# A Framework for C++ Exception Handling Assistance

Endre Fülöp, Attila Gyén and Norbert Pataki*

*Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University*

**Abstract**

Exception handling is a beneficial language construct in the modern programming languages. However, C++'s type system does not really conform to these elements. As a consequence, developers have to pay attention to avoid mistakes because of the missing compiler support. Moreover, exception handling is a typical runtime activity that has less impact on the compile time approaches. Static analysis is an approach in which we reason about a program based on the source with no execution of the analyzed code. It can be used for many purposes, for instance, finding bugs, refactoring the code, or measure code complexity. In this paper, we present our framework which analyses the exception handling structure of whole software projects. Using the analysis results, the tool highlights locations in the source code where impactful changes can be made to improve exception handling. Our solution visualizes the details of exception handling, assists the developers to comprehend subtle details. Our approach uses static analysis approach and the implementation takes advantage of the Clang compiler infrastructure.

**Keywords**

C++, exception handling, static analysis, Clang, visualization framework

## 1. Introduction

Exceptions as an error handling mechanism appear in many languages, some of the most notable examples of widely used languages with exceptions are Java, Python, and C++. Exceptions are an alternative to exhaustive precondition checking and error code handling. One of the main benefits of using exceptions is that there exists exception handling implementation, which incur no runtime overhead in case the exceptional path is not taken [1]. Checked exception thrown by functions is part of the signature in case of Java, but for example, in case of Python, exceptions do not appear in the type signatures of functions and callable language structures. Exceptions introduce an alternative control flow in the codebase they appear in, so the post-conditions of functions and methods have to be carefully examined if exceptions are in-use. In Object Oriented Programming (OOP), a method of a class can be reasoned about from the point of view of the invariant properties that the data of an object has during the lifetime of the object.

The alternative control thanks to exceptions can alter the guarantees of method, and for example in the case C++ Standard Library (STL) using the generic containers with types that can throw exceptions during their lifetime, can lead to less robust or less performant code being generated, just to keep the implementation correct. The exception guarantees of containers in

*SQAMIA 2022: Software Quality, Analysis, Monitoring, Improvement, and Applications, September 11–14, 2022, Novi Sad, Serbia*

*Corresponding author.

gamesh411@gmail.com (E. Fülöp); gyenattila@gmail.com (A. Gyén); patakino@elte.hu (N. Pataki)
0000-0002-7519-3367 (N. Pataki)

this case depend on whether the contained type can throw exceptions or not. In C++ standards, C++11 is a more recent (which is referred to as "modern C++" in this paper) in which exceptions appear in the signature of the function in an inverse manner [2]. In modern C++, the encouraged way telling a compiler about a function's or callable language structure's behavior regarding exceptions is specify when they are guaranteed to **not** throw an exception. This information can be explicitly written, but is not inferred by the C++ compiler automatically. Also if the binary consists of multiple libraries linked together, the specific structure of the compilation units can happen to avoid throwing exceptions even when the individual function calls are prepared to handle exceptions. A static analyzer tool can explore the possible executions paths, to determine which functions or callables can throw which exceptions.

Clang is a C++ compiler which supports the modularized development of static analyzer tools [3]. Clang has a libary (LibTooling) for implementing custom tools that use the AST for extracting information about source code. We have developed a solution using Clang's LibTooling library to extract exception information about a C++ project. This information can be used to generate descriptions for IDEs to help developers understand their source code easier, and use this information to mark their functions as a non-throwing function. This could then lead to more extensive use of move semantics in case of STL containers, more possibilities for optimization.

This rest of this paper is organized as follows. In Section 2, we show related work. In Section 3, we briefly present how the exceptions worked earlier and recently. After, our work is discussed. Section 4 presents how the proposed static analysis method collects information from the source code. The execution of scanning algorithm is detailed in Section 5. Section 6 presents our approach which takes advantage of the processed exception information and prepares it for the comprehension. In Section 7, the Monaco Editor-related efforts are shown. Finally, this paper concludes in Section 8.

## 2. Related Work

The Clang compiler offers warnings for using deprecated exception specification in C++, and identifies if one uses exception handling language constructs in an unintended way [4]. Other than those warnings the compiler frontend do not provide any feedback on the exception-related behavior of the program.

The LLVM optimizer tool `opt` has a pass named `prune-eh` that can remove unused exceptions in the intermediate language representation [5]. This, however, does not affect the C++ language specific overload resolution differences. The optimization possibilities coming from this can not be exploited unless the programmer injects this information in the C++ parsing stage. The infrastructure proposed can help generate this information for the programmer to use.

Clang Tidy is a static analysis tool implemented as part of the LLVM. Clang Tidy has checks, which enforce style guides, identify possible bugs in programs, and for some of these, offer automatic fixes. Clang Tidy has 2 exception-related checks, `modernize-use-noexcept` [6] and `bugprone-exception-escape` [7].

Ada has a similar construct for exception handling. However, Ada does not support user-defined exception types. A static analysis method for exception analysis is presented [8]. Because

```
void f() {
   throw "Exception";
}

void g() throw(double) {
  throw "Exception";
}
```

**Figure 1:** Dynamic exception specification

of the older construct of Ada exception handling, the approach is oversimplified for the modern C++ realm.

Prabhu et al. presented an interprocedural exception analysis approach for C++ [9]. This approach utilizes the control-flow induced by exceptions and transforms it into an exception-free counterpart. The proposed lowering transformations do not affect the precision and accuracy of any subsequent program analysis. However, this method does not support effective visualization for suggestion how to handle exceptions.

## 3. Elements of Exception Handling in C++

Exception handling is an essential control structure in modern programming languages. However, C++ has a strong relation to the C programming language that does not support exception handling with specific control structures. This relation makes C++'s exception handling hard.

C++ enables exception handling with *try* blocks that indicate a block is inspected at runtime whether an exception is occured. After a try block, *catch* clauses are intended to handle the problems. Exception can be raised with the *throw* construct.

Any type of object or variable can be thrown in C++. However, there is a polymorphic base class in the standard library that should be a base class for raised exception. According to the C++ Core Guidelines, throw of built-in and `std::exception` should be avoided.

Since the missing constructs of exception handling in C and the C++'s backward compatibility to C, it is not required to express in the function signature if a function may throw an exception. Ada has a similar construct [8]. On one hand, the type of an exception can be specified which can be raised during the execution of the function. On the other hand, this information is not validated at compilation time, only at runtime, so it is called dynamic exception specification. Consider code snippets in Figure 1.

Nevertheless, it is not advised to throw string literals as exception. However, this is valid in C++. Both functions do compile. Moreover, the call of `g` results in the invocation of `std::unexpected` function that terminates the execution. Runtime validation of exception specification is not a sophisticated approach and not advised comprehensively [10].

C++11 improves the constructs of exception handling. The one of most significant construct is the `noexcept` specifier that states that the associated function does not throw any exceptions. Unfortunately, this behavior is also checked at runtime. Therefore, the code snippet in Figure 2 can be compiled.

```
void f() noexcept {
  throw "Exception";
}
```

**Figure 2:** Noexcept specifier

To be more perplexing, a function which does not emit any exception can be compiled without noexcept specifier. For instance, the code snippet in Figure 3 also compiles without any warning message.

```
int f( int i ) {
  return i;
}
```

**Figure 3:** Valid code with missing no exception specifier at an exception-free function

## 4. Generating exception information for projects

Clang LibTooling is a community-developed open-source software that operates on compilation databases. A compilation database is a standard JSON formatted encoding of the projects structure, listing every compiler action needed to build the project. Each compiler action specifies a main C++ file, which with all its included headers constitutes a translation unit (TU), and possibly linker flags for specifying where is the implementation of functions, that have no definition in the current TU. We developed a static analysis tool that uses Clang LibTooling as a source level library and takes a compilation database as input and a C++ file for which there is a corresponding entry in the compilation database. The tool then generates the abstract syntax tree (AST) of the TU, and traverses the AST to generate the exception-handling-related information of the functions. The information at this stage is on only local, and does not take the interprocedural connections into consideration. After the local exception information is calculated, the tool determines the possible caller-callee relations that exist in the translation unit. The interprocedural exception information is serialized into an indexfile, and can be used to generate the project specific exception information [11].

### 4.1. Local exception information

The tool traverses the translation unit with a recursive visitation strategy to identify every callable function or callable language structure. These include free functions, class methods, constructors (referred to as callables). In the first stage, the tool takes advantage of the AST to traverse the TU, using a recursive descent in the declaration structure of the TU. Clang's AST representation uses nested declarations (called Declaration Contexts) to represent C++ namespaces, class and function bodies. First all the top-level function declarations are detected, and put in a worklist data structure for processing. During the exploration of the functions, the

```
// f is noexcept if g() does not throw or
// can only possibly throw int typed exceptions
int f() {
  try {
    return g();
  } catch (int i) {
    return -i;
  }
}
```

**Figure 4:** Conditional noexcept

```
int main() {
  try { // try-1
    f(); // call-1 contained and directly contained by try-1
    try { // try-2 contained and directly contained by try-1
      if (g()) { // call-2 contained by try-1, directly contained by try-2
          throw 1;
          // throw-1 contained by try-1, try-2, directly contained by try-2
      }
    } catch (...) {
      // catch-1 contained by try-1, try-2, directly contained by try-2
      //...
      throw; // throw-2 contained only by try-1
    }
  } catch (int i) { // catch-2 contained and directly contained by try-1
    throw i+1; // throw-3 contained and directly contained by try-1
  }
}
```

**Figure 5:** Relations of exception-related language elements

Clang-internal identifier (Unified Symbol Resolution, USR) is used to match to disambiguate functions. USR can be used to identify functions across TUs.

For each function, it is noted whether there is a definition for the function in the current TU. If the function has a body, then it is traversed, and all `try`, `throw` expressions and `catch` statements are extracted. For each of these, the source location and the type information of the related subexpressions (e.g.: what type of exceptions a try block handles through its catch statements) is collected. Additionally, the containment relations are also extracted from the source, meaning there is an ordering relation between each try expression, between any catch or throw expression and a try block.

A throw expression or a catch statement is also contained within one or more try blocks. During the analysis of a callable's body, call- and constructor expressions are also detected, and they are also added to worklist of functions to be explored. This means that analysis will take declaration of non-top level callables also into account. The relation of expressions that transfers control to another lexical scope (call expression) and the try blocks are also computed by this stage of the analysis. For call expressions, a placeholder entry is generated, which can be used to mark a function noexcept on the condition that a particular function does not throw exceptions (Listing 4). The containment and direct containment relations are computed for pairs of try blocks, for the throw expressions and the try blocks and the call expressions and the try blocks (Listing 5). The result of this stage of the analysis is the intraprocedural exception-related information of a TU's callables, and this computation can be executed on every single translation unit independently.

## 4.2. Algorithm for computing possibly thrown exceptions

After computing the intraprocedural exception-related information of a TU's callables, it can be used to create the interprocedural relations of the same callables. The computation uses fixed-point algorithm to generate the intraprocedural results. At the beginning of the algorithm, all callables are added to a worklist. The computation ends, when there are no more items in the worklist. The worklist items are processed in an arbitrary order, but the logic of this step is similar to exception pruning done by the LLVM optimizer pass, so a preorder visitation of the inverse callgraph would be most fitting [5].

However, Clang's current callgraph implementation for C++ code which also contains exception-handling nodes is currently insufficient for the current tool. And this visitation strategy would only be usable if there were no cycles in the inverse callgraph, which is the same property in callgraph, which in turn indicates recursive calls. To handle recursion, strongly connected components (SSC) could be used as a partitioning technique but the current problem is solvable with the fixed-point iteration and a union operator on the closed set all possible exception types.

The inverse of the call relation is used to get the possible callers of a callable. In each iteration, a callable is taken from the worklist. The current callable is then used to possibly extend the exception information of all of its predecessors. If a predecessor information is changed, the predecessor is put back in the worklist. The update of the predecessor is done by identifying the callsite's containment relations with other exception-related language structures. If the newly calculated set of the possible exceptions changes the exception specification of the callable that contains the callsite, that means that callsite's callable can in turn affect its callers, therefore it has to put back into the worklist.

## 4.3. Propagating exceptions for project-specific results

After the interprocedural stage of the algorithm finishes, we have a refined set of relations, but if the definition of a function is inside another TU, the analysis can further be enhanced by taking those into account. There exists a Clang tool which can generate a mapping from function declarations and the source files they are defined in. This information and the build system's information encoded in the compilation database can be used to generate exception information that is project-specific. Library code cannot always assume that a function will not throw, because that would violate generality. Using project-specific build information, however, the details of the called functions become available in case of static linkage.

For project-specific exception information, all the compiler arguments that are necessary to build a specific binary are collected. Knowing all the related compiler arguments, the interprocedural algorithm is run for each TU, then the exception information is extracted from them. It is assumed that there is no definition for the same callable declaration in two different TUs, as that would mean there is a One Definition Rule (ODR) violation. If there is an ODR violation, the algorithm can detect it at this point. Then the callables are all processed with the same worklist algorithm as in the interprocedural stage. This method gives a third level of refinement for the exception information, and can then finally be used to detect callables that have less strict exception specification than it is indicated by the algorithm.

## 5. Execution of the exception-scan algorithm

Our tool takes only one TU of the project into consideration. To have a project-wide result, we need to execute the tool with every TU of the project. To achieve this, we develop a script that ensures the traversal of the projects' build description. To run the exception-scan algorithm to detect whether any of the functions in the source files can throw any exception, we need to have our own project directory located on our machine.

We also can execute the algorithm in freshly cloned repository from git remote server. When we have the aforementioned project the following steps must be performed:

- Under Unix OS a terminal window, on Microsoft Windows machine a PowerShell window needs to be opened. The current working directory must be the project's root folder where we want to run the script.
- Check for file named `CMakeLists.txt`. In case of it does not exist, unfortunately, we are not able to run the detection algorithm because this file defines how the cmake command needs to be executed.
- Create a folder inside the project's root directory where the cmake build command will be executed and the build files will be located. Any name can be given to this folder. It is a good practice to give some unique name because there are projects on git which already have a build folder. In this particular case, we will name our folder as `pane-build`.
- Make sure that the current working directory is the newly created `pane-build` folder then from terminal window run the command in Figure 6:

```
$ cmake .. -D CMAKE_EXPORT_COMPILE_COMMANDS=ON
```

**Figure 6:** CMake invocation for generating compilation database

This will create the necessary files to run the exception-scan script. There will be a file called `compile_commands.json` in the `pane-build` folder after the previous command successfully finished. This file defines how to execute the source files, with which command for example and where the source files are located within the file system. Its content is a list which holds JSON (JavaScript-Object-Notation) objects, as many source files are in the project. Every one of these objects has three keys:

  - `directory`: path to the pane-build folder
  - `command`: the execution command, how to execute the source file
  - `file`: path to the source file
- The next step is to copy this `compile_commands.json` file from pane-folder into the root directory.
- The last step is to run the algorithm for the files we want to scan. So we need to open the previously mentioned `compile_commands.json` and copy the files path from it then run the command in Figure 7 from terminal.

```
$ <path to exception-scan> <copied path to the source file>
```

**Figure 7:** Invoking the exception-scan tool

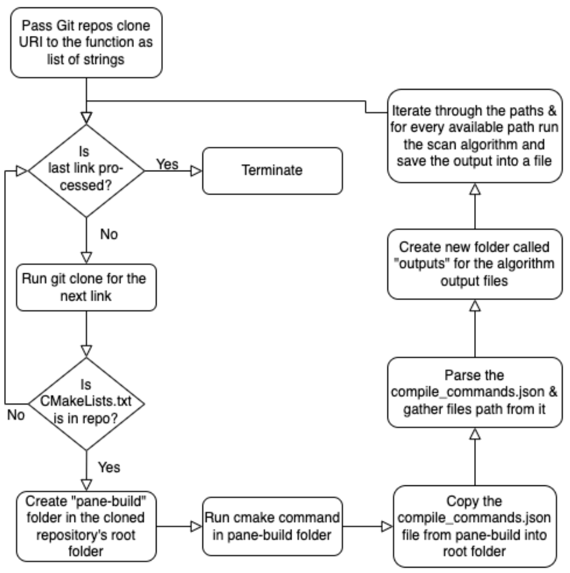
As it seems, there are lot of steps need to be done to be able to run the exception-scan algorithm. In case of the project has a lot of files, we must execute the algorithm for every single file which can be very time consuming. Because of this, to facilitate a more convenient usage we made a script which will do all the steps above and run the exception-scan algorithm for every available source file listed in the `compile_commands.json`.

The script must be placed next to the project folder. It has a class named `ExceptionScan` which can be instantiated without any constructor parameter. On the instance object, we have to call the `new_repo_scan()` method which takes a list of strings as input parameter. It will iterate on list items and execute the previously described steps. If there is no repository link in the list or in case the last link is consumed, the script will terminate. It also performs an extra step because it creates a folder in the project's root directory called `outputs` and it will place all the output txt files there. The operation of the algorithm can be seen on Figure 8.

The problem with previously defined script is that in every case we have to clone a new repository from remote server even if it was already cloned before. To solve this problem, we also implemented another method in `ExceptionScan` class. It is called `run_exception_scan()` which flow-chart diagram can be seen on Figure 9. It takes one optional input parameter. If it is not provided, it will default to none. In case of the user does not provide any input parameter, the method will detect all folders in the main folder where the script is located and it will iterate over on every detected folder and execute the necessary steps to run the exception-scan algorithm. If the algorithm was already executed before in the repository, it will delete the old files and will regenerate them. If the user decides to pass a parameter to the function, it must be a list of strings where every list item must refer to a folder name for which the user wants to run the algorithm. In this case, a check will be made to make sure the given folder if exists. If the last folder is scanned, the algorithm will terminate.

## 6. Parsing the output of the exception-scan tool

After the exception-scan algorithm generated the output files, we need to parse them to extract the information that is important to us. We currently care about function calls and whether



Pass Git repos clone URI to the function as list of strings
Is last link processed?
Yes
Terminate
No
Run git clone for the next link
Is CMakeLists.txt is in repo?
No
Yes
Create "pane-build" folder in the cloned repository's root folder
Run cmake command in pane-build folder
Copy the compile_commands.json file from pane-build into root folder
Parse the compile_commands.json & gather files path from it
Create new folder called "outputs" for the algorithm output files
Iterate through the paths & for every available path run the scan algorithm and save the output into a file

**Figure 8:** The flow-chart diagram of the `new_repo_scan()` method

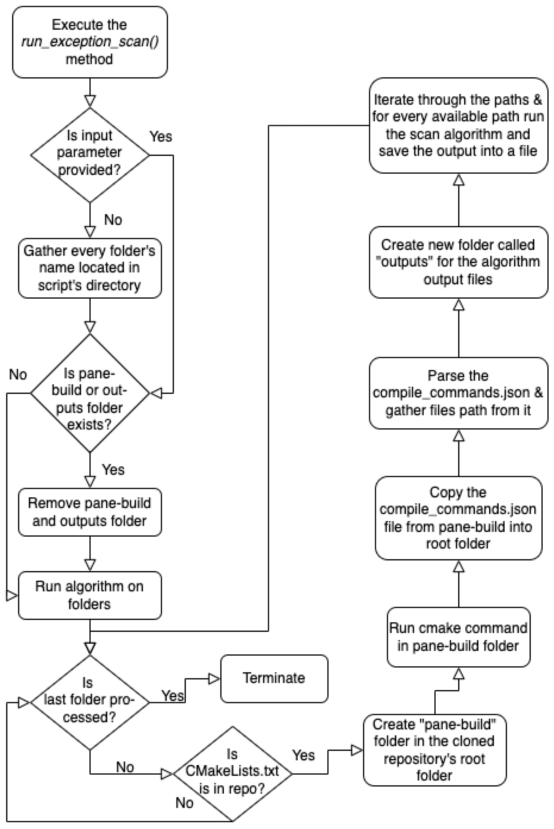
they can throw any exceptions. This parser is written with the help of Python programming language and its output is a Python list which holds dictionaries. They are very similar to JavaScript's arrays and objects and for this reason they can be easily assigned to a variable inside the Monaco editor which is Microsoft's open-source browser based code editor. The implementation details can be found in Section 7.

A part of the generated file which is relevant to us can be seen in Figure 10. The highlights of this figure are the `calls`, `tries`, `throws` and `catches` keys.

- `calls` – refer to a function call and give information about the call's location within the file. The necessary part is after the last slash: `tinyxml2.cpp:1403:12, col:34`. We can interpret this part in the following way: in `tinyxml2.cpp` in row `1403` is a function call and to be more precise it starts from column `12` and ends at column not `34` but `35`. This is because column 34 is before the closing bracket and 35 is after the closing bracket which indicates the function call.
- `tries, throws, catches` - these parts indicate whether there is a try-catch block in the function and if yes, then it can throw any exception and what type of exception.

To get the required information from the files, we have developed a script which parses the files line-by-line and with the help of a regular expression filters out only those lines which hold a function call and transforms them in a way which can be interpreted by Monaco editor. So from the previously mentioned part `tinyxml2.cpp:1403:12, col:34` it will create the following object: `{ row: 1403 }`.

The final output of the parsed files will be a list of objects where every object will have the same shape with row key as shown above. Currently, we do not care about the start and end columns, because we want to highlight the full row where the function was called. The output



Execute the *run_exception_scan()* method
Is input parameter provided?
Yes
No
Gather every folder's name located in script's directory
Is pane-build or out-puts folder exists?
No
Yes
Remove pane-build and outputs folder
Run algorithm on folders
Is last folder processed?
Yes
Terminate
No
Is CMakeLists.txt is in repo?
Yes
No
Create "pane-build" folder in the cloned repository's root folder
Run cmake command in pane-build folder
Copy the compile_commands.json file from pane-build into root folder
Parse the compile_commands.json & gather files path from it
Create new folder called "outputs" for the algorithm output files
Iterate through the paths & for every available path run the scan algorithm and save the output into a file

**Figure 9:** The flow-chart diagram of the `run_exception_scan() method`

```
c:@N@tinyxml2@S@XMLUnknown@F@Accept#*$@N@tinyxml2@S@XMLVisitor#1:
calls:
 c:@N@tinyxml2@S@XMLVisitor@F@Visit#&1$@N@tinyxml2@S@XMLUnknown#@
   </Users/user/exc-scan/tinyxml2/tinyxml2.cpp:1403:12, col:34>
tries:
throws:
catches:
```

**Figure 10:** Output of the exception-scan tool

file's extension will be a `.json` for compatibility reasons between Python and JavaScript and because it has a user-friendly data visualization and it is easily comprehensible.

## 7. Visualization of the results

Our goal was not only to extract the necessary output from the raw source files, but also wanted to visualize it with highlighting the rows in the source files where function calls were made.

```
const editor = monaco.editor.create(
    document.getElementById(‘container’), {
        value: sourceOrigin,
        language: ‘cpp’
});
```

**Figure 11:** Creating Monaco Editor instance

```
let decoratorList = [{
    range: new monaco.Range(...row),
    options: {
        isWholeLine: true,
        className: ‘functionCall’
    }
}];
```

**Figure 12:** Creating decorators in Monaco Editor

```
    decorations = editor.deltaDecorations([], decoratorList);
```

**Figure 13:** Applying decorators in Monaco Editor

We chose the Monaco online editor to do the visualization job, because it fully meets the needs we were looking for like add custom styles, code-lenses, jump-functions doing all of this right in the editor itself.

Monaco editor is maintained by Microsoft and it is available worldwide for free. It has a playground with interactive examples and provides wide access to the editor instance and supports features like colorizing the editor line-by-line, adding custom hover messages using JavaScript, CSS, and HTML for the corresponding parts [12].

As mentioned earlier, the output of the parser will be a list of objects where every object holds a key-value pair, where the name of the key is `row` and the corresponding value is the function call’s row number in the source file.

Monaco editor defines a so called editor object with the help of which we can interact with the editor itself. To do this, the `deltaDecorator()` method must be called on the editor instance. It takes two input parameters. The first one is a list with the old decorators and the second one is a list too, but with the new decorators which will be applied. In the latter, we can refer to a CSS class name as a string and add our own style to it. First we need to instantiate the editor object, then provide the appropriate programming language as a string what we want to use as seen in Figure 11.

The `sourceOrigin` global variable holds the raw C/C++ code itself as a string. Its value comes from the editor widget. Now we are able to create the decorators (see Figure 12).

The `functionCall` string refers to a CSS class name, where we defined how we want to display those lines where it is applied.

When the decorators are created, we can assign them to the editor with the help of the previously mentioned `deltaDecorations()` method as seen in Figure 13.

The tools identifies source locations where impactful changes can be made. The Monaco Editor visualizations help the developer to identify and navigate these points of interests and also provide justifications for the suggested changes in the form of code-lens as a metainformation of the source.

## 8. Conclusion and Future Work

Exception handling is a beneficial language element in C++. However, the compiler cannot handle many serious bits of exception-related information in the source code. Moreover, the main idea of exception specification has changed. In this paper, we analysed what are the major problems and defined a comprehensive framework for visualization and assistance for improved exception handling. Our framework takes advantage of the Clang compiler and the Microsoft Monaco editor for improved comprehension of the exception handling. Another important feature of the framework is the missing noexcept indication.

We have many future work items. For instance, to make our algorithm parts more usable for every user, we want to create a tool that will automate every step which we now have to do manually. This tool will have a graphical user interface from where the user can select one or more directories for he/she wants to execute the exception-scan algorithm, or clone a whole new repository from git. When the algorithm has finished its execution, the tool will automatically show a list with the parsed source files in their corresponding folders. The user select any file, then the appropriate source file's content will be loaded into the editor with the corresponding output txt file and it will highlight the rows where there are function calls, try-catch blocks or throw exceptions.

## References

[1] C. de Dinechin, C++ exception handling, IEEE Concurrency 8 (2000) 72–79. doi:10.1109/4434.895109.

[2] B. Stroustrup, The C++ Programming Language, 4th ed., Addison-Wesley Professional, 2013.

[3] R. Torres, T. Ludwig, J. M. Kunkel, M. F. Dolz, Comparison of Clang abstract syntax trees using string kernels, in: 2018 International Conference on High Performance Computing & Simulation (HPCS), 2018, pp. 106–113. doi:10.1109/HPCS.2018.00032.

[4] The Clang Team, Diagnostic flags in Clang, https://clang.llvm.org/docs/DiagnosticsReference.html, 2022. [Online; accessed 19-June-2022].

[5] LLVM Project, LLVM's analysis and transform passes, https://llvm.org/docs/Passes.html, 2022. [Online; accessed 20-June-2022].

[6] The Clang Team, Clang tidy's modernize-use-noexcept, https://clang.llvm.org/extra/clang-tidy/checks/modernize/use-noexcept.html, 2022. [Online; accessed 20-June-2022].

[7] The Clang Team, Clang tidy's bugprone-exception-escape, https://clang.llvm.org/extra/clang-tidy/checks/bugprone/exception-escape.html, 2022. [Online; accessed 20-June-2022].

[8] C. F. Schaefer, G. N. Bundy, Static analysis of exception handling in Ada, Software: Practice and Experience 23 (1993) 1157–1174. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.438023107. doi:https://doi.org/10.1002/spe.4380231007. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380231007.

[9] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, A. Gupta, Interprocedural exception analysis for C++, in: M. Mezini (Ed.), ECOOP 2011 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 583–608.

[10] Herb Sutter, Exception safety and exception specifications: Are they worth it?, http://www.gotw.ca/gotw/082.htm, 2009. [Online; accessed 20-June-2022].

[11] G. Horváth, P. Szécsi, Z. Gera, D. Krupp, N. Pataki, Challenges of implementing cross translation unit analysis in Clang Static Analyzer, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2018, pp. 171–176. doi:10.1109/SCAM.2018.00027.

[12] E. Fülöp, A. Gyén, N. Pataki, Code comprehension for read-copy-update synchronization contexts in C code, in: S. Bourennane, P. Kubicek (Eds.), Geoinformatics and Data Analysis, Springer International Publishing, Cham, 2022, pp. 187–200.