translated multiple times, differently. When used in a C compilation, the transformation process may either ignore it or could provide a C compatible transformation.

However, it also provides no information the way different headers are used: as it only wraps the compiler command and builds no external database about the files, it has no way to check if a source file is used with both languages. Compared to the previous example, where if a source code fragment was not disabled by a preprocessor definition, it was always transformed in the same way, in this case a program could end up with both a modified and an unmodified version of the source code, causing linking errors, or possible runtime problems.

One issue is caused by the linker: if an inline function has definitions in several object files, the linker will choose only one of them. In this scenario, when the C compiler does not instrument a function, but a C++ compiler does, the linker could choose any of the implementations.

```
// some_header.h
#ifdef __cplusplus__
extern "C" {
#endif
inline foo() {
  CPP_ONLY_INSTRUMENTATIN_MACRO;
  printf("bar\n");
}
#ifdef __cplusplus__
}
#endif
```

A possible workaround that during the compilation, the tool could convert global inline functions it has to modify to static functions. The issue with this approach is that with this change, the address of the function will be different in every object file, possibly changing the behavior of the program, if it depends on equality checks of the function addresses. This limitation has no possible automatic fix: While a wrapper function could guarantee that the same object address is used in every translational unit, it would also reintroduce the original issue in a more limited form. A tool also has no reliable way if the function pointer is used in a comparison. For this the best a transformation tool could do is to provide a diagnostic if it encounters the situation, and require the developer to solve it or silence the warning.

Another issue is presented by functions which only have declarations in the source file, and only affects tools that change function signatures: In this case,

it is possible that the tool correctly updates the signature, and changes the implementation of the function, which is in a C++ source file but does not update calls to the function in C code. Similarly, it is possible that the function is implemented in C, in which case the implementation will be unchanged, but the C++ callers will provide an additional parameter to it. As this is an extern C function, none of the above would result in linker errors – but both would result in runtime issues, where the exact results depend on the used calling conventions. While this is a more limited issue compared to the inline functions, it similarly has no automatic solution.

5. Dealing with conditional macro expansions

In the previous sections we discussed several issues presented by conditional preprocessor directives, but only in the context, that preprocessor directives will cause different parts of the source code to be compiled. Another issue presented by the preprocessor is macro expansion: when the transformation code has to modify a source code fragment which is at least partially is a result of the expansion of one or more preprocessor macro.

```
#define FACTORY_FUNCTION(T)       \
  T* create_or_return() {         \
    static T* instance = new T(); \
    return instance;              \
  }

// ...

INLINE_MACRO FACTORY_FUNCTION(TYPE_NAME_MACRO(foo,bar));
```

In a permanent transformation, the goal would be the transformation of the code behind the macro - so the code using it would remain the same, but the macros would expand to a different source. In an automatic tool, however, it could be easier to expand the macros to the actual source code generated by them, and then transform that source code. This could prevent several edge cases which could not be solved by the tool: for example, if the macro is defined in a header file, the tool can not be sure that every single of its use has to be instrumented.

Also, as every file is transformed uniquely by every translation unit, expansion will not cause issues even when a macro is defined differently for different compiler invocations. However it does not solve the previously mentioned issue, where one header, without an include guard, is included multiple times in a translational unit, but with differently expanding macros. In this case, the macros are expanded multiple times differently in the same file and could cause problems during the source code transformation.

As this can not happen when the `pragma once` compiler extension is used – that would prevent the second, different expansion –, we can provide a perfect workaround by cloning the file: when a transformation problem as described earlier is detected – a macro expansion is detected, but at a location where a macro was already expanded previously, but differently –, the file should be duplicated, and the include before, and after the conflicting should be changed to refer to the second file. The downside of this approach is that it assumes that the transformation tool has a detailed data structure about its transformations, and can perform this detection.

Alternatively, a simpler approach can be implemented in two separate phases:

(1) The first hooks into the preprocessing phase of clang tooling
   (a) It collects every source file used by the preprocessor into a list
   (b) If it detects that a source file was already used, and it again emits non-whitespace tokens, it also marks the location where it happened
(2) If it detected a multiply used source file, it creates multiple clones of that file and changes the invocing include directives. This can be implemented using a virtual in-memory filesystem in clang tooling, without writing anything to a file system handled by the operating system. After that, it starts over with using this virtual file system.
(3) if it did not detect any source file used multiple times, it runs the second phase, which is the real tool as previously described, after the AST was parsed.

In this process, we can guarantee that the second rerun will not contain any source files used multiple times and that the preprocessor tool should not result in any noticeable change in the behavior of the program. The AST tool also requires no modifications, as if a macro in the source is expanded differently multiple times, it is hidden behind the preprocessor tool.

The disadvantage of this approach is that the tool will duplicate files even when it did not have to, as it can not tell if the tool will modify them. However this is a rarely used possibility in the language, and as such, will not result in any noticeable performance hit for most projects. In the detection step, it also assumes that the process only transforms actual C++ code, not macro definitions - it will not duplicate files which have no header guard, but only change preprocessor symbols.

After these modifications, the tool will be able to safely expand macros that have possibly different results but depend only on information defined by the project. However, macros depending on external information result in different issues:

- There are standard macros, which could be possibly changed by the tool, such as `__FILE__` or `__LINE__`.
- There are nonstandard macros, which in practice work the same way for every compiler, such as `__COUNTER__`
- There are macros which are often different for compilers, such as `__clang__` or `_MSC_VER`.

A transformation tool has to deal two possible situations with these: macros conditionally depending on these expressions – either directly, or indirectly, by a used macro –, or macros using these macros in expansions.

Some answers are clear in the previous list – for example, unless the tool can be sure that the real compiler will do the same steps as the clang tool, it should not expand a macro. In this case, the good answer is most likely only providing a diagnostic – based on that the developer or the tool developer may investigate the issue further, and possibly improve it. As an example, if the tool detects that the real compiler is GCC, and the condition is only based on `GNUC` macro, without depending on a clang specific macro, the expansion can be safely done. However, if the preprocessor did not take a previous condition only because while it allowed GCC, but disallowed clang, it is no longer expandable. This analysis requires a rather complex logic and understanding of different compiler internals. These decisions also require logging how the preprocessor evaluates conditions in the clang tool, making it realistic only for tools that often encounter these special cases.

Other decisions are not easy to decide: expanding macros such as `__COUNTER__` or `__PRETTY_FUNCTION__` could be perfectly safe even if the real compiler would interpret them somewhat differently. Another good example for this is the `__DATE__` macro, which is provided by every compiler, but it is evaluated differently every time.

Instead of expanding them, however, another approach is a more limited, and slower macro expansion: while the mentioned macros are often used in various C/C++ projects, it is unlikely that a tool has to transform the actual tokens generated by these macros. A more likely situation is that these macros are used in another macro - which also generates the source code which has to be transformed. In this case, instead of the full expansion of the source code fragment the tool has to transform, it could try to use a more restrained approach by only expanding macros one level at a time, and stopping as soon as the source locations where the transformation hat to take place are actual tokens.

This also means that if a transformation has to modify two locations – for example, wrapping a function call in another –, only the macros that contain

the location before and after the function call has to be expanded - macros within the function parameter list can stay unexpanded.

While this limitation does not solve the original issue, as there can be still situations the tool can not handle, it will greatly reduce their number. By implementing a similar approach, a tool has to emit fewer diagnostics about unexpandable locations.

6. Conclusion

In this article, we presented a methodology for developing C(++) source transformation tools, which is simple for developers to implement but also reduces the possible compilation or runtime issues caused by it. While we were not able to provide an automatic solution for every possible corner case, we provided workarounds to reduce the times the software has to provide diagnostic and/or require active interaction from the developer using it.

The methods we described are primarily intended for automatic AST based source code transformations, but most of the techniques we described could be adapted for other tools: for example, while most of our used methods are unsuitable for automatic refactoring, or permanent transformations, the described ideas could be used to implement the error detection and recovery capabilities of these tools in large projects.

We also have to note that while our results improve the usability and precision of these source transformation tools, it could be improved – both with the open questions in the mixed language projects and macro expansions. There are interesting improvement possibilities, such as logging the possible transformation conflicts during the build process, using an external code-database for finding how functions or headers are used or reconstructing actual macro expansions based on the differences between the preprocessor output of the actual compiler and clang.

We also only described the methods of these techniques without providing an actual library implementing these features: most of the methods described are generic and could be implemented in a generic reusable way for any tool, but current implementations only exist as a part of actual code instrumentation tools. The development of a ready to be used simple toolset would certainly reduce the cost of writing code transformation tools.

We only focused on the build process and the preprocessor, without mentioning the additional issues caused by accidental semantic changes in the program. With the capabilities of clang tooling, it would be possible to develop a framework which could validate that a given source change would not result in any unintended side effect - that is, apart from adding the additional

instrumentation, logging, or validation, it will not cause changes in the original program flow.

While code transformation and analysis is part of the development process for a long time, the increasing number of available tools for developing such software makes it an interesting research area and makes the development of usable and reliable tools easier.

References

[1] "An Empirical Analysis of C Preprocessor Use". In: *Software Engineering, IEEE Transactions on* 28 (Jan. 2003), pp. 1146–1170.

[2] boost. *Boost C++ libraries*. 2018. URL: https://www.boost.org/ (visited on 03/31/2018).

[3] FrogLogic. *Coco coverage analysis tool*. 2018. URL: http://www.froglogic.com/coco (visited on 03/31/2018).

[4] Alejandra Garrido and Ralph Johnson. "Challenges of Refactoring C Programs". In: *International Workshop on Principles of Software Evolution (IWPSE)* (Jan. 2002), pp. 6–14.

[5] GCC. *gcov - A test coverage plarform*. 2018. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on 03/31/2018).

[6] Google. *Syzygy Transformation Toolchain*. 2018. URL: http://github.com/google/syzygy/ (visited on 03/31/2018).

[7] ISO. "ISO/IEC 14882:2014 Information technology — Programming languages — C++". In: Geneva, Switzerland: International Organization for Standardization, 2014.

[8] Kitware. *CMake*. 2018. URL: https://cmake.org (visited on 03/31/2018).

[9] Nagy Laszlo. *Build EAR*. 2018. URL: http://github.com/riszotto/Bear/ (visited on 03/31/2018).

[10] LLVM. *Clang Memory Sanitizer*. 2018. URL: https://clang-analyzer.llvm.org/ (visited on 03/31/2018).

[11] LLVM. *Clang Static Analyzer*. 2018. URL: https://clang-analyzer.llvm.org/ (visited on 03/31/2018).

[12] pierricgimmig. *Orbit Profiler*. 2018. URL: http://github.com/pierricgimmig/orbitprofiler/ (visited on 03/31/2018).

[13] strace. *strace*. 2018. URL: https://strace.io/ (visited on 03/31/2018).

[14] Valgrind. *Valgrind instrumentation framework*. 2018. URL: http://valgrind.org/ (visited on 03/31/2018).

[15] Laszlo Vidacs. "ICSOFT 2009 - 4th International Conference on Software and Data Technologies, Proceedings". In: vol. 1. Jan. 2009, pp. 232–237.

[16] Daniel Waddington and Bin Yao. "High-fidelity C/C++ code transformation". In: *Science of Computer Programming* 68.2 (2007). Special Issue on ETAPS 2005 Workshop on Language Descriptions, Tools, and Applications, pp. 64–78. ISSN: 0167-6423. URL: http://www.sciencedirect.com/science/article/pii/S0167642307000718.

Department of Programming Languages and Compilers, Eötvös Loránd University

Email address: zsoltparragi@caesar.elte.hu, gsd@elte.hu