



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

## Bachelor Thesis

# Static Code Analysis for HPC Use Cases

submitted by

Frank Röder

Faculty of Mathematics, Informatics and Natural Sciences  
Department of Informatics  
Research Group Scientific Computing

course: Informatics  
student number: 6526113  
email: 3roeder@informatik.uni-hamburg.de

first reviewer: Dr. Michael Kuhn  
second reviewer: Alexander Droste

advisors: Dr. Michael Kuhn & Alexander Droste

Hamburg, 26-07-2017

# Abstract

The major objective of this thesis is to approach the procedure of getting into compiler-based checks. The focus are high-performance computing use cases. Especially the Message-Passing-Interface (MPI) is used to execute parallel tasks via inter-process communication, including parallel reading and writing of files are taken into account. A motivation states why it is remarkable to use static analysis. Following this, techniques and tools to improve software development with static analysis are introduced. Nowadays parallel software has large code bases. With rising complexity the possibility of generating bugs is undeniable. Tools to reduce the error-proneness are important factors of efficiency. The infrastructure of LLVM as well as the Clang Static Analyzer (CSA) are introduced to understand static analysis and how to capture information of the relevant compile phases. Based on this, the utility of an existing check is explained. Problems exposing at runtime are observed through code simulation in the frontend named symbolic execution. In what follows, the comprehension is transferred to the use case of purpose. Common mistakes to overlook like issues with readability and bad code styles are checked through analysis of the abstract-syntax-tree. For this intention the LLVM tool Clang-Tidy has been extended with new checks.

The checks regarding symbolic execution involve MPI-IO related double closes and operations concerning file access. The routines to find these bugs have been added to the CSA. This thesis makes use of the already existing structure named MPI-Checker, which provides the handling of MPI.

As a summary the benefits of working on checks to detecting serious bugs are mentioned.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	Nowadays Software in HPC . . . . .	6
1.3	Foundation . . . . .	8
1.4	Overview . . . . .	9
<b>2</b>	<b>Components of LLVM</b>	<b>10</b>
2.1	LLVM . . . . .	10
2.1.1	Structures of LLVM . . . . .	13
2.1.2	Setup of LLVM . . . . .	15
2.2	Clang . . . . .	17
2.3	The Clang Abstract Syntax Tree . . . . .	21
2.3.1	Another AST Representation . . . . .	25
2.3.2	AST Matchers . . . . .	26
<b>3</b>	<b>MPI - Message Passing Interface</b>	<b>28</b>
3.1	MPI Communication Example . . . . .	29
3.2	MPI-IO Example . . . . .	30
<b>4</b>	<b>Clang-Tidy</b>	<b>32</b>
4.1	Example of Clang-Tidy . . . . .	33
4.2	Invoke Clang-Tidy Checks . . . . .	35
<b>5</b>	<b>Clang Static Analyzer</b>	<b>36</b>
5.1	Introduction . . . . .	36
5.2	Path-Sensitive . . . . .	37
5.3	Invoke The Analyzer . . . . .	38
5.3.1	About the Checks . . . . .	41
5.3.2	scan-build with MPI . . . . .	43
<b>6</b>	<b>MPI-Checker</b>	<b>45</b>
6.1	Files of the Implementation . . . . .	45
6.2	New Features . . . . .	46
6.2.1	AST Analysis . . . . .	46
6.2.2	Path-Sensitive Implementation . . . . .	49
<b>7</b>	<b>Related Work</b>	<b>50</b>

<b>8 Conclusion</b>	<b>51</b>
<b>Bibliography</b>	<b>52</b>
<b>Appendices</b>	<b>57</b>
<b>List of Figures</b>	<b>63</b>
<b>List of Listings</b>	<b>64</b>
<b>List of Tables</b>	<b>66</b>

# 1 Introduction

## 1.1 Motivation

The importance of high-performance computing (HPC) is indisputable. Sectors of businesses, science and military uses HPC to simulate behavior of settings, even before they start with the real case. Geo and climate research needs high calculation power for complex computation while aiming for great throughput. This is necessary to make precise predictions and to draw conclusions from big amounts of collected data. For this reason it is not only necessary to concentrate on computing power but also on the performance of the in and output. Most of the time there are mathematical models to state insights with computer simulations. Moreover, the means of machine learning are realized with the help of HPC. Analyzing and learning from large amounts of data is in great demand.

There is a strong need for best maintainable, efficient, and readable software. Trade-offs have to harmonize these characteristics. Therefore, a lot of tools can support the engineers in software architectures by checking for issues that are not obvious to capture. Debuggers, which are specialized to handle parallel programs with MPI, are just a couple commonly used.

However, debugging is not limited to these specific debuggers. GDB, the debugger of GNU [Pro], can debug by manually control multiple sessions. Each session corresponds a process of the parallel program. In general, most single-process debuggers not particular created for parallel programs can debug parallel by starting a session for every process involved. Allinea's DDT [All] is specialized in parallel debugging and can handle a lot processes while visualizing communication and also detecting typical bugs of parallel programs. Another one endorsed for HPC is TotalView by RogueWave [Wav]. TotalView also debugs multiple processes and enables control over the program execution. Both DDT and TotalView have similar demands for debugging of parallel applications and are recommendations by the official Open MPI homepages FAQ [OM]. TotalView restricts the user to have all executables on the local machine, where the debugger is started from. As all of these tool examples interact with the running program, faulty states are not easy to reproduce or to keep track of thousands or even millions of processes involved. While the mentioned multi-process debuggers are popular to use, other tools like STAT, the Stack Trace Analysis tool, grant insight by merging stack traces from parallel applications visualized as graph. It is also used to locate groups of processes and can pass this information to DDT or TotalView for more in depth analysis [LLN17]. This is a big advantage because these parallel debuggers are limited to an amount of processes

and STAT can capture even more. Static code analyses can observe errors even before they occur in the running program. Based on the abstract-syntax-tree representation of the source, static analysis is performed to find violations, wrong use of interfaces and bad code style. With symbolic execution respectively path-sensitive analysis, a graph of program points is used to detect failures, which occur at runtime by using symbols as assignments to variables. The abstract-syntax-tree, or AST in short, is explained in section 2.3. Simulating the program run while the frontend is processing the source is an advantage of the CSA.

Since it is possible to have checkers for a limited amount of bugs, which is also augmented by used interfaces, there is a gap supplying checks needed for up-to-date programming tools and libraries.

## 1.2 Nowadays Software in HPC

Due to the size of benchmarks and libraries like IOR (Interleaved Or Random) and HDF5 (Hierarchical Data Format 5), the reason for static analysis is motivated once again. The following tables were emitted by the tool Count Lines of Code [Dan]. The command `cloc` is called for the whole source folder. The resulting lines for IOR are shown in table 1.1 (a) and the one for HDF5 in table 1.1 (b).

Language	files	code	Language	files	code
C	12	4 870	C	670	467 912
C/C++ Header	6	526	C/C++ Header	270	31 337
Other	7	1 733	Other	651	167 412
total	25	7 129	total	1 591	666 661

(a) Lines of Code in IOR-2.10.3                      (b) Lines of Code in hdf5-1.10.1

Table 1.1: CLOC Results

### Statistics about Bugs

For these huge applications it might not be feasible to find bugs by manual testing. However, the frequent usage of them reveal bugs, which are reported by users. These issues might get fixed with a version upgrade.

### HDF5 Bugs Statistics

The homepage of HDF5 offers no information about located bugs in general. Because this data is kept intern, there are no reports. A contact with their HDF Helpdesk brought

the knowledge of table 1.2. With assumption, that only C/C++ include parallel and MPI related manner, the ratio of 478 359 lines of code compared to 410 issues is not to neglect.

The issues captured by their statistics exclude a specific amount of undetected ones. The quantity of solved bugs in relation to open cases denotes how important it is to use tools supporting the creation of program code.

Year	Total Number of Issues	Parallel (closed)	MPI (closed)
2016	410	24(8)	16(3)
2015	629	23(9)	11(6)

Table 1.2: Total Number of HDF5 Issues

## IOR

There were no clear results about bugs found in IOR. Different repositories had different amounts of commits and the source of LLNL (Lawrence Livermore National Laboratory) is not actively developed anymore. This information is provided from direct contact to their support. The biggest amount of commits were found at [LAN] with latest commit in April 2017.

## Static Analysis in Present Use Cases

With the rising importance of nowadays software embedded into a lot of different areas, static analysis is a relevant tool to find vulnerable and safety-critical locations. An example of medical software that is very difficult to test, shows an infusion pumps software under real conditions before it gets connected to the hardware. Since it is very critical to have bugs in the program code, especially in this case, the so-called technique, static code analysis, is used next to manual code reviews, exhaustive testing and simulating the program on a computer to verify the correctness [Adm16].

Software of nuclear power plants imposes the same requirements in their reactor protection system. In general there are technical assignments for computer systems to accomplish, like the standard of the International Electrotechnical Commission named IEC 60880:2006 for nuclear power plants of 2006 [IEC06].

## 1.3 Foundation

As this thesis is based on analysis of representations generated at compile time, a brief initiation elucidates the structure.

With lexical analysis a compiler is scanning the source code as sequence of characters. After this step a sequence of tokens is passed to the parser, which analyzes the syntax and translates these into the AST. As last step of the frontend, the semantic analysis takes place to check for example types of variables. Thereafter a so called intermediate form will be generated, which is also the last step of the frontend phase. The optimizer, or also called middleend, can do additional processing before the final step of generating the executable is carried out. This takes place in the backend where the machine code is generated. The named frontend steps are visualized in figure 1.1.



Figure 1.1: Frontend Process

Certainly, the parts of a compiler can be built self-generated, but as there is limited time for this thesis and the section of nowadays computer science has plenty of really good open source implements with beneficial licensing, the focus is set on implementing new features and understanding given compiler infrastructures.

To name a few C/C++ compilers used in HPC use cases that might provide the interface to get access to the different parts of compilation, Clang is a very popular sample with a well documented and reusable API. The licensing is an advantage because it allows to use, copy and change the tools, while it is necessary to keep the copyrights in the files. This licensing is called BSD, the Berkeley Software Distribution. For a long time it was not possible for GCC to just supply the frontend functionalities to the programmer. GCC is distributed under the GPL license allowing the usage and modification but disallows the changing of the license named Copyleft.

A further frequently used compiler in HPC is the Intel C++ Compiler or ICC [Int], which offers a cluster edition. Special are the Intel performance libraries included to optimize the performance for their own processors, as well as an OpenMP support. Intel also integrated tools for analysis and debugging into their compiler package.

Another common used compiler is part of the collection of tools named PGI Compilers & Tools by the Portland Group [Gro]. Included are optimizers, debuggers, profilers and an IDE. At its early stages it was specialized for high-performance Fortran, a language extension of Fortran 90 made for parallel tasks. In 2013 NVIDIA Corporation bought PGI and integrated it into their system use.



The choice of frontend processing functionality is determined by the programming language. In the case of HPC it is very common to use C/C++ and also languages like Fortran. Debugging applications written with MPI for a multitude of processes can be a hard task. Susceptibilities of making mistakes should be minimized by supporting tools. In an optimal setting every bug is reported before the batch of computing is started. While the compiler outputs errors and warnings at compile time based on the frontend, other bugs might be unnoticed until the calculation is already in progress. It is important to remember that various compilers find and generate different dispatches about mistakes found.

Table 1.3 shows the key features that needs to be mentioned about frequently used compilers in HPC in short.

Compiler	License	Features
GCC	GPL	mature compiler with a lot of languages supported
Clang	BSD	good documented API, re-useable
ICC	license server	high optimization for Intel CPUs, OpenMP support
pgcc	NVIDIA End-User License	big package with compilers, profilers, debuggers and IDE supporting OpenMP

Table 1.3: List of Compilers

## 1.4 Overview

This thesis is structured as follows:

Chapter 2 observes the main facilities to handle different points of the compilation process. The setup and also the establishment of the tools are considered as well as the basic structures to understand Clang and the surroundings of static analysis are explained.

Chapter 3 introduces MPI with examples of commonly used functionalities.

Chapter 4 covers basic usage and an example of Clang-Tidy.

In chapter 5 path-sensitive analysis regarding the Clang Static Analyzer is introduced. With chapter 6 the implementation of the added checks are explained. Considered files in the LLVM project to change are addressed.

Chapter 7 covers different approaches and techniques of debugging. Chapter 8 contains the summary with sight on future improvements of the MPI-Checker.

## 2 Components of LLVM

As the foundation in chapter one already stated, building the lexical analysis, parser and other involved components would exceed the amount of time in addition to the creation of a checker. The implementation of the working checks are based on the infrastructure of LLVM and the Clang frontend.

Clang provides the AST classes and features to generate and observe these information. To understand the tools and contexts of LLVM, this chapter exhibits an introduction.

### 2.1 LLVM

LLVM is a collection of compiler and tool-chain technologies used to design compiler front- and backends, as well as optimizers between both of these phases [Lat12]. This umbrella project is greatly known for its Clang compiler, which has a lot of advantages compared to GCC (GNU Compiler Collection) [Tea17a]. With its beginning in year 2000 at the University of Illinois it was developed with the purpose to have re-usable libraries with a well documented interface. The repository of LLVM consists of two main and one optional part. The first one is the LLVM suite containing all libraries, tools and header files. Also included are an assembler, a disassembler, a bitcode analyzer and a bitcode optimizer. Further implied are regression tests used to verify Clang and LLVM tools. The test suite, which is the optional part, is used to validate the correctness of functionality and efficiency. This infrastructure is also directly applied to approve the implementation of Clang-Tidy, while especially lit, the LLVM Integrated Tester, is used. The second big part is Clang, which can compile C/C++ and Objective-C/C++ code and also transfer it into different representations with the possibility to be processed by LLVMs optimizer and backend facilities [LLV17k]. In general the core of LLVM is the intermediate representation (IR). IR is an internal data representation describing the already processed source code by the frontend. It is quite similar to assembly language and located in the middle layer of the compiler. LLVM takes the general IR and represents it in an optimized form. The optimizer tries to perform different transformations on the representation generated from the AST. The aspiration is to reduce the runtime of the program code through removal of redundant computations. At last the code generator, also named backend, maps the code fitting to the instruction set of a given architecture. As there are various ways in

LLVM Logo [LLV17m]



LLVM to process a language in the frontend and different options of generating the code for the backend are possible, the compilation is very adaptive. This concept is called the Three-Phase-Compiler, illustrated in figure 2.1, in which every part is rather independent and therefor maintenance and writing of one component is mainly limited to one phase.

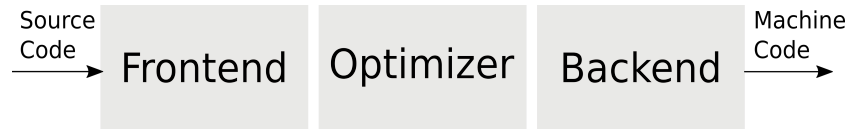


Figure 2.1: Three-Phase-Compiler LLVM [Lat12]

The problem of supporting  $M$  source languages and  $N$  targets would require  $M \cdot N$  compilers to build. Figure 2.2 shows the advantage of the concept LLVM is supporting. This retarget-ability makes the tool-chain usable for many programmers by supporting an amount of languages for different target machines [Lat12].

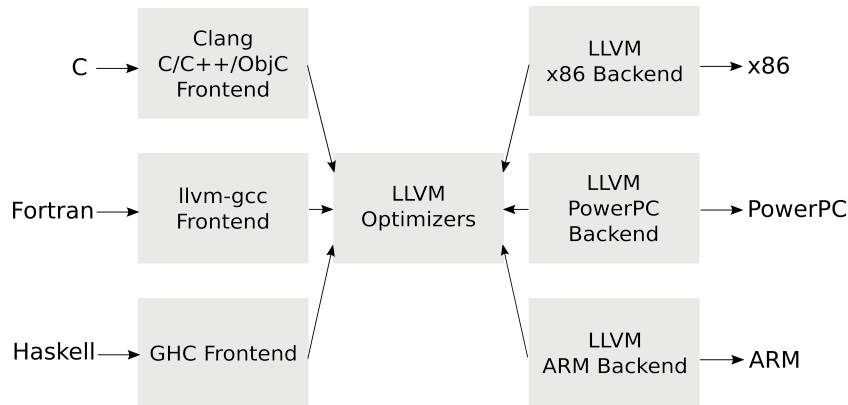


Figure 2.2: Retarget-Ability LLVM [Lat12]

Further, more introductions to write new languages and compilers are provided by LLVM. A very well known result of the good infrastructure is the OpenCL implementation. It is a framework to share the workload of tasks between different processing devices using a C-like language. Especially the graphics processing unit is a major part of optimizing computations and transfer a program object containing the compute kernel to.

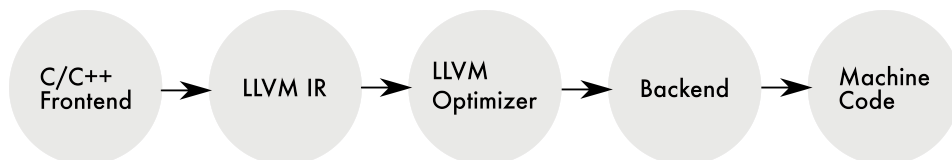


Figure 2.3: LLVM Compiler Parts

The frontend parses and checks the source for errors. Following this, an AST is created and converted into IR. Afterwards it is possible to process it with a set of optimizers and

analyses, increasing the overall performance and verify the correctness. At the end the code generator creates the native machine code. LLVM has a lot of tools to improve the parts of the three phases. Programmers specialized for frontend can completely focus on their part without taken other parts of the compilation into account. This was and still is not smoothly possible with GCC because it uses global data structures, which are creating dependencies between the front- and the backend. This structs contain for example debugging information.

*“The compiler frontend and backend interact with each other using callback functions called language hooks. All hooks are included into a global variable struct ...”* [Wik13]

The following listing gives an overview of LLVM in short [LLV17n]:

- LLVM Core
  - source- and target-independent optimizers
  - based on the LLVM intermediate representation (LLVM IR)
  - facilities to invent an own language and to use the LLVM as code generator and optimizer
- Clang
  - very fast C/C++/Objective-C frontend with low memory use
  - emphasizes expressive diagnostics
  - is a powerful compiler-fronted for different kind of code analysis and other processing
- LLDB for debugging
  - a native debugger as combination of Clang and LLVM qualities
  - uses Clang expression parser and ASTs as well as LLVM disassembler and LLVM JIT
  - can be faster and more efficient in memory usage than GDB
- libc++
  - a project with focus on high-performance implementation of the C++ standard library
  - quick execution, small memory usage, fast compile time
  - better implementation of standard containers
- compiler-rt
  - refers to well tuned low-level generator support routines

- run-time libraries for dynamic tests with tools like
  - \* AddressSanitizer for detecting memory errors
  - \* ThreadSanitizer for detecting data races
  - \* MemorySanitizer detects uninitialized reads
  - \* DataFlowSanitizer for dynamic analysis of the data flow
- OpenMP
  - a sub-project of LLVM to support the implementation in Clang
  - supports offloading computations
  - since Clang 3.8.0 support of OpenMP 3.1
- lld
  - is a linker in replacement for system linkers
  - supports linkers like ELF PE/COFF
  - replaces the GNU linkers

### 2.1.1 Structures of LLVM

Since LLVM is a really large project, this section takes a closer look at some relevant parts for the implementation and understanding of checks. Picking the right data structure is an important decision. All of them are located in the folder `llvm/ADT` and just included in to the program code. Based on the algorithmic context and use case, there are different kinds of containers to choose from. The folder structure of LLVM is explained in section 2.1.2.

Commonly used data structure in checks and also inside of the MPI-Checker, are the `SmallVector` [LLVe] and the `SmallSet` [LLVd]. They are used as containers to store relevant data element for the analysis. As they are really efficient to use, both vector and set are limited to the basic operations. How both are decelerated is shown in listing 2.1 and listing 2.2.

```
1 SmallVector<Type, Size> Name;
```

Listing 2.1: SmallVector

```
1 SmallSet<Type, Size> Name;
```

Listing 2.2: SmallSet

Like their names already tells, they are used for small sizes of elements. Important to mention is that if the set for example gets larger than the specified size, it falls back to a `std::set` to keep an efficient lookup time. The `FoldingSet` [LLVc] is a class implemented as single-linked chained hash table. A given node is stored as bucket and created, if there is no instance already contained in the set. The `FoldingSet` is used to generate a set of unique elements corresponding a program state as singular instance. This is useful in symbolic execution to differentiate nodes and improve the access. The example in listing 2.3 is used to store the `FoldingSetID` value rather than recompute the node every time it is needed. This is used in the case of path-sensitive analysis of the MPI-Checker.

```
1 void Profile(llvm::FoldingSetID &ID) const override {
2     ID.AddInteger(CurrentState);
3 }
```

Listing 2.3: FoldingSet

Furthermore, custom run-time type information like dynamic casts are used to verify a reference or a pointer pointing to an instance of a certain class. These operations are similar to the ones in C++ but have their disadvantages removed like a virtual method table. All of them are contained in the file `llvm/Support/Casting.h` and important to use in static analysis. The `isa<>` operator is used to check if a referenced object is of a certain class. In other words listing 2.4 returns true if the parameter `Val` is an instance of the type argument.

```
1 isa<Type>(Val)
```

Listing 2.4: isa

The instruction `cast` is a check cast shown in listing 2.5. It returns the cast of a parameter argument for a specified type. When the type is incorrect, `null` is returned.

```
1 cast<Instruction>(Val);
```

Listing 2.5: cast

The operation `dyn_cast` of listing 2.6 does the same as `cast` but it also denotes the correctness of a type as well as the success of the cast.

```
1 dyn_cast<Instruction>(Val);
```

Listing 2.6: dyn\_cast

## 2.1.2 Setup of LLVM

For the fastest access, it is possible to download the pre-built binaries directly for one of the listed operating systems at their homepage [LLV17l]. When it is needed to have the latest updates and changes, the repository can be directly copied [LLV17f]. The repository at GitHub is a mirror of Subversion and can be one possibility of building LLVM. For this purpose it is possible to use the version-control git to receive the needed source.

There are different directories of tools, which need to be considered one by one. As the `/llvm` directory may contain all utilities to use LLVM IR and the core libraries, the directory `llvm/tools/clang` contains the CSA, `scan-build` and main parts to work with Clang. Another directory called `llvm/tools/clang/tools` contains Clang-Tidy and much more.

It is usual for large projects, that packages are put into one another and also needed to be updated with git commands by themselves. A general pull in the root directory `/llvm` only works for itself and not for the other mentioned folders. In addition, it is recommended to use `git pull --rebase` because the upstream repository is in Subversion and it is beneficial to avoid a non-linear history of the cloned repository [Gee15].

LLVM can be built for example with `ninja` or `make`. In the first step it is required to create a top level directory to clone the source and create a folder for the build.

A sample of a script is attached to the appendix. To clone the three basic repositories, an example is shown in listing 2.7.

```
1 git clone http://llvm.org/git llvm
2 cd llvm/tools
3 git clone http://llvm.org/git/clang.git
4 cd tools/extra
5 git clone http://llvm.org/git/clang-tools-extra.git
```

Listing 2.7: Git and LLVM

To configure LLVM the command in listing 2.8 is executed in the build folder `build/debug`. While Ninja is specified as build system generator through the CMake option `-G`, it can be replaced with any other common building system as argument. For `make` it is “Unix Makefiles” to generate standard UNIX makefiles and “Xcode” to generate Xcode project files. Many options are available.

The debug option `CMAKE_BUILD_TYPE=DEBUG` activates all assertions, compile libraries unoptimized and generates debugging information. This option is set by default. Other configurations like `Release`, `RelWithDebInfo` and `MinSizeRel` are also available. `Release` disables debugging information and includes optimization. The default value is `-O3` but can be changed through `CMAKE_CXX_FLAGS_RELEASE` to add flags as one would usually do it in a normal compile command. To enable the output of compile commands during generation, `CMAKE_EXPORT_COMPILE_COMMANDS=ON` activates this feature. The

option set through `LLVM_USE_SANITIZER="Address"` defines the sanitizer, which is used to build binaries and tests. In this case it is the **AddressSanitizer**. The last argument hands `cmake` the path to the source containing the `CMakeLists.txt`.

```
1 cmake -G Ninja -DCMAKE_BUILD_TYPE=DEBUG \  
2   -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \  
3   -DLLVM_USE_SANITIZER="Address" \  
4   <path-to-source>
```

Listing 2.8: Build LLVM

In the next step LLVM is built like in listing 2.9 while there are different options available. All rules to build are listed in the file `ninja.rules` inside of `build/debug` or through the command `ninja help`.

```
1 ninja  
2 # executes default  
3  
4 ninja install  
5 # to install header files, libraries, tools and documents  
6  
7 ninja install clang  
8 # installs and rebuild clang  
9  
10 ninja check-all  
11 # runs all LLVM regression tests  
12  
13 ninja check clang-tidy  
14 # clang-tidy tests
```

Listing 2.9: Setup LLVM



## 2.2 Clang

Clang is a central part and with its static analyzer a important unit of the LLVM project. This compiler can confirm the correctness of code through analyzing the AST. Clang is a compiler-frontend created with the purpose to be an improvement of the back then already existing compiler GCC. With intention to create an easy to understand AST, Clang is well-conceived [LLV17a], while the old code-base of GCC aggravate it for new developers to get into compilers [LLV17h]. Due to its nature, Clang is built as an API to make it comprehensible to reuse. This is not the only reason for Google to set it as a main compiler for Google Chrome [pML15], also Apple is using it since Xcode version 3.2 and integrated it directly into the IDE [LLV17b]. As Clang uses a different frontend than GCC, there are pros and cons of choosing one of them. The past showed that GCC has and had some disadvantages concerning information kept for further activities. One of them is trivial constant folding. This potentially disables a lot of IDE typical features like refactoring, auto completion and some more. Since 1990 till today, GCC is not serving the needs for example the IDE of Apple due the lack of various properties. Clang saves even more information for static and dynamic processes of analysis afterwards. Besides static analysis, dynamic analysis is enabled through the flag `-fsanitize=<check>` with different sanitizers to choose from. This flag is just passend to the normal compile command `clang`. The result of storing additional information with Clang leads to the typical IDE features.

*“Clang does not implicitly simplify code as it parses it like GCC does. Doing so causes many problems for source analysis tools: as one simple example, if you write “x-x” in your source code, the GCC AST contains “0”, with no mention of ‘x’. This is extremely bad for a refactoring tool that wants to rename ‘x’.”*[LLV17g]

For dynamic analysis Clang uses for example its MemorySanitizers, which checks the code at runtime to find present errors like reading of uninitialized memory [Tea17b] or the AddressSanitizer to find flaws like `use-after-free`, `use-after-return`, `double-free` and some more [Tea17c].

Because it makes use of Clangs frontend, the already mentioned debugger LLDB by LLVM performs faster in most cases than GNUs GDB. The Clang compiling can be much faster while straining less memory [LLV17r].

For all that, there are languages like Java, Fortran, Ada supported by GCC, whichever are not even planed to be provided by Clang. In addition, there are also more targets available like through LLVM itself. While examining different benchmarks of both compilers, it really depends on the case and environment in which they are tested [Pho17]. LLVM had a project called dragonegg, which implemented parts of the GCC frontend into the LLVM environment until 2015. Today many GCC extensions are supported inside of LLVM. As a current example `llvm-gcc`, an implementation of GCC with parts of LLVM, supports a lot of GCC specific features and options [Teaa].

Figure 2.4 shows how similar the compilers are in structure, while figure 2.5 shows the

time taken to compile a small `.c` file. The measurement was repeated 100 times for every command. Afterward the mean was used to visualize the result. `llvm-gcc` performs with 90 ms better than GCC with 140 ms. Clang takes in this case 619 ms to compile the file, which might be different in larger settings.

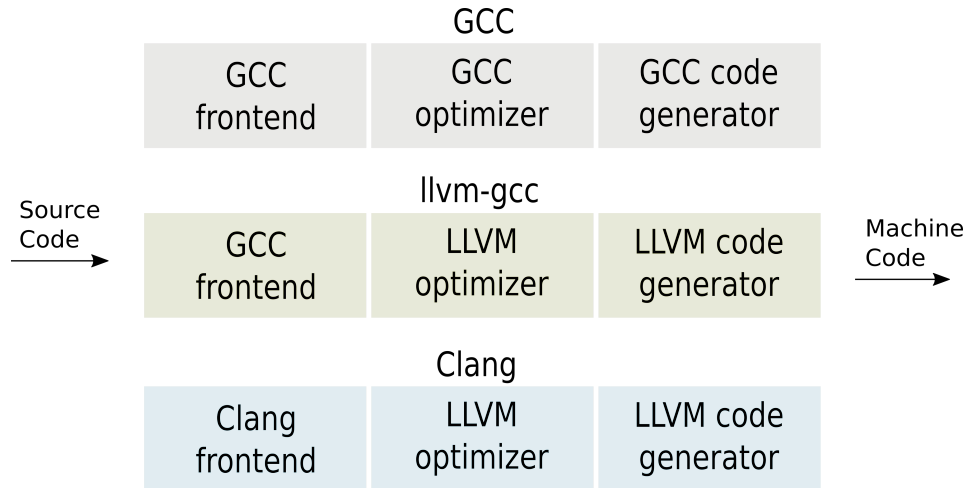


Figure 2.4: Phases of Compilers [DKV17]

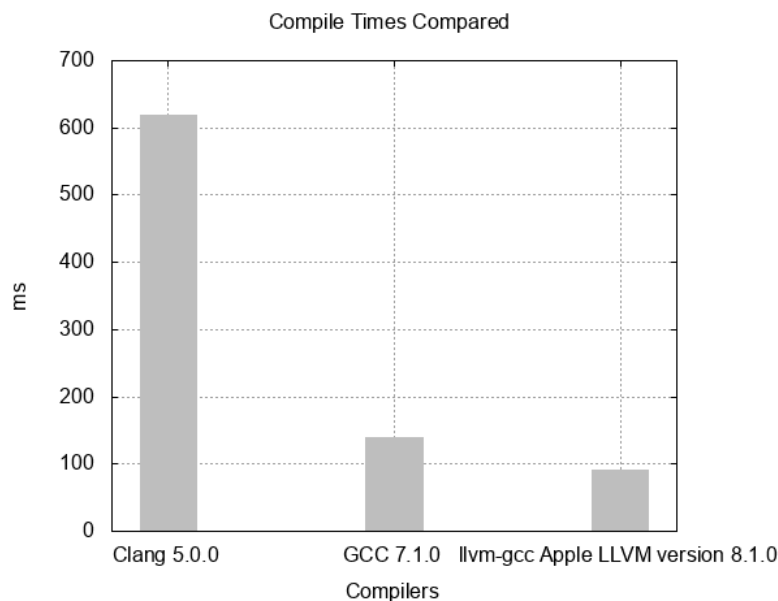


Figure 2.5: Measurement of Compile Times

Since 2009, Clang is used in production also for compiling LLVM and itself [Gre09]. For the intention to work with LLVM it is a big advantage to use its own tool Clang than frontends like the one of GCC. As other compilers, Clang is invoked in the command line like in listing 2.10.

```
1 clang program.c
```

Listing 2.10: Clang Command Line

There are a lot of options, which can manipulate the different phases of the frontend. As already mentioned, dynamic analysis with different sanitizers can be activated and many more. The following table 2.1 shows some sample flags that can be passed to the compile command `clang`. All options are listed when the command `clang -help` is used.

Option	Description
-Xanalyzer <arg>	Arguments for the static analyzer
-Xpreprocessor <arg>	Arguments for the preprocessor
-fsanitize=<check>	Enable runtime checks for different forms of behavior.
-c	Only run preprocess, compile, and assemble steps
-S	Only run preprocess and compilation steps
--analyze	Run the static analyzer
-g	Generate source-level debug information

Table 2.1: Example Flags for Clang

In the following a brief summary of the project Clang:

- Clang Frontend
  - Lexer, Parser, Semantic Analysis
  - Clang AST
  - Static Analyzer with frontend and checkers
- LibClang
  - high level interface to Clang
  - used with Xcode and other languages
  - backwards compatible
  - defines the API for most functionality of Clang
- LibTooling
  - C++ interface to write standalone tools
  - grants full control over the AST
  - build on top of LibTooling tools
    - \* clang-check for syntax checking
    - \* clang-fixit fixing of compile errors
    - \* clang-format for code formatting
    - \* refactoring tools

## 2.3 The Clang Abstract Syntax Tree

To understand the coherence in which the AST is used, the following initiation provides a brief overview. Based on the code of listing 2.11 and the graph of figure 2.6 the components of LLVMs frontend and backend are very flexible in use.

```
1 #include <stdio.h>
2 int main()
3 {           // CompoundStmt { group of statements }
4     int i = 2; // DeclStmt
5     i += i;    // CompoundAssignOperator
6     return i;  // ReturnStmt
7 }
```

Listing 2.11: Clang Command Line

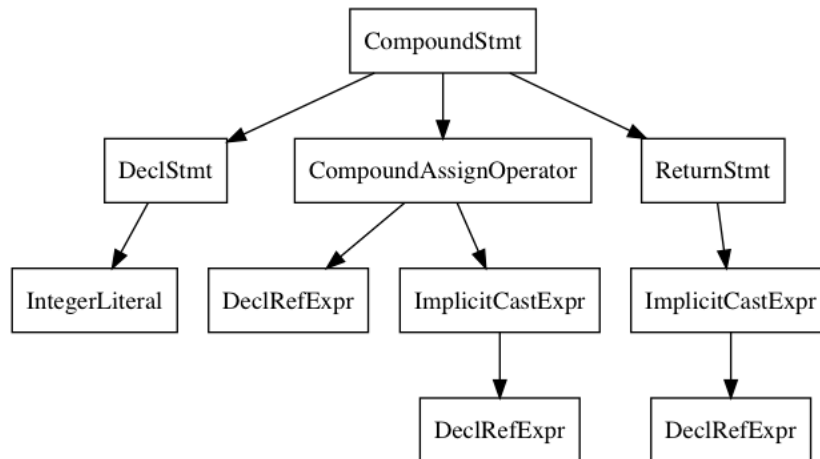


Figure 2.6: AST

As this thesis exhibits the work with LLVM, Clang and Clang tools, it is essential to know the functionality of their AST. Other compilers have a different representation and capabilities to process. Clang represents important information in a form that refactoring and further analysis can be archived. The problem of simplifying code and arithmetics in the phase of parsing is discussed again and illustrated in listing 2.12. The x would not be mentioned with GCC anymore and also can not be considered by procedures like refactoring. This is true for particular cases, where the settings of compiling are not set to avoid those simplifications. Additional this is an often named example for the comparison of Clang and GCC.

```

1 // GCC
2 x = 5
3 y = 25 + x
4 // afterwards
5 x = 5
6 y = 30
7
8 // Clang
9 x = 5
10 y = 25 + x
11 //afterwards
12 x = 5
13 y = 25 + x

```

Listing 2.12: Constant Folding GCC and Clang

The Clang AST is a very comprehensive representation “fully type resolved”, what means that it can create a node for every type leading to a very large size of the AST [Tea17d]. The nodes of Clangs AST are ordered in a hierarchy of classes. The class `ASTContext` contains all information about the AST of a translation unit, the final input for Clang to generate the IR. This unit is the result of the preprocessor, which captures also directives specified by `#include`, `macros` and `#ifdef`.

```

1 Preprocessor -> C compiler -> Assembler -> Linker

```

Listing 2.13: Compile C

With the function `getTranslationUnitDecl()` the entry point of the AST can proceed to one of the core classes named `Decl`, `Stmt` and `Type`. Graphs of these classes are added to the appendix. Instances of these classes have identity and also pointer identity, therefore two pointers can be compared.

## RecursiveASTVisitor

The AST is traversed with a class called `RecursiveASTVisitor`. It is used to visit all reachable nodes, beginning with the `TranslationUnitDecl`. At each node it also can walk up the class hierarchy until the uppermost like `Decl`, `Stmt` or `Type` is reached. `RecursiveASTVisitor` can also find class node combinations determined by the user. This is achieved with an `override` of the function `VisitNode()`. In listing 2.14 the function `VisitArrayType` will be triggered for every `ArrayType` node. The name of the array will be assigned as string to the variable `arrayTypeName_`.

```

1 bool VisitArrayType(clang::ArrayType *at) {
2     arrayTypeName_ =
3         at->getDecl()->getQualifiedNameAsString();
4     return true;
5 }

```

Listing 2.14: Visitor Function

## Source Code and the AST

In figure 2.7 the uppermost node is an instance of the class `FunctionDecl` representing a declaration. From all core classes, there are many reachable child nodes. As it is illustrated in figure 2.7, the `DeclStmt` is the parent node of the `VarDecl`. While `VarDecl` denotes the declaration of the variable `result`, the following child nodes are the statement. It is important to know that a `DeclStmt` is an “*Adaptor class for mixing declarations with statements and expressions.*”[LLV17i]. This feature can lead to complex coherences. Every node in the AST got its own special methods to survey the connections of the tree.

To generate a comprehensible AST dump on the command line, it is required to denote the position of the source code for every class of the `ASTContext`. For this reason a so called `SourceLocation` encodes the position. The `SourceManager` can decode the location to get all the information like column, line and full include stack saved in the object. The lines of listing 2.15 will not correspond to those of figure 2.7 because the source is represented in a shortened form.

A descriptive example of how an AST looks is shown in the following figures 2.7 and 2.8.

```

1 int sqr(int x){
2     int result = x * x;
3     return result;
4 }

```

Listing 2.15: Simple Program for the AST

```
1 clang -Xclang -ast-dump -fsyntax-only test.c
```

Listing 2.16: Console AST Dump

While `-Xclang` passes arguments directly to the clang compiler, `-ast-dump` creates an output of the AST in the console. The option `-fsyntax-only` is used to check the syntax and prevent the creation of an objective file.

```
-FunctionDecl 0x62100011a900 <test.c:3:1, line:6:1> line:3:5 sqr 'int (int)'  
|-ParmVarDecl 0x6210000f23e8 <col:9, col:13> col:13 used x 'int'  
|-CompoundStmt 0x62100011ab90 <col:15, line:6:1>  
|-DeclStmt 0x62100011ab00 <line:4:5, col:23>  
|  |-VarDecl 0x62100011a9c8 <col:5, col:22> col:9 used result 'int' cinit  
|  |  |-BinaryOperator 0x62100011aad0 <col:18, col:22> 'int' '*'  
|  |  |  |-ImplicitCastExpr 0x62100011aa90 <col:18> 'int' <LValueToRValue>  
|  |  |  |  |-DeclRefExpr 0x62100011aa30 <col:18> 'int' lvalue ParmVar 0x6210000f23e8 'x' 'int'  
|  |  |  |  |-ImplicitCastExpr 0x62100011aab0 <col:22> 'int' <LValueToRValue>  
|  |  |  |  |  |-DeclRefExpr 0x62100011aa60 <col:22> 'int' lvalue ParmVar 0x6210000f23e8 'x' 'int'  
|  |-ReturnStmt 0x62100011ab70 <line:5:5, col:12>  
|  |  |-ImplicitCastExpr 0x62100011ab50 <col:12> 'int' <LValueToRValue>  
|  |  |  |-DeclRefExpr 0x62100011ab20 <col:12> 'int' lvalue Var 0x62100011a9c8 'result' 'int'
```

Figure 2.7: AST Dump



### 2.3.1 Another AST Representation

Another more graphical representation can be achieved with the following command in listing 2.18.

```
1 clang -cc1 -ast-view test.c
```

Listing 2.17: Clang AST with -ast-view

This option generates a `.dot` file representing the AST in a special format. Graphviz, a tool particularly for visualization of graphs, can transfer the given structural information into these while serving an amount of types for common visual outputs [Gra17]. The program itself is open source and one way of generating graphs from `.dot` files. The flag `-ast-view` is a frontend only option and can just be used by passing the flag `-cc1`. The frontend only option is actually for Clang developers because it is not guaranteed to be stable. The flag `-Xclang` can pass arguments to the driver, which also can pass it on to the frontend [Teab].

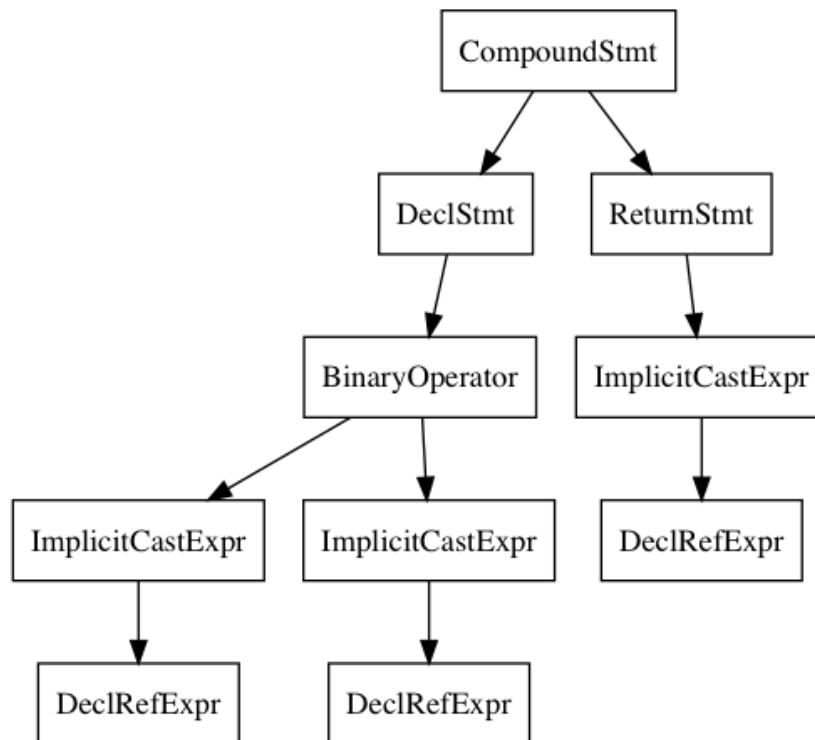


Figure 2.8: AST with Graphviz

## 2.3.2 AST Matchers

For certain problems it is a good choice to write a matcher specified for one's own interests. To work with the AST a specific Domain Specific Language (DSL) is provided. To match a function declaration of figure 2.8, the matching looks like listing 2.17. If the match was successfully found in the AST, the class `MatchCallback` is invoked [doxa].

```
1 match functionDecl(  
2     hasName(  
3         "sqr")
```

Listing 2.18: Matching Example

There are 3 different categories the matcher offers [LLV17d]. The first ones are the **node matchers**. They track down which type of node to find. In the example of listing 2.18, `functionDecl()` searches for all matching nodes of type `FunctionDecl`. As this call alone would fit hundreds of matches including the internal declaration of Clang, it needs to be specified. The second type of matchers are the **narrow matcher**. They observe the attributes related to a node. For the already mentioned example `hasName("sqr")` differentiates if the `FunctionDecl` is named like the string passed to the function. The third category are the **AST traversal matchers**. This matchers survey the relationship of the given node to others. For the `FunctionDecl` there are given functions to be called on that node, like `hasAnyParameter(<ParmVarDecl>)`, `hasBody(<Stmt>)` or `returns(<QualType>)`. With this knowledge and basic information about the LLVM environment, a source-to-source translation tool could be implemented with the help of Clangs LibTooling and LibASTMatchers. It is possible to write syntax checking tools observing the nodes and relationships of them in the AST.

## clang-query

Besides building tools for matching specific node conditions, an interactive tool for evaluation of match expressions can be used, called `clang-query`. As it is also included in the LLVM sub directory `clang-tools-extra/` the binary might be already available system-wide. By basically running the command `clang-query` while passing a `.c` file as argument, an environment pops up to interactively interpret match expressions to view the matching points in the source. For the named example in figure 2.8, the result looks like listing 2.19. The programmers have a quick idea about the accuracy of their matcher expression.

```
1 clang-query> match functionDecl(hasName("sqr"))
2
3 ... /test.c:3:1: note: "root" binds here
4 int sqr(int x){
5 ^~~~~~
6 1 match.
```

Listing 2.19: clang-query Example

## 3 MPI - Message Passing Interface

The Message Passing Interface is an implementation to hand the programmer the power of inter-process communication. Furthermore, simultaneous reading and writing is also included called MPI-IO. Conveniently MPI is used by every HPC program, which includes parallel processes. Since its beginning of development in year 1992, MPI can make distributed computer systems cooperate with each other and solve a problem through dividing it into sub-problems. Each sub-problem is solved in less time than the whole problem. Data of the computation are exchanged between processes. A master process gathers all results at the end of every partial-computation, to combine them for the final result. As powerful MPI is, it is hard to handle. Bugs like race conditions, deadlocks, livelocks and issues with the message itself might occur [Squ17]. Reproduction of errors and detecting the cause can be an exhaustive tasks to solve.

## 3.1 MPI Communication Example

This section discusses some basics and clues about the functionality of MPI in short. For further information it is recommended to have a look at <http://mpi-forum.org/>. Listing 3.1 shows the main parts every program with MPI should have. This is followed by the initialization of the environment, which is executed by every process. The individual rank as well as the total number of processes in the communicator are stored in variables for later purposes. The environment is closed with `MPI_Finalize`.

```
1  #include <mpi.h>
2
3  // Initialize the environment.
4  MPI_Init(&argc, &argv);
5
6  int processRank;
7  MPI_Comm_rank(MPI_COMM_WORLD, &processRank);
8
9  int processCount;
10 MPI_Comm_size(MPI_COMM_WORLD, &processCount);
11
12 // main program
13
14 // Finalize environment.
15 MPI_Finalize();
```

Listing 3.1: MPI Basics

The following models in listing 3.2 and 3.3 show the communication of two processes. As each process has its own unique rank, the communication is addressed for a specific destination to send to or a source to receive from. To know what kind of data to send and how much, the parameter `count` denotes the number of elements, whereas the `MPI_Datatype MPI_INT` describes the datatype of the buffer. In `MPI_Recv` it is possible to inspect a status of type `MPI_Status`. This is used to verify the receive operation. Like in listing 3.1, the already shown communicator group `MPI_COMM_WORLD` is used and needed by the functions while there are more possibilities. There is usually no need for splitting the communicator in small programs. It is used for dividing the group of processes into smaller groups to encapsulate different sub-groups of communication.

However, it has to be mentioned that the use of the `tag` is only needed for special cases, otherwise it can be set to 0. An example would be to send two different messages of same length and same type but with different information to the same destination. At this point it is needful to set different tags to differentiate the data in the communication of two processes, as it can not be guaranteed that the messages are received in the order they are sent [CSE17].

Important to know is, that this kind of communication represented in listing 3.2 is classified as blocking communication. When the routine comes to the point of calling `MPI_Send` or `MPI_Recv` it does not return until the data is stored and the buffer is free again. The other kind is called nonblocking communication in which the routine comes to an immediately call like `MPI_Isend` and `MPI_Irecv`. This functions return immediately back to their task without waiting for the buffer being copied and the communication to terminate successful. The programmer needs to implement specific points, where the process verifies the integrity of the communication. This is realized with the function `MPI_Wait` or `MPI_Test`. For both immediately calls a **request** handle is returned to manage the enduring status of the message. Especially this both properties of MPI calls lead to different kinds of bugs named in the introduction beforehand.

```

1  if(rank == 0)
2      MPI_Send(data, count, MPI_Datatype,
3              destination, tag, comm)
4  if(rank == 1)
5      MPI_Recv(data, count, MPI_Datatype,
6              source, tag, comm, status)

```

Listing 3.2: MPI Communication Mode Formal

```

1  if(rank == 0)
2      MPI_Send(sendbuf, count, MPI_INT,
3              rank + 1, 0, MPI_COMM_WORLD)
4  if(rank == 1)
5      MPI_Recv(recvbuf, count, MPI_INT,
6              rank - 1, 0, MPI_COMM_WORLD, status)

```

Listing 3.3: MPI Communication Mode

Of course the communication is not limited to the point-to-point example of listing 3.2 and 3.3. In addition there are different communication modes integrated into MPI. While `MPI_Bcast` spreads a message to all processes in the communicator, a `MPI_Gather` collects all data of all processes in one. To keep in mind that MPI is used to solve computation problems more efficient than one single process, the interface offers plenty functions and specifications to improve a program in its efficiency and performance.

## 3.2 MPI-IO Example

The sample in listing 3.4 shows how different processes can write a sequence of characters into the same file. In this case, `fh` is the file handle of the special MPI type `MPI_File`. To prevent overwriting each others data, an offset of type `MPI_Offset` indicates the displacement of the individual pointer inside the file. As usual, the process with `rank == 0` is considered as master. Regarding to the rank a specific value for `char *data`

is hand to the `MPI_File_write` operation. The result of four processes looks like the illustrated text file of figure 3.1.

```
1 MPI_File fh;
2 MPI_Status status;
3 MPI_File_open(MPI_COMM_WORLD, "test.txt",
4               MPI_MODE_CREATE | MPI_MODE_RDWR,
5               MPI_INFO_NULL, &fh);
6
7 const char *data = rank == 0 ? "Master " : "Slave ";
8
9 const MPI_Offset offset = rank * strlen(data);
10
11 // Update individual file pointer.
12 MPI_File_seek(fh, offset, MPI_SEEK_SET);
13
14 MPI_File_write(fh, data, strlen(data), MPI_CHAR,
15               &status);
16
17 MPI_File_close(&fh);
```

Listing 3.4: MPI-IO Example Code

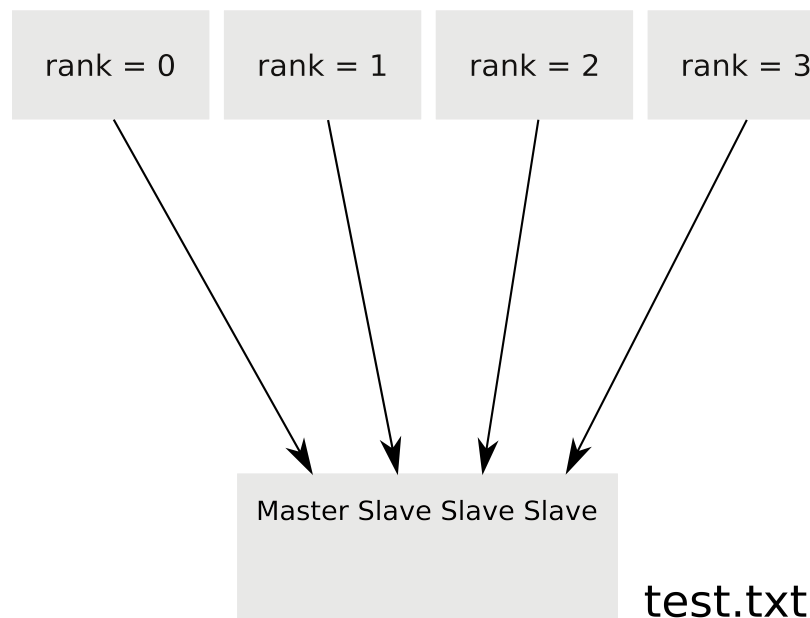


Figure 3.1: MPI-IO Example

## 4 Clang-Tidy

*“Clang-tidy is a clang-based C++ “linter” tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors” [LLV17j].*

The diagnosis is based on the information provided by the AST using matchers or the preprocessors `PPCallbacks` [doxc]. Checks which are not depending on run time behavior and occur through distracted programming can be created. Certainly, it can analyze even bigger problems with just the AST information, but a check might get very soon very complex. In addition, there are no assumptions made about values of variables for most cases.

The command `clang-tidy` has its own checks but can also invoke those of the CSA. This is achieved by passing the flag `-checks=` with the desired check. A big advantage is to write a custom check straightforward. The checks are organized in modules based on their class of what they detect. Issues related to readability, performance, MPI and many more are included. Clang-Tidy is made for problems according style, misconstruction of source code and misuse of interfaces by analyzing the syntax based on the AST. There can be hints generated to apply proposed fixes to a detected hitch.

To pass a compile command database for analysis is mentioned as recommendation.

The tool `clang-query` is used to create matchers directly in the console as mentioned in section 2.3.2. Despite it already contains a great number of checks, including `modernize-loop-convert` [LLV17o], `readability-else-after-return` [LLV17q] and many more, the adding of a new check is explained in the following section.



## 4.1 Example of Clang-Tidy

This section explains how a certain check works. To have a quick overview of all, listing 4.1 shows how to invoke the right command. Because the option `-list-checks` only displays enabled ones, the `-checks=*` is additionally passed to list all available.

```
1 $ clang-tidy -list-checks -checks='*'
```

Listing 4.1: All Clang-Tidy And Analyzer Checks

As the names of the listed checks reveal, CSA routines can also be inducted by the `clang-tidy` binary. The details on how to activate the analysis is explained in the following. Located in `clang/tools/extra/clang-tidy` is a python script named `add_new_check.py`.

```
1 $ ./add_new_check.py readability checker-name
```

Listing 4.2: Add a Clang-Tidy Check

The command in listing 4.2 creates a template of a checker as `.cpp` file, a header file `.h`, denotes the checker in the `CMakeLists.txt` and also generates a default test file inside of `clang/tools/extra/test/clang-tidy`. With rebuild of LLVMs tools the checker would be ready to go.

### Example - Readability ElseAfterReturn

In what follows the already implemented check **ElseAfterReturnCheck.cpp** is reviewed. This check detects additional `else` statements after a potential return in the body of an `if` statement.

Every check has two main functions, which needs to be adapted for the desired purpose. The function `registerMatchers(MatchFinder *Finder)` registers a Clang AST Matcher to provide the source locations of interest.

Afterwards `check(const MatchFinder::MatchResult &Result)` is called to perform actions on the locations of match, which is referred. For this particular case, the matcher finds every `if` with a then statement, that is defined as **ControlFlowInterruptor-Matcher** in listing 4.3 on line 3 or is part of a group of statements. This just finds and binds the defined statements to a given string. As certain statements are tracked for every `if` body, an `else` statement is also bound with the string `else` on line 15.

```

1 void ElseAfterReturnCheck::registerMatchers(
2     MatchFinder *Finder) {
3     const auto ControlFlowInterruptorMatcher =
4         stmt(anyOf(returnStmt().bind("return"),
5                     continueStmt().bind("continue"),
6                     breakStmt().bind("break"),
7                     cxxThrowExpr().bind("throw")));
8
9     Finder->addMatcher(
10         compoundStmt(forEach(
11             ifStmt(hasThen(stmt(
12                 anyOf(ControlFlowInterruptorMatcher,
13                     compoundStmt(has(
14                         ControlFlowInterruptorMatcher))))),
15                 hasElse(stmt().bind("else"))))
16             .bind("if"))),
17         this);
18 }

```

Listing 4.3: ElseAfterReturnCheck::registerMatchers()

At this point the matching logic for the wished source locations to find is viable. To process whats happening after the matcher is registered, the method ElseAfterReturnCheck creates diagnostics about the AST node and location where the fault is detected.

```

1 void ElseAfterReturnCheck::check(
2     const MatchFinder::MatchResult &Result) {
3     const auto *If = Result.Nodes.getNodeAs<IfStmt>("if");
4     SourceLocation ElseLoc = If->getElseLoc();
5     std::string ControlFlowInterruptor;
6     for (const auto *BindingName : {"return", "continue",
7                                     "break", "throw"})
8         if (Result.Nodes.getNodeAs<Stmt>(BindingName))
9             ControlFlowInterruptor = BindingName;
10
11     DiagnosticBuilder Diag = diag(
12         ElseLoc, "do not use 'else' after '%0'"
13         << ControlFlowInterruptor;
14     Diag << tooling::fixit::createRemoval(ElseLoc);
15
16     ...
17 }

```

Listing 4.4: ElseAfterReturnCheck::check()

## 4.2 Invoke Clang-Tidy Checks

To obtain the benefits of Clang-Tidys checks, a script named `run-clang-tidy.py` is invoked to call default checks on all files considered by a compilation-database, a JSON file. The flags and commands for compilation contained in the JSON file are captured. The `clang-tidy` binary and the `clang-apply-replacements` have to be in the users `$PATH` to enable the function of this Python script.

```
1 clang/tools/extra/clang-tidy/tool/run-clang-tidy.py
```

Listing 4.5: Call run-clang-tidy

Another way to access the checks is shown in listing 4.6, which was already used to list the available checks.

```
1 clang-tidy [options] <source>
```

Listing 4.6: Call clang-tidy

### ElseAfterReturn Example

The present code of listing 4.7 is analyzed with the command of listing 4.8. The check `ElseAfterReturn` outputs a warning like in figure 4.1.

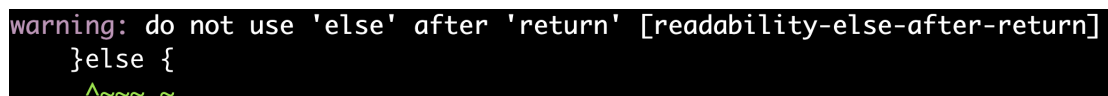
```
1 if(j > 1){  
2     return 1;  
3 }else{  
4     return 0;  
5 }
```

Listing 4.7: elseAfterReturn Example

With the option `*,-readability-else-after-return'`, all checks are disabled while `ElseAfterReturn` is enabled.

```
1 $ clang-tidy -checks='*,-readability-else-after-return' \  
2     elseAfterReturn.c
```

Listing 4.8: clang-tidy ElseAfterReturn



```
warning: do not use 'else' after 'return' [readability-else-after-return]  
    }else {  
      ~~~~~
```

Figure 4.1: ElseAfterReturn Diagnosis

# 5 Clang Static Analyzer

## 5.1 Introduction

The CSA enables with its facilities an even deeper analysis beyond the AST but still based on the already mentioned representation. The key is to simulate the execution of the source code without completely compiling or even running it. Because it reasons about possible execution not depending on input parameters, random generated data or even properties of library elements, bugs which might occur on very unusually program paths can be found. The advantage of analyzing every reachable program point and state is a feature other common testing methods can not fulfill. Certainly there are limitations like loops, which are simulated for up to 4 iterations. After that the CSA can only approximate the following iterations [Zak13].

Through symbolic execution, it is possible to abstract from concrete values with symbols and make assumptions about the path of execution that led to a certain error. The pathway of execution is represented as **ExplodedGraph**, a reachability graph. The graph is an object while the nodes are **ExplodedNode** containing a program state and program point. The program point is related to the location of execution, the stack frame, post- and pre-statement. The program state encloses the memory location, constraints on symbolic values, a generic data map and environmental expressions [LLVa]. Even bugs whichever been overseen by test suits or through testing by hand are precipitated. The path which led to the bug is tracked and minded for the result of analysis.

Different APIs lead to different behavior of the final program, especially if there is something like MPI. Nowadays landscape of computers provide software developers with an amount of cores on one chip. Not only this setting might be difficult enough to picture most mistakes happening trough parallel interaction in the code. Inter-process communication with MPI makes it possible for processes on different nodes to cooperate, which is a very powerful but also susceptible to failures tool. The CSA will not find all prospects of difficulties by using just the already present checks. It is needed to build a check, that extracts the necessary information out of the data offered by Clang. The programmer of such a check needs to capture predefined nodes in the **ExplodedGraph**, store them and observe dependencies of symbolic values of different states. If a defect covered by the check is found, it reports the state specified by the developer. The handling of an event occurring during analysis can be customized. It is possible to continue the analysis until the end of execution. Because it is up to the designer of the check what to trace and when to stop the pursuit. The behavior at detection can be designed different. The CSA is a very powerful tool to keep HPC programs from faulty

executions and improve the overall ruggedness of the application. It is completely open source and a component of the Clang project. The CSA is used during development and makes it possible to find bugs in early stages, which makes them simpler to fix.

*“Like the rest of Clang, the analyzer is implemented as a C++ library that can be used by other tools and applications.”* [CA17]

## 5.2 Path-Sensitive

The benefit of the CSA is, that it uses path-sensitive analysis. Clang-Tidy with its purely AST based analysis can not really reason about values of variables. Symbolic execution make assumptions about ranges of values. This leads to different program execution paths. Even for-loops can be simulated up to the 4th iteration, as mention in the introduction of the CSA. The difference of both methods are clearly pointed when looking at their results of analysis. As Clang-Tidy emits a message about the location of the bug, the CSA will show a whole path, with conditions leading to the issue.

```
1 void pathSensitive(int a){
2     if(a < 5)
3         a = a - 4;
4     if(a > 3)
5         a = 10 / a;
6 }
```

Listing 5.1: Example for Path-Sensitive

Listing 5.1 leads to different paths of execution, depending on input *a* of the function `pathSensitive()`. All states *P* of figure 5.1 are valid, except the faulty state *P4*, where a division by zero is detected.

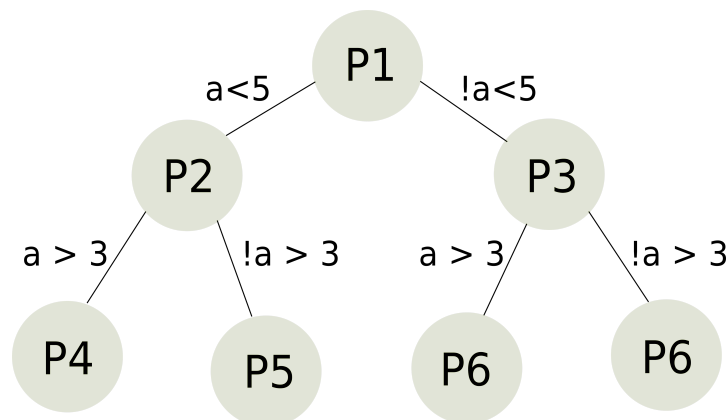


Figure 5.1: Path-Sensitive Graph

## 5.3 Invoke The Analyzer

Currently the static analyzer can be executed in Xcode or as an autonomous tool from the command line. To check source code mainly from the command line, there is an utility called `scan-build`. For Mac OS X it is possible to download pre-built binaries for direct use [LLV17p]. There are two location to obtain `scan-build` from. The first one is the LLVM repository, where it is contained in Clang and the second one is the Python Package Index `pip`. Another way of executing the analyzer is to pass the `--analyze` flag to the compile command `clang` already mentioned in section 2.2. Both routines contain all required parts for analysis. This includes the checks, algorithms, libraries and the parts explained in section of LLVM. The related MPI options being part of `checker-279` release [Rel16] are new to the analyzer. A following test program shows the result of calling `scan-build`. The usage is conceptualized to be straightforward. For this example there is a simple program in listing 5.2 where at some point an error occurs. `scan-build` overrides the `CC` and `CXX` environment variables and uses a wrapper compiler in between of the one which would have done the task. This wrapper compiler guesses the default one and uses for most cases `gcc` or `clang` [`scan-build -h`]. It is also possible to hand `scan-build` a specific compiler by using the option `--use-cc <path-to-compiler>` for C or `--use-c++ <path-to-compiler>` for C++. Because of `mpicc` is being a wrapper with different configuration of environment variables and linked libraries, it would not work directly with `scan-build`. To solve this issue, a different solution is mentioned later. Its important to know that every file, which is not compiled is also not considered by the analyzer.

```
1 int divisionByZero(int i, int j){
2     int result = i / j;
3     return result;
4 }
5 int main (){
6     divisionByZero(0,0);
7 }
```

Listing 5.2: Simple Program with Bug

The script `scan-build` is just placed in front of the build command like in listing 5.3 or additional with a specification for the compilation and the file to analyze illustrated in listing 5.4. In case of listing 5.4 it just invokes the preprocess and compile steps activated through the option `-S`. This flag offers enough processing to analyze the AST. There is no need for a binary, especially in cases where compilation would take several minutes. It is useful to reduce the analysis to the only needed steps.

```
1 $ scan-build -o . make
2 # with current folder as output
```

Listing 5.3: Invoke scan-build

```
1 $ scan-build clang -S divbyzero.c
```

Listing 5.4: Invoke scan-build with options

```
scan-build: Using '/usr/local/bin/clang-5.0' for static analysis
divbyzero.c:4:17: warning: Division by zero
    int result = i / j;
                   ~^~^
1 warning generated.
scan-build: 1 bug found.
```

Figure 5.2: Analyze result

Obviously this produces an error shown in figure 5.2 because of the division by zero. `scan-build` analyzes this example and provides a report with the order of control flow. To visualize how the program execution led to the bug, another script called `scan-view` is included into the utility. `scan-view` opens the report generated by `scan-build` in the default browser.

[Summary](#) > **Report 61465b**

#### Bug Summary

File: divbyzero.c

Warning: [line 4, column 17](#)  
Division by zero

[Report Bug](#)

#### Annotated Source Code

```
1 #include <stdio.h>
2
3 int divisionByZero(int i, int j){
4     int result = i / j;
5
6     return result;
7 }
8 int main (){
9     divisionByZero(0,0);
10
11 }
```

3 ← Division by zero

1 Passing the value 0 via 2nd parameter 'j' →

2 ← Calling 'divisionByZero' →

Figure 5.3: HTML Report scan-build

There is one more option to work with this tool. The commands `intercept-build` and `analyze-build` does the same job as `scan-build` but with more options to choose from.

`intercept-build` builds the program without analyzing or running it and `analyze-build` only do the analysis and delivering of the output. With `intercept-build` a compilation database is created, which is a JSON file containing all compile commands required to build the project as already mentioned in section 2.1.2. The up-to-date version of this command is part of the pip installation [Nag17], while in LLVM it is still not in default use. The generated JSON file is different to the database generated by CMake but both can be used for the analysis. The counterpart `analyze-build` runs the analyzer based on the generated compiler calls in the compilation database [LLVf]. Chapter 3 explained the structure of MPI. As the already mentioned unreleased scripts might not created the expected results, the use of them is limited to circumstances.

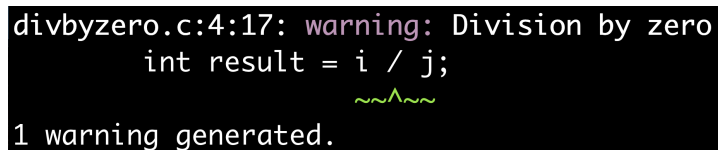
It is important to use both `intercept-build` and `analyze-build`, especially if there is an API used like MPI, which needs to be wrapped before analyzing. This is not that clear by just using `scan-build`.

A good hint to consider is to always analyze the project in its debug configuration because the CSA can extract more information from `assertions` about unfeasible paths, whichever can be shortened. False-positive are reduced through assertions of the program by checking certain conditions like preconditions, postconditions and invariants to verify the reliability of the programs executions [Ree95]. The analyzer omits messages about false-positives and generates a more solid result.

The CSA improves the quality of program code because it can find a lot of flaws and should be used frequently in the process of developing. By comparison, the CSA finds most potential bugs in the source that would have been hard to find by manual testing. An example of invoking the analyzer just with `clang` is shown in listing 5.5.

```
1 $ clang --analyze divbyzero.c
```

Listing 5.5: Clang with analyze option



```
divbyzero.c:4:17: warning: Division by zero
    int result = i / j;
                      ~~~
1 warning generated.
```

Figure 5.4: Clang with analyze option activated

This leads to the same report shown in figure 5.2 but precludes the HTML view of `scan-view`. Instead a special formatted XML file `.plist` is created and can be observed with Xcode. The view is shown in figure 5.5.



Key	Type	Value
▼ Root	Dictionary	(3 items)
clang_version	String	clang version 5.0.0
▼ files	Array	(1 item)
Item 0	String	divbyzero.c
▼ diagnostics	Array	(1 item)
▼ Item 0	Dictionary	(10 items)
▼ path	Array	(5 items)
▼ Item 0	Dictionary	(6 items)
kind	String	event
▼ location	Dictionary	(3 items)
line	Number	8
col	Number	19
file	Number	0
▼ ranges	Array	(1 item)
▼ Item 0	Array	(2 items)
Item 0	Dictionary	(3 items)
Item 1	Dictionary	(3 items)
depth	Number	0
extended_message	String	Passing the value 0 via 2nd parameter 'j'
message	String	Passing the value 0 via 2nd parameter 'j'
Item 1	Dictionary	(6 items)
Item 2	Dictionary	(5 items)
Item 3	Dictionary	(2 items)
Item 4	Dictionary	(6 items)
description	String	Division by zero
category	String	Logic error
type	String	Division by zero
check_name	String	core.DivideZero
issue_hash_content_of_line_in_c...	String	d6d2a1bfb6b8af7d0598d35a2d63558f
issue_context_kind	String	function
issue_context	String	divisionByZero
issue_hash_function_offset	String	1
▼ location	Dictionary	(3 items)
line	Number	4
col	Number	17
file	Number	0

Figure 5.5: .plist in Xcode

### 5.3.1 About the Checks

When the CSA is invoked with `scan-build`, there are default checks that are always turned on. Part of them are **Core Checks** in which the `core.DivideZero` is included. Besides, there are special checks for C++, OS X and unused code. Table 5.1 exhibits a brief overview [LLV17e].

Name	Description
Core Checkers	General purpose checks
C++ Checkers	Specific for C++
OS X Checkers	Specific for Objective-C and the SDK of Apple
Dead Code Checkers	Detects unused code
Security Checkers	Security checks for APIs and the standard of CERT
Unix Checkers	Specific for the POSIX API and Unix

Table 5.1: Available Checkers

Furthermore, there are more checks available like the **Alpha Checks** with limitations and likely to occur false positives. Because of this, these checks are not enabled by default [LLV17c]. Checks of the **Core Alpha Checkers** include for example the detection of bool variables assigned with non bool values and a dozen more. To list all enabled checks the command in listing 5.6 is invoked.

```
1 $ scan-build --help-checkers
```

Listing 5.6: Display Enabled Checks

For all available checks, the commands of listing 5.7 are used. This will also print all non-path-sensitive checks.

```
1 $ clang-tidy -list-checks -checks=*
2 # or
3 $ clang -cc1 -analyzer-checker-help
```

Listing 5.7: Display Available Checks

A particular check can be enabled or disabled. This is determined with the `scan-build` option `-enable-checker <NameOfCheck>` or `-disable-checker <NameOfCheck>`. It will add or delete a check from the list of enabled checks the command in listing 5.6 showed.

### 5.3.2 scan-build with MPI

As there is no possibility to execute scan-build primitive like in listing 5.3, options to make use of wrapping MPI are hidden inside the usage of `intercept-build` or other options of revealing the right flags to the analyzer. The replacement through the wrapper compiler explained beforehand is not working for this particular case, the build command does not pass the right compiler and linker flags in order to understand and analyze MPI.

`scan-build` guesses the systems default one in this case and fails. There are options to pass different compiler wrappers to `intercept-build`. The option `-use-cc <path>` specifies a path to the one of interest. By typing `intercept-build -h` the option `-use-cc` displays the default compile command set by the `CC` and `CXX` environment variables. The default of `intercept-build` denotes and run the MPI compiler `mpicc`, if it is set via `export CC=mpicc`. This can also be directly achieved by passing `-use-cc mpicc`. To have a look at an example of the static analyzer being invoked at projects containing MPI, the example of Alexander Droste [Dro16] gives a well insight.

It is important to keep in mind again that `intercept-` and `analyze-build` are official released in the package of `pip` but not in the repository of LLVM.

Listing 5.8 exhibits the result of just passing the parameters but without the right information the wrapper would deliver. This error occurs through the lack of the not linked MPI library. A wrapper for the MPI implementations came in the the mid-90's. It just passes the right compiler flags and links to the underlying C/C++ compiler.

```
1 $ scan-build -o . -enable-checker optin.mpi.MPI-Checker \  
2   --use-analyzer $(which clang) make  
3  
4   basic.c:27:10: fatal error: 'mpi.h' file not found  
5     #include <mpi.h>
```

Listing 5.8: Invoke scan-build at MPI error

To solve this problem of missing include paths listing 5.9 shows a solution for that. The compiler and linker flags are obtained directly from the `mpicc` command with the option `--showme`. With assignment to environmental variables the analyzer recognizes the MPI specific features to use the underlying compiler in a correct context. Afterwards the commands in listing 5.10 will analyze the related files with MPI checks activated.

```
1 $ export LDFLAGS=`mpicc --showme:link`  
2 $ export CFLAGS=`mpicc --showme:compile`  
3 $ export CXXFLAGS=`mpicc --showme:compile`
```

Listing 5.9: MPI Flags

```

1 $ scan-build -enable-checker optin.mpi.MPI-Checker \
2   --use-cc=$(which clang) make
3 # or
4 $ intercept-build make
5 $ analyze-build -o . -enable-checker optin.mpi.MPI-Checker \
6   --use-analyzer $(which clang)

```

Listing 5.10: Analyse MPI

It is possible to have some issues with the JSON Compilation Database, which is not created reliably by `intercept-build`. Another tool to experience the same effect as with `intercept-build` is `bear`:

*“Bear is a tool that generates a compilation database for clang tooling.”* [Nag]

This tool based on `scan-build` and should therefore have a similar behavior. Afterwards `analyze-build` or `clang` with `--analyze` and `-p compiledb` can be invoked. Instead of using `intercept-build` it is possible to create the database with a simple CMake flag as a parameter for the build command like in listing 5.11 or add the line of listing 5.12 to the prime `CMakeLists.txt` file.

```

1 $ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON

```

Listing 5.11: Analyze based on CMake's compilation database

```

1 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

```

Listing 5.12: Analyze Based on CMakeLists.txt Option

After this procedure the analyzer is invoked like in listing 5.13 by leaving out `intercept-build`.

```

1 $ analyze-build -o . -enable-checker optin.mpi.MPI-Checker \
2   --use-analyzer $(which clang) -o .

```

Listing 5.13: `intercept-` and `analyze-build` for MPI

## 6 MPI-Checker

This chapter captures the added features and files of the MPI-Checker implementation. New functions of MPI are supplemented and used in the extension for a wider range of analysis. The concepts of checks implemented and planned are explained and visualized. Test files verify the correct implementation and detection of bugs.

Since the actual MPI-Checker implementation is particular for MPI communication regarding non-blocking calls, this thesis modifies already implemented features as well as expand the checker with MPI-IO related functionality's. This also takes care about the implementation of Clang-Tidy, which includes checks for type-mismatches of MPI functions and wrong referenced buffers.

The repository for this implementation is available at Github:

[https://github.com/Bassstring/llvm\\_mpi\\_checks](https://github.com/Bassstring/llvm_mpi_checks).

### 6.1 Files of the Implementation

The files in listing 6.1 are more or less important for the adaption of the MPI-Checkers functionality. Clang-Tidy needs the help of the `MPIFunctionClassifier` to capture the calls and relevant data of the AST.

The static analyzer is part of Clang, while Clang-Tidy belongs to **clang-tools-extra**.

```
1 # Clang Static Analyzer
2 /clang/StaticAnalyzer/Checkers/MPIFunctionClassifier.h
3 /lib/StaticAnalyzer/Checkers/MPI-Checker/
4 /test/Analysis/MPIMock.h
5
6 # Clang-Tidy
7 /tools/extra/clang-tidy/mpi/
8 /tools/extra/test/clang-tidy/mpi-buffer-deref.cpp
9 /tools/extra/test/clang-tidy/mpi-type-mismatch.cpp
10 /test/clang-tidy/Inputs/mpi-type-mismatch/mpimock.h
```

Listing 6.1: Paths MPI-Files

## 6.2 New Features

### 6.2.1 AST Analysis

For the pure AST analysis, function of MPI-IO are added to the `MPIFunctionClassifier.cpp` and `MPIFunctionClassifier.h`. With this step the AST can capture the relevant functions of MPI in containers. These containers are realized with the already introduced `llvm::SmallVectors`. All added functions are listed in the appendix section C. Next to `MPI_File_open` and `MPI_File_close`, the most common non collective MPI-IO functions are added to the MPI-Checker. The already presented communication functions have similar patterns in array of their parameters. For this check there are two important parameters to take care of. The first one is the `buffer` and second the `MPI_Datatype`. Depending which function is observed, the position of these two values are stored and analyzed. The assignment of this section organize MPI-IO functions in pendency of their parameters and use the already present logic of comparing the given types.

#### Details MPI-IO

As a continuation of chapter 3, where MPI was introduced, there are different classes of data access routines MPI-IO makes use of. Like the present functions in table 6.1 are added to the MPI-Checker, they are classified into different handling of positioning on data and synchronism.

A nonblocking data access routine is continuing the computation while the read or write is executed in the background. The user needs to assure the IO request similar to nonblocking communication. For the blocking operations the call do not return until the operation is completed.

**Explicit offset** does not use a file pointer. The access of data is directly performed at the location given as `MPI_Offset`. MPI functions of this kind have a `_at` at the end of their call.

There are two possibilities of implicit positioning on files. Every process have a **individual file pointer** for each file handle. The position of this pointer is updated with each operation in relation to the view of the file. The offset is not delivered by the programmer but is incremented by pointing to the next elementary type. This `etype` describes an item of positioning and accessing of the data. Functions of this kind contain an `i` standing for immediately.

A **shared file pointer** is for example used with the call `MPI_File_open`. There is only one shared pointer, which is used among all processes in the communicator. Only this

pointer is updated, while the individual file pointers remain unused. It is important that all processes have the same file view. If there are several operations called with use of the shared file pointer, the behavior is same as a serial sequence. For the related non collective data access routines of table 6.1, the user needs to define a specific order. Non collective routines are not deterministic in serial sequences. The functions of this kind have a `_shared` at the end of their name.

position	synchronism	function like
explicit offset	blocking	MPI_File_read_at MPI_File_write_At
	nonblocking	MPI_File_iread_At MPI_File_iwrite_At
individual pointers	blocking	MPI_File_read MPI_File_write
	nonblocking	MPI_File_iread MPI_File_iwrite
shared pointer	blocking	MPI_File_read_shared MPI_File_write_shared
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared

Table 6.1: MPI-IO Function Classification [Fora]

## Type-Mismatch

Write and read functions have the same arrangement of their parameters shown in listing 6.2. Because of this reason the capturing of the buffer and data type is arranged.

```
1 int MPI_File_write(MPI_File fh, void *buf, int count,  
2     MPI_Datatype datatype, MPI_Status *status)  
3  
4 int MPI_File_iread (MPI_File fh, void *buf, int count,  
5     MPI_Datatype datatype, MPI_Request *request);
```

Listing 6.2: Type-Mismatch MPI-IO

In figure 6.1 the output of detecting such a mismatch is illustrated. The original buffer type and the expected `MPI_Datatype` are compared and an issue is displayed.

```
/Users/Franky/MPIclangTidy/ctidychecks.c:20:21: warning: buffer type 'float' does not match  
the MPI datatype 'MPI_INT' [mpi-type-mismatch]  
  MPI_File_read(fh, readbuf, filesize, MPI_INT, MPI_STATUS_IGNORE);  
                   ^
```

Figure 6.1: Type-Mismatch Example

## Buffer-Dereference

The dereference of buffers has the same needs like the type-mismatch. In this case the only important parameter is the buffer. Based on the check, incorrectly access to buffers with missing or incorrectly placed `&` are detected. The error message is very precise, because it counts the amount of references and considers this in the output. Figure 6.2 shows an example, where the buffer `readbuf` was declared as `char* readbuf`. This leads to a `pointer -> pointer` issue.

```
/Users/Franky/MPIclangTidy/ctidychecks.c:20:21: warning: buffer is insufficiently dereferenced:  
pointer->pointer [mpi-buffer-deref]  
  MPI_File_read(fh, &readbuf, filesize, MPI_INT, MPI_STATUS_IGNORE);  
                   ^
```

Figure 6.2: Buffer-Dereference Example



## 6.2.2 Path-Sensitive Implementation

### Double-Close

The call `MPI_File_close(MPI_File fh)` synchronizes and then closes the file related to the given file handle `fh`. This synchronization is equal to the call of the collective operation `MPI_File_sync(MPI_File fh)`. The collective routine `MPI_File_close(MPI_File fh)` should only be used when all routines incorporated to `fh` have finished their task. An unwanted double-close can lead to unpredictable behavior in uses of the file between these calls. The CSA can now track program-states calling `MPI_File_close(MPI_File fh)` and captures two closes of a file handle happening successive. An example for the working report is shown with figure 6.3.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  MPI_File fh;
5  void open() {
6      MPI_File_open(MPI_COMM_WORLD, "test.txt",
7                    MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
8      MPI_File_close(&fh);
9  }
10 int main(int argc, char **argv) {
11     MPI_Init(&argc, &argv);
12     open();
13     MPI_File_close(&fh);
14     MPI_Finalize();
15     return 0;
16 }
```

2 ← File is previously closed here. →

1 Calling 'open' →

3 ← Returning from 'open' →

4 ← Double close on file 'fh'.

Figure 6.3: Double Close Example

## 7 Related Work

A lot of HPC and critical software demand assurance through static analysis. Another tool named MUST [ra17a], the MPI Runtime Error Detection Tool, is also more of a static analyzer to assure the correct usage of MPI, than a debugger with options to halt the execution and mark break points. It helps the developer to write correct code while assuring the MPI-Standard during development and before the application gets ported to a system [ra17b, p. 3]. MUST consists of three packages with different properties. As MUST contains the main functionality to assure MPI, another tool named  $P^n$ MPI is used to intercept the calls.  $P^n$ MPI is a wrapper library with a standardized interface [Forb]. The Generic Tool Infrastructure, GTI, is used to support event-based tools in parallel systems and is the third package MUST needs. While providing these infrastructures, developers can concentrate on implementing functionality to their tool, than handling the structure around it. Other than scan-build, MUST replaces `mpiexec` with `mustrun`. A code generation step is conducted to run the application with MUST. The result of analysis is a HTML file containing all issues and locations of occurrence. Graphs for visualization are also included. MUST detects a lot of bugs like type mismatches, send-send deadlocks, leaked datatypes, overlapping buffers and many more. Since features of a debugger are missing, the integration with DDT is supported [ra17b, p. 15-18]. This enables the usage of the normal DDT surface with extras of MUST.

Whereas static code analysis is one part of detecting issues of a specific class, other techniques like fuzzing have a different approach. With fuzzing the ruggedness of an application is assured through negative testing with random input. Following this, the execution is examined for misconduct. While there are tools with primitive attempts, American Fuzzy Lop [Zal] is also observing the program paths. This is realized through compiling with a wrapper and favoring the investigation of certain paths in the process of fuzzing.

LLVM contains a library for fuzzing itself, named libFuzzer [LLVb]. This was already used with Clang and the AddressSanitizer in projects like Chrome for many years [ACN12]. With in-process fuzzing there is the possibility of testing more input values a second, than with out-of-process fuzzing. In-process fuzzing means, that there is no new process launched for every test case and inputs are directly modified in the memory. Fuzzing is next to other testing methods another one to improve the properties of software through discover of hidden issues.

## 8 Conclusion

The purpose of this thesis was to explore the field of static analysis. This was achieved with the help of LLVM. After the setup and structure of the umbrella project was initiated, tools to help the programmer in relevant cases were explained. Clang as major component was observed to manipulate the compilation and receive the needed information for purely AST and path-sensitive analysis. The AST as key part with its classes and methods to gain knowledge for the analysis was considered. Clang-Tidy, `clang-query` and `scan-build` were illustrated with examples. Different views of understanding the infrastructure and tools were given. Cases of usage were considered, while the process of building checks was explained. MPI was the main target to deal with. As wrong buffer referencing and type mismatches were detected with Clang-Tidy, symbolic execution was used to simulate run time dependencies regarding IO specific calls. As this only involves the case of double-close, there are even more options of checks now available to add to the MPI-Checker. The infrastructure was extended with many MPI-IO calls as well as a function to capture relevant information of file handles.

Like in the case of MPI communication, immediate MPI-IO calls have to be matched. Overlapping writes are another option to verify. Checks to assure collective routines like a file open can be added to the MPI-Checker.

Due to the amounts of interfaces and libraries of LLVM, this thesis lead to regard the structures than implementing a lot of new features in the MPI-Checker.

# Bibliography

- [ACN12] Abhishek Arya and Chrome Security Team Cris Neckar. Fuzzing for Security. *Chromium Blog*, April 2012. Taken from: <https://blog.chromium.org/2012/04/fuzzing-for-security.html>.
- [Adm16] U.S. Food & Drug Administration. Infusion Pump Software Safety Research at FDA. April 2016. Taken from: <https://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm>.
- [All] Allinea. *Allinea DDT: The debugger for C, C++ and Fortran threaded and parallel code*. Taken from: <https://www.allinea.com/products/ddt>.
- [CA17] LLVM Clang-Analyzer. Clang Static Analyzer. *LLVM Homepage - Clang-Analyzer*, 2017. <http://clang-analyzer.llvm.org/>.
- [CSE17] CSE. MPI TAGS. 2017. Taken from: [https://cseweb.ucsd.edu/classes/sp99/cse160/programming/mpi\\_info/tags.html](https://cseweb.ucsd.edu/classes/sp99/cse160/programming/mpi_info/tags.html).
- [Dan] Al Danial. *cloc - Count Lines of Code*. <https://github.com/AlDanial/cloc>.
- [DKV17] LLVM DKV. LLVM Overview. *Website*, 2017. Taken from: <https://sites.google.com/site/llvmdvk/llvm-sogae>.
- [doxa] doxygen. *clang::ast\_matchers::MatchFinder::MatchCallback Class Reference*. Taken from: [https://clang.llvm.org/doxygen/classclang\\_1\\_1ast\\_\\_matchers\\_1\\_1MatchFinder\\_1\\_1MatchCallback.html](https://clang.llvm.org/doxygen/classclang_1_1ast__matchers_1_1MatchFinder_1_1MatchCallback.html).
- [Doxb] Doxygen. *clang::Decl Class Reference*. Taken from: [https://clang.llvm.org/doxygen/classclang\\_1\\_1Decl.html](https://clang.llvm.org/doxygen/classclang_1_1Decl.html).
- [doxc] doxygen. *clang::PPCallbacks Class Reference*. Taken from: [https://clang.llvm.org/doxygen/classclang\\_1\\_1PPCallbacks.html](https://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html).
- [Doxd] Doxygen. *clang::Stmt Class Reference*. Taken from: [https://clang.llvm.org/doxygen/classclang\\_1\\_1Stmt.html](https://clang.llvm.org/doxygen/classclang_1_1Stmt.html).
- [Doxe] Doxygen. *llvm::Type Class Reference*. Taken from: [https://clang.llvm.org/doxygen/classclang\\_1\\_1Type.html](https://clang.llvm.org/doxygen/classclang_1_1Type.html).
- [Dro16] Alexander Droste. MPI-Checker. *Github Repository* <https://github.com/0ax1/MPI-Checker>, November 2016. Taken from: <https://github.com/0ax1/MPI-Checker>.

- [Fora] MPI Forum. *MPI: A Message-Passing Interface Standard*. Taken from: <http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
- [Forb] MPI Forum. *PnMPI: Low-overhead Wrapper Library*. Taken from: <https://computation.llnl.gov/projects/pnmpi>.
- [Gee15] Marcus Geelnard. A tidy, linear Git history. December 2015. Taken from: <http://www.bitsnbites.eu/a-tidy-linear-git-history/>.
- [Gra17] Graphviz. Graph Visualization Software - Envisioning connections. 2017. Taken from: <http://www.graphviz.org/>.
- [Gre09] Doug Gregor. Clang Builds LLVM. 12 2009. Taken from: <http://blog.llvm.org/2009/12/clang-builds-llvm.html>.
- [Gro] Portland Group. *PGI Compilers & Tools*. <http://www.pgroup.com/>.
- [IEC06] IEC. NORME INTERNATIONALE INTERNATIONAL STANDARD , 2006.
- [Int] Intel. *Intel® C++ Compiler in Intel® Parallel Studio XE*. <https://software.intel.com/en-us/c-compilers/ipsxe>.
- [LAN] LANL. *IOR*. Taken from: <https://github.com/IORe-LANL/ior>.
- [Lat12] Chris Lattner. LLVM. <http://www.aosabook.org/en/llvm.html>, July 2012. Taken from: <http://www.aosabook.org/en/llvm.html>.
- [LLN17] LLNL. STAT. 2017. <https://computing.llnl.gov/code/STAT/>.
- [LLVa] LLVM. *Checker Developer Manual*. Taken from: [https://clang-analyzer.llvm.org/checker\\_dev\\_manual.html](https://clang-analyzer.llvm.org/checker_dev_manual.html).
- [LLVb] LLVM. *libFuzzer – a library for coverage-guided fuzz testing*. Taken from: <https://llvm.org/docs/LibFuzzer.html>.
- [LLVc] LLVM. *llvm/ADT/FoldingSet.h*. Taken from: <http://llvm.org/docs/ProgrammersManual.html#llvm-adt-foldingset-h>.
- [LLVd] LLVM. *llvm/ADT/SmallSet.h*. Taken from: <http://llvm.org/docs/ProgrammersManual.html#llvm-adt-smallset-h>.
- [LLVe] LLVM. *llvm/ADT/SmallVector.h*. Taken from: <http://llvm.org/docs/ProgrammersManual.html#dss-smallvector>.
- [LLVf] LLVM. *scan-build*. Taken from: <https://github.com/llvm-mirror/clang/tree/master/tools/scan-build-py>.
- [LLV17a] LLVM. A simple and hackable code base. 2017. Taken from: <https://clang.llvm.org/features.html#simplecode>.
- [LLV17b] LLVM. A simple and hackable code base. 2017. Taken from: <https://clang-analyzer.llvm.org/xcode.html>.

- [LLV17c] LLVM. Alpha Checkers. *Website*, 2017. Taken from: [https://clang-analyzer.llvm.org/alpha\\_checks.html](https://clang-analyzer.llvm.org/alpha_checks.html).
- [LLV17d] LLVM. AST Matcher Reference. 2017. Taken from: <http://clang.llvm.org/docs/LibASTMatchersReference.html>.
- [LLV17e] LLVM. Available Checkers. *Website*, 2017. Taken from: [https://clang-analyzer.llvm.org/available\\_checks.html](https://clang-analyzer.llvm.org/available_checks.html).
- [LLV17f] LLVM. Checkout LLVM from Subversion. 2017. Taken from: <http://llvm.org/docs/GettingStarted.html#checkout-llvm-from-subversion>.
- [LLV17g] LLVM. Clang vs Other Open Source Compilers. 2017. Taken from: <https://clang.llvm.org/comparison.html>.
- [LLV17h] LLVM. Clang vs Other Open Source Compilers - Pro's of clang vs GCC. 2017. Taken from: <https://clang.llvm.org/comparison.html>.
- [LLV17i] LLVM. clang::DeclStmt Class Reference. 2017. Taken from: [https://clang.llvm.org/doxygen/classclang\\_1\\_1DeclStmt.html](https://clang.llvm.org/doxygen/classclang_1_1DeclStmt.html).
- [LLV17j] LLVM. Extra Clang Tools 5 documentation CLANG-TIDY. *LLVM Homepage - Clang*, 2017. <http://clang.llvm.org/extra/clang-tidy/>.
- [LLV17k] LLVM. Getting Started with the LLVM System. May 2017. Taken from: <http://llvm.org/docs/GettingStarted.html>.
- [LLV17l] LLVM. LLVM Download Page. 2017. Taken from: <http://releases.llvm.org/download.html#4.0.0>.
- [LLV17m] LLVM. LLVM Logo. *Website*, 2017. <https://llvm.org/Logo.html>.
- [LLV17n] LLVM. LLVM Overview. 2017. <http://llvm.org/>.
- [LLV17o] LLVM. modernize-loop-convert. 2017. Taken from: <https://clang.llvm.org/extra/clang-tidy/checks/modernize-loop-convert.html>.
- [LLV17p] LLVM. Obtaining the Static Analyzer. 2017. Taken from: <https://clang-analyzer.llvm.org/installation>.
- [LLV17q] LLVM. readability-else-after-return. 2017. Taken from: <https://clang.llvm.org/extra/clang-tidy/checks/readability-else-after-return.html>.
- [LLV17r] Clang LLVM. Fast compiles and low memory use. 2017. Taken from: <https://clang.llvm.org/features.html>.
- [Nag] László Nagy. *Build EAR*. <https://github.com/rizsotto/Bear>.
- [Nag17] László Nagy. Clang's scan-build re-implementation in python. 2017. Taken from: <https://github.com/rizsotto/scan-build>.

- [OM] Open-MPI. 1. *How do I debug Open MPI processes in parallel?* Taken from: <https://www.open-mpi.org/faq/?category=debugging#general-parallel-debugging>.
- [Pho17] Phoronix. GCC 7.0 vs. LLVM Clang 4.0 Performance With Both Compiler Updates Coming Soon. January 2017. Taken from: <https://www.phoronix.com/scan.php?page=article&item=gcc7-clang4-jan&num=2>.
- [pML15] phoronix Michael Larabel. Why Google Chrome Switched To The Clang Compiler On Linux. 2015. Taken from: [https://www.phoronix.com/scan.php?page=news\\_item&px=MTg4MDI](https://www.phoronix.com/scan.php?page=news_item&px=MTg4MDI).
- [Pro] The GNU Project. *GDB: The GNU Project Debugger*. Taken from: <https://www.gnu.org/software/gdb/>.
- [ra17a] rwth aachen. Project MUST. 2017. Taken from: <https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST>.
- [ra17b] rwth aachen. Project MUST - MUST Installation. *Documentation*, 2017. Taken from: <https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST>.
- [Ree95] John Reekie. How to use assertions in C. December 1995. Taken from: <http://ptolemy.eecs.berkeley.edu/~johnr/tutorials/assertions.html>.
- [Rel16] Clang Analyzer Checker Release. Release notes for checker-XXX builds. *LLVM Homepage - Releasenotes Clang-Analyzer*, November 2016. Taken from: [http://clang-analyzer.llvm.org/release\\_notes.html](http://clang-analyzer.llvm.org/release_notes.html).
- [Squ17] Jeff Squyres. MPI: Debugging – Can You Hear Me Now? March 2017. Taken from: <http://www.clustermonkey.net/MPI/debugging-can-you-hear-me-now.html>.
- [Teaa] LLVM Team. *LLVM C front-end*. Taken from: <http://releases.llvm.org/2.8/docs/CommandGuide/html/llvmgcc.html>.
- [Teab] The Clang Team. *Frequently Asked Questions*. Taken from: <https://clang.llvm.org/docs/FAQ.html>.
- [Tea17a] GCC Team. GCC, the GNU Compiler Collection. <https://gcc.gnu.org/>, 2017. <https://gcc.gnu.org/>.
- [Tea17b] The Clang Team. Clang 5 documentation - MemorySanitizer. 2017. Taken from: <https://clang.llvm.org/docs/MemorySanitizer.html>.
- [Tea17c] The Clang Team. Clang 5 documentation - MemorySanitizer. 2017. Taken from: <https://clang.llvm.org/docs/MemorySanitizer.html>.
- [Tea17d] The Clang Team. Introduction to the Clang AST. 2017. Taken from: <https://clang.llvm.org/docs/IntroductionToTheClangAST.html>.

- [Wav] Rogue Wave. *TotalView for HPC*. Taken from: <https://www.roguewave.com/products-services/totalview>.
- [Wik13] Wikibooks. GNU C Compiler Internals/GNU C Compiler Architecture - GCC Initialization. *Wikibooks*, April 2013. [https://en.wikibooks.org/wiki/GNU\\_C\\_Compiler\\_Internals/GNU\\_C\\_Compiler\\_Architecture#GCC\\_Initialization](https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture#GCC_Initialization).
- [Zak13] Anna Zaks. Clang Static Analyzer execution path for loop. *Mailing List*, March 2013. Taken from: <https://lists.llvm.org/pipermail/cfe-dev/2013-April/028947.html>.
- [Zal] Michal Zalewski. *american fuzzy lop (2.49b)*. Taken from: <http://lcamtuf.coredump.cx/afl/>.



# Appendices

```

1  #!/bin/bash
2
3  ping -c 1 www.google.com
4  if [ $? -eq 0 ]; then
5      branch=master
6      address='http://llvm.org/git'
7
8      mkdir llvm-top && cd llvm-top
9      git clone -b $branch $address/llvm llvm
10
11     tools='llvm/tools'
12     git clone -b $branch $address/clang $tools/clang
13
14     ctools='llvm/tools/clang/tools'
15     git clone -b $branch $address/clang-tools-extra \
16     $ctools/extra
17
18     mkdir -p build/debug && cd build/debug
19     cmake -G Ninja -DCMAKE_BUILD_TYPE=DEBUG \
20           -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
21           -DLLVM_USE_SANITIZER="Address" \
22           ../../llvm
23     ninja
24     ninja check-all
25 fi

```

Listing 1: LLVM Setup Script

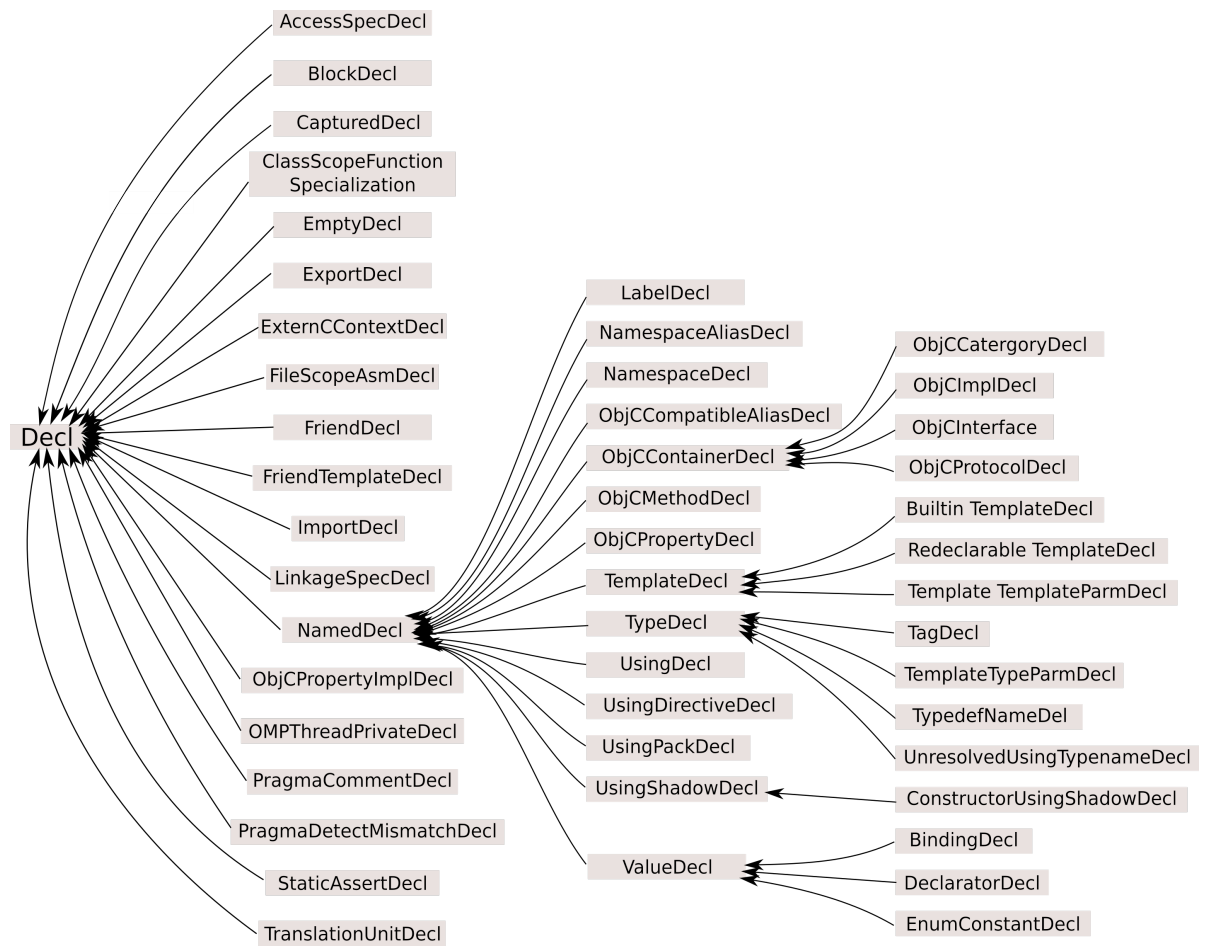


Figure from [Doxb]

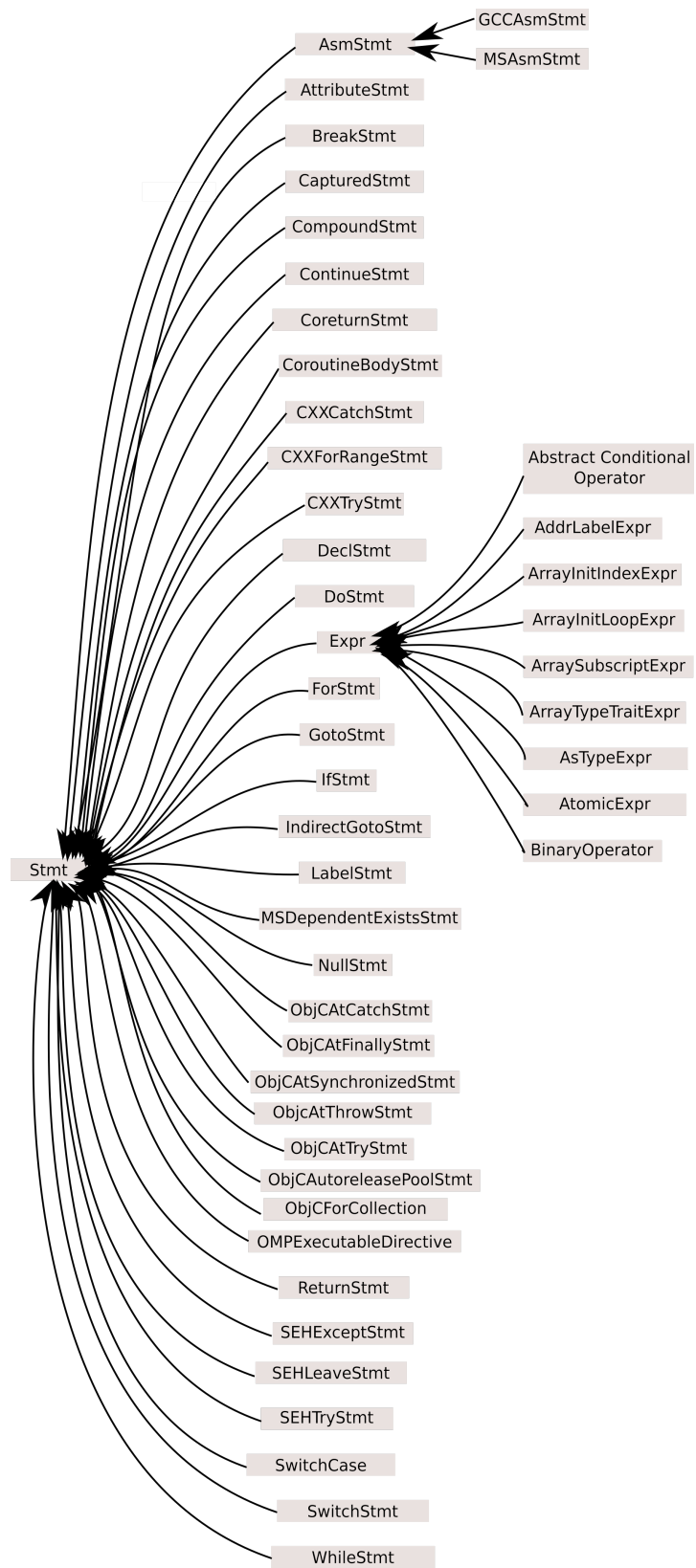


Figure from [Doxd]

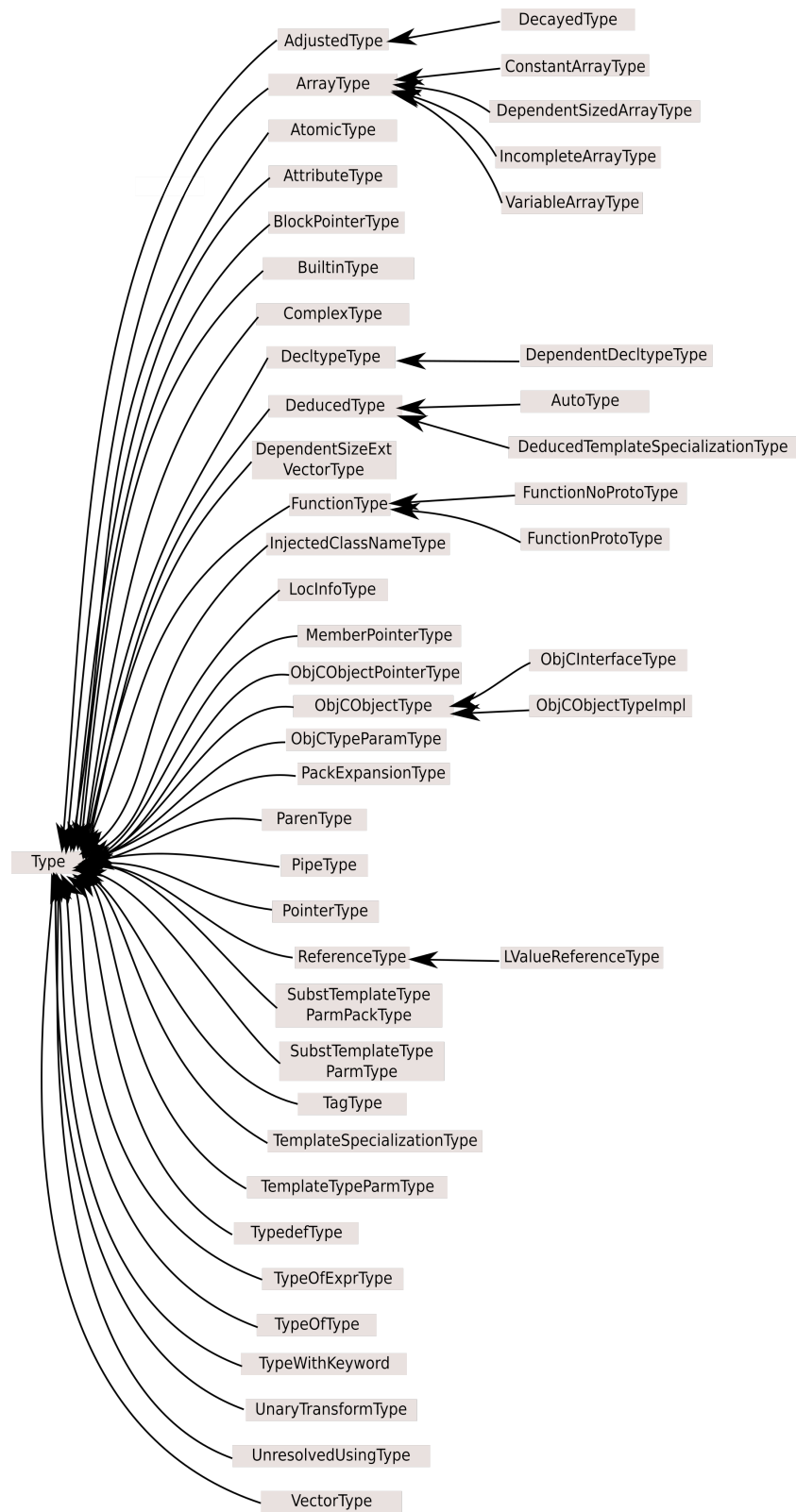


Figure from [Doxe]

- MPI\_File\_open
- MPI\_File\_close
- MPI\_File\_read
- MPI\_File\_write
- MPI\_File\_seek
- MPI\_File\_seek\_shared
- MPI\_Type\_create\_subarray
- MPI\_File\_write\_at
- MPI\_File\_read\_at
- MPI\_File\_iread
- MPI\_File\_iwrite
- MPI\_File\_iread\_at
- MPI\_File\_iwrite\_at
- MPI\_File\_iread\_shared
- MPI\_File\_iwrite\_shared
- MPI\_File\_read\_shared
- MPI\_File\_write\_shared
- MPI\_Get\_count

# List of Figures

1.1	Frontend Process . . . . .	8
2.1	Three-Phase-Compiler LLVM [Lat12] . . . . .	11
2.2	Retarget-Ability LLVM [Lat12] . . . . .	11
2.3	LLVM Compiler Parts . . . . .	11
2.4	Phases of Compilers [DKV17] . . . . .	18
2.5	Measurement of Compile Times . . . . .	18
2.6	AST . . . . .	21
2.7	AST Dump . . . . .	24
2.8	AST with Graphviz . . . . .	25
3.1	MPI-IO Example . . . . .	31
4.1	ElseAfterReturn Diagnosis . . . . .	35
5.1	Path-Sensitive Graph . . . . .	37
5.2	Analyze result . . . . .	39
5.3	HTML Report scan-build . . . . .	39
5.4	Clang with analyze option activated . . . . .	40
5.5	.plist in Xcode . . . . .	41
6.1	Type-Mismatch Example . . . . .	48
6.2	Buffer-Dereference Example . . . . .	48
6.3	Double Close Example . . . . .	49

# List of Listings

2.1	SmallVector . . . . .	13
2.2	SmallSet . . . . .	13
2.3	FoldingSet . . . . .	14
2.4	isa . . . . .	14
2.5	cast . . . . .	14
2.6	dyn_cast . . . . .	14
2.7	Git and LLVM . . . . .	15
2.8	Build LLVM . . . . .	16
2.9	Setup LLVM . . . . .	16
2.10	Clang Command Line . . . . .	19
2.11	Clang Command Line . . . . .	21
2.12	Constant Folding GCC and Clang . . . . .	22
2.13	Compile C . . . . .	22
2.14	Visitor Function . . . . .	23
2.15	Simple Program for the AST . . . . .	23
2.16	Console AST Dump . . . . .	24
2.17	Clang AST with -ast-view . . . . .	25
2.18	Matching Example . . . . .	26
2.19	clang-query Example . . . . .	27
3.1	MPI Basics . . . . .	29
3.2	MPI Communication Mode Formal . . . . .	30
3.3	MPI Communication Mode . . . . .	30
3.4	MPI-IO Example Code . . . . .	31
4.1	All Clang-Tidy And Analyzer Checks . . . . .	33
4.2	Add a Clang-Tidy Check . . . . .	33
4.3	ElseAfterReturnCheck::registerMatchers() . . . . .	34
4.4	ElseAfterReturnCheck::check() . . . . .	34
4.5	Call run-clang-tidy . . . . .	35
4.6	Call clang-tidy . . . . .	35
4.7	elseAfterReturn Example . . . . .	35
4.8	clang-tidy ElseAfterReturn . . . . .	35
5.1	Example for Path-Sensitive . . . . .	37
5.2	Simple Program with Bug . . . . .	38



5.3	Invoke scan-build . . . . .	38
5.4	Invoke scan-build with options . . . . .	39
5.5	Clang with analyze option . . . . .	40
5.6	Display Enabled Checks . . . . .	42
5.7	Display Available Checks . . . . .	42
5.8	Invoke scan-build at MPI error . . . . .	43
5.9	MPI Flags . . . . .	43
5.10	Analyse MPI . . . . .	44
5.11	Analyze based on CMake's compilation database . . . . .	44
5.12	Analyze Based on CMakeLists.txt Option . . . . .	44
5.13	intercept- and analyze-build for MPI . . . . .	44
6.1	Paths MPI-Files . . . . .	45
6.2	Type-Mismatch MPI-IO . . . . .	48
1	LLVM Setup Script . . . . .	58

# List of Tables

1.1	CLOC Results . . . . .	6
a	Lines of Code in IOR-2.10.3 . . . . .	6
b	Lines of Code in hdf5-1.10.1 . . . . .	6
1.2	Total Number of HDF5 Issues . . . . .	7
1.3	List of Compilers . . . . .	9
2.1	Example Flags for Clang . . . . .	19
5.1	Available Checkers . . . . .	42
6.1	MPI-IO Function Classification [Fora] . . . . .	47

## Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

---

Ort, Datum

---

Unterschrift