Universidad Carlos III de Madrid
Escuela Politécnica Superior
Bachelor Degree in Computer Engineering

Bachelor Thesis

Contract-Based Programming on Modern C++

Author: José Cabrero Holgueras
Supervisor: José Daniel García Sánchez

Leganés, Madrid, Spain
July 2018

"Mentors are there in the moments of truth"

Anonymous

Acknowledgements

There are so many people who have been involved in the development of this project. First of all, I would like to thank all my family for all their support and unconditional help, especially over this last year. I could not have borne the load if you would not have been present with all this help. There are not enough lines in this document to thank what you have given me.

I want to especially thank Javier López. I did not think I would live to know a genius, but I did. He has computer science in his veins and transmits his passion to anyone close to him. It is astonishing what he can do with a computer. He has taught me everything he knew when I needed and helped me like no one has in this work. A real mentor is the one that sits next to you and guides you in solving your problems, and Javier has been mine. There is probably no one in the group as dedicated to its task as Javier is. I wish I had had more time to share with him since it has been a real inspiration for this work. If someone deserves credit in the ideas of this project, it is Javier who does.

I would also like to give a huge thank to Manuel F. Dolz. You also contributed to this project offering me to work with you. When I received the offer, I did not think I would earn what I have. It has been an incredible experience. If I look back to September, I was a completely different person, and things that change you, are what define who you are.

I am specially thankful to David del Rio, to Javier Fernandez and to Mario Vasile, because they were present on the last days of this project. Each of them has given something to this project that makes it unique and without their collaboration, this project would not have been what it is. I do not really imagine how this would have ended up without your collaboration. Thank you.

My labmates also deserve some words here. They have been my classmates, my friends and my partners during this journey that has been the degree. Thanks to Laura Martín for helping me in those desperation moments in the lab, you really helped me out. And thanks to Nerea Luna for all those train travels back and forth, your support during all these years is immeasurable. Thanks for talking to me on stress moments and for always believing in me.

Thanks to Saúl Alonso, because he has always guided me and telling me about any opportunities that there are out there; to Javier Prieto, who has cared of me from the very first moment that I joined the group. He has always been there and it has been an inspiration to follow what I really liked to do, and his support at all points. And to Guillermo Izquierdo, who helped me a lot on the decision of the Master Course and I really believe that thanks to that, I will follow my dream. Thanks to you all, you have all given me something that I will always keep, and you have all been an inspiration for me to be better.

Thanks to Ignacio Guillermo Martínez and to Alejandro Rey. They have been my classmates and I could not imagine a better group of friends. Thanks to all the work you put up to help me with practices and for those good moments we spent.

Finally, thanks José Daniel García and the ARCOS group for offering me a job in the department and lending me all the equipment and tools that I needed.

This project has been more than a learning process, it has been a personal growth. Thanks to all of you.

Contract-Based Programming on Modern C++

Bachelor Thesis

José Cabrero Holgueras

Abstract

Contract-based Programming or Design By Contract (DBC) is a discipline for system construction that in recent years has postulated to be one of the most solid and reliable models for software creation. It is well known that in the software industry the number of projects not being successfully developed is huge. The main cause of the failure is that projects do not meet user expectations. In this context, Design By Contract seems to emerge as a solution to decrease this failure rate. This philosophy provides a set of mechanisms for the validation of part of the requirements specification.

In recent years, several programming languages started to implement DBC, either as part of the language or an external library. The main programming languages that support contract-based programming are Ada 2012, Spark, Eiffel, D, C# CodeContracts or Microsoft Source-Code Annotation Language (Microsoft SAL). Traditionally, C++ has been a programming language focused on flexibility, performance and efficiency. This has attracted many people to carry out projects using this programming language. However, trends make programming languages change, and the interests of the industry are leaning towards solid solutions. Those solutions shall include frameworks that are reliable. With this same purpose, C++ has designed a specification for the implementation of Design By Contract in the programming language. This new specification has been accepted by the ISO C++ committee to be included in C++20. The specification includes several clauses that allow the user to write pre/post-conditions on the code. This allows part of the requirement specification to be merged into the code, enabling traceability between the phases of the software project.

The specification of a new feature in a programming language implies changes in how the language is understood by a compiler. For the implementation of a new specification, several changes are required at different levels. This document describes these changes. Additionally, it provides an overview of the structure of a compiler, and a brief description of all the parts of the Clang C++ compiler.

Keywords: · Contract Based Programming · Contract · C++ · Clang · Compiler

Programación basada en Contratos en el C++ moderno

Trabajo Fin de Grado

José Cabrero Holgueras

Resumen

La programación por contratos es una disciplina de construcción de sistemas que recientemente se ha postulado como una de las más solidas y fiables para la creación de sistemas software. Se sabe que la industria de desarrollo de software no está siendo exitosa debido en parte a la tasa de fallos que hay en éstos. En este contexto, la programación por contratos emerge como una solución para reducir esta tasa de fracaso en la industria. Esta tendencia de desarrollo provee a los usuarios con mecanismos para la validación de los requisitos.

En los últimos años, varios lenguajes de programación han comenzado a implementar la programación por contratos, bien como parte del lenguaje o como una biblioteca externa. Los principales lenguajes de programación que a día de hoy soportan programación por contratos son Ada 2012, Spark, Eiffel, D, C# CodeContracts or Microsoft Source-Code Annotation Language (Microsoft SAL). Tradicionalmente, C++ ha sido un lenguage de programación centrado en proveer al usuario con flexibilidad, rendimiento y eficiencia. Estas características han atraido a muchos clientes de cara a utilizar este lenguaje de programación en proyectos. Sin embargo, las tendencías fuerzan cambios en los lenguajes de programación, y los intereses de las empresas actualmente se están inclinando hacia soluciones robustas. Estas soluciones, deben incluir marcos de trabajo que sean fiables. Con esto en mente, se ha diseñado una especificación para la programación por contratos en el lenguaje de programación. Esta nueva especificación, ha sido aceptada para por el Comite ISO C++ para ser incluida en C++20. Esta especificación provee al usuario con varios mecanismos que permiten verificar condiciones en el código. Esto permite directamente enlazar la especificación de requisitos con la implementación de los mismos.

La especificación de una nueva característica dentro de un lenguaje de programación implica cambios en como el lenguaje es entendido por un compilador. Para la implementación de estos nuevos requisitos se requiere de realizar modificaciones en el compilador en distintos niveles de análisis. En este proyecto, se describe un resumen de los cambios que son necesarios dentro de un compilador. Estos cambios incluyen un resumen de la estructura del compilador, posteriormente se desglosa la estructura del compilador de C++ Clang y por último se describen las modificaciones en cada una de las partes involucradas.

Keywords: · Programación Basada en Contratos · Contratos · C++ · Clang · Compilador

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The first chapter exposes briefly what it is going to be treated along the document. In Section 1.1, a little background on the question that is commented. In Section 1.2, the objectives of this document are introduced. Finally, in Section 1.3 an overview of the document is established.

1.1 Background

The correctness of software has been a concern in recent years because of the high amount of software projects failure. It has been found that the lack of a proper requirements phase is the main problem. In this sense, many solutions have been explored around the idea of decreasing this failure rate. Specifically, current solutions focus on the idea of understanding the requirements, from both the consumer side, and the provider side. However, they fail to properly transfer the collected knowledge across all the project phases.

To tackle with this issue, contract-based programming emerges as a strategy aimed at the creation of highly reliable software systems. Design By Contract is built around the high-level idea of contract, i.e. an agreement between two or more parties for doing something. It establishes rights and obligations that both parties shall comply to during the project. These constraints, if broken by any of the involved parties, may immediately cause the contract to be revoked. Based on the previous definition, requirements of a software component may be specified as part of the source code and verified at compile-/run-time.

For this reason, the creation of certain structures that allow the verification of the code is something desirable in a programming language. As a matter of fact, programming languages such as Ada, Spark or C#, are increasingly starting to provide native support for Design By Contract. These structures state the responsibilities within the code, and ensure the proper behaviour of software modules according

to the requirements. This programming approach aims to merge parts of the specification, design and testing into the implementation.

On the other hand, C++ is a well-known programming language usually targeted to high performance applications. Modern C++ has been recognized by the industry as a great language for the development of huge software projects, since it provides a higher abstraction level with respect to other languages like C. Additionally, C++ focuses on efficiency and usability by relieving developers from tasks that are usually error prone such as the memory management. In this project, we provide support for contract-based programming in the Clang compiler.

1.2 Objectives

As stated before, this work aims at supporting Design by Contract in Modern C++. As part of this objective, several secondary goals emerge:

- **O1. Basic Functionality**. The creation of a basic infrastructure that gives the programmer a set of directives whose use implements DBC.
- **O2. Advanced Functionality**. A set of features that allows the programmer to use the tools for further usages such as testing.
- **O3. Efficiency**. The implementation seeks not to increase a lot the execution time of a executable without contract directives.
- **O4. Efficiency of the compiler**. The implementation aims to minimize the overhead that is created on the compiler by implementing the new features over it.

1.3 Document Structure

For the sake of clarity, this document is divided into chapters which are briefly described in the following list:

- Chapter 1, *Introduction*, introduces the motivation and main background this work. It also describes its objectives and the document structure.
- Chapter 2, *State of the Art*, gives an overview on how other programming languages implement Design By Contract and briefly evaluates the implementation in each of those.
- Chapter 3, *Background*, details what is a compiler, the compilation process and the main examples of compilers.

- Chapter 4, *Analysis of the problem*, introduces the general functionality that the project shall have. It presents the requirements specification of the project with a breakdown into user requirements and system requirements.
- Chapter 5, *Design of the solution*, introduces the basic structure of a compiler. Then it is used for the development of the design within the solution.
- Chapter 6, *Evaluation*, starts by evaluating the functionality that was implemented. Then it explains some benchmarks that were created with the purpose of evaluating the efficiency of the implementation. Finally, we provide some concluding remarks on the evaluation.
- Chapter 7, *Project Plan*, includes the overall scheduling of tasks and their deviations with respect to the expected execution time. The methodology used for the project is described and specified in this chapter.
- Chapter 8, *Socio-economic Environment*, shows the breakdown of all the costs derived of the execution of the project.
- Chapter 9, *Legal Framework*, introduces legislation and its applicability to this project. Finally, it details the licence terms under which this work is distributed.
- Chapter 10, *Conclusion*, elaborates on the outcome of the project and its future work.
- Appendix A, *Implementation*, shows the implementation that was carried out to support Design By Contract over the Clang C++ compiler.

Chapter 2

State of the Art

In this chapter, we comment the state of the art, a summary of the different alternatives that allow the usage of Contract-Based Programming in a native manner. First of all, in Section 2.1, we analyse Eiffel the first programming language to offer an implementation of Contract-Based Programming. Later on, in Section 2.2, an evaluation of Ada 2012 is carried out since it is one of the all-purpose language that offers Design By Contract. In Section 2.3, we analyse the Spark programming language, which has Design By Contract as a design principle. Later on, in Section 2.4 we review C# a programming with great importance in recent years. Additionally, in Section 2.5, we analyse D , a programming language which has its origins in C and C++, and provides new features with respect to C and C++. Finally, Section 2.6, we carry out a small evaluation of the Design By Contract library Microsoft SAL.

2.1 Eiffel

In this section we expose what Eiffel is and its main characteristics. Later on in Section 2.1.1, we expose the implementation of Design By Contract in Eiffel. Finally in Section 2.1.2, we make an evaluation of the language.

Eiffel is the first programming language to explain from the aforementioned programming languages for two main reasons. Firstly, because this language implements contract-based programming which makes it interesting to investigate about before dealing with our project. But the fact that attracts the attention is that the creator of this programming language is Bertrand Meyer. This is especially meaningful since the creation of the contract-based programming technique is also attributed to him.

Firstly we will talk about *The Eiffel Method*, a method for software development which aimed to make software development easier. The method is focused on some key really innovative ideas such as:

- Design by Contract. "Defines a software system as a set of components that interact through precisely specified contracts"[1]. Applying this philosophy as a design principle results in a system which, for sure, will provide the functionality as it is needed. Nevertheless, it does not imply the absence of errors. Contracts target the minimisation of bugs and the reliability of the code however they might not ensure that all programming errors are found.

- Single-Product Model. Aims for a software life-cycle process which really complements each of the phases with each other, resulting in giving a different view of the system with each of the views.

The Eiffel programming language process of creation followed *The Eiffel Method*. The programming language was implemented as a tool to support the software creation through *The Eiffel Method*. Therefore, it is important to remark that whenever we talk of Eiffel we are including both the software development method and the programming language. However, from now on in this section, we refer to the capabilities that this programming language provides but omitting the methodology.

Eiffel focused on improving what all its competitors offered. The approach was to create a language in which developers could program much faster, without evident bugs and speeding up the process of software development. The main competitors they found were C and C++, which, in the end, shared the same consumer segment as Eiffel. The objective was to provide similar features than C or C++, avoiding the low-level structure and the associated complexity that they imply to any programmer. The goals that Eiffel tried to achieve on its design are the following:

- Reliability. The reliability as a software characteristic makes difficult to include bugs and programming errors that lead to a more complex development and to an improper integration in a system. The main mechanisms that Eiffel provides to ensure reliability are the following:

  - *Real Static Typing.* The types are known at compile time. It eases programming since allows a reader of the code to explicitly know what type is inside each variable. This approach is the same followed by C++, C or Java. The alternative approach is dynamic typing, featured in languages such as Python or Perl. This makes the type resolution a matter of the run-time environment, and although it provides the language with more flexibility, it is not the best approach for production applications.

  - *Assertions and Design By Contract.* The programming language provides some mechanisms that perform run-time checking. These mechanisms are a solid approach for solving programming errors and guarantee the success of software projects.

  - *Automatic Garbage Collection.* Unlike C and more or less C++ (not considering the new C++ mechanisms such as smart pointers), the memory is a huge concern in current systems. Actually, it is rare to find a program that does not require any dynamic memory mechanism, and

freeing the memory is something to care about. Eiffel provides a unique alternative, a code with a similar performance as C, but avoiding any memory problem that a user can find.

- Reusability. The software that is developed is used for purposes other than the one for which they were originally generated. The following are some meaningful characteristic to support this:
  - *Abstract classes.* These summarised versions of a class provide interface information and contracts information. Through them, any programmer should be able to deduce its behaviour. This also allows the implementation of a class in different manners, adapting them to the necessities of the user on each specific case.
  - *Deprecated Code.* Eiffel proposes a framework where there is a version control of libraries. With it, features that are likely to be removed in a future version of the library are marked. Whenever a user includes something marked in the code, he is warned to be using an older version of the library or a feature likely to be removed.
- Extendibility. It permits features to be easily extended, and make them comprehensible so that anyone needing to modify others implementation finds the least difficulties.
  - *Inheritance.* It allows the elements to be reused and to apply them to the necessities of the corresponding project by the re-implementation of certain methods. This characteristic is also provided in other programming languages, but not directly in C (allows structures composition, though).
  - *Multiple Inheritance.* Eiffel permits inheriting from multiple classes. This is something that not many programming languages provide but that Eiffel actually does.
- Efficiency. It tries to enhance both the development process and the execution stage. Some characteristics to improve it are:
  - *Small runtime engine.* The runtime engine is small so that it doesn't require much memory. In addition to that, it is programmed in C making it low-level and very fast.
  - *Ice Melting Compilation.* Provides a fast compilation method in which the object code is compiled in parts. When the programmer has to recompile the code, it only does it for the parts that have been modified. This makes the compilation process faster and therefore the development becomes faster.

2.1.1 Design By Contract in Eiffel

The most important features that Eiffel included was the Design By Contract implementation [2]. It was something revolutionary that up to that point time had only been seen theoretically. But with its implementation, a new paradigm for software creation was created. It provides three different statements

that allow contracts specification. These statements are sections within classes, that allow the validation of conditions. Those are:

- *Requires.* It is an expression for a precondition. Its behaviour is like an *if statement* that will be evaluated just before starting the execution of the function. If any precondition is not met the execution is stopped, what is all the preconditions shall be met for the execution of the function. The client code is responsible for fulfilling all the preconditions.

- *Ensures.* It serves as an expression of postcondition. It behaves similarly to the precondition but acts on the return values. It evaluates the result of a function and returns if and only if all the postconditions are met. The postconditions are part of the responsibility of the provider. It means that in case any of the postconditions is not met, the agreement is not being fulfilled by the provider.

- *Class Invariants.* A class invariant is a condition that is evaluated on a member of a class. It is a check that is performed every time we modify a class member and it ensures that any modification to that member is going to be proper and it is going to be performed according to some predefined rules. As we evaluate in Section 2.1.2, this kind of conditions are not the most efficient.

2.1.2 Advantages and Disadvantages of Eiffel

In this section, we depict the main advantages and disadvantages that Eiffel supposes against other languages.

While researching for the elements of this programming language, everything seemed to be advantages. This is contradictory since Eiffel does not appear among the most used programming languages. And having such advantages it should be more used than some of the most used programming languages. If we search a bit, we can easily spot some of the biggest disadvantages that make this language not used at all.

The main advantage that Eiffel supposes over C and C++ is memory management, which saves from a lot of errors. In the end, it simplifies the programming to users. That causes that, in principle, people should prefer to use it. Over other high-level programming languages, they do not have the performance to compete with Eiffel.

Another advantage against C++ is the inclusion of class invariants. Class invariants permit a level of restriction over the programming language that is not implemented in C++. However, the main reason for that is performance. Class invariants usually imply a lot of checks and condition evaluations at run-time, that not good in for the performance. It hides to the user a lot of checks that are making the

performance worse.

If we look at disadvantages, the first of them is the price. Although the reference is open to everybody, the compiler that they provide comes associated to an Integrated Development Environment (IDE) called Eiffel Studio which contains the only compiler for Eiffel that exists. Eiffel Studio is free for academic use. As it is mentioned on its web page, it is very used for teaching as a first programming language, because it provides a great basis on good programming practices. The main disadvantage comes when it has to be used for commercial purposes. The licence for commercial use of Eiffel Studio costs per computer from 7000 € to 10000 € which makes it a very costly alternative to be used in a project.

Among other disadvantages of this language, we can find the lack of overriding for functions and operators. The overriding of functions is a mechanic that nowadays implement most programming languages because it allows a great simplification of the code. In addition to that, it doesn't provide any bit level semantics, which makes it unusable for certain applications that require to manipulate bitmaps and require this level of precision. As a matter of fact, it does not provide any support for callback functions, a feature that is more and more demanded and being implemented in modern programming languages.

Something that drew my attention was the critics against the support for advanced language features and the shortage of libraries. This shortage of libraries is especially worrying because it does not allow the creation of applications in different contexts.

In the end, C++ which is the language against which we are comparing now, C++ seems a much better alternative. By the time at which Eiffel was launched, the memory management was very useful, but at this point in time, the C++ Standard Template Library (STL) overcomes this situation. The STL provides an efficient and standard implementation of the most common memory structures called containers. It allows the programmer to use them instead of the classical structures such as arrays or dynamic memory. In addition to that, those structures free the programmer from some tasks that they needed to care about such as freeing the memory or allocating more. All of this functions are seamless and invisible to the programmers. In fact, it adds a feature called *smart pointers*. These smart pointers permit a user having a sole region of memory shared by many structures, and whenever this region is not used anymore, it is automatically freed.

With relation to the overloading and callback functions, C++ provides both, the first by creating a function with the same name, and the second with the creation of a lambda function, a function object or a function pointer. Finally, the most important fact is that C++ is open-standard, and there are compilers free and available for open and public use. And that is what makes a programming language really

used, giving the possibility not only of learning it for free but for enterprises to use it for their projects without a cost. As a little reflection, in a project with five programmers, assuming a salary of 50000€ annually, the total price of the licence of Eiffel Studio for five programmers would be the same as hiring a sixth programmer.

Table 2.1 shows a summary of the main advantages and disadvantages of Eiffel that we depicted in this section.

Feature	C++	Eiffel
Contract support	Current development	✓
Memory management	Manual/Smart Pointers	Automatic Garbage Collector
Pricing	Free (with Free Compilers)	Up to 10000€
Libraries	Great Support	Shortage of libraries
Overriding	✓	✗
Bitmap support	✓	✗
Callback functions	✓	✗
Preconditions	✓	✓
Postconditions	✓	✓
Class invariants	✗- Replaceable by assert	✓
Assertion	✓	✗

Table 2.1: *Comparison of C++ and Eiffel*

2.2 Ada 2012

In this section we describe Ada programming language main characteristics. In Section 2.2.1 we describe Design By Contract in Ada. Afterwards, in Section 2.2.2 we depict the main advantages and disadvantages of the language.

Ada, named after Ada Lovelace[1], is a programming language created by a French company under a contract for the United States Department of Defence. Ada is mainly based on the Pascal programming language [3, 4, 5]. Ada has held several versions of its programming language. Currently, the most recent one is Ada 2012 and it is the version that this section is going to be centred on because it provides Design By Contract implementation.

As some professionals argue, Ada is the most complete language that was ever created [6], that is because Ada has a set of features which makes it suitable for any kind of system. Some features that make this programming language powerful are:

[1] Ada Lovelace is considered to be the first person to write about the idea of a computer

Structured Programming. It is a kind of programming that divides the code into several logical sections. Those sections permit several benefits such as improving the readability of the code and improvements to memory management. This feature increases modularization allowing logical sections to be reused in multiple programs because they are relocated along different parts of the memory. This would later evolve in what we know as the Object Oriented Programming paradigm [7].

Statically Typed. As it was already mentioned in Eiffel (Section 2.1), this type of languages resolve the types at compile-time, meaning that no further check is performed at run-time(contrary to dynamically typed).

Imperative. This type of programming specifies what a program must do by explicitly stating *how something must be done*. The opposite to Imperative Programming is Declarative Programming which specifies a *what a program should perform* without specifying the process on how it shall be done[8, 9].

Wide-Spectrum. It is a programming paradigm which allows a programming language to be used both as high-level and low-level. It applies progressive refinements to the high-level code in order for it to be optimized as a low-level code[10].

Object Oriented Programming (OOP). It is a paradigm that allows users to generate objects which are specific instances of classes. Classes determine which are the main characteristics that we will see in an object, whereas objects are specific classes which determine specific individual instances of a class. This specific paradigm favours reusability and flexibility.

High-level. A programming language of this kind favours comprehension of the code and it is closer to natural language. This principle goes against the efficiency of the code because it is normally thought that lower level programming languages allow a better level of detail, and are therefore seem to be better in performance [11].

2.2.1 Design By Contract in Ada 2012

Reliability is one of the key design principles that Ada followed from its origin. With the aim of pursuing the reliability, the contract-based design appears as a feature in Ada in the version published in 2012. However, previous versions of Ada still provided certain mechanisms to be checked at run-time. Among them we can highlight the following [12]:

- Type Conversion. It checks if the types among which the user was converting were the same or not, and therefore determined whether they were completely compatible or not. This check is usually performed at compile time.

- Parameters passing. Ada allows specifying input and output parameters. This check verifies whether the direction in which parameters are passed is correct or not.

- Range checks. This verifies whether the accessed position of an array is correct or is outside the reserved space. This check is also followed by other programming languages such as Java. This verification implies the execution of the program in most of the cases, therefore, it cannot be performed at compile-time. This differs a lot from the memory model followed in C or even in C++ where memory can be managed freely sometimes supposing problems.

- Sub-types with ranges. When we deal with a sub-type[2] we are also checking the memory layout and whether we are accessing forbidden positions. This check is mostly similar to the check of the arrays and it is usually performed at run-time.

- Static coverage tests. These are some statements that are evaluated before compile-time.

These elements that we have described are some kind of contracts, that already many languages implement without considering Design By Contract. What Ada started providing in its 2012 version as some explicit clauses that allow establishing conditions on different elements of the code. The following contract elements were provided and included in the language in the 2012 release:

- Preconditions and postconditions. They are enumerated conditions that are checked before executing a function in the case of preconditions, and after the execution of the former in the case of postconditions. These elements explicitly establish the rights and responsibilities of each of the parties. They force the implementation to fulfil some sets of requirements and therefore they are a fundamental part of software development.

- Type Invariants. They are equivalent to a class invariant. In Ada, everything is considered a type. These predicates are specified over a private element. In this case, we have a statement that is applied to a private field of our type, resulting in the verifiability of that field fulfilling the pre-agreed conditions.

- Dynamic sub-type predicates are equivalent to class invariants applied to visible subtypes. Since visible types are usually highly used then we will distinguish between static and dynamic. According to the Ada specification [13], we can enable a given dynamic subtype predicate and disable it on demand.

- Static sub-type predicates. This type of predicate performs the same verification of a class invariant but associated to a subtype. Ada restricts the conditions that can be evaluated on this type of predicates. This enforces that the checks are not huge and take a lot of time. The specification

[2]In Ada every variable has a type, and when we deal with a sub-type is equivalent to dealing with a subclass

of Ada establishes that the main difference between static and dynamic is that static sub-type predicates are evaluated always and cannot be deactivated.

Figure 2-1 shows a small classification of how Ada type invariants are classified into subsets according to the conditions that they allow to execute.

These elements allow us to generate complex subtypes such a type representing prime numbers, something that would ensure the satisfaction of some conditions and would, in fact, increase the reusability of certain components.

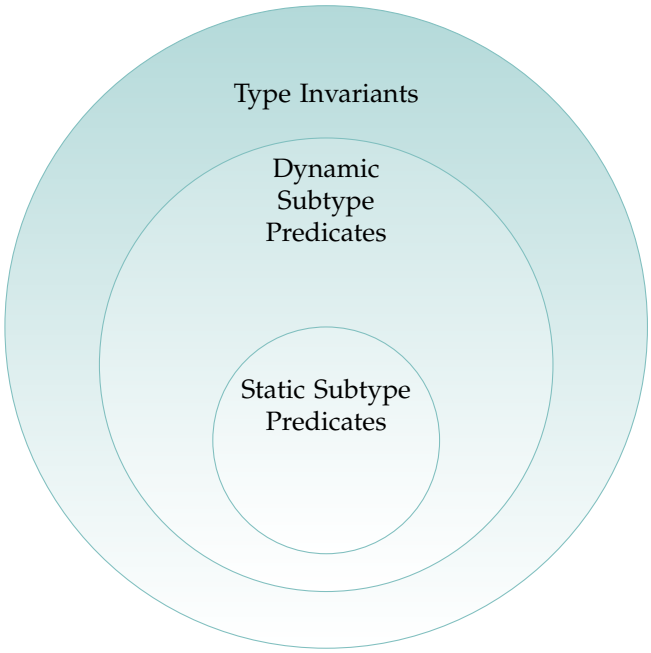

Figure 2-1: *Ada 2012 - Type Invariants Classification.*

2.2.2 Advantages and Disadvantages of Ada 2012

This section describes the advantages and disadvantages found in Ada 2012.

At a first sight, Ada seems to be a more complete programming environment in what contracts based programming regards. In relation to our implementation, Ada provides more alternatives that allow static and dynamic checks or as in Eiffel the Class Invariants alternative.

First of all, the implementation is very confusing even within the reference, not being able at all times to express which are the differences between one type of predicate and another.

With that regard, the C++ implementation has a more clever and simple solution in order to activate and deactivate the contracts. It establishes certain levels of assertion, with which every contract is annotated. Given the levels of assertion, we can establish when to activate each level of assertion. Although

C++ allows more flexibility when establishing the assertion level, a bad programming practice may make inadequately achieve a good performance, for example, making the code run a huge assertion in cases where it is not necessary. On the other hand, Ada 2012 is less flexible, provides limitations on when to use static subtype predicates which in the end improve the performance.

Actually, elder features that Ada implements are the most relevant with respect to what C++ provides. Even though it is not the most efficient alternative, they try to avoid frequent programming errors. Several relevant programming errors are the vector length check or the type range tests. The most similar alternative that we have is the usage of the STL containers which perform these evaluations.

With regard to what are C++ general advantages and features, we can find several advantages. First of all, we can observe the pricing. Although as language, Ada is a "open standard" language, the compiler, GNU NYU Ada Translator (GNAT), is only free for academic purposes and for Free/Libre and Open Source Software (FLOSS) projects. For the rest of the cases, C++ supposes the great advantage of not having to assume any cost for a compiler since there are free compilers. In particular, history agrees on this point, because Ada was one of the most prominent languages. Nevertheless, the fact of being a paid compiler did not permit users to start using and learning it. The price of the cheapest compilers was over a 100$ per licence. In that situation, the communities looked for other languages, such as our alternative (C++) where there are free compilers. The basis of having free compilers permits a community to grow around a language, by learning it unitedly and participating in its development[14, 15].

What most people claim to be the biggest error of Ada was its main design principle, reliable and solid programming. This assumed that a programmer is going to introduce errors. This idea was taken to the extreme and developed into making the language so strict that it also became inflexible. As a result, it prevented programmers from doing wrong things, but it also became hard to make things right. From many programmers point of view that is its main flaw. It tried to be the most secure and reliable language. It tried to overcome many programmers mistakes. But it ended up constraining the programmer. In conclusion, Ada makes a programmer focus more writing syntactically correct code rather than on the actual programming task.

Table 2.2 shows the comparison of the main characteristics with its advantages and flaws of C++ and Ada 2012.

Feature	C++	Ada 2012
Philosophy	Programer knows what he does	Programmer commits errors
Commitment	All-purpose	All-purpose
Objective	Efficiency	Reliability
Pricing	FLOSS	Not Free
Preconditions	✓	✓
Postconditions	✓	✓
Assertion	✓	✗
Type invariants	✗- Equivalent to assert	✓
Static Subtype Predicates	✗- Equivalent to assert	✓
Dynamic Subtype Predicates	✗- Equivalent to assert	✓

Table 2.2: *Comparison of C++ and Ada 2012*

2.3 Spark

In this section we describe Spark. Specifically in Section 2.3.1 we describe the Design By Contract in Spark. Finally in 2.3.2 we develop a summary of advantages and disadvantages of this programming language.

This section introduces Spark, a language that was initially developed from Ada Core and has from that point on been in a parallel line to the development of it. Spark focuses on the development of highly reliable tools. As the main objective of the language has always been to provide a very solid framework in which data integrity and reliability were always ensured, contract-based programming was a feature that has always been included. The main objective when creating Spark was to elaborate a set of instructions from the Ada full set of instructions that was non-ambiguous at any point. With that Spark enables the formal verification of any code generated with that instruction set.

In contrast to Ada, that proposed an all-purpose programming language, Spark appears to be a specific language for some specific and counted applications. It provides a similar approach as some POSIX Profiles of the C programming language[3] or Java different versions[4].

[3]The C programming languages POSIX profiles, such as PSE50, PSE51, PSE52, PSE53, restrict the features available on the language for embedded systems making them easily verifiable and certifiable. In addition to that, they allow the removal of some features from the language that might not be necessary for a specific project and allow the development of a lightweight application.

[4]Java, offers three different versions Enterprise, Standard or Micro Editions that provide either complete versions, general purpose version or reduced versions respectively.

Spark provides several important features, namely[16]:

Flexibility. It provides compatibility of a newly built software in Spark with legacy systems implemented in Ada or in C. This feature permits switching old code to a new language without compromising the availability or previous functionality of the system. This feature is an incentive intended to ease switching a project from a programming language into Spark with no compromises at all.

Ease of Adoption. It tries to differentiate itself from other approaches by claiming to be one of the most reliable languages. The utilisation of the contract-based programming allows it to be highly verifiable and reliable, being applicable to industry sector where it is a basic requirement the need of verification in the language.

Powerful Static Verification. With static verification we refer to the verification of the correctness code at compile time. It means that all kinds of analysis are applied to the code to prove its correctness without needing to execute it. In fact, it can prevent and ensure that no run-time errors will appear. Spark implements the usage of a mechanism that allows partial pre-compilation. In addition, Spark allows a mathematical verification of the code. It allows to mathematically prove that a code elaborated with this language will not throw any exception. That is something critical and that not many languages can provide.

Reduced Cost of Unit Testing. Unit testing is usually a job that programmers have to develop by themselves by making each and every one of the tests by hand or at least specifying the test cases. Unit testing allows justifying the correct behaviour of a software. In the case of safety-critical software, this cost supposes a great part of the budget of the process since, otherwise, it would not be trustworthy and could not be certified for certain applications. Spark Pro provides a framework which is able to generate all the unit test suite. They justify that the tests are generated automatically if some annotations are placed on the objects of the testing.

2.3.1 Design By Contract in Spark

This section describes the Design By Contract implementation that Spark provides. It is divided into two main divisions before Ada 2012 and after it.

As we already mentioned in the Ada section, previous to Ada 2012 the support for Design by Contract was not included. The strong links that Ada and Spark have did not affect this feature. Spark required very powerful verification tools, therefore, it implemented from the very beginning of the Design by Contract. Contracts previous of Spark previous to Ada 2012 were described as comments were represented by the following structure “ --#”. Contracts in those versions fall into two categories:

Core Annotations. They are related to data and information flow. They specify which should be the flow of the program and how the relations of the program with global variables have to work[17]·

Proof Annotations. These are the formal annotations of Design By Contract with which we have dealt in this document so far. They verify the compliance of the code to any document. They are normally used for formal verification but they can be used for mathematical verification. The two main types are preconditions and postconditions[17].

With Ada 2012, Design by Contract was introduced as a new feature, and it allowed to introduce a new set of features that allow further checks on the syntax of a program[18, 19, 20]. There are a few of them which include several interesting features:

Subprogram Contracts. These contracts assertions are the main ones. They include several basic statements (preconditions and postconditions) along with some more advanced elements that allow a great control of the runtime environment.

- Preconditions. They ensure that the value of some element is correct before the procedure starts. In Design By Contract, it is considered that the Preconditions compliance is a task of the calling procedure.
- Postconditions. They are assertions which ensure that the value returned by a procedure is inside the acceptable range specified in the contract. The implementation is in charge of fulfilling them.
- Global Variables. When modifying this kind of variables in code, we have to take care of the value since it might affect several procedures elsewhere. With regard to that, Spark provides a mechanism for protecting the global variables in procedures from reading, writing or both permissions. This allows deducing further information on how information is treated by only reading the prototype.
- Information Flow. Some very useful elements that, establish dependencies among the elements in the function, giving further information to the compiler. This is something that ensures that a variable of the program depends on several input variables.

Loop Invariants. They are assertions that ensure that a loop is being performed properly. It checks, e.g. the whether the control variables of the loop are leaving the expected range of values, or whether those same control variables are being incremented or decremented on each iteration.

Testing and Proof Contracts. Spark includes the ability to use several directives similar to preconditions, postconditions and regular assertions that are only used in testing environments. They pursue a better coverage of the tested software and the ability to include heavier proofs.

2.3.2 Advantages and Disadvantages of Spark

This section describes the main advantages and disadvantages that Spark supposes against other alternatives.

As we have seen, Spark includes some very interesting features that might be very important for C++. With regard to the function that preconditions, postconditions and assertions provide, it is pretty similar to what our implementation provides. But it is interesting to analyze loop invariants, these elements that might be saving some efforts from programmers task. There are two main concerns with regard to these elements. The first whether they are interesting enough to be implemented in C++. And if the implementation of the former would comply with the C++ philosophy to introduce features in the language.

C++ provides an Assert Statement, which assesses a condition at a certain point of the code. The loop invariants evaluation of the code can be beneficial since it is included near the loop statement. But in the end it ends up being easily transformable to an assert statement and provided that C++ does not implement a notation for ranges, it ends up not giving any advantage with respect to the usage of asserts.

With regard to whether the implementation follows the philosophy given to C++ contracts, the answer is no. C++ philosophy in what respects to features included in the languages is always tending to efficiency and giving the best performing applications. In that sense and although C++ contracts, a priori, do not improve the efficiency of the code, they are pretty flexible allowing the programmer to place them in order not to hurt performance. Loop invariants try to provide reliability, and reliability and efficiency do not always cope properly. This means that assuming a check on each iteration of the loop is not always assumable. Depending on the application, loops can suppose many iterations and that means considering that the code of the contract is executed that many times.

With regard to testing and proof contracts, C++ offers an equivalence. In C++ there are several assertion levels. Those assertion levels are included in the contract and the specification provides some guidelines on which assertion level to use in each case. Compiling with a build level or another will ensure that some contracts are checked or not. In the same way that Spark offers some directives that will be executed only under a testing or a proof environment. In that manner, heavy proofs are only included in testing versions whereas release versions do not include it.

Table 2.3 shows the comparison of C++ and Spark with respect to their features, advantages and disadvantages in a summarized tab

Feature	C++	Spark
Philosophy	Programer knows what he does	Programmer commits errors
Commitment	All-purpose	High-reliable and secure systems
Objective	Efficiency	Reliability
Pricing	FLOSS	Not Free
Complete Features	✓	✗
Preconditions	✓	✓
Postconditions	✓	✓
Assertion	✓	✗
Global variable protection	✗- Replaceable by assert	✓
Loop invariants	✗- Replaceable by assert	✓
Information Flow Statements	✗	✓
Runtime Programming Checks	✗	✓

Table 2.3: *Comparison of C++ and Spark*

2.4 C# - CodeContracts

This section describes C# as programming language and its main characteristics. In Section 2.4.1 an overview of the CodeContracts library is given. Finally in Section 2.4.2 we summarize the advantages and disadvantages of the programming language.

C# is a programming language from the family of C-like programming languages. CodeContracts is a library of C# which provides support for Design By Contract. It is a simple, object-oriented and type-safe programming language. The main characteristics that make C# interesting are:

Component-Oriented. Similarly to the concept of OOP, this idea arises. It simplifies the way of constructing software creating components that perform a function. A difference usually between a component and a class is that a component is a self-contained and self-describing package[21]. The main difference is that a class can provide a functionality in relation with other classes and not be self-contained whereas a component implies self-contention and the functionality included in it being completely independent of anything else present elsewhere. In fact, it determines a paradigm called Component Object Model (COM) which provides tools for generating Components in many other languages including C#. Furthermore, it allows the execution even remotely of a component

and creating components in other programming languages.

Garbage Collection. It is an automatic mechanism that acts to free the memory whenever the program reaches a point where some variable or region stops existing. This implies that we do not have to care to free the memory however it creates a heavier runtime environment. As in Java, the garbage collector avoids many programming errors. In addition to that, it is important to remark that in order for the programming language to provide this service, it transforms objects in array-like structures that as some programmers complain, makes difficult to care about memory management. This transformation makes hard to guess in which region of memory a variable is being allocated. Or when dealing with big applications, which is the memory region that has been assigned to any of the elements.

Exception Handling. When an unexpected behaviour appears in the code, an exception is raised and the user is able to act and manage that exception. It is contrary to the idea of returning error codes.

Unified Type System. Creates a unified type hierarchy in which all the types inherit from a common type *object*. This allows generating a set of common operations that all types must support. All the characteristics are inherited and can actually be user-defined if they are declared in the *object* type.

Versioning. This feature claims to solve problems that many programming languages have to deal with when any of its components is updated to a newer version. At that point, some of the characteristics that were offered are no longer available. This means that some of the objects might require a re-implementation. For this purpose, C# offers the keywords *virtual* and *override*.

2.4.1 Design By Contract in C# - Code Contracts

We always have to relate to Code Contracts, when we address Contract Based Programming in C#. Code Contracts is a library that composes part of the diagnostics engine of C# and enables the user to establish contracts that are checked over some conditions. There are three kinds of contract directives offered in C# [22]:

- Preconditions. Contract condition evaluated before executing a function. The user is in charge of fulfilling the conditions in order for the function to execute correctly.

- Postconditions. Contract evaluated after the execution of a function. The responsibility of the contract accomplishment is part of the software developer.

- Class Invariants. Conditions that are checked at the class member level. They ensure that the value of any of the members is going to be fulfilling some conditions. This class invariants must be true for all instances. We consider a class instance to be correct if all the class invariants are met.

- **Special Postconditions**. Those special postconditions allow the user to act on *old values* of the variables. It would permit a user the elaboration of a postcondition which evaluates how the value has changed over the execution of a function. It also offers the user the alternative of putting postconditions over the return value of a function and the pass-by-reference values that we transfer to the function. It uses a special syntax which makes it very meaningful. It is important to remark that special postconditions have limitations. This means that some of this syntax to refer to *old values* and *return values* might not be interoperable.

The elements that C# provides are the same that other programming languages already provided to this project. The interesting point of this language is the set of features that come associated with this contract execution and its evaluation, namely:

Automatic Testing Tools. When developing the testing phase, contracts allow to remove meaningless tests by comparing the preconditions and postconditions with the object of each of the unit tests. This is a feature that enhances, especially in big projects the time spent in testing, something that is very useful to integrate within other programming languages.

Static Verification. It is a feature that makes an analysis of the conditions of the code and even before the execution it allows the user to know whether some contracts are failing or not. As we said it only evaluates certain contracts that it is able to execute at a pre-compilation time such as implicit contracts (contracts that the language already evaluates by default such as array boundaries or null accesses) and explicit contracts (just in case that the evaluation is possible).

Dynamic Verification. Also known as runtime verification, it consists of checking the contracts while the code is being executed. The classical way of doing this procedure was using *if statements*. With this alternative, it allows a simpler syntax and more focused on the purpose of code verification.

Reference documentation. The preconditions, postconditions and class invariants are all used when generating the documentation for the sake of completeness.

2.4.2 Advantages and Disadvantages of C# - Code Contracts

This section evaluates C# and its implementation of contracts. It evaluates the advantages and disadvantages of using it.

When we have to evaluate C#, we find as in the case of Ada an all-purpose programming language. But in contrast to all the counterparts, that we had with Ada, with C# we find that it is a very useful language. In C# we find a huge community and a vast source of information. It has one of the best and new-user-friendly documentation that we can find about a programming language. C# provides many functionalities to perform anything that the user desires, without compromising the user freedom.

With regard to contract-based programming, its framework is very complete, providing few but very powerful functionalities. Something that is very interesting to see, is the automatic generation of documentation from the contract that the users write. Another of the great advantages that the implementation that C# with respect to ours is the simplicity for the representation of the *special postconditions*. Those include a specific syntax, that is better for the programmer and reader of the code.

It was already discussed the fact that class invariants are not the best performance-wise alternative but it helps as a functionality for high-level languages. As a substitution for class invariants, the assertion in our implementation is a great substitute which would remove certain checks from some functions.

Something that C# also provides is the ability to remove all the contracts from the code whenever its needed by defining a *preprocessor directive*. However, this is better resolved in the C++ alternative permitting the elimination of only part of the contracts.
[23] Finally, we remark the main counterpart is the fact that the CodeContracts library depends a lot on the Windows platform. Among the complains that users have are also related to this. It is not that C# is a bad programming language, it is that .NET, the platform that executes the C# code, slows the code making it not comparable to native code. What C++ strives is looking to create implementations of a code that could compile on several platforms, generating native code.

Table 2.4 shows the comparison of the characteristics of the C++ and C# with CodeContracts.

Feature	C++	C# - Code Contracts
Commitment	All-purpose	All-purpose
Multi-platform	✓	Multi-platform centered in Windows
Platform	Native	.NET
Preconditions	✓	✓
Postconditions	✓	✓
Assertion	✓	✗
Class Invariants	✗- Replaceable by assert	✓
Generation of Documentation	✗	✓

Table 2.4: *Comparison of C++ and C# - Code Contracts*

2.5 D Programming Language

This section describes an overview of the D programming language. Specifically Section 2.5.1 describes how is Design By Contract implemented in D. Finally Section 2.5.2 describes the main advantages and disadvantages of D.

The D programming language is an evolution of a C/C++ programming language with a syntax sim-

ilar to Java. The origin of D is having an efficient programming language such as C or C++ but without constraining the user on having to write low-level code[24, 25]. D has been used by many important enterprises such as Netflix, Facebook or eBay for their projects since it provides a great alternative for an easy-to-write and efficient programming language. D main characteristics are the following[24]:

Object Oriented Programming. It provides with an interface similar to the C/C++ syntax to create *classes* that can encapsulate different attributes. The instances of a *class* generate an object. It also implements *operator overloading* with which complex classes can be treated similarly to basic data types.

Functional Programming. It gives a user the ability to work with functions in different ways. Functional programming is a way of programming with functions at different points of the program. The D programming language gives a user the possibility to use *pure functions* (the usual basic functions), *inmutable types and data structures* (classes whose member functions do not modify the calling instance but return a new instance of the modified object), *lambda functions* (functions that can be declared within a context in order to be later on used with different names).

Productivity. The possibility to use modules in substitution of libraries is something that many programming languages are trying to implement in newer versions. It also permits using generic programming by means of templates.

Functions. D uses a similar concept to lambda functions which are the term of *dynamic closures*, which permit the access to reference variables of other contexts within a called function. The *in, out and ref parameters* are a kind of parameters that eliminate the necessity of using pointers since they determine which is the usage that parameters are going to use in the functions.

Arrays. D arrays carry with them the dimensions of the array. In addition to that, *strings* have direct support in the language and therefore we can create them as a separate object. This allows to explicitly create an array of strings without the need of having to use *char ** types which are harder to control for the programmer.

Ranges. It provides an interface to the user that allows him to specify sequences of values.

Resource Management. One of the most important points, it implements Resource Acquisition Is Initialization (RAII) so that user avoid finding *null references*. In addition to that it implements garbage collection, custom memory management and low-level programming.

Performance. It is part of what we called down and dirty programming. It provides the possibility of including explicitly written assembly language between the usual D code by using no special directive.

Reliability. D programming language provides an interface with which the user is able to create contracts. This will be detailed in section 2.5.1.

Compatibility. Given that the C community is bigger, it gives access to the C functions by implementing cross-compatibility with this two languages. This allows that D uses any C library.

Project Management. D brings us with the ability to generate multiple versions of a program with the same text.

.

2.5.1 Design by Contract in D

In this section, we will detail the Design By Contract approach that D follows.
The Design by Contract approach in D is completely different from any other language. It is implemented through a set of directives which are executed before and after the execution of the body of the function. These directives enable execution of arbitrary code, thus being quite powerful. There are three different kinds of directives that are applicable to functions:

- *in*. The *in* clause defines a context where the *preconditions* shall be placed. It permits the execution of any code before the beginning of the function. Since preconditions are not supposed to modify the behaviour of the function unless the parameters are wrong, it has no possibility of modifying the arguments of the function.

- *do*. The *do* clause defines a context where the actual body of the function shall be placed.

- *out (result)*. The *out* clause defines a expression for *postcondition*. It defines the context of the function that is executed previous to the return to the calling function. It receives as parameter *result* which is a variable holding the result of the function. With that variable, we are able to evaluate the correctness of the return value. This region is delimited to be used for contract postconditions placement and should not modify the result that the function returns.

We have remarked the fact that the result of the code on the regions of *in and out* shall not modify the actual function execution. The main argument supporting that is, that the release version might not include those regions. In such case, any functionality included in those regions would be deleted, resulting in an incomplete code.

With respect to class contracts, we can apply class invariants. Class invariants, as in other languages, protect the validity of the members of a class. Class invariants are declared under the class body and implemented under the clause *invariants*. As with the previous clauses, they can include any kind of

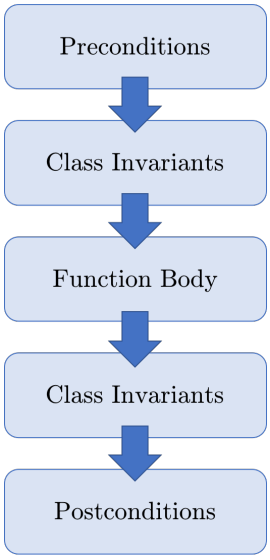

Figure 2-2: *D - Order of contracts execution*

code related to inner members of a class.

In the case that any function modifies any of the members of a class, in order to speed up the verification of the fields, the validation of them is only executed at the beginning and at the end of the functions that modify them. The strict order is first preconditions, then invariants are checked, later on, the function body is executed, next, the invariants are validated again and finally, the postconditions are executed.

Figure 2-2 shows the order in which contracts are executed in D programming language when a function is executed.

Due to the nature of the implementation, no other keyword different than *assert* is included in the language. This clause stops the execution of the program raising an exception in the case that the condition under its brackets is not correct. The main reason not to include any other directive is that we can put any *statement* within the *in, out and invariant* clause simplifying the implementation a lot[26].

Finally, we comment that as in other languages, D provides some implicit contracts built inside the language. This is what we called implicit contracts in C#. Those kinds of contracts are related to protecting the boundaries of arrays or protecting the transformation of types. As we see, this is a trend that many programming languages tend to follow.

2.5.2 Advantages and Disadvantages of D

In this section, we depict the advantages and disadvantages that the usage of D provides in a project.

When evaluating D as a programming language it seems like we are evaluating C++. That is because of the similarity that both programming languages share. D shares many of the things that differentiate C from C++ and it makes it very resembling. Among the main advantages that we find in D with respect to C++, we find some elements that suppose a hop ahead of C++[27, 28].

Compile time execution is one of those features. C++ enables users to make compile-time execution by means of the metaprogramming. Although the framework is very complete in C++, D provides a much complex framework giving access to more advanced features. In addition to that, D provides a simpler syntax for arrays management. It has a garbage collector that on the one hand, makes easier the learning process for newcomers to the language. On the other hand, the garbage collector is as much as possible avoided. Even experts in the use of this language try bypassing it. We already commented that garbage collectors are not a good option since they end up wasting resources, and D's garbage collector is not an exception.

With regard to the contract implementation, it is controversial how some features are delivered to the user. First of all, we take a look at having to use contexts to define the code of the preconditions and postconditions. It permits creating an explicit process of validation of parameters before executing a function. The execution of a code makes the condition and the purpose of preconditions and post-conditions wider and does not close it into a specific requirement which is the purpose of Design By Contract.

However, what is more contradictory, is the implementation of class invariants. They are used for the validation of the members of a class. However, they are validated with a piece of code that surprisingly is only executed at the beginning and at the end of functions. Although it is true that in this paradigm the values of a function are usually modified inside the procedure, it is easy to depict cases where the manual modification can lead to some error.

To conclude, we can say that D is a pretty good programming language for some public, however, its design purpose is not clear at all. Firstly, because it claims to be a replacement of C and C++ because of its performance, and then sacrifices efficiency by introducing a garbage collector or some checks in the arrays. On the one hand, it gives the user many alternatives to protect itself from low-level code. On the other hand, it ends up being an abstraction over C and C++, so anyone deepening enough in the language will find necessary to use pointers and some other features that try to be restricted.

Table 2.5 shows a side by side comparison of C++ and D programming language summarising what

we have commented.

Feature	C++	D
Commitment	All-purpose	All-purpose
Design principle	Efficiency	Ease for programming / Efficiency
Domain	Low-Level	High-Level
Garbage Collector	✗	✓(Automatic and Manual)
Contract specification	Clause	Function Envelopment
Preconditions	✓	✓
Postconditions	✓	✓
Assert	✓	✓
Class Invariant	✗- Replaceable by assert	✓

Table 2.5: *Comparison of C++ and D*

2.6 Microsoft Source-Code Annotation Language (Microsoft SAL)

In this section we will introduce Microsoft Source-Code Annotation Language (Microsoft SAL). Later on in 2.6.1 we will comment the advantages and disadvantages of using Microsoft SAL.

Microsoft SAL, as its own name indicates is a notation language used to describe the parameters that a function uses. Microsoft SAL is not a new programming language as any of the examples that we were talking about before. Instead, it is an extension to the C++ programming language. Microsoft SAL enables the compiler to automatically make checks on the code[29].

The code verification is done through the Visual Studio Framework. In order to speak about this, I will make a little incise to illustrate some precedents about that. Even though C++ is considered a multiplatform language because it is used in both Linux based and Windows operating systems, it is not as interoperable as it might appear. The main reason for this lack of interoperability is actually the Operating System. We have some common mechanisms in both OS, such as the STL (which is a multi-platform implementation). But the wide majority of features differs from one to another. The OS directory structure, the treatment of elements or the availability of libraries conditions the way in which programmers develop according to the platform.

In this way, there are different libraries that are only available for the Windows release of C++. And given that this programming language is extensively used in the Windows context, the company controls how the implementation of it works. In this case, the usual way for developers in Windows to use C++ is with Visual Studio. It is an IDE which provides a complete environment for C++ development in

Windows platforms. There are two versions of the IDE, the one we are interested in is only Windows compatible because it is included within the full environment.

In fact, the two main purposes that Microsoft SAL[29] provides are the following:

- *Automatic Code Validation.* The annotations provide the analyzer with information that later on can be used to guess whether those assumptions are being fulfilled or not.
- *Improvement of User Readability.* This annotations that are giving information to the compiler, are at the same time giving information to the programmer. With this information, the programmer can infer what are the things that can be done with the elements passed as parameters.

For example,lets assume that a programmer that is implementing the *add* function, that receives two parameters *a* and *b*. Both parameters are annotated with *_In_*, which means that both are only input and have to be treated as read-only. We would be making two assumptions, on the one hand, the code analyzer would be checking whether there is any statement violating the condition and writing on any of this two variables. Additionally, the programmer would be able to know if he is doing something wrong and analyze the state of its function[30].

Microsoft Microsoft Source-Code Annotation Language is a static code analysis technique that is included in C++ for Windows. We will now focus on what static analysis of code can provide us. On the one hand, it is a powerful technique, which can discover errors even before having to compile the code. On the other hand, the fact of not compiling it makes it inequivalent to runtime execution testing. The information that annotations give to this compiler allow to infer many conditions. For example, given that we are passing the length of an array to a function, the static analyzer would be able to analyze whether at some point the code is going out of its bounds. In fact, since we are not dealing with the same clauses than in other cases, the static analyzer is powerful enough to check the validity of each and every condition that the user establishes. Anyways, as it is always said, the absence of errors does not justify its proper behaviour[31].

2.6.1 Advantages and Disadvantages of Microsoft SAL

In this section we depict the main advantages and disadvantages of the implementation of Microsoft SAL.

Microsoft SAL provides a completely different alternative than the contracts that we are used to dealing with. It gives the user the ability to establish assumptions over the functions (behaviour and

parameters), classes, structures that it is going to develop, and later on, the static analyzer and the programmer itself is able to validate those assumptions. Whether it is better or worse can only be specified in each particular case. It is different and enables the user with a different set of tools.

It is important to remark at this point the differences between our implementation, and this one. This implementation works properly to check the code for the available annotations, but it is by no means a generic annotation language. It provides structures that allow validating a wide range of cases, but there are some that the user would not be able to validate simply because of the need of execution. In case we wanted to validate for example whether a computation is made correctly, this alternative would not be useful.

In the same way, the implementation of the contracts lacks some conditions that can check at runtime. For example, the ability to check the calling function without modifying largely the code. Nevertheless, the runtime implementation always holds more flexibility since it permits a wider range of checks than a static analysis execution. It is true that static analysis is the only possible way to know whether the code will fail at some point, however, runtime analysis is valid for a wide majority of programs.

In fact, the Microsoft SAL implementation tries to fulfil some necessities C++ is missing. C++ does not implement any boundary check on the arrays, however, Microsoft SAL enables a user to statically check if for some reason is having an *array out of bounds* exception. Or the case of modifying a reference. In such casuistic, the user would not know if the data with which he invoked the function has been modified. With these kinds of annotations, he can be sure of what to expect of a function.

Another huge disadvantage of this framework is being Windows only. Although Windows is highly present, it is a paid alternative, and we have to take it into account. Of course, contract-based programming is for particulars, but it is more an approach for big enterprise projects rather than little ones. So pricing has to be taken into consideration.

In conclusion to this section, depending on the characteristics of the project, we will have to deal with different problems, depending on the characteristics, both frameworks could even be used at the same time. However, the big controversial fact about this set is the fact of being Windows only and therefore supposing a cost per licence.

Figure 2.6 shows the comparison between C++ Contracts and Microsoft SAL.

Feature	C++ Contracts	Microsoft SAL
Flexibility	✓	Limited Features
Number of clauses	Few	Many
Code Analysis Type	Dynamic	Static
Condition Specification	✓	✗
IDE Dependent	✗	✓(Visual Studio)
Operating System	Windows, Linux, Mac	Windows
Orientation	Compiler Oriented	Code Analyzer

Table 2.6: *Comparison of C++ and Microsoft SAL*

Chapter 3

Background

In this chapter, a description of the main background needed to develop the project is described. In Section 3.1, a background of the design of a compiler is given. Later on, in Section 3.3 a description of what a generic compilation process is depicted. Then in Section 3.4, the GCC compiler is overviewed. Then in Section 3.5 an detailed description is reviewed in Clang Compiler. Afterwards, in Section 3.6 a comparison of GCC and Clang is performed evaluating them. Finally, in Section 3.7, different compilers are

3.1 Reasons to Modify the Internals of a Compiler

This section discusses the benefits of implementing the proposal inside the compiler. These are:

- Introducing a new syntax is not possible unless it is correctly parsed by the compiler.
- Contracts might be perfectly taken into account in static code analysis.
- Class inheritance shall be taken into account.
- Efficiency is a must.

For all these reasons, a modification to the compiler internals is required.

3.2 Compiler Definition

In this section, we depict the concept of a compiler and its definition. Apart from that, we explain several types of compilers.

> A **compiler** is a tool that is used to transform from a language into another language. It is usually conceived to transform a program from a high-level language into machine code.

We usually define machine code as the native code that an architecture is able to interpret to make operations. It is usually assembler code even bytecode. Although the main objective of a compiler is to make the translation, their job has become complex since the transformation requires some checks in the meantime. A compiler evaluates the code looking in it for any kind of error, and trying to solve it.

Usually, we tend to deal with compilers for the creation of object code, however, compilers are not limited to this field of action. The usage of compilers covers a huge different set of disciplines:

- When we deal with interpreted programming languages such as *Python* or *Javascript*, we are dealing with **interpreters** which are a kind of compilers which analyze the code for its validity and later on they execute it.
- **Source-to-source** compilers translate from one high-level language to another high-level language, for example translating from Java to C.
- **Cross-compilers** are in charge of generating code for a different platform. For example, the Arduino platform works this way. The platform is not powerful enough to compile its own code, therefore we use a compiler in order to generate from a computer the code that then is sent to the device by a *serial port*.

3.3 Compilation Process

In this section we describe the generic compilation process of a compiler and its subparts. Specifically in Section 3.3.2, we detail the behaviour of the preprocessor. Later on, in Section 3.3.3, we break down the lexer behaviour. In Section 3.3.4, we explain the compiler parsing process. Afterwards, in Section 3.3.5, we overview the Semantic Analysis process. Next, in Section 3.3.6, we detail the code generation process. Lastly, in Section 3.3.1, we break down the behaviour of the compiler driver. Finally, in Section 3.3.7 and 3.3.8, we review the functions of the Abstract Syntax Tree (AST) and the Symbols Table respectively.

According to what we previously said, we would define compilation as the action of compiling a program, however, we will treat it as the process of transforming a high-level code into machine code. Usually, when we speak of compilation there are two main parts, the front-end and the back-end. When we speak of front-end, we usually refer to the part of the analysis of the code and generation of intermediate code. We refer to intermediate code since we create an intermediate representation of the code from which we are able to generate machine code for different architectures. After the generation of the intermediate code, we usually perform some further optimizations over assembly code and perform the transformation of the intermediate code to architecture specific code. The compilation refers to the actions carried out in the front-end and within the front-end, there is usually a subdivision of the formerly

called analysis phase which covers the first three phases of the analysis.

We can distinguish 2 main types of compilers:

- **One-pass compiler**. In this kind of compiler, the code is analyzed by reading and going through it one time. In this case, the compiler generates the machine code from the source code.
- **Multi-pass compiler**. In this kind of compiler, some intermediate representations of information are created. In this way, we refine the representation of the language and we are able to perform a proper analysis. According to experts, compilers tend to be safer if the analysis is done multi-pass way. In fact, these intermediate representations permit the elaboration of further abstractions which are more adequate for optimizations. For validation of the former, it is always easier to validate smaller pieces that work together than a big compiler doing everything at once.

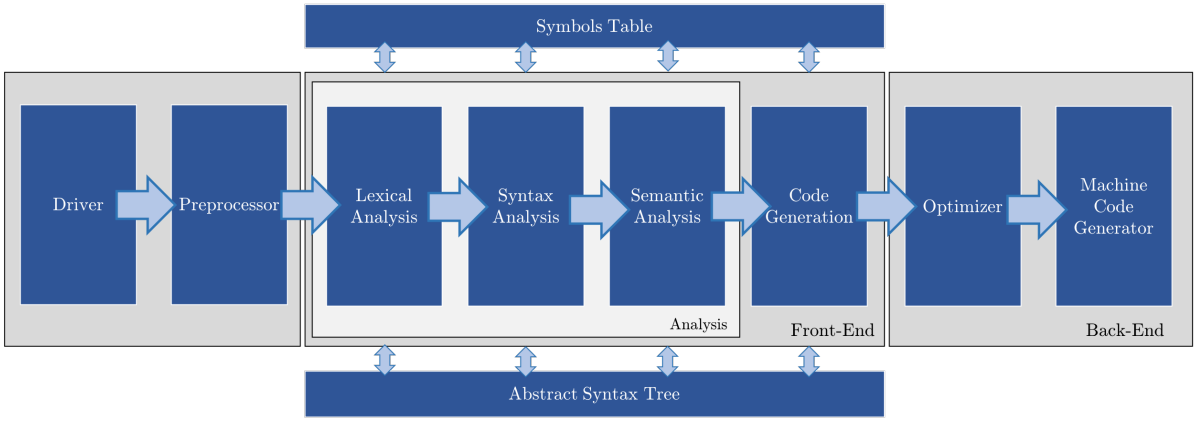

Figure 3-1: *Generic Compiler Execution Flow*

Figure 3-1 shows the generic flow of a compiler. The figure shows clearly what is considered the front-end and the back-end. This scheme is going to be followed and described in following sections.

3.3.1 Driver

In this section, we describe the main functionality that the Driver provides in a compiler.

This element does not have much relation to the process of analysis of the code. However, we are not only interested in the theoretical implementation of a compiler. We are also interested in the details that usually a compiler programmer is interested in.

The driver is the first element of the compiler to be executed. It is in charge of making the execution of each of the other elements of the compilation. All the compilation options are directed to it, and it is

in charge of managing those depending on where they are targeted to. With that we can distinguish two cases:

- If the compilation option is targeted to the Driver, it will be handled by the Driver and removed from the list of options.
- On the other hand, we have other types of options which target other steps of the compilation. Those options are managed by the driver and left in lists which will later on go to the corresponding phases of the compilation.

Another of the responsibilities of the driver is to open the files to be compiled. Whenever we try to compile something that does not exist in the file system, it is usually the driver who warns it.

Finally, the driver is able to use different toolchains. A toolchain is the set of actions to be carried out in order to obtain a compiled code in a specific architecture. Depending on the Operating System, the Driver identifies the proper toolchain and launches each of the elements of that toolchain.

After the execution of the Driver, the Preprocessor is executed over the files that we have just treated.

3.3.2 Preprocessor

This section describes the Preprocessor behaviour, it happens right after the execution of the Driver and deals with the file it just opened.

This phase is not considered part of the compilation, it is a pre-compilation stage for the code. The preprocessor is in charge of substituting elements in the code so that the compiler can understand them.

One of the elements that is under its responsibility is looking for preprocessor directives and transforming them. More in the C world than in the C++ world, it is usual the usage of preprocessor directives. For example, by means of the clause # *define* we are able to generate an easy substitution for a value. What the compiler does with these elements is looking for them in the code and substitute them by their equivalences. The result of this phase is a code with no equivalences.

In fact, the function that we are interested in preprocessor is the import of libraries. Libraries are declared with the clause #*include*. When the preprocessor finds any of the clauses, it looks for the library name in the include directory paths and pastes the code of the library directly in the place where the #*include* clause is.

The result of this phase is going to be the same as the source code but with all preprocessor directives already substituted in the code so that the lexer can transform it.

3.3.3 Lexer

In this section, we describe the functions of the Lexer or Lexical Analyzer, which generates a transformation from the preprocessed file.

The lexer is also known as the lexical analyzer is a subsection of the compiler which is in charge of creating subunits of code by reading the code. The lexer is in charge of reading the source code, and with some information given in advance, it tries to create the different small units that we can, later on, compare with the symbols in the grammar. Those basic units are usually called tokens. This constitutes the first phase of the code analysis.

In the lexer, we usually have a table with equivalences. What the lexer does is picking characters, one by one, and try comparing with any of the equivalences of the table. Whenever it finds an equivalence, it creates a *token* of that type.

There is usually a token per reserved keyword in the language. In addition to that, we create a special token for each different operator. Finally, what remains unclassified are usually variable names and constants. The variable names are usually assigned identifiers and the constants are usually stored with its same name.

Though we are creating an abstraction over the language, usually, we are not interested in losing detail from the code so we still store the string of code that gave the result to this *token*. We usually try to keep the information that later on can be useful for any diagnosis engine or the compilation.

To perform the lexer process, we usually take into account either a grammar which includes all the types of tokens or regular expressions. The grammar allows us to differentiate each type of token and the characters.

Lexers in some high-level languages have to fight a problem when finding the tokens and that is ambiguity. In C or C++ we tend to use some operators in a non-distinctive way. We use operators differently depending on the utility, for example, multiplication and pointer access use the * operator. The problem is usually ambiguous to the lexer for certain strings. The problem relies on the fact that we introduce a chain that might be interpreted in many different ways and depending on the context it means different things. One of the examples of this chain is *(X)*Y*. This chain without further information could be interpreted in different ways, such as the multiplication of two variables (*X * Y*)or the dereference of a

pointer (*(X)(*Y)*)[32].

The most widespread solution is to implement some contextual information feedback from the Semantic Analyzer to this section. In this way, we are able to analyze which is the context of the token and solve it. Another solution is the possibility of lexing multiple times the file in case that the parser finds inconclusive the search in this way. However, one of the most well-known alternatives is the carried out by the Clang Compiler where it uses its own library and its own knowledge to be able to deduce which is the kind of token it is dealing with[33].

After this section, a sequence of tokens has been generated from the source file. This sequence of tokens needs to be processed to check whether it matches the grammar. This process is done by the Parser.

3.3.4 Parser

In this section, we talk about the parser. The parser is in charge of verifying that the code is compliant to the grammar.

We usually call Parser to the Syntax Analyzer of the grammar and constitutes the second phase of the code analysis. Oppositely to the idea that we have of the compiler as a pipeline, it is far from that conception. It usually goes back and forth trying to find structures that fit the grammar that in principle was defined and governing the behaviour of the compiler. Compilers tend to be Parser-Driven in the sense that it is the parser who demands things to other components of the compiler.

The parser asks the lexer for tokens. It uses those tokens to, in the order that they were received, to compare them with the grammar. The grammar specifies the order of tokens that are admissible for the programming language. However, there are multiple ways in which a grammar can be analyzed. We can distinguish two main classes of parsers:

- **Top-down.** This kind of parser goes from the axiom making derivations over all the productions in order to match a production rule. This is usually implemented in a form of *Recursive Descent Parser*. A recursive descent parser suffers from backtracking. Although recursivity is not something good at all it is difficult to find implementations of compilers not using this. A *top-down recursive descent parser* is constructed from a tree from the input which is read from left to right until it generates a terminal symbol that matches. If it does not find it, it backtracks and tries with other derivation. There is a kind of *Recursive Descent Parsers* called *Predictive Parser* which generates all possible derivations and in that way, it avoids having to backtrack.

- **Bottom-up.** This is a kind of parser that evaluates from the result tokens and tries to fulfil the

grammar from the leaf nodes up to the root. If it is able to reach the root node from the leaf nodes it means that the sentence is correct and accepted by the grammar.

As we have seen what both of this types of parsers do is analyzing an input and trying to reach the output from it. In case that for some reason it is not able to find a production that satisfies the input in the grammar, then the compiler warns an error. If a sentence does not match a production, it does not strictly mean that it is not correct, as we have seen there might be several productions from a non-terminal symbol.

In case that we want to add something to the language which does not follow the syntax supported by the grammar, then changes should be made to the production rules of the grammar.

Once we end this phase, we are sure that the sentences that we have are syntactically correct. However, some statements might not be correctly placed in some context. In the following section, we describe the Semantic Analyzer which is in charge of analyzing the context information of the code.

3.3.5 Semantic Analyzer

In this section, we describe the Semantic Analyzer. The Semantic Analyzer is in charge of validating the context information of the code.

The semantic analyzer has a wide functionality. It constitutes the third and last phase of code analysis. It takes the context information and evaluates the sentences checking if they are correctly placed in the code. It might happen that we are dealing with an expression that is correct from the point of view of the grammar, however, the expression is not properly placed in the programming language. For example, placing an executable statement outside a function in C or C++. Among the main functionalities that it has we can see the following:

- **Type checking.** We have already talked for long about the type checking. It is a process which consists of evaluating everything related to a type. The semantic analyzer evaluates if the assignation of variables and values are being properly done. It tests whether a type is being operated with an incompatible type.
- **Type compatibility.** It compares expressions and checks assignations in case that is made from different types checking that both are equivalent.
- **Polimorphism checking.** It evaluates whether an instance of a class is correctly taking the form that it should. It goes one step ahead of type compatibility, it is done with inherited classes when we want to give a subclass the aspect of a superclass. It is not always compatible since it depends on whether the members that it holds inside have the same size as the superclass.

These are some of the most important characteristics that it checks for, however, it does not stand there. Semantic analyzer does among other things the following: type coincidence, declaration of variables, reserved identifiers usage, multiple variable declarations within the same scope, out-of-scope variable access or parameters type coincidence[34, 35].

At the end of this section, the code is already correct. With this code, the Code Generator is able to create the intermediate representation.

3.3.6 Code Generator

In this section we describe the Code Generator, whose main functionality is to generate the intermediate code.

The Code Generator is in charge of transforming to intermediate code, however, this would limit the ability of the compiler to generate code for multiple architectures. That is why the code generates an intermediate representation. It is usually assembler-like code that keeps all the information that might be needed for the back-end to generate machine specific code. Apart from that one of the main advantages of intermediate-code is that it is designed to make optimizations easier. In fact, if everybody is able to optimize the intermediate-code the results are better for every architecture and not just one [36].

Summarizing this phase, it picks any of the sentences that have been proven to be correct and generates the equivalent sentence in intermediate code. The compiler has some predefined structures that allow the conversion from a high-level code sentence into intermediate-code statements.

But the sentences are not picked directly from the code. They are picked from the Abstract Syntax Tree (AST), which is an intermediate structure that the front-end generates and the Code Generator uses to carry his function. The AST is described in next section.

3.3.7 Abstract Syntax Tree (AST)

In this section we describe the concept of Abstract Syntax Tree (AST) which is a fundamental structure of the compilation process.

The Abstract Syntax Tree (AST) is a tree that holds the structure of the program by means of nodes. Each node holds the information of the code and the characteristics of it. It is present everywhere since almost any component could theoretically extract information from it. However it is normally is managed by the parser, from there the concept of a parser-driven compiler.

The creation of it is done in the parser. The order of action of the compiler tends to be the following:

1. When the parser needs, it asks the lexer for a token and the lexer provides the parser with a token.
2. The parser verifies the token with the grammar.
3. Once the compiler is able to match a sentence with the grammar, it invokes the semantic analyzer to check for the correctness of the statement.
4. If everything goes fine, the semantic analyzer has a successful analysis and the parser generates the corresponding nodes.
5. Then, the parser is in charge of adding those nodes to the tree below the corresponding parent node.

Each node of the ASTholds enough information from the source so that when it is needed it can be retrieved and so that we are not losing detail when any transformation to it is performed.

With this in mind, we do not have to be generating code as we receive sentences, we just store it temporarily in the AST and later on whenever we need it, we can extract information of it. Actually, the fact of it having the shape of a tree allows extracting context information by looking at the parent and child nodes. Given all that, the process of code generation just imposes the read of the AST in a certain order and generating the information in it.

3.3.8 Symbols Table

In this section, we describe the Symbols Table which is a fundamental structure of the compilation process.

Whenever we deal with variables, we can get a common error "Variable 'x' is not declared in this scope". This error is mainly due to the symbol table. It is a structure which stores all the information regarding the variables, objects that we create in our program.

It stores all kinds of information inside to be able to follow variables during its lifetime. Some of the information that it stores is the scope where variables exist, its access restrictions (for private class members), the type of a variable, the line where they were declared (in case that someone wants to use a variable before it was created), etc...

In fact, the symbols table includes some clever mechanisms to follow the redeclarations of types that we can create on the classes and structures. Whenever a new type is created it stores information on that

type and links it to the types it already knows. In the beginning, it links it to basic data types such as *int, char, float or double*, however, once the language starts to be richer, the symbols table recognizes all those types.

3.4 GNU Compiler Collection (GCC)

In this section, we describe an overview of GNU Compiler Collection (GCC), which is one of the most extended compilers.

GCC is the default compiler that GNU Operating System use. This compiler is intended to work with *C, C++, Objective-C, Objective-C++, Fortran, Java, Ada and Go* along with many of the internal libraries implementation of those languages such as *libstdc++*. In this case, we are interested in the working for the C++ language since it is the language that we are dealing with in our project. In the first versions of the compiler, GCC was implemented as a Look-Ahead LR Parser (LALR) for some *front-ends* but recently it has switched all its front-ends to be a Recursive-Descent Parser. The intermediate code is only generated in some cases where it is needed, in the rest of the cases, the machine code is generated from the AST.

GCC is usually divided in 2 phases, *front-end* and *back-end*. The first phase is the *front-end* which provides a specific parser for each language. The *front-end* of each language generates the AST. The AST is already a standard representation from which GCC is already able to generate a specific architecture code.

However, in some cases, GCC generates a intermediate code representation in two different languages named *GENERIC* and *GIMPLE* [37]. In the origins of GCC, instead of using these modern representations, a more basic language was used, called Register Transfer Table (RTL). This alternative was later replaced because it supposed an advantage in terms of low-level optimizations. That is mainly because Register Transfer Table had some disadvantages such as the inability to deal with structures and arrays and the early introduction of the stack limits. *GENERIC* is a language-independent representation used for intermediate code. This language is generally used for the representation of functions in trees. *GIMPLE* is another language-independent representation that is a subset of *GENERIC*. Its purpose is related to code optimization since *GIMPLE* retains the structure of the parse trees, lexical scopes and control structures [37].

With regard to the *back-end*, it is in charge of generating machine code and usually calls the linker if an executable is the desired objective. Since the *front-ends* are dependent on the specific language, we

need a specific *front-end* for each language. However, GCC has only one *back-end* since the *front-ends* generate a common specific representation.

GCC is probably the most well-known compiler for C. It usually comes preinstalled with most Linux distributions (distros). GCC is mostly programmed in C and C++. As many people argue it is a complex task to modify it since it requires a deep knowledge of it. In addition, there are few documents on how its internals work. This added to the complexity of modifying a compiler makes it a not so proper alternative for the inclusion of new features.

3.5 Clang

In this section we describe Clang, the main alternative to GCC in what C/C++ language free compilers respect. Specifically, in Section 3.5.1, we describe LLVM, the intermediate language representation over which Clang is built. Later on, in Section 3.5.2 we describe the internals of Clang.

Clang is another of the great alternatives which is present nowadays for compiling C and C++. The Clang project aims to provide a *front-end* for C, C++ Objective-C, Objective-C++, OpenCL C and others. As we already know, the *front-end* is usually a part which is in charge of generating the intermediate representation which in this case is in IR. Whereas the *back-end*, LLVM, would be the part in charge of generating the executable.

Although the usual alternative and default option for many Operating System distributions is to use GCC, there is one well-known operating system that uses Clang, that is Mac OS. Clang interest has grown in recent times from many well-known enterprises such as Google, Apple or Microsoft. Those enterprises have gained a lot of interest over this compiler and LLVM because of the modularity, its relative ease of use and the possibility of implementing new tools thanks to its interface.

As in GCC, Clang provides an implementation of the C++ Standard Library. The C++ Standard Library is just a specification in which the ISO C++ committee establishes the guidelines and features that it should guarantee. Since Clang design principles vary from the implementation offered by GCC, they implemented a different version of the C++ Standard Library.

3.5.1 Low Level Virtual Machine (LLVM)

In this section we are describing Low Level Virtual Machine (LLVM), which is a intermediate language that the Clang compiler uses.

LLVM is a project born to develop a low-level compiler which optimized the alternatives that were present. Its objective was to create a language were optimizations were able to be carried out independently from the source and also independently on the output machine.

The project includes a programming language called Intermediate Representation (IR). This programming language is a low-level assembly-like representation which holds a lot of the information from the original programming language. It ends up being a very complete assembler code. The ability to retain all that information allows that later on the transformation of the code into machine code is done in a consistent way and without losing the detail that was expressed in the code.

In fact, it has another huge advantage. Usually building the own compiler for each programming language, would mean that we have to build the *front-end* and the *back-end*. The problem of the *back-end* is having to implement all possible transformations of the chosen intermediate code representation into a machine code representation. In case a compiler for only desktop computers was developed, this would imply at least supporting *Intel* and *AMD* processors with *amd64*. This would mean having to re-implement all the transformations of our intermediate code into only one machine code language but taking into account the differences of implementation of both manufacturers. And even in that case, we would be leaving out a set of processors that are more than present nowadays such as *ARM*, *MIPS* or *PowerPC*.

LLVM provides with a tool that everyone can use for its programming language. It eases the implementation of a programming language in a compiler allowing anyone to have its programming language used in many platforms. This means that we would have just to implement a *front-end* transforming to IR, and we would be obtaining all the optimizations and the ability to transform to a wide variety of platforms independently from the source programming language.

3.5.2 Clang Internals

In this section we describe the Clang Internals, that is, the different modules that comprise it[38]. In Section 3.5.2 we describe the Clang Driver. Later on, in Section 3.5.2 we give an overview of which are Clang analysis phase. Followingly, in Section 3.5.2 we describe the Clang Preprocessor. Then, in Section 3.5.2, we analyze the lexer implementation in Clang. Next, in Section 3.5.2 we show the function of

Clang Parser. Afterwards, in Section 3.5.2 we describe the Semantic Analysis in Clang compiler. Then, in Section 3.5.2 we look at the concept of AST and the concept of visitor. Finally in Section 3.5.2 we look at the code generation.

Driver

In this section, we describe the Clang Driver module.

As in a normal compiler, the first element to appear in the scene is the Driver. In Clang is the main process launched and it is there for the remaining of the execution. The Driver is in charge of parsing the arguments and it also extracts platform-specific characteristics for the compilation. The platform information is extracted for it to, later on, be able to choose a toolchain. A toolchain is the set of guidelines that the execution suffers in order for it to adapt to the different platforms where the compiler is available. For example, we could be setting up the compilation to affect differently depending on whether we are on a Windows on a Unix computer. Now we detail the steps that are carried out in Clang during the Driver execution. From the information extracted from the Clang webpage the steps followed by the Clang Driver [39] are the following:

1. **Argument Parse**. It takes all the arguments and parses them into specific objects so that the compiler analysis can treat them with ease. These arguments are later on passed to the specific toolchain. The Driver chooses depending on the argument whether to pass it or not to the next phase. In addition, this serves to transform the attributes for the specific toolchain something that is equally done in GCC.

2. **Compilation Construction** . The compiler creates a compilation from the input arguments line. From the files obtained at first in the compilation line, it establishes which are the necessary actions to carry on with each of them. The set of actions is usually a set of steps followed by a compiler in order to compile a source file.

3. **Tool and Filename Selection**. This step is one of the most complex of the Driver selection. From the list of actions to be carried out, the compiler infers which tools are going to be performing each action. Once everything is done, the Driver loads the different toolchains to use the different tools that they need. A normal execution might require Tools from different toolchains.

4. **Tool Specific Argument Translation** . Once we know which are the different arguments that are needed for each of the phases and the Tools that we are going to use, the arguments are passed to each of the tools. The main reason for doing this in this step is that each tool is independent and generates input and output. Thus, the arguments affect only specific tools.

5. **Execution of the actions**. When we know all the steps, to be carried out the different execution pipelines are executed for the different files.

Analysis Phase

In this section, we overview the analysis phase of the Clang compiler.

With all the previous knowledge in mind, we can now start talking about the analysis phase of the compiler. The analysis phase takes several steps and involves several classes that we are going to try to handle in a straight way. Although the compiler pipeline usually starts by the Lexer, Clang works differently and all start with the Parser. The process in which Clang does the compilation is the following:

1. **Parser**. Clang compilation process starts with the creation of a Parser. We already commented that it is very common to have "parser-driven" compilers. Clang creates the Parser. The Parser is in charge of generating the different modules needed for the compilation. The parser main function is to generate the AST.

2. **Basic Compiler Phases Generation**. Before starting the Parser, a Sema object is initialized. In Clang the Semantic Analyzer is usually called Sema. In Clang, Sema validates the nodes that are later on inserted in the AST with the information that the code provides. Next, a Preprocessor is created. Although we have talked of Preprocessor outside the analysis phase, in Clang it is taken and considered inside the compilation. The Preprocessors main function is to translate the input file substituting all directives and including all libraries that are necessary. The important fact about the Preprocessor is that holds the Lexer inside. When we are tokenizing the file, it is the preprocessor who holds the Lexer object inside.

3. **Parse and Code Generation**. Once we have all the elements needed, the compiler analyzes the code by top-level declarations. A top-level declaration is any declaration that we can find global in the code. The simplest example of a top-level declaration is a global variable. The compiler parses each of the top-level declarations generating the AST and once it is ready, the Code Generator generates the code the corresponding node. This is done for all the top-level declarations that it finds.

4. **Final Generation**. When compiling, we are not always able to generate the code for some elements the first time we go over it. Those symbols are delayed and are generated at the end of the compilation and once everything else has been parsed. That is because the compiler needed some information that when parsing the first time was not possible to find. This phase is executed once all the code has been parsed.

5. **Code Generation**. At the same time as we were parsing the Code Generator, CodeGen in Clang, we were generating an intermediate representation in Intermediate Representation (IR). This intermediate representation can be transformed into machine code.

Figure 3-2 shows the Clang compiler execution flow with the modifications that it has from a usual compiler.

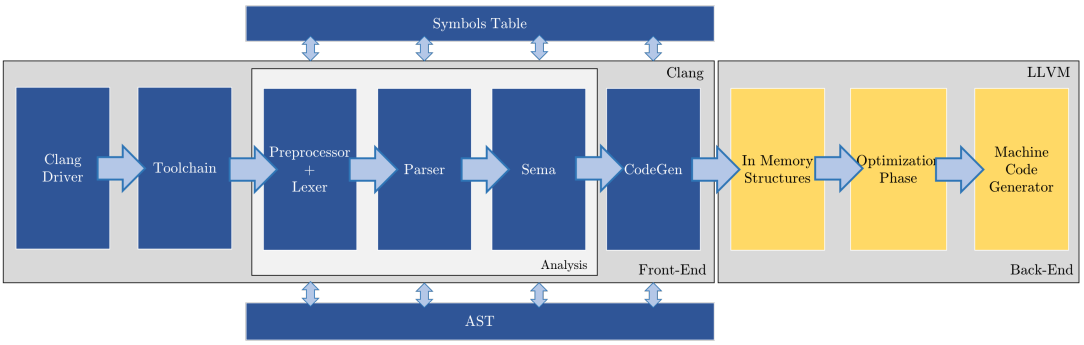

Figure 3-2: *Clang + LLVM Compiler Execution Flow*

Preprocessor

In this section, we describe the Clang Preprocessor.

The Preprocessor is the main class representing the management of files up to the beginning of its analysis. Clang uses an abstraction layer for files which is similar to any usual file system. It creates different file objects and identifiers which allow to open and treat each of the files involved in the compilation process in a specific way. The object used to identify the files is usually the *FileID* object. However, the preprocessor creates an abstraction through the SourceManager object so that whatever uses this file does not have to deal with the specific characteristics of the object.

The Preprocessor class is also in charge of creating the Lexer in Clang. The Lexer class is contained inside the Preprocessor class and it is the Preprocessor class who governs the Lexer by initializing its execution.

After the execution of the Preprocessor, the Lexer is analyzed.

Lexer

In this section, we break down the main functionality that is present in the Clang Lexer process.

The Lexer class is in charge of performing the Lexical Analysis of the source code. The Lexical Analysis consists of the transformation of the different characters into meaningful sequences. It is performed

by the class *TokenLexer* that picks character by character trying to match each of the identifiable sequences of characters. Once we are able to match one sequence it creates a Token object. As it was described in section 3.3.3, the Clang compiler features a special Lexer which takes into account the context information in the creation of the Tokens. This avoids having to deal with information flows from Sema to this part. The *Token* object implements several methods which make it able for us to compare with following tokens and the type of the token. This type of the Token allows us to compare it with the grammar.

Since the Lexer in Clang is kind of mixed with the preprocessing of files, both are carried out at the same time, but by changing the parsing mode. In case that the compiler finds a # symbol, it will switch the mode to *ParsingPreprocessorMode*. In this mode, the reading of the file changes and takes into account different parameters such as the End Of File (EOF) being switched to End Of Directive (EOD). Another of the modes is *ParsingFilename* which is used when we find a *#include* directive. This allows that we perform at the same time the Lexing and Preprocessing of a file.

Parser

In this section, we analyze the Parser and its main functionality.

The Parser class is one of the fundamental classes in the compiler. It is in charge of verifying the grammar. It is done by successive condition evaluations whenever a Token is received. It contains the recursive-descent parser.

Although the responsibility is reduced to checking the grammar, the Parser controls the rest of the execution.

The Parser task is closely related to Sema actions. Actually, most of the analysis is performed by these two classes. If we look at it from the code perspective, the Lexer is just in charge of generating tokens that it finds in the code. This creation of tokens, unless some evident errors, will not validate the code. Instead, the main task comes from the side of the Parser and Sema.

Sema is described in the following section.

Sema

In this section, we describe Sema, the class in charge of performing the Semantic Analysis of Clang.

The Semantic Analysis is a crucial step in the process of compilation of Clang. It has two main tasks, first, it is the generation of an AST node from the information received both from the Token and the Parser. With that information, Sema is able to instantiate nodes with very different characteristics.

The first phase is the validation of the different pieces that Sema receives from the Parser and Lexer. The Sema verification implies whether the node is correctly placed within the context. This information is usually further information that the grammar cannot verify, for example, whether the attributes are associated to correct types of functions or if the values of each of the fields are correctly placed.

After having performed all the checks, the Sema generates an instance of the node being analyzed and returns it to the Parser so that it adds it to the AST.

C

Abstract Syntax Tree (AST) and Visitors

In this section we describe two important concepts. First of all, what is the Abstract Syntax Tree (AST) and secondly what is a AST Visitor.

In Clang, one of the most interesting structures is the AST. It is programmed to hold all the information from the source code. This is of a huge importance when we relate it to the generation of code. Storing all the information in the node permits us later using this information to accurately generate anything.

One concept to take into account at this point is the concept of Translation Unit Declaration. It is a AST node is important that is the root node of a compilation. The Translation Unit Declaration Node is created at the beginning of compilation and from it are appended to the rest of the nodes. This concept is important to take into account when we deal with the code generation because, as we will observe in section 3.5.2, is performed in two steps.

Another concept to take into account is the concept of Top Level Declarations. Any Top Level Declaration is a declaration whose parent node is the Translation Unit Declaration. Again this is important because of the compilation steps. The compilation takes first into account the Top Level Declaration and in the last step takes into account the Translation Unit Declaration.

Before deepening into the task that CodeGen provides, we need to mention the term visitors. What has made Clang a well-known compiler is its ability to make tools over the interface that it provides. Visitors are a great example of this. Visitors give the programmer the capacity of generating a tool to make an analysis of the code very easily. They provide an interface through which we can traverse a generated AST for any code and perform local operations on nodes of our interest.

In fact, this tool allows an easy modification of the code. With this technology, we would be able to easily create a cross-compiler by transforming the structure of any node to the destination language.

This tool is a generalization of what in fact Clang does inside it within the CodeGen functionality.

A visitor is a special kind of metaprogrammed element. It works by specifying functions for each kind of the statements. For the vast majority of them we might not be interested in visiting them and for them, an empty implementation inherited from the parent class. However, we can override those implementations to visit and execute custom code for the nodes we need.

CodeGen

In this section, we describe the process carried out by the Code Generator in Clang.

If we recap the concept of visitors, is because a visitor is an abstraction of what CodeGen is. CodeGen is a complex implementation of a visitor. CodeGen is called in two ways:

1. **Top Level Declaration**. The Parser and Sema are in charge of generating each of the Top Level Declarations. Once each of them is generated, CodeGen generates the IR code for everything that is able to do. This means that for example the instantiations of template functions of C++ might not be translated directly at that point since no usage might have been registered yet.

2. **Translation Unit Declaration**. Once every of the compilation elements have ended, and the Parser and Sema have already processed all the code, the emission of the elements that could not be generated previously is done. In this step, CodeGen generates the instantiations of template function or delayed generation elements.

The functionality of the CodeGen is performed in a similar way as a visitor that recursively visits all nodes, then checks for validity of the node and in case that it considers it proper generates the equivalent LLVM code.

The CodeGen module contains all the implementation of the different translations from a node to the IR code.

3.6 GCC vs Clang

In this section, we describe the main advantages and disadvantages of using each of the compilers described before.

With this, we already have a little background on what compilers do. Although this is a more or less detailed explanation, it reaches no more than the tip of an iceberg of what a compiler is.

With these two compilers, we have to choose for the implementation of the new features. In this section, we will observe the advantages that one supposes against the other and we will take a look at some performance evaluations taken over both compilers.

At a first sight, we can observe that our explanation of Clang is much wider and deep than GCC. The main reason for that is the documentation. By far, Clang provides a much more complete documentation than GCC. The documentation of GCC compiler is very poor. When we were trying to look for information and documentation on the internals the only things that we reached is a page where you find a general description of some functions that might or might not be useful for your development. This, in fact, makes development a huge obstacle. Compilers are not especially little and simple pieces of engineering. Clang compiler code contains around 2.5 million lines of code. This means that looking inside it not one of the easiest tasks. If we consider that in our design we have to modify the compiler, the learning curve without documentation increases a lot with GCC.

This does not imply that Clang has a good documentation. Clang documentation is almost auto-generated by Doxygen. Doxygen is a documentation software that uses annotations that are made on the code to generate automatically the documentation of the compiler. This is a great alternative for the beginning, however, it does only provide information on all the things that want to be hidden such as private class members. For example, if we want to make a *constant function*, we have to dive into the code and look for the implementation. This is because the implementation of that is done in certain functions that are not public and hidden to the user. Those functions are performing a private function, and since the user is not supposed to know that, it is almost impossible to get to that point without touching the real code.

Either way, if we compare the documentation of both, it is a clear win on the Clang side since the documentation is well-organized and allows the implementation of features in an easy manner. Something that at first-glance GCC did not provide.

In [40], a list of the advantages of using one compiler or another is shown. Among the main relevant features of using GCC against Clang we find the following:

- **Nested functions**. Grammars usually enclose strange behaviours that we can use, however, we are not usually able to predict. In this case, the behaviour that we are describing is the ability to declare a function within another function. This is not permitted in Clang whereas in GCC it is allowed. This problem in the case of C++ is easily solvable because of functional programming. It is not the same in the case of C where there is not an equivalent. But that is something to choose depending on the implementation and in our case that we are dealing with C++ is not a problem.

- **Source Languages**. Clang acts as *front-end* for several languages. Even though Clang supports a wide variety of them, GCC supports more. We are interested in C++ support in this project so that does not affect us.

- **Target Architectures**. GCC supports more target architectures than Clang. That is mainly due to the fact of using LLVM underneath and that is what limits the target architectures. In our case we are not interested in the *back-end* but in the implementation of the functionality in the *front-end* just working in the main platforms. And with LLVM is more than enough.

With respect to the advantages that the usage of Clang supposes against the usage of GCC we have the following:

- **Understandability**. We already commented that Clang is much more documented and understandable than GCC. It provides a clear interface to create new tools.

- **API design**. Clang has been designed to provide its services as an API. When we look at the internals each of the modules seems to be self-contained and provide a functionality to others. This allows the integration of it with other tools.

- **GCC design decisions**. GCC has been designed to be a compiler and that is what it is. It does not aim to provide an interface, therefore, using it is not always as easy as it could be.

- **GCC code simplification**. When we have the AST, we still have the full information of the source code and we are able to generate the initial information. However, GCC simplifies the information once refinements are done.

- **Clang AST serialization**. A file can be generated containing the AST. This allows us to later use it in other tools. We can generate a file that another program receives and uses for code analysis.

- **Clang efficiency and memory consumption**. Clang is more efficient and consumes fewer resources than GCC.

- **Clang diagnostics.** GCC lacks of a meaningful system of diagnostics. Clang provides a better diagnostic which allows the programmer to find easier the errors.

- **Clang is licensed under BSD licence**. This type of licence permits the user to use the system and embed it in another functionality. This license permits any company to use Clang in commercial projects. This is not only good for the programmer but for everyone. It will allow the expansion of Clang in different projects and its usage will make it better and in all terms.

- **LLVM elements**. It shares some elements with LLVM since it is part of the *back-end*. This allows having more tools for the analysis of the code. The support and the way of optimizing code are shared with LLVM, and that makes the code analysis better.

- **Clang and C++**. The support that Clang has over C++ is much better and complete than the one it GCC has.

- **Language Extensions**. Clang provides many tools to extend language understanding. It provides tools for thread safety checking and some extended vector types.

Now we take a look at a performance evaluation for the compilers. Those measurements establish a bias in which we are able to differentiate between both compilers in an objective way. We use as measurements compilation times, considering the time that it takes to compile an executable file. And the execution and performance of the compiled code.

First, we have to differentiate between compilation times. There are different studies and performance tests that we have observed, note [41, 42, 43, 44]. In [42], the efficiency of the compilation with different optimization flags is evaluated. Both Clang and GCC are included in this test. The results are good on the Clang side. We see that in terms of flags *-02 and -03* Clang spends much less time than GCC. This means that the compilation times are much better in Clang according to this study.

In fact, as it is developed in [41, 43], which is related to one of the newest versions of Clang, the performance seems to be improving a lot. In newer versions of Clang, performance is getting similar results in the tests for compile time. In this sense, and from what we have observed all through the articles, Clang seems to be a better alternative if what we seek is optimizing compilation times.

However, we are strictly talking about compilation times and Clang. If we consider linking the executable also in our test, something that in Clang is related to LLVM, times seems to change. As it is developed in [44], most of the LLVM-based compilers lose mainly because of it. However, the data from this study was taken and analysed carefully. The study did not seem to be objective at all since the versions of the programs compiled differed a lot in time. This is probably what mainly provoked this result. In fact, we have to remember the fact that benchmarks sometimes do not represent real programs. This means that we might not be evaluating the real efficiency of the compiler.

With regard to the execution times, again it depends a lot. In [41], we were dealing with early versions of LLVM and Clang. This meant that the performance difference between the two is very noticeable and in favour of GCC. During the past few years, the differences have relaxed, and in [43] where we make the same benchmarks with the newest versions of both compilers we obtain that Clang is getting really close if as efficient. Although GCC does better in benchmarks, Clang is not getting far behind. It means that it is getting as a better and better alternative for development. In fact, results for big benchmarks where one of the two loses does not result in a huge difference. This means that choosing one or the

other is not a big deal and only the experience of compiling the program with one of the two can result in a better or worse result.

3.7 Other Compilers

In this section, we take a look at other minor compilers that are available. These compilers were not chosen mainly because their public is limited or they are paid. Implementing something in a compiler with a wider public range makes our feature available in an easier manner to all that people. In fact and although differences are not big, using something that people is familiar with is better if we want to adapt our product to a bigger public.

PGI C++ Compiler (PGC) is a C++ compiler that is developed by Nvidia and supports many features related to hardware acceleration such as OpenMP and OpenACC. OpenACC which is a way of programming Nvidia Tesla Graphical Processing Unit (GPU)s. This compiler has two versions, a community version which is free and a paid version which is said to be professional. It seeks to find and develop a compiler which optimizes information to be delivered to multicore systems and High-Performance Computer (HPC). This system has the purpose of allowing to create a single code for heterogeneous applications both HPC and GPU.

Intel C++ Compiler (IntelC++) and **AMD Optimizing C/C++ Compiler (AOCC)** are two compilers with the same purpose and made by different companies. Intel C++ Compiler (IntelC++) is a compiler created by Intel Corporation whose aim is providing a optimization specific for Intel processors. It is optimized for the latest C++ features and OpenMP standard. In the case of AMD Optimizing C/C++ Compiler (AOCC), it is compiler made by AMD for C and C++ which include improvements in code generations so that it creates a optimal code. AOCC uses the LLVM library on its implementation.

As we can see, the rest of compilers that are compared in this section, do not have a general purpose as GCC or Clang. They are specific for certain platforms and generate better results in one or another. Appart from that, it is very likely that due to the aim of optimizing, dealing with the source code of these compilers was not as easy as it is with GCC or Clang. Either way, it is important to mention the existence of compilers of this kind.

Chapter 4

Analysis of the problem

This chapter provides a definition of which are the project objectives. In section 4.1 an overview of how the user shall be able to act and the features to be implemented is given. In section 4.2 the main constraints when developing the system are depicted. A description of the prototype of a user that will use the system is given in 4.3. In section 4.4 it is described the environment where the project is expected to work. Finally in section 4.5 the user requirements are written down.

Within the second part of the chapter, the system requirement specification is carried out. The first step of the system requirement specification is to develop the use case specification the main actions that a user can carry out(section 4.6. Right after that, in section 4.7 an overview of the system technical details are given. Finally in section 4.8 the system requirements are depicted. As a final section, the requirement traceability matrices are shown.

4.1 General Capabilities

This project aims to develop several features to support Design By Contract (DBC) in C++. It consists on the implementation of several new keywords that are usually known in DBC. Those are *expects* for preconditions, *assert* for assertions and *ensures* for postconditions. This new keywords will have the following syntax:

- *[[assert assertion-level: condition]]* for assertions.
- *[[expects assertion-level: condition]]* for preconditions.
- *[[ensures assertion-level return-variable: condition]]* for postconditions.

This new clauses will be applied to different facets of the code. *Assertions* will be associated to a executable section of code and *preconditions and postconditions* will be associated with the *function dec-*

laration or definition. This contracts will evaluate the condition. In case that the condition is evaluated to be true, nothing will happen. For example, if we say *[[assert: x > 0]]* and *x = 1* then, the assertion will be evaluated as correct and the code execution will not be modified. In the opposite case if *x = -1*, then the assertion will not be correct and the implementation would provoke an exceptional behaviour.

We are interested in the fact of implementing them as keywords so that the programmer cannot influence what the contract behaviour is. Something that could be viable in case that the implementation was performed with macros.

On a second phase of the project, some advanced features were implemented. As we have seen, the contracts imply a processing time overhead. This is not assumable for a release version. However, it is a good idea for the verification of the functionality and might be used for alpha and beta versions. The inclusion of assertion levels allows, by means of a mechanism, to activate and deactivate the different contracts. This avoids having the overhead on the release version and permits having verification on pre-release versions. The assertion level is a keyword of the set of *axiom, always, default and audit*. The assertion level is compared with the build level in order to see which contracts to generate. The build level values are *off, default and audit*. The following comparison determines which attributes are generated each time:

- With the build level set to *off* only *always* contracts are generated.
- With the build level set to *default*, *always and default* contracts are generated.
- With the build level set to *audit*, *always, default and audit* contracts are generated.
- *Axiom* assertion level never generates code.

In the same manner, the system is supposed to abort when the evaluation of a conditions fails. However depending on the case, the user might want to continue the execution, for this purpose, a continuation flag is implemented which allows the continuation even if the application fails with a contract.

Finally, as last meaningful feature, the function executed when the evaluation of a condition is false will be by default the interruption of the program. However, the possibility for the user to have further control and define the violation handler is also given. A user can define the function that it is executed whenever a contract fails. The implementation shall also give a mechanism to provide the name with a violation handler.

4.2 General Constraints

In this section, a general overview of the constraints that are found when having to deal with the implementation of the code is given.

One of the first restrictions that we find is related to the positioning of the contract clauses that will be present. Both *preconditions and postconditions* will be associated with a function header, however, the *assertions* have to be placed in an executable piece of code. With that assumption in mind, C++ functional programming implementation shall interact properly with the contract specification. That implies that we have to define the behaviour with *lambda functions*, *function pointers* or *generic programming*. Apart from that, with regard to the specification of a custom handler, the user shall not be able to execute this custom handler as a normal function. That means that the implementation shall take care that the user is not doing anything against the integrity of the program.

4.3 User characteristics

In this section, we are going to explain which are the characteristics of a potential user of the system to be developed. In order for a proper development of the requirements, it is very important to take into account the user. If the system is adequate for users necessities, they will find it familiar and use it whenever they have to solve this problem. Otherwise, if we do not identify properly the potential users, the system will not be adequate for their necessities.

This project is not a high-level project aiming to make easy the life of normal people. Two main groups are the target of this project. Firstly, C++ developers which want to improve the quality of their projects. For them, contracts not only suppose the assurance of reliability, but their usage can help them finding programming errors, and therefore speed up the development. Secondly, enterprises. The fact of having tools to ensure that what the user asked in a project is properly implemented is an appealing feature for the usage of the language in future projects.

Either way, any C++ programmer can use the contracts since they should not suppose any difficulty. They are easily placeable and checked automatically by the compiler. In addition to that, no additional compilation options are needed for the basic functionality (without build level, aborting and without a custom handler).

4.4 User Operational Environment

In this section, we have to take a look at the environment where the software is going to be deployed. In fact, for the correct adaption of the system to the requirements, this step has to be carefully developed. The operational environment is any computer where a compiler can be run. This means medium-end and low-end computers have to allow the execution of the implementation.

To develop this phase, we have to go one step ahead and think about the implementation of the system. For many of the features that the system is going to include, we have to modify the grammar of the programming language. That means, that for the features to work a simple implementation is not enough. We have to modify the internals of the compiler and that means what it will, later on, interpret the compiler.

With that in mind, we have to make sure that the modifications that we make to the compiler are similar to all the processes that the compiler does for the rest of implementation. In that way, we can know that the implementation has to work on any system where the compiler chosen for its implementation does.

4.5 User Requirements

In this section, we are going to develop the User Requirements (UR) of the problem that we have developed. This implementation is based on a paper released by the ISO C++ committee and every requirement has been extracted from it[45]. The main objective that we want to fulfil with the UR section is the creation of a high-level abstraction of what the problem is. The step of elicitation of the requirements is crucial for a proper project development. That is why in this section we are developing this set of requirements. The UR phase is carried out right after having an interview with the client. In our case, the first elicitation of the requirements was carried out from the following paper [46]. With both papers the elicitation of requirements is carried out.

Since the description must not include a lot of detail, UR are going to make an overview of the features that want to be implemented. The User Requirements will be divided into two main categories:

Capability requirements. Those refer to the functions and operations required by the user from the system. They refer to what the user needs to solve a problem and fulfil a goal.

Constraint requirements. These requirements refer to how the software shall be built and operate. These are the user constraints when trying to solve a problem.

For the description of the requirements, we are going to use tables. In each of the tables, some relevant fields are going to be present in order to establish some common features that are present. Among those features we have the following:

- **UR-XX-YY.** This will represent the user requirement identifier. The identifier is composed by:
  - *XX.* It is the kind of requirement that we are dealing with, it can be either CA if its User Capability Requirement or CO in case it is a User Constraint Requirement.
  - *YY.* It is the numeric identifier of the requirement, it will start in 1 and go increasing by one. It is unique for each kind of requirement, so an identifier can be repeated only for different requirement categories.

  As an example, we could have UR-CA-10 representing the 10th user capability requirement.

- **Definition.** This field is a little text with the description of the requirement. An example of it could be: The user shall be able to generate tables in LaTeXformat.

- **Necessity** It establishes the importance within the project that the requirement has. It can have three values either *Essential*, *Desirable* or *Non-Essential*.

- **Priority.** This field establishes which is the order in which features are meant to be implemented. There are three different values this field can hold *High* (if the feature is very needed), *Medium* (if the priority is related to high priority and is not that important) or *Low* (if the feature is not that important in the order or is similar to an extra feature).

- **Stability.** This feature represents the likelihood of it to be modified or has changed from the original requirement. It can hold two possible values within it *Stable* (if it is not likely to be changed) or *Unstable*(if it is likely to be changed).

- **Verifiability.** This feature establishes how verifiable are the requirements. That is the ease of the implementation to justify that this requirement has been successfully implemented. It can have three different values depending on the difficulty that its verification has to suppose, *High*(if it is easy to be verified), *Medium* (if it is not that easy to be verified) or *Low*(if it is difficult to be verified).

- **Status.** This represents the step of the requirements lifecycle in which the requirement is. The different steps that we have are *Proposed, Verified, Validated, Rejected or Suspended.*

UR-XX-YY	
Necessity:	Essential | Non-essential
Priority:	Low | Medium | High
Stability:	Stable | Unstable
Verifiability:	Low | Medium | High
Status:	Proposed | Verified | Validated | Rejected | Suspended
Definition:	Requirement description and explanation

Table 4.1: *User Requirements Template Table*

4.5.1 Capability Requirements

UR-CA-01	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to place an assert contract on the code.

Table 4.2: *User Capability Requirement UR-CA-01*

UR-CA-02	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to place an expects contract on the code.

Table 4.3: *User Capability Requirement UR-CA-02*

UR-CA-03	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to place an ensures contract on the code.

Table 4.4: *User Capability Requirement UR-CA-03*

UR-CA-04	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall not be affected by an affirmative condition evaluation.

Table 4.5: *User Capability Requirement UR-CA-04*

UR-CA-05	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	A user shall be prompted with a violation handler execution in case that the condition is negatively evaluated.

Table 4.6: *User Capability Requirement UR-CA-05*

UR-CA-06	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall specify a condition for each contract clause.

Table 4.7: *User Capability Requirement UR-CA-06*

UR-CA-07	
Necessity:	Essential
Priority:	Medium
Stability:	Unstable
Verifiability:	High
Status:	High
Definition:	The user shall omit or repeat the contract when specifying the definition of a previous function declaration.

Table 4.8: *User Capability Requirement UR-CA-07*

UR-CA-08	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to abort in case of a violation handler execution.

Table 4.9: *User Capability Requirement UR-CA-08*

UR-CA-09	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to continue the execution in case of a violation handler execution.

Table 4.10: *User Capability Requirement UR-CA-09*

UR-CA-10	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to specify an assertion level on the contract clause.

Table 4.11: *User Capability Requirement UR-CA-10*

UR-CA-11	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to place an assertion level which will be always evaluated.

Table 4.12: *User Capability Requirement UR-CA-11*

UR-CA-12	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to place an assertion level which will never be evaluated.

Table 4.13: *User Capability Requirement UR-CA-12*

UR-CA-13	
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to determine the build level.

Table 4.14: *User Capability Requirement UR-CA-13*

UR-CA-14	
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The assertion level shall be taken into account with respect to the build level.

Table 4.15: *User Capability Requirement UR-CA-14*

UR-CA-15	
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall be able to determine the violation handler.

Table 4.16: *User Capability Requirement UR-CA-15*

UR-CA-16	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	High
Definition:	The user shall have information about the violation of the code.

Table 4.17: *User Capability Requirement UR-CA-16*

4.5.2 Constraint Requirements

UR-CO-01	
Necessity:	definition.
Priority:	Essential
Stability:	High
Verifiability:	Stable
Status:	High
Definition:	The user shall place the expects clause in the function declaration

Table 4.18: *User Constraint Requirement UR-CO-01*

UR-CO-02	
Necessity:	definition.
Priority:	Essential
Stability:	High
Verifiability:	Stable
Status:	High
Definition:	The user shall place the ensures clause in the function declaration

Table 4.19: *User Constraint Requirement UR-CO-02*

UR-CO-03	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The user shall place the assert contract in a executable section.

Table 4.20: *User Constraint Requirement UR-CO-03*

UR-CO-04	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The condition of the ensures clause shall be related to the return value.

Table 4.21: *User Constraint Requirement UR-CO-04*

UR-CO-05	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The user will not be able to access the structure values by runtime means.

Table 4.22: *User Constraint Requirement UR-CO-05*

UR-CO-06	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The default policy for continuation mode shall be aborting the execution.

Table 4.23: *User Constraint Requirement UR-CO-06*

UR-CO-07	
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The default policy for build level shall be default.

Table 4.24: *User Constraint Requirement UR-CO-07*

UR-CO-08	
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The user shall not be able to directly invoke the violation handler.

Table 4.25: *User Constraint Requirement UR-CO-08*

UR-CO-09	
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The functionality shall work on with C++ Programming Language features.

Table 4.26: *User Constraint Requirement UR-CO-09*

4.6 Use Cases

This section defines the use cases that are extracted from the user requirements. The use cases are a formalization of what the user must be able to do and are requisites to be able to extract the system requirements. The use case diagram represents which are the capabilities that in general, a user has when using a system.

A use case is defined by the following fields:

- **UC-XX**. It defined the identifier of the use case. The XX will define the identifier number of the use case and it is unique for each Use Case.
- **Name**. The name is a brief description of what the use case will imply.
- **Actor**. It determines who is in charge of performing the action. In this case, the actor is always going to be the user.

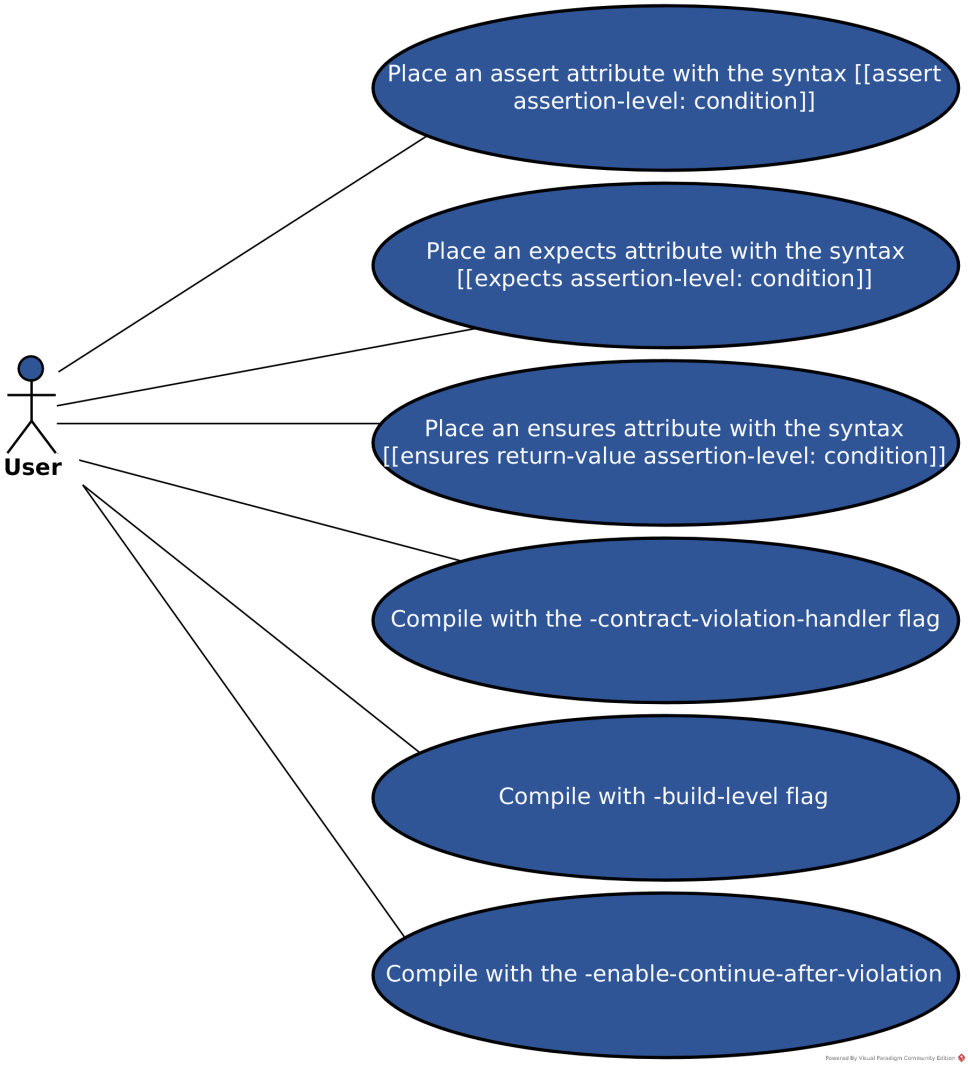

Figure 4-1: *Design By Contract Use Case Diagram*

- **Preconditions**. It determines the conditions under which the use case ensures that the functionality will be carried out correctly.

- **Postconditions**. It determines the characteristics that can be always verified after the execution of the functionality.

- **Description**. It establishes what how the use case needs to carry out the actions and what it shall allow the user to do.

UC-XX	
Name:	Name
Actor:	User | Administrator
Preconditions:	The conditions that have to be fulfilled for the proper execution.
Postconditions:	The conditions that will be fulfilled after the execution.
Description:	The expansion of the use case

Table 4.27: *Use Case Template Table*

UC-01	
Name:	Placing an assert attribute with the syntax [[assert assertion-level: condition]].
Actor:	User
Preconditions:	• The syntax shall be respected according to the use case. • The assertion shall be placed within an executable section of code. • The assertion shall have a condition that is possible to be transformed to a boolean expression.
Postconditions:	• The system shall have evaluated the condition correctly and abort in case that the condition is evaluated to false
Description:	The user places in an executable section of the code an assert attribute. This attribute is compiled into a binary. When it is executed the code evaluates the condition placed on the contract. In case it is false the system aborts.

Table 4.28: *Use Case UC-01*

UC-02	
Name:	Placing an expects attribute with the syntax [[expects assertion-level: condition]].
Actor:	User
Preconditions:	• The syntax shall be respected according to the use case. • The assertion shall be associated to the function type. • The assertion shall have a condition that is possible to be transformed to a boolean expression.
Postconditions:	• The system shall have evaluated the condition correctly and abort in case that the condition is evaluated to false
Description:	The user places in an executable section of the code an assert attribute. This attribute is compiled into a binary. When it is executed the code evaluates the condition placed on the contract. In case it is false the system aborts.

Table 4.29: *Use Case UC-02*

UC-03	
Name:	Placing an ensures attribute with the syntax [[ensures return-value assertion-level: condition]].
Actor:	User
Preconditions:	• The syntax shall be respected according to the use case. • The assertion shall be associated to the function type. • The assertion shall have a condition that is possible to be transformed to a boolean expression. • The return variable shall be able to be compared to the value that is created.
Postconditions:	• The system shall have evaluated the condition correctly and abort in case that the condition is evaluated to false
Description:	The condition placed on the contract is evaluated by the system and in case that it is false, the system aborts.

Table 4.30: *Use Case UC-03*

UC-04	
Name:	Compile with *-enable-continue-after-violation* flag.
Actor:	User
Preconditions:	• The flag shall follow the syntax *-enable-continue-after-violation*
Postconditions:	• No postconditions.
Description:	The evaluation of a condition to false does not cause the program to abort.

Table 4.31: *Use Case UC-04*

UC-05	
Name:	Compile with *-build-level* flag.
Actor:	User
Preconditions:	• The build level shall be either *off, default or audit*. • The flag shall follow the syntax *-build-level*
Postconditions:	• The system shall have generated *always* contracts if build level is *off*. • The system shall have generated the *always* and *default* contract if build level is *default*. • The system shall have generated the *always*, *defaul* and *audit* contracts if build level is *audit*.
Description:	The condition placed on the contract is evaluated if and only if the comparison of the build level and the assertion level determines that it is necessary to generate it.

Table 4.32: *Use Case UC-05*

UC-06	
Name:	Compile with *-contract-violation-handler* flag.
Actor:	User
Preconditions:	• A function with that name shall be existing in the code.
Postconditions:	• Any violation of a condition will result in the execution of that function.
Description:	When a condition fails, the contract violation handler established with the flag is executed.

Table 4.33: *Use Case UC-06*

4.7 Function and Purpose

In this section, we will detail the behaviour that the system will have. We will emphasise the technical details of the implementation. This is an intermediate step between the User Requirements Specification and the System Requirements phase. It allows us to extract all the details in a developed description that later on will be transformed into a specification which will serve as an input for the design phase.

The implementation of the contracts is going to be developed by means of new attributes in the language. Attributes are usually associated with any kind of statement in the code. They are introduced in relation to any statement where they induce the compiler to act differently on them. There are different types of attributes depending on the origin that they have. Depending on this, they usually have different syntaxes. For example, two of the main providers of attributes in C++ are the C++ Standard and the GNU Compiler Collection (GCC). Both have different syntaxes and provide different directives that modify the behaviour either of the result of the compilation or the functions that the compiler will perform during its execution. Their usage is something that is very extended in some contexts because they give a lot of information and allow the compiler to avoid unnecessary computations. In addition to that, they allow for some high-performance features such as inlining that is not performed in other situations. The system will have to recognize three different syntaxes for the new attributes, one for each of the different possibilities of the elements:

- **Assertion.** For the assertion, the supported syntax is going to be *[[assert assertion-level : condition]]*. This expression has to be associated to a *null statement*. Associating the expression to a *null statement* permits the clause of it being associated only with an executable section of code.
- **Preconditions.** For the preconditions we used this syntax *[[expects assertion-level : condition]]*. This expression has to be associated to a *function type*. It is important to differentiate the *function type* from the function return type itself because the association is kept in different structures. This causes that associating an attribute to the *function type* will modify how the function will be treated during the compilation. In this way, we would be able to find any modification done to the function by explicitly finding the changes to the structure that represents it. For example, modifying the type of the function would include *inlining*. This is a technique which omits the function call and substitutes it by the code in its body.
- **Postconditions.** For the postconditions, a syntax like *[[ensures assertion-level ret-variable : condition]]* is going to be used. In this syntax another special member is included which is *return-value*. This is an identifier, which is bound to the return value of the function. It serves as an identifier, for a variable that will be later on created or initialized. Indeed it is used just to identify the return value, in case that the user wants to use it in the condition. But the important point is that we need the return values since the postconditions are always related to that value.

Now a description of the common values will be detailed. The first common value is the condition. The condition evaluation can result in two different behaviours. In case that the condition is *true*, then the program has to do no further execution. On the opposite case, if the condition is evaluated to be *false*, the program has to execute the *violation handler*. The condition is usually evaluated at runtime, however, as the second part of this project, the compiler should be modified to allow the compile-time evaluation of the conditions.

There is an element that we did not take into account in the previous syntax and that is the *assertion-level*. This assertion level will be used to activate and deactivate conditions. This field can have several values *axiom, always, default, audit or no value*. Their functionality is highly related to another feature the *build level*, so we will first comment it so that it remains clear. The build level is a limitation that the user imposes on the code, to limit the number of attributes that will be transformed into real condition evaluations. It has again certain limited values, *off, default and audit*, and its value is given to the compiler before its execution. With regard to putting no value in the *assertion-level* field, the specification forces it to be assumed to be *default*, so from this point on we will assume that putting *nothing* is the same as *default*. When we have the *build level* set to *off*, we will only take into account contracts marked with the assertion level *always*. When the *build level* is turned to *default*, only attributes marked as *default or always* are taken into account. Finally, if we set the *build level* to *audit*, attributes with *always, default or audit* will be taken into account. *Always* attributes are always evaluated, no matter which build-level is used. On the other hand, *axiom* attributes are never evaluated, no matter what the build-level is.

As one of the last main features, some modifications had to be done to the violation handler. The violation handler is what is executed right after a condition evaluated to *false*. Usually, the violation handler that will be used will be the default of the system. This handler, however, can be customized in some ways. First of all, the violation handler can be defined by the user. This will be done by a compilation flag. With it, the user will be able to determine the name of the compilation handler. Apart from that, the user will be able to choose whether to abort after a contract violation or to activate a continuation mode in which it will not halt the execution. The main purpose of that mode is allowing the user to see whether the system is robust and can continue working even if some elements of it fail. In order to do this, another compiler flag will be implemented.

With this in mind, we have to take into account *functional programming*. This kind of programming is related to for example lambda functions. In this case, the implementation excludes the usage of contracts over lambda functions. The main reason for it is that behind it, the hidden type is not a function but a *functor*, an object with the *parenthesis operator* overridden so that it executes the function that the user decides. Since that is the case, we have to modify a lot the implementation to affect directly the operator

instead of the object created and that is not worthy at all since the implementation of contracts on this kind of functions would not cope a lot with the ideas of Design By Contract. For this reason, lambda functions and functors are excluded from Design By Contract.

4.8 System Requirements

In this section, we are going to explain the System Requirements. This specification is going to be detailed and accurate. Each of the requirements in this phase is designed to be specific, measurable, achievable, realistic and time-bound (SMART). The correctness of this section is crucial for the later development of the project, in which we will depend on this section for the development of it. In addition to that, it has been demonstrated that a proper requirements specification phase reduces substantially the failure in projects, the cost of the whole project and decreases the cost on the rest of the phases.

In this section, the requirements will be divided into two categories:

Functional requirements. A functional requirement specifies what a process must accomplish, what the system shall do.

Non-Functional requirements. It specifies how software shall be done.

For the description of the requirements, we are going to use tables. In each of the tables, some relevant fields are going to be present in order to establish some common features. Among those features we have the following:

- **SR-XX-YY.** This will represent the system requirement identifier. The identifier is composed by:
  - *XX.* It is the kind of requirement that we are dealing with, it can be either FR if its System Functional Requirement or NFR in case it is a System Non-Functional Requirement.
  - *YY.* It is the numeric identifier of the requirement, it will start in 1 and go increasing by one. It is unique for each kind of requirement, so an identifier can be repeated only for different requirement categories.

  As an example, we could have UR-CA-10 representing the 10th user capability requirement.
- **Origin.** This field specifies the identifier of the user requirement that originated this system requirement. A requirement can have several origins.
- **Definition.** This field is a little text with the description of the requirement. An example of it could be: The user shall be able to generate tables in LaTeXformat.
- **Necessity** It establishes the importance within the project that they have. It can have three values either *Essential*, *Desirable* or *Non-Essential*.

- **Priority.** This field establishes which is the order in which features are meant to be implemented. There are three different values this field can hold *High* (if the feature is very needed), *Medium* (if the priority is related to high priority and is not that important) or *Low* (if the feature is not that important in the order or is similar to an extra feature).

- **Stability.** This feature represents the likelihood of a requirement to be modified or to be changed from the original requirement text. It can hold two possible values within it *Stable* (if it is not likely to be changed) or *Unstable*(if it is likely to be changed).

- **Verifiability.** This feature establishes how verifiable requirements are. That is the ease of the implementation to justify that this requirement has been successfully implemented. It can have three different values depending on the difficulty that its verification has to suppose, *High*(if it is easy to be verified), *Medium* (if it is not that easy to be verified) or *Low* if it is difficult to be verified.

- **Status.** This represents the step of the requirements lifecycle we are. The different steps that we have are *Proposed, Verified, Validated, Rejected or Suspended.*

SR-XX-YY	
Origin:	UR-XX-YY
Necessity:	Essential | Non-essential
Priority:	Low | Medium | High
Stability:	Stable | Unstable
Verifiability:	Low | Medium | High
Status:	Proposed | Verified | Validated | Rejected | Suspended
Definition:	Requirement description and explanation

Table 4.34: *System Requirements Template Table*

4.8.1 Functional Requirements

SR-FR-01	
Origin:	UR-CA-01 UR-CA-06
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall recognize the assert contract syntax [[assert assertion-level: condition]]

Table 4.35: *System Functional Requirement SR-FR-01*

SR-FR-02	
Origin:	UR-CA-02 UR-CA-06
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall recognize the expects contract syntax.[[expects assertion-level: condition]]

Table 4.36: *System Functional Requirement SR-FR-02*

SR-FR-03	
Origin:	UR-CA-03 UR-CA-06
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall recognize the ensures contract syntax.[[ensures modifier identifier: conditional-expression]]

Table 4.37: *System Functional Requirement SR-FR-03*

SR-FR-04	
Origin:	UR-CA-01 UR-CA-02 UR-CA-03 UR-CA-06
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall transform a contract into code for its evaluation.

Table 4.38: *System Functional Requirement SR-FR-04*

SR-FR-05	
Origin:	UR-CA-04
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall execute a violation handler if the condition evaluation turns to be true.

Table 4.39: *System Functional Requirement SR-FR-05*

SR-FR-06	
Origin:	UR-CA-05
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall execute no additional code if the condition evaluation is false.

Table 4.40: *System Functional Requirement SR-FR-06*

SR-FR-07	
Origin:	UR-CO-01
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall verify that the expects clause is associated to the function type.

Table 4.41: *System Functional Requirement SR-FR-07*

SR-FR-08	
Origin:	UR-CO-02
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall verify that the ensures clause is associated to the function type.

Table 4.42: *System Functional Requirement SR-FR-08*

SR-FR-09	
Origin:	UR-CO-03
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall verify that the assert clause is associated to a null statement.

Table 4.43: *System Functional Requirement SR-FR-09*

SR-FR-10	
Origin:	UR-CA-10
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall generate a structure in which it will hold the assertion level for each contract clause

Table 4.44: *System Functional Requirement SR-FR-10*

SR-FR-11	
Origin:	UR-CA-07
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall verify that the contracts are either repeated or omitted in the definition of the function with respect to the declaration.

Table 4.45: *System Functional Requirement SR-FR-11*

SR-FR-12	
Origin:	UR-CA-10
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall recognize 4 assertion levels: axiom, always, default and audit.

Table 4.46: *System Functional Requirement SR-FR-12*

SR-FR-13	
Origin:	UR-CA-12
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall never evaluate axiom contracts.

Table 4.47: *System Functional Requirement SR-FR-13*

SR-FR-14	
Origin:	UR-CA-11
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall always evaluate always contracts.

Table 4.48: *System Functional Requirement SR-FR-14*

SR-FR-15	
Origin:	UR-CA-13
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide a mechanism for the user to choose the build level.

Table 4.49: *System Functional Requirement SR-FR-15*

SR-FR-16	
Origin:	UR-CA-13
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall recognize 3 build levels: off, default and audit

Table 4.50: *System Functional Requirement SR-FR-16*

SR-FR-17	
Origin:	UR-CA-14
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall compare the build level with the assertion before the evaluation of the condition.

Table 4.51: *System Functional Requirement SR-FR-17*

SR-FR-18	
Origin:	UR-CA-14
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall evaluate always contracts if build level is off.

Table 4.52: *System Functional Requirement SR-FR-18*

SR-FR-19	
Origin:	UR-CA-14
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall evaluate always and default contracts if the build level is default.

Table 4.53: *System Functional Requirement SR-FR-19*

SR-FR-20	
Origin:	UR-CO-05
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall allow the evaluation of direct values.

Table 4.54: *System Functional Requirement SR-FR-20*

SR-FR-21	
Origin:	UR-CA-14
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall evaluate always, default and audit contracts if the build level is audit.

Table 4.55: *System Functional Requirement SR-FR-21*

SR-FR-22	
Origin:	UR-CA-15 UR-CA-08 UR-CA-09
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide a a mechanism for the user to choose the continuation mode.

Table 4.56: *System Functional Requirement SR-FR-22*

SR-FR-23	
Origin:	UR-CA-16
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide information on the contract violation by means of a contract violation structure

Table 4.57: *System Functional Requirement SR-FR-23*

SR-FR-24	
Origin:	UR-CA-16
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide information of the line number

Table 4.58: *System Functional Requirement SR-FR-24*

SR-FR-25	
Origin:	UR-CA-16
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide information on the file name where the contract violation occurred

Table 4.59: *System Functional Requirement SR-FR-25*

SR-FR-26	
Origin:	UR-CA-16
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide the function name where the violation occurred

Table 4.60: *System Functional Requirement SR-FR-26*

SR-FR-27	
Origin:	UR-CA-16
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide the a comment on the violation ocurred.

Table 4.61: *System Functional Requirement SR-FR-27*

SR-FR-28	
Origin:	UR-CA-16
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide the assertion level that ocurred.

Table 4.62: *System Functional Requirement SR-FR-28*

SR-FR-29	
Origin:	UR-CA-15
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide a mechanism for the user to choose a custom violation handler.

Table 4.63: *System Functional Requirement SR-FR-29*

SR-FR-30	
Origin:	UR-CO-08
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall protect the code from the execution of the violation handler.

Table 4.64: *System Functional Requirement SR-FR-30*

SR-FR-31	
Origin:	UR-CO-01 UR-CO-02
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall warn the user when using lambda functions

Table 4.65: *System Functional Requirement SR-FR-31*

SR-FR-32	
Origin:	UR-CO-01 UR-CO-02
Necessity:	Essential
Priority:	Low
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall warn the user when using function pointers.

Table 4.66: *System Functional Requirement SR-FR-32*

SR-FR-33	
Origin:	UR-CO-04
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall ensure that the conditional expression does not access the member that should not be accessed by the caller function

Table 4.67: *System Functional Requirement SR-FR-33*

SR-FR-34	
Origin:	UR-CA-01 UR-CA-02 UR-CA-03
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall provide the functionality for conditions including a template variable.

Table 4.68: *System Functional Requirement SR-FR-34*

SR-FR-35	
Origin:	UR-CO-04
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall warn if a parameter is modified within a function and its used within a postcondition.

Table 4.69: *System Functional Requirement SR-FR-35*

SR-FR-36	
Origin:	UR-CO-07
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall use the default level of assertion in case that it is not specified.

Table 4.70: *System Functional Requirement SR-FR-36*

SR-FR-37	
Origin:	UR-CO-06
Necessity:	Essential
Priority:	Medium
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall abort on a contract violation if nothing is specified.

Table 4.71: *System Functional Requirement SR-FR-37*

4.8.2 Non-Functional Requirements

SR-NFR-01	
Origin:	UR-CO-09
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The implementation shall work properly with C++ 17.

Table 4.72: *System Non-Functional Requirement SR-NFR-01*

SR-NFR-02	
Origin:	UR-CO-09
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall be implemented in Clang Compiler

Table 4.73: *System Non-Functional Requirement SR-NFR-02*

SR-NFR-03	
Origin:	UR-CO-09
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The system shall be compliant with LLVM

Table 4.74: *System Non-Functional Requirement SR-NFR-03*

SR-NFR-04	
Origin:	UR-CO-09
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The developed code shall be interoperable between different platforms (Windows, MacOS and Linux).

Table 4.75: *System Non-Functional Requirement SR-NFR-04*

SR-NFR-05	
Origin:	UR-CA-13
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The mechanism that the user will use to provide the build level shall be a compilation flag.

Table 4.76: *System Non-Functional Requirement SR-NFR-05*

SR-NFR-06	
Origin:	UR-CA-08 UR-CA-09
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The mechanism to specify the continuation mode shall be a compilation flag.

Table 4.77: *System Non-Functional Requirement SR-NFR-06*

SR-NFR-07	
Origin:	UR-CA-15
Necessity:	Essential
Priority:	High
Stability:	Stable
Verifiability:	High
Status:	Validated
Definition:	The mechanism to specify the user defined handler shall be a compilation flag.

Table 4.78: *System Non-Functional Requirement SR-NFR-07*

4.9 Traceability Matrix

	UR-CA-01	**UR-CA-02**	**UR-CA-03**	**UR-CA-04**	**UR-CA-05**	**UR-CA-06**	**UR-CA-07**	**UR-CA-08**	**UR-CA-09**	**UR-CA-10**	**UR-CA-11**	**UR-CA-12**	**UR-CA-13**	**UR-CA-14**	**UR-CA-15**	**UR-CA-16**	**UR-CO-01**	**UR-CO-02**	**UR-CO-03**	**UR-CO-04**	**UR-CO-05**	**UR-CO-06**	**UR-CO-07**	**UR-CO-08**	**UR-CO-09**
UC-01	✓			✓	✓	✓	✓	✓		✓	✓	✓							✓		✓				✓
UC-02		✓		✓	✓	✓	✓	✓		✓	✓	✓					✓				✓				✓
UC-03			✓	✓	✓	✓	✓	✓		✓	✓	✓						✓		✓	✓				✓
UC-04								✓	✓													✓			✓
UC-05										✓	✓	✓	✓	✓									✓		✓
UC-06															✓	✓								✓	✓

Table 4.79: *Traceability Matrix User Requirements to Use Cases*

	UR-CA-01	**UR-CA-02**	**UR-CA-03**	**UR-CA-04**	**UR-CA-05**	**UR-CA-06**	**UR-CA-07**	**UR-CA-08**	**UR-CA-09**	**UR-CA-10**	**UR-CA-11**	**UR-CA-12**	**UR-CA-13**	**UR-CA-14**	**UR-CA-15**	**UR-CA-16**	**UR-CO-01**	**UR-CO-02**	**UR-CO-03**	**UR-CO-04**	**UR-CO-05**	**UR-CO-06**	**UR-CO-07**	**UR-CO-08**	**UR-CO-09**
SR-F-01	✓					✓																			
SR-F-02		✓				✓																			
SR-F-03			✓			✓																			
SR-F-04	✓	✓	✓			✓																			
SR-F-05				✓																					
SR-F-06					✓																				
SR-F-07																	✓								
SR-F-08																		✓							
SR-F-09																			✓						
SR-F-10										✓															
SR-F-11							✓																		
SR-F-12										✓															
SR-F-13												✓													
SR-F-14											✓														
SR-F-15													✓												
SR-F-16													✓												
SR-F-17														✓											
SR-F-18														✓											
SR-F-19														✓											
SR-F-20																					✓				
SR-F-21														✓											
SR-F-22								✓	✓						✓										

Table 4.80: *Traceability Matrix SR-FR-01 to SR-FR-22*

	UR-CA-01	**UR-CA-02**	**UR-CA-03**	**UR-CA-04**	**UR-CA-05**	**UR-CA-06**	**UR-CA-07**	**UR-CA-08**	**UR-CA-09**	**UR-CA-10**	**UR-CA-11**	**UR-CA-12**	**UR-CA-13**	**UR-CA-14**	**UR-CA-15**	**UR-CA-16**	**UR-CO-01**	**UR-CO-02**	**UR-CO-03**	**UR-CO-04**	**UR-CO-05**	**UR-CO-06**	**UR-CO-07**	**UR-CO-08**	**UR-CO-09**
SR-F-23																✓									
SR-F-24																✓									
SR-F-25																✓									
SR-F-26																✓									
SR-F-27																✓									
SR-F-28																✓									
SR-F-29															✓										
SR-F-30																								✓	
SR-F-31																	✓	✓							
SR-F-32																	✓	✓							
SR-F-33																				✓					
SR-F-34	✓	✓	✓																						
SR-F-35																				✓					
SR-F-36																							✓		
SR-F-37																						✓			
SR-NF-01																									✓
SR-NF-02																									✓
SR-NF-03																									✓
SR-NF-03																									
SR-NF-04													✓												
SR-NF-05								✓	✓																
SR-NF-06															✓										

Table 4.81: *Traceability Matrix SR-FR-23 to SR-NFR-06*

Chapter 5

Design of the solution

In this chapter, we describe the design of the project that departs from the implementation developed by Javier López Gómez. In this work, we tackle with the different compilation flags and the custom contract violation handler. Therefore, in this section, we introduce modification needed to introduce the functionality aimed in this project. In order to better understand the contributions of this work, in the following, we describe the basis of this project.

In section 5.1 we evaluate the alternatives for the development of the project. In section 5.2 we overview a breakdown of the compilation process. In section 5.3, we describe the design of the solution for the project.

5.1 Compiler Selection: Discussion of Alternatives

In this section, we evaluate the different compilers available for the development of the project. As it was mentioned in Chapter 3, we need to decide which is the most suitable compiler for the purposes of the project. In order to select the compiler, we have taken in to account the following aspects:

- *Documentation.* Finding documentation about GCC was not an easy task, and for the proper development of the project, it is important to find support especially from the documentation. Solving a problem is a much more complex task in case that no documentation is available. In fact due to the small community of people that deals with compilers, finding information on people bumping into similar problems is very difficult. In this part Clang is a better option.

- *Modularity.* A basic design principle of Clang is making it pretty modular. This characteristic is something that tends to make easier the development of any project.

- *Adoption.* GCC is a more widespread compiler. Having a more used compiler makes easier to reach a lot of public.

- *Licence.* Clang and LLVM are released with a licence that is much lighter in the obligations that the programmer has. Furthermore, it gives the programmer more rights and freedom over the code that they are writing.
- *C++ support.* We are very interested in having a good implementation of the C++ standard because it shall provide mechanisms to design the newer ones. The implementation of the C++ standard is more advanced and has a better implementation in Clang.

With all that in mind, due to its better documentation, the modularity that it provides, the freedom of the licence and the C++ support, we decide to use Clang for the basis of the project.

5.2 Overview of the Original Clang Design

In this section, a breakdown of the processes and the modules that compose a compiler is done by means of diagrams. The compiler architecture and design process do not follow the usual architectural design of a normal software product. The reason behind is that the newly implemented element implies relations to many components of the compiler, and it causes that what is a sole and simple functionality has to be fragmented into several modules which interact between them.

Specifically in Section 5.2.1, we give an overview of the component design of Clang compiler. Afterwards, in Section 5.2.2, we depict the interactions between the different system components.

5.2.1 Architectural Design

In this section, we describe the architectural design of the project. The architectural design is the description of the different components that configure the system.

The different modules that are identified in this description are:

- Driver. The first element of the compiler to be launched. The module is in charge of performing the analysis of the arguments that are received.
- Parser. The central module of the compiler. Its function is generating the AST.
- Preprocessor. It is in charge of managing the source file and to give it to the Lexer.
- Lexer. It tokenizes the file and provides it as they are requested to it.
- Sema. Its responsibility is generating each of the nodes of the AST.
- CodeGen. The aim of this module is generating intermediate code from the representation of the code.

- AST. This class is in charge of managing the attachment of nodes in a hierarchy.
- Symbols Table. It is a module to which other modules can appeal to obtain information on existing symbols.

Figure 5-1 represents the breakdown in components of Clang compiler main modules. The most important module of this component diagram is the Parser since it is the common point. It is the one in charge of controlling the compilation process following Parser-Driven Compilation. There are some additional modules in charge of giving different services such as the Preprocessor, the Lexical Analysis (Lexer), the Syntax Analysis (Parser) and the Semantic Analysis (Sema). The Driver and the Code Generator(CodeGen) are modules that act independently of the Parser. In addition to that, two additional modules are comprised of the design which are the Symbols Table and the Abstract Syntax Tree (AST). The interfaces are named by the with a key *Provider - Consumer*. The Consumer is the part of the relationship which makes use of the functionality, and the Provider is the relationship part which implements the functionality.

5.2.2 Functional Design

In this section, we describe the design of the process from an interaction point of view. We describe dynamically which are the interactions that are carried out in the project.

Algorithm 5.1 represents the Clang compilation process. The process has three main phases. The first phase is the Driver analysis of the compilation arguments. Those compilation arguments are processed and are used during the different phases of the compilation. After finishing the execution of the Driver, a Parser instance is created. The Parser instance generates all the modules it needs for the proper compilation of the code. Afterwards, the compilation process is performed. The sequence of acquiring a token, processing the token according to the grammar and adding it to the AST is repeated until the Abstract Syntax Tree (AST) is completely built. The third and last phase is the Code Generator (CodeGen) which uses the AST nodes to generate the intermediate code. This process is done iteratively with each of the subparts of the AST.

Figure 5-2 represents the interactions of the different components during the compilation process. It represents how each of the elements develops their own tasks. We can also depict the three phases that were described previously. The first phase consists on the analysis of arguments that the Driver performs. The Parser is the central part of the code and its main task is the construction of the AST. It is in charge of calling the rest of the modules for the compilation and it uses different calls to obtain elements from the other modules of the compiler. The final phase is the Code Generation done with each of the top-level declarations of the AST.

Algorithm 5.1 Clang Compilation Process

```
1: Driver := CreateDriver()
2: Driver.process_compile_options()
3: Parser := Driver.CreateParser()
4: Preprocessor := Parser.CreatePreprocessor()
5: Preprocessor.process_file(File)
6: Lexer := Preprocessor.CreateLexer()
7: Sema:= Parser.CreateSema()
8: AST := Parser.CreateAST()
9: while (There are more tokens) do
10:     Token := Lexer.token_request()
11:     Statement := Parser.create_stmt(Token)
12:     Statement := Parser.verify_stmt_with_grammar(Statement)
13:     ASTNode := Sema.verify_sema(Statement)
14:     AST.attach_node(ASTNode)
15: end while
16: CodeGen.code_gen_top_level_decl()
17: while (There are more Top Level Declarations) do
18:     TopLevelDeclaration := Parser.get_top_level_decl()
19:     CodeGen.generate_code(TopLevelDeclaration)
20: end while
21: Return BinaryFile
```

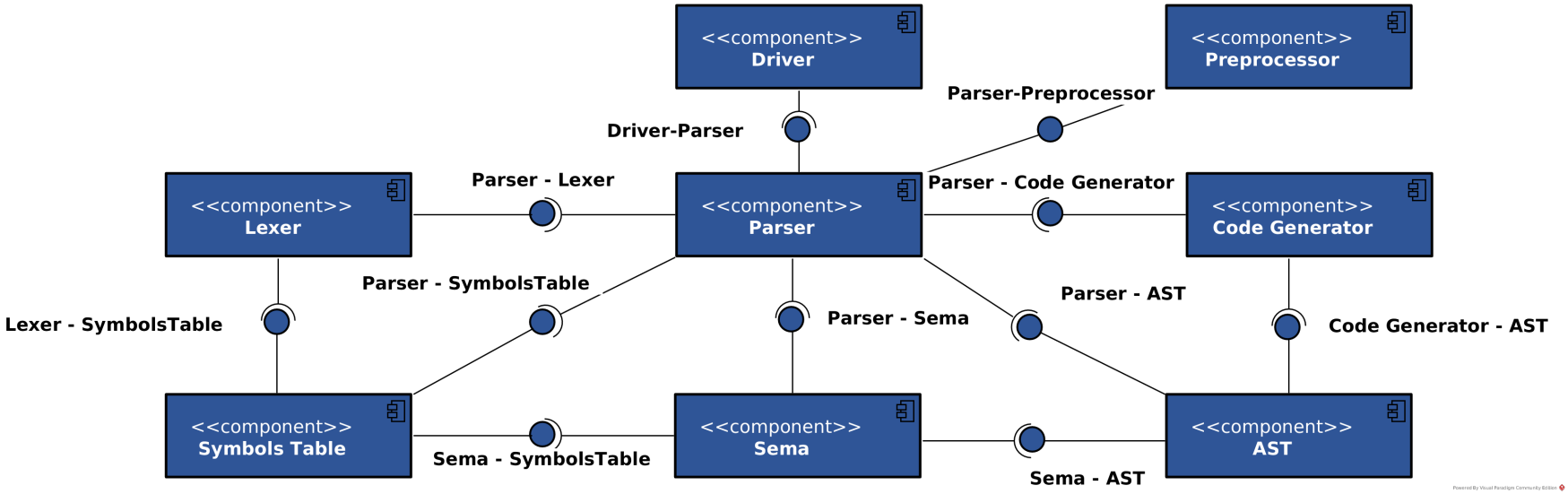

Figure 5-1: *Clang Component Diagram*

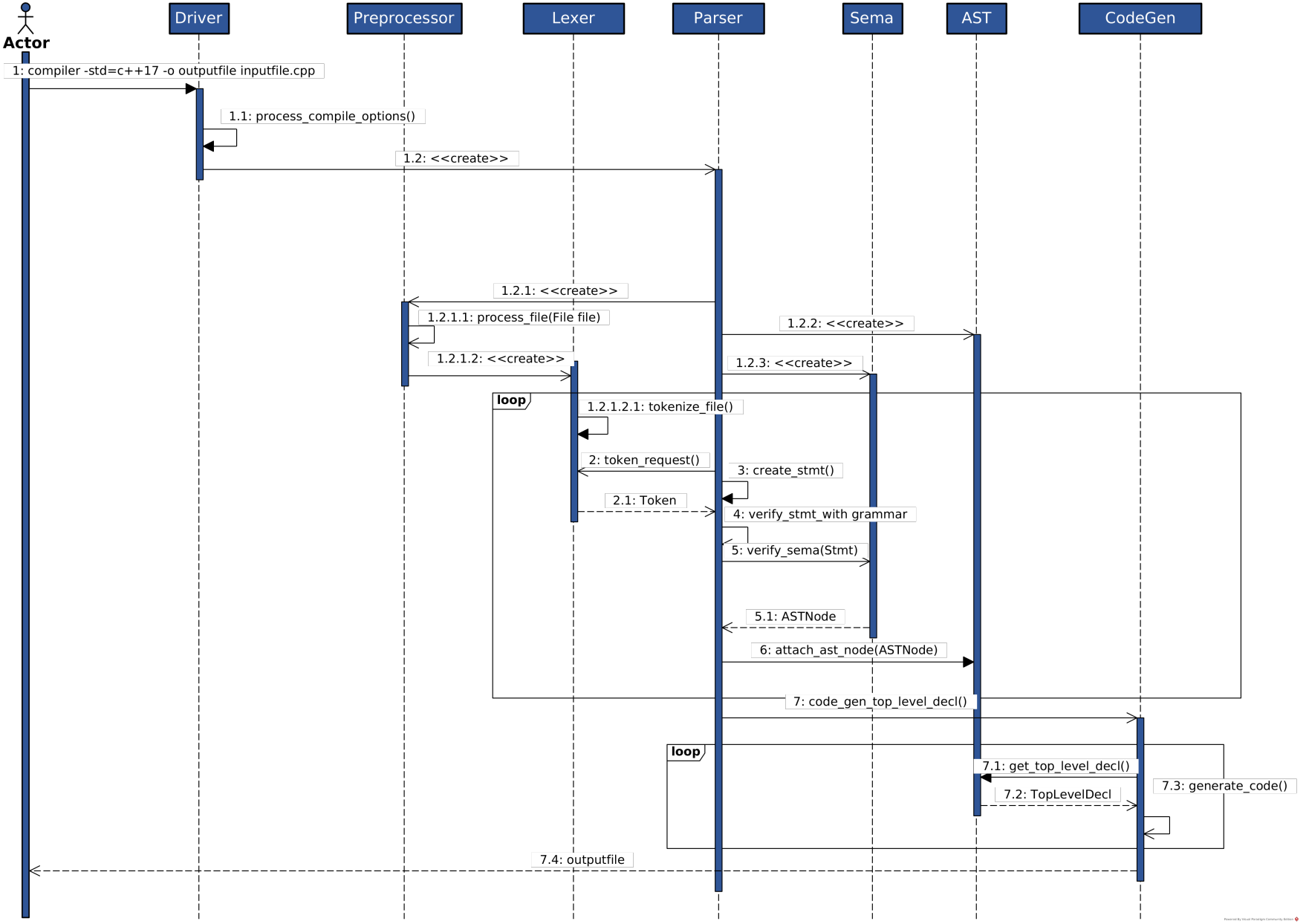

Figure 5-2: *Clang Sequence Diagram*

5.3 Proposed Clang Design

This section describes the modifications in the design reviewed in Section 5.2 for the inclusion of the features in requirement specification. Specifically, Section 5.3.1 explains which are the different modules included in the compiler for the development of the functionality. Apart from that, Section 5.3.2 explains the new interactions and relations included.

5.3.1 Architectural Design

This section defines the new modules that are necessary for the design of the functionality. The new modules are the following:

- *Driver.* This module receives a slight modification to support the argument parsing of the new compilation flags.
- *Contract Parsing Module.* This new module is used for the parsing of the new syntax for the attributes.
- *Contract Semantic Analysis Module.* This new module used for the semantic analysis of the new attributes.
- *Contract Code Generator Module.* It is used for the generation of code associated with a contract expression.

Figure 5-3 shows the connections between the new modules with the system. The modules coloured in blue are the modules that did not receive any modification. Modules coloured in purple are modules that received a modification. Modules coloured in yellow are newly included modules. We can observe that the Driver is the only module which was modified. Additionally, the three newly included modules are depicted. The Contract Parsing Module has relation exclusively with the Parser. The Contract Semantic Analysis module has an exclusive relationship with the Semantic Analysis module. The Contract Code Generator module has a relation with the Code Generator module.

5.3.2 Functional Design

In this section, we are describing the different interactions and relations of the new modules. We depict the different actions to be carried out by each of this modules.

The main tasks that each of the modules has to produce for the generation of a complete system functionality are:

- *Driver.* Since the main functionality of the Driver is to process compilation arguments, the new responsibility of the module is providing the processing of the new flags. The flags to be processed are the *build level* flag, the *continuation mode* flag and the *custom violation handler* flag.

- *Contract Parsing Module.* This module is designed to cooperate with the Parser. The functionality to be provided is to parse the new syntax of the contracts.
- *Contract Semantic Analysis Module.* This module interacts with Sema (Semantic Analysis module). The functionality that it provides is the semantic analysis of the new contracts.
- *Contract Code Generation Module.* This module is intended to relate with the Code Generation Module. It shall translate a contract in the code intermediate code that represents the action that wants to be performed.

Algorithm 5.2 represents in pseudo-code the modified compilation process with the addition of new features. The red marked parts are the newly introduced functionalities. Within this parts, we can specify four blocks. The first block consists of the calls that are performed in the modified Driver module. The second call represents the functionality associated with parsing the attributes. The third call represents the modifications to the Semantic Analysis. And the fourth block represents the calls related to the Code Generation of the contracts.

Figure 5-4 represents the order of the interactions in which the new modules act in the compilation. With respect to the Figure 5-2, we can observe the addition of the three new modules and the changes to one. We can observe that the functionality related to the Driver includes the flag argument processing. The Contract Syntax Module acts on the syntax analysis of the code together with the Parser. The Contract Semantic Analysis Module complements the functionality of the Semantic Analysis for the generation of the contracts AST Nodes. Finally, the Contract Code Generation Module complements the functionality of the Code Generation module adding the contracts intermediate code generation and the combination with the compilation flags.

Algorithm 5.2 Proposed Compilation Process

```
1: Driver := CreateDriver()
2: Driver.process_compile_options()
3: Driver.process_violation_handler()
4: Driver.process_build_level_flag()
5: Driver.process_continuation_flag()
6: Parser := Driver.CreateParser()
7: Preprocessor := Parser.CreatePreprocessor()
8: Preprocessor.process_file(File)
9: Lexer := Preprocessor.CreateLexer()
10: Sema:= Parser.CreateSema()
11: AST := Parser.CreateAST()
12: while (There are more tokens) do
13:     Token := Lexer.token_request()
14:     Statement := Parser.create_stmt(Token)
15:     Statement := Parser.verify_stmt_with_grammar(Statement)
16:     ContractParsingModule.parse_contract_attribute()
17:     Sema.verify_sema(Statement)
18:     ASTNode := ContractSemanticModule.generate_contract_node()
19:     AST.attach_node(ASTNode)
20: end while
21: CodeGen.code_gen_top_level_decl()
22: while (There are more Top Level Declarations) do
23:     TopLevelDeclaration := Parser.get_top_level_decl()
24:     CodeGen.generate_code(TopLevelDeclaration)
25:     if (ContractCodeGeneratorModule.compare_build_level_and_assertion_level()) then
26:         ContractCodeGeneratorModule.generate_if_stmt()
27:         ContractCodeGeneratorModule.generate_call_to_custom_handler()
28:         if (ContractCodeGeneratorModule.evaluate_continuation_flag()) then
29:             ContractCodeGeneratorModule.generate_abort_call()
30:         end if
31:     end if
32: end while
33: Return BinaryFile
```

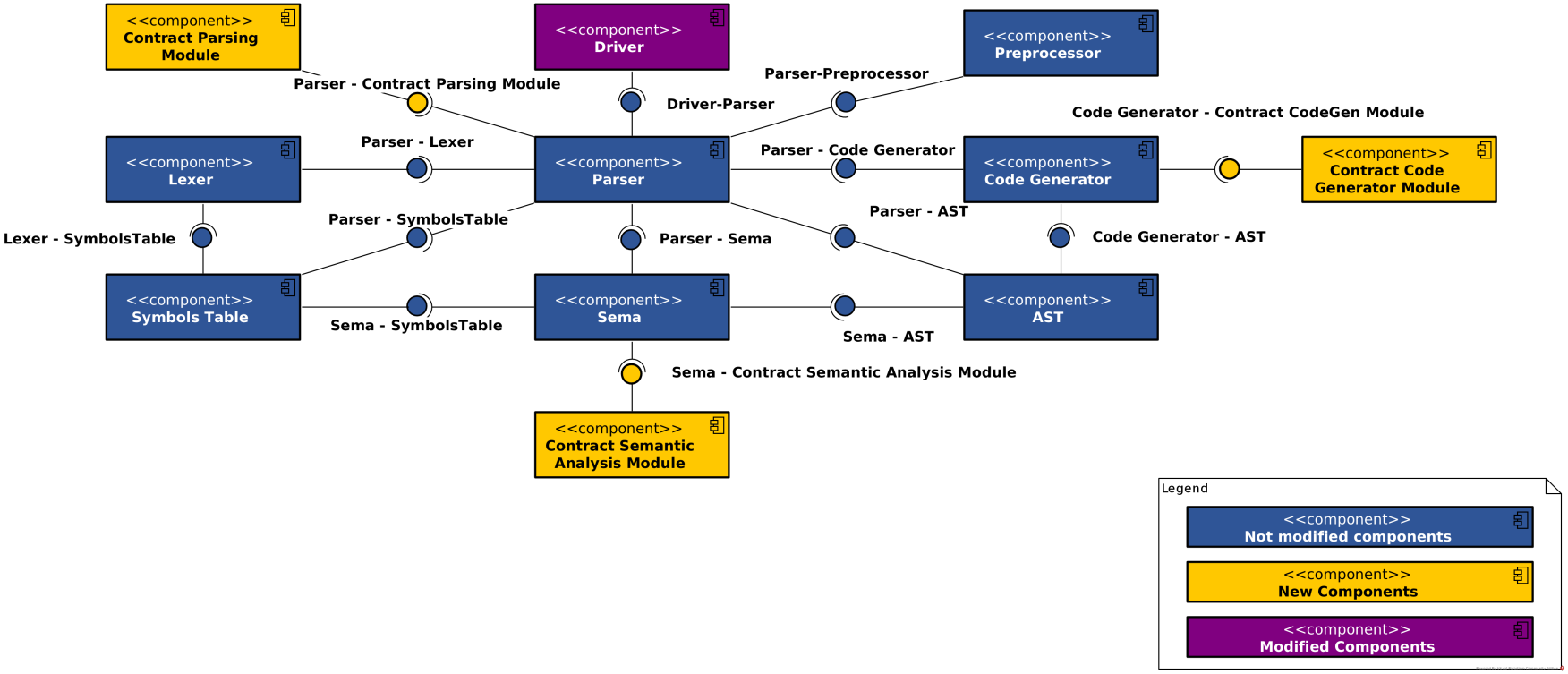

Figure 5-3: *Proposed Design Component Diagram*

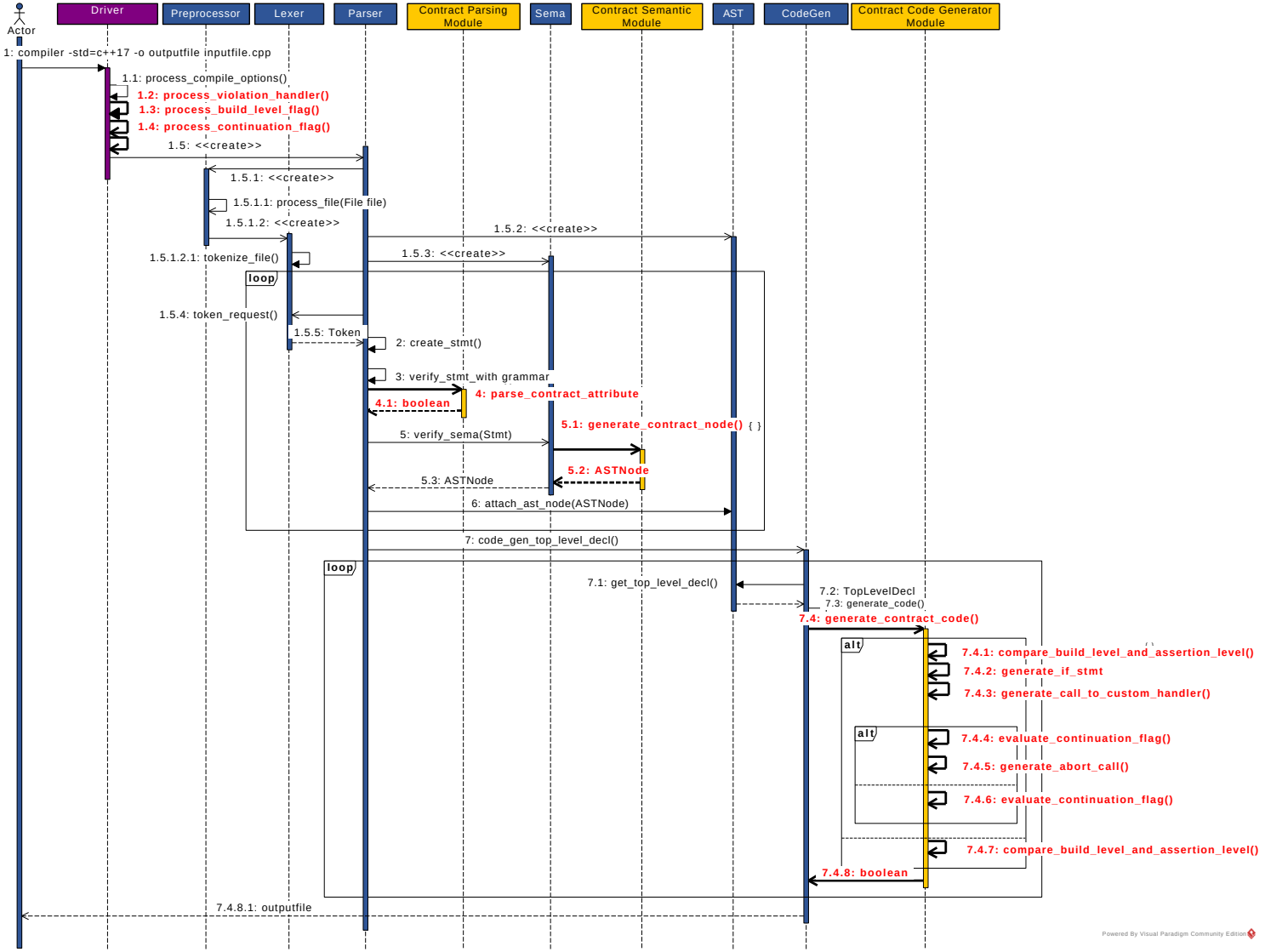

Figure 5-4: *Proposed Design Component Diagram*

5.4 Detailed Design of the Solution

In this section, we present the detail design of the project. In the detailed design, we describe which are the specific changes that need to be performed and how they have been designed performed. Specifically, in Section 5.4.1, we detail the flags that need to be introduced for the user to control the compiler. Afterwards, in Section 5.4.2, the specific design of the basic functionality of the contracts is broken down into the different modules that have been specified. Finally, Section 5.4.3 represents the modifications on the modules to allow the execution of a custom violation handler.

5.4.1 Compiler Flags

This section describes the changes performed to the Driver module so that it supports the new compilation flags.

In order to give support to some of the interactions of the user with the compiler, we need to support certain compilation flags. A flag is a specific mechanism of communication between the user interface and the modules. The compiler internally stores the different flags introduced so that any of the modules that make use of it, is able to access it and verify the value associated to it (in case it has it). For this communication, the following compilation flags have been introduced:

- *Build Level.* The build level flag is used for the activation or deactivation of contracts according to their assertion levels. In what respects to the Driver module, it has to ensure that the syntax followed is either *--build-level=<level>* or *-build-level=<level>* where level is either *off*, *default* or *audit*.

- *Continuation Mode.* This flag is used to avoid the program execution to abort in case that there is a contract violation. This means that to abort on a violation the flag shall not be present. The Driver module shall verify the syntax to be either *--enable-continue-after-violation* or *-enable-continue-after-violation*.

- *Custom Violation Handler.* This flag defines the name of the function that the user wants to use as violation handler. The Driver shall ensure that the syntax followed is either *--contract-violation-handler=<function>* or *-contract-violation-handler=<function>* where function shall be the function name of the violation handler.

5.4.2 Basic Functionality

In this section we describe the main functionality included in the different modules in order to support a basic functionality of the contract specification.

Firstly, we recap the basic functionality that the modules shall comprise. The basis of the specification is the generation of a set of new statements that are called contracts. These contracts evaluate a condition. In case that the evaluation of condition is true, the execution of the program is not affected. However, if the evaluation of the condition is false, a violation is raised. There are three different keywords supported in the specification *assert*, *expects* and *ensures*.

- The *assert* contracts are placed on a executable region of the code (associated to a *null statement*). The condition of an *assert* is evaluated when the contract it is reached by the program flow.
- The *expects* contract is associated to the *function type*. The condition subject can be an accessible[1] pre-existing variable or a function attribute. The *expects* condition is evaluated before the execution of the function.
- The *ensures* contract is associated to the *function type*. The condition subject can be an accessible[1] pre-existing variable or a function argument but it is meant to be related to the future return value. The *ensures* contract clause permits pre-defining a name to identify the return value and operate with it. The condition is evaluated after the execution of the function.

Each of this contract have a different syntaxes. The *assert* syntax is *[[assert assertion-level: condition]]*. The *expects* syntax is *[[expects assertion-level: condition]]*. Finally the *ensures* syntax that must be fulfilled is *[[ensures assertion-level return-value: condition]]*.

The assertion level present in the syntax is used to activate and deactivate the evaluation of contracts when compiling. The assertion level is compared with the value given in the build level flag and this comparison determines whether a contract is evaluated or not.
From this point, this section describes the changes that needed to include the different modules in order to allow the functionality explained.

Contract Parsing Module

In this section, we describe the design of the Contract Parsing Module. The Contract Parsing Module shall implement changes to the grammar in order to support the new syntax.

Listing 5.1 shows the changes to the grammar for the support of contracts. Terminal symbols are differentiated by being surrounded with single quotation marks. The rest of symbols are considered non-terminal. Each of the different lines of a production represents the alternatives of a production from a non-terminal. It implies the change of one production rule and the addition of two new rules. The modified rule is the *attribute* non-terminal rule. This rule permits the non-terminal to transform into

[1]We determine to be accessible to any variable that could be accessed within the body of the function to which the contract is associated.

a C++ attribute or a C++ attribute list. The modification consists of adding the transformation of the non-terminal into a contract attribute specifier. The *contract attribute specifier* non-terminal expansions are the supported syntaxes for each of the attributes (*assert*, *expects* and *ensures*). With regard to the last rule, it formalizes the transformations of the *contract-level* non-terminal into the different accepted values (*axiom, always, default and audit*).

Listing 5.1: *Grammar Changes*

```
1  attribute:
2          attribute-token
3          attribute-argument-clause[opt]
4          contract-attribute-specifier
5  contract-attribute-specifier:
6         '['']' 'expects' contract-level[opt] ':' conditional-expression ']'']'
7         '['']' 'ensures' contract-level[opt] identifier[opt] ':' conditional-expression ']'']'
8         '['']' 'assert' contract-level[opt] ':' conditional-expression ']'']'
9  contract-level:
10         'axiom'
11         'default'
12         'audit'
13         'always'
```

Contract Semantic Analysis Module

This section describes the design of the Contract Semantic Analysis Module. This module shall evaluate semantically the placement of the attribute and it is in charge of generating the attribute from the parts obtained. Among the responsibilities that the Semantic Analysis module we distinguish:

- *Validate association.* The contract shall be placed with a *null statement* in the case of *assert* and associated to a *function declaration* in case of *expects* and *ensures*.
- *Validate condition.* The Semantic Analysis Module validates that the introduced condition is contextually convertible to a boolean. This means that once the condition can be evaluated with the information at that point to true or false.
- *Validate the arguments.* The Contract Semantic Analysis Module shall validate that the number of arguments associated with a contract is at most the expected in the syntax.
- *Create the AST Node.* The Contract Semantic Analysis Module shall be responsible for creating the AST Node of the Contract if the data matches the expected

Algorithm 5.3 presents the pseudo-code of the verifications described for a contract. It validates all the conditions that a new attribute shall have and in case that all the verifications are passed correct, a new AST Node is created.

Algorithm 5.3 Proposed Semantic Analysis

```
1: if (Contract has expected number of attributes) then
2:     if (Condition is contextually convertible to boolean) then
3:         if (Contract is associated to correct statement) then
4:             ASTNode := generate_ast_node_for_contract()
5:             Return ASTNode
6:         end if
7:     end if
8: end if
```

Contract Code Generation Module

This section evaluates the design of the Contract Code Generation Module. The responsibility of this module is to transform the contract AST Node into intermediate code. The transformation shall be done so that the evaluation works as it is specified in the requirements. The responsibilities of this module are:

- *Evaluate the code generation.* The Contract Code Generation module shall evaluate whether the contract intermediate code has to be generated or not. The evaluation is done by comparing the build level and the assertion level of each contract.
- *Generate a contract validation.* The contract code shall be transformed into an *if statement* where the condition is the negated contract condition.
- *Generate the call to abort.* The module shall evaluate the value of the continuation flag that was depicted in 5.4.1 and generate an execution interruption in case the flag is not present.

Listing 5.2: *assert attribute code*

```
1 ...
2 [[assert: x > 0]];
3 ...
```

Listing 5.3: *assert attribute equivalence*

```
1 ...
2 if (!(x > 0)){
3     violation_handler();
4     stop_execution();
5 }
6 ...
```

Listings 5.2 and 5.3 represents the equivalence in code of what an assert attribute provokes in the code. As it is depicted the condition is negated and the body of the if statement has two consequences,

the execution of a violation handler and the abort of the execution.

Algorithm 5.4 presents the behaviour that the Contract Code Generation Module. It represents the actions that shall carry out when dealing with the AST Node of a contract. We can observe the comparison of the assertion level with the build level flag and the verification of the continuation mode. We can also see the negation of the contract condition and the generation of the *if statement* with that condition.

Algorithm 5.4 Proposed Code Generation

```
1: if (BuildLevel > AssertionLevel) then
2:     Condition := NegateCondition(Contract_Condition)
3:     ThenStatement := generate_contract_violation_handler();
4:     if (Continuation Flag is not present) then
5:         CallToAbort := generate_call_to_abort()
6:         ThenStatement += CallToAbort()
7:     end if
8:     IfStatement := generate_if_statement(Condition, ThenStatement);
9:     EmitStatement(IfStatement)
10: end if
```

Finally we are depicting the breakdown of the functionality for *expects* and *ensures*. In the case of assert transforming it to an if statement is possible because of its placement in an executable region of code. However in the case of the *expects* and *ensures* it shall be modified. A transformation shall be done from the original function which is annotated with contracts to a function which performs the verifications and is not annotated with contracts.

Listings 5.4 and 5.5 represent the transformation that is performed to a code with *expects* and *ensures* contracts. The first change is that the function is added the suffix "_unchecked". A new function is created which evaluates the preconditions, then executes the *unchecked* function and then checks the post condition.

Listing 5.4: *expects and ensures code*

```
1  int square(int x)
2  [[expects: x > 0]]
3  [[ensures ret: ret == x * x]]
4  {
5      return x * x;
6  }
7  int main(int argc, char* argv[]){
8      int x = square(10);
9      return 0;
10 }
```

Listing 5.5: *expects and ensures equivalence*

```
1  int square_unchecked(int x)
2  [[expects: x > 0]]
3  [[ensures ret: ret == x * x]]
4   __attribute__((always_inline)) //Inline
5   {
6      return x * x;
7  }
8  //New function
9  int square(int x)
10 {
11     //Preconditions evaluation
12     if(!(x > 0)){ std::terminate();}
13     //Function execution
14     int ret = square(x);
15     //Postconditions evaluation
16     if(!(ret == x * x)){ std::terminate();}
17     return ret;
18 }
19 int main(int argc, char* argv[]){
20     //Modified function call
21     int x = square(10);
22     return 0;
23 }
```

5.4.3 Violation Handler

In this section, it is presented how the user is able to introduce custom violation handlers. The introduction of custom violation handlers belongs to the Contract Code Generation Module responsibilities. This functionality permits the execution of a custom handler when a violation is raised. For this functionality, we need the name of the function that the user passed with the compilation flag in Section 5.4.1. The steps that are needed for the execution of a custom violation handler are:

- *Finding the function declaration*. The Contract Code Generation Module needs to find the symbol of the custom violation handler in the symbols table. This functionality is also needed to be generated.

- *Generate contract violation information*. The information of the file, line, condition and comment of

the violation is extracted and stored within the instance of an object automatically generated.

- *Generate the function call*. The module shall generate a function call to the function declaration found. The information on the contract violation is passed as a parameter.

Algorithm 5.5 shows the pseudo-code of the different interactions that are needed for the generation of the custom violation handler. In a first step, the extraction of a function call is done from the symbols table. Then, the violation information is extracted. Finally, the call to the violation handler is generated and executed.

Algorithm 5.5 Proposed Violation Handler Generation

```
1: ViolationHandler := SymbolsTable.find_function_call(FunctionName)
2: ViolationInformation := extract_violation_information()
3: ViolationHandlerCall := generate_call_to_handler(ViolationHandler, ViolationInformation)
```

Chapter 6

Evaluation

This chapter presents the evaluation of the proposed C++ contract implementation. Specifically, Section 6.1 demonstrates the correctness of the implementation. Afterwards, Section 6.2 shows the performance evaluation of the proposed implementation of C++ contracts.

6.1 Conformance Tests

In this section, we evaluate the correctness of the proposed C++ contract implementation. Specifically, we check the accomplishment of the requirement specification through a set of tests. To do so, we leverage **lit**[47], a tool part of the LLVM infrastructure that allows to automatise the execution of the test suite. Additionally, this tool is able to evaluate the correct execution of the test.

To evaluate the correctness of the proposal, we classify the tests into two main groups:

1. **Compiler Tests.** These lit-based tests have been designed to identify common programming errors, such as the wrong placement of elements in the code or syntax errors. Additionally, these tests cover all of the contract statements and assertion levels.

2. **Behaviour Tests.** These tests evaluate the run-time behaviour of the generated binary, i.e. the program aborts its execution if a contract is violated.

6.1.1 Compiler Behaviour Tests

This section describes the different Compiler Behaviour Tests that have been developed and the main focus that this tests target.

These tests evaluate the correctness of the compiler when having to analyze a statement in C++. These tests evaluate the following actions:

- The correct placement of the attribute in a normal code sentence. In the case of *assert*, it shall be done associated to a *null statement* and in the case of *expects* and associated to the function type.
- The wrong placement and association of the attribute in a normal code sentence.
- The automatic cast to a boolean expression.
- The usage of the attributes within C++ Method Declarations which are treated differently than Function Declarations.
- The wrong usage of a non-existing assertion level.

These tests are performed for each of the contract clauses, *assert, expects* and *ensures* and for each *assertion level* of the former. In the special case of *expects* and *ensures* some tests evaluate special cases that the C++ syntax allowed. Those tests regard the following aspects:

- The application of contracts to lambda functions or functors that shall not be permitted.
- The inheritance of attributes (by now is not supported).

6.1.2 Executable Behaviour Tests

This section describes the Executable Behaviour Tests. These tests evaluate the expected actions of an executable compiled with the contracts. These tests are exception-based-test that are made thanks to the custom contract violation handler functionality.

These set of tests are performed in batteries of 60 unitary tests. A battery is performed for each of the build levels (*off, default and audit*) and no build level. Each battery of test includes testing on the following features:

- The evaluation of the condition is checked both in an affirmative case (no violation expected) and in a negative case (violation expected).
- All the assertion levels (*axiom, always, default and audit*) are checked.
- The continuation flag is evaluated since the test work without aborting.
- The build level flag is evaluated for each of the levels.
- The custom violation handler is checked since it is a fundamental element.
- The tests annotations cover both functions and class methods.
- The tests are performed with normal variables.

- The tests are performed with templatized variables. Templatized variables are differently treated in compilers. Templates are covered in three ways template class, template function and templatized member of a templatized class.

6.1.3 Test Suite

In this section we give a description of the testing that is performed over each of the requirements that were depicted in previous section. This tables are very big since explaining each of the tests would imply a lot of time. We have to remind that this tables are executed for the specific contract that they are associated to, but they are executed for all the build levels that are possible. The fields of each of the tests are the following:

- **UT-XX.** This represents the test identifier, where XX is the number of the test.
- **Name.** This field is a short description of what the test does.
- **Requirements Covered.** It defines the list of the requirements that are verified with this tests.
- **Result.** This field represents the result of executing the tests over the implementation. It can have two results, either ✓ Validated in case the test is passed or ✗ Not Validated in case the test is not passed.
- **Description.** This field describes what are the characteristics that are evaluated over the text with regard to the requirements evaluated.

Tables 6.2, 6.3, 6.4, 6.5, 6.6 and 6.7 represent the different tests that were developed for this functionality.

Tables 6.8 and 6.9 show the tests that verify each of the requirements. The important point on this matrices is that no requirement is left unchecked. Since the functionality usually depends on several features, multiple tests might be verifying the proper behaviour of a requirement.

UT-XX	
Name:	Test Name
Requirements Covered:	SR-FR-XX
Result:	✓ Validated ✗ Not Validated
Description:	Test Description

Table 6.1: *Test Template Table*

UT-01	
Name:	Compiler Behaviour Test - *assert*
Requirements Covered:	SR-FR-01, SR-FR-04, SR-FR-09, SR-FR-10, SR-FR-12, SR-FR-13, SR-FR-14, SR-FR-20, SR-FR-36
Result:	✓ Validated
Description:	In this test we evaluate the compiler identifying common errors in the placement of *assert* contract such as: • Using a the proper syntax. • The code transformation into a condition. • The code association to a null statement. • The code supplying four assertion levels. • The generation of *axiom* contracts. • The generation of *always* contracts. • The evaluation of direct values. • The use of default assertion level.

Table 6.2: *Unit Test-01*

UT-02	
Name:	Compiler Behaviour Test - *expects*
Requirements Covered:	SR-FR-02, SR-FR-04, SR-FR-07, SR-FR-10, SR-FR-11, SR-FR-12, SR-FR-13, SR-FR-14, SR-FR-20, SR-FR-31, SR-FR-32, SR-FR-33, SR-FR-35, SR-FR-36
Result:	✓ Validated
Description:	In this test we evaluate the compiler identifying common errors in the placement of *expects* contract such as: • Using a proper syntax. • The code transformation into a condition. • The code association to a function type. • The code supplying four assertion levels. • The generation of *axiom* contracts. • The generation of *always* contracts. • The generation for different build levels. • The evaluation of direct values. • The warning in the usage on lambda functions. • The warning on the usage on function pointers. • The access to forbidden variables. • The use of default assertion level.

Table 6.3: *Test-02*

UT-03	
Name:	Compiler Behaviour Test - *ensures*
Requirements Covered:	SR-FR-02, SR-FR-04, SR-FR-08, SR-FR-10, SR-FR-11, SR-FR-12, SR-FR-13, SR-FR-14, SR-FR-20, SR-FR-31, SR-FR-32, SR-FR-33, SR-FR-35, SR-FR-36
Result:	✓ Validated
Description:	In this test we evaluate the compiler identifying common errors in the placement of *ensures* contract such as: • Using a proper syntax. • The code transformation into a condition. • The code association to a function type. • The code supplying four assertion levels. • The generation of *axiom* contracts. • The generation of *always* contracts. • The generation for different build levels. • The evaluation of direct values. • The warning in the usage on lambda functions. • The warning on the usage on function pointers. • The access to forbidden variables. • The use of default assertion level.

Table 6.4: *Test-03*

UT-04	
Name:	Executable Behaviour Test - *assert*
Requirements Covered:	SR-FR-01, SR-FR-04, SR-FR-05, SR-FR-06, SR-FR-10, SR-FR-12, SR-FR-13, SR-FR-14, SR-FR-15, SR-FR-16, SR-FR-17, SR-FR-18, SR-FR-19, SR-FR-20, SR-FR-21, SR-FR-22, SR-FR-23, SR-FR-24, SR-FR-25, SR-FR-26, SR-FR-27, SR-FR-28, SR-FR-29, SR-FR-30, SR-FR-31, SR-FR-34, SR-FR-36, SR-FR-37
Result:	✓ Validated
Description:	In this test we evaluate the compiler performing the proper behaviour of an executable with *assert* contract verifying the following conditions: • Using a proper syntax for assert. • The code transformation into a condition. • The code aborting if condition is not fulfilled. • The code continuing if condition is fulfilled. • The code supplying four assertion levels. • The generation of *axiom* contracts. • The generation of *always* contracts. • The generation for different build levels and its support. • The execution of contracts according to the build level. • The evaluation of direct values. • The usage of the continuation flag to provide support to the test suite. • The information of the violation is given to the user. • The ability to generate a custom contract violation handler. • The warning in the usage on lambda functions. • The warning on the usage on function pointers. • The access to forbidden variables. • The use of default assertion level. • The usage of templates variables of any kind. • The generation of a default handler.

Table 6.5: *Unit Test-04*

UT-05	
Name:	Executable Behaviour Test - *expects*
Requirements Covered:	SR-FR-02, SR-FR-04, SR-FR-05, SR-FR-06, SR-FR-10, SR-FR-12, SR-FR-13, SR-FR-14, SR-FR-15, SR-FR-16, SR-FR-17, SR-FR-18, SR-FR-19, SR-FR-20, SR-FR-21, SR-FR-22, SR-FR-23, SR-FR-24, SR-FR-25, SR-FR-26, SR-FR-27, SR-FR-28, SR-FR-29, SR-FR-30, SR-FR-31, SR-FR-34, SR-FR-36, SR-FR-37
Result:	✓ Validated
Description:	In this test we evaluate the compiler performing the proper behaviour of an executable with *expects* contract verifying the following conditions: • Using a proper syntax for expects. • The code transformation into a condition. • The code aborting if condition is not fulfilled. • The code continuing if condition is fulfilled. • The code supplying four assertion levels. • The generation of *axiom* contracts. • The generation of *always* contracts. • The generation for different build levels and its support. • The execution of contracts according to the build level. • The evaluation of direct values. • The usage of the continuation flag to provide support to the test suite. • The information of the violation is given to the user. • The ability to generate a custom contract violation handler. • The warning in the usage on lambda functions. • The warning on the usage on function pointers. • The access to forbidden variables. • The use of default assertion level. • The usage of templates variables of any kind. • The generation of a default handler.

Table 6.6: *Test-05*

UT-06	
Name:	Executable Behaviour Test - *ensures*
Requirements Covered:	SR-FR-02, SR-FR-04, SR-FR-05, SR-FR-06, SR-FR-10, SR-FR-12, SR-FR-13, SR-FR-14, SR-FR-15, SR-FR-16, SR-FR-17, SR-FR-18, SR-FR-19, SR-FR-20, SR-FR-21, SR-FR-22, SR-FR-23, SR-FR-24, SR-FR-25, SR-FR-26, SR-FR-27, SR-FR-28, SR-FR-29, SR-FR-30, SR-FR-31, SR-FR-34, SR-FR-36, SR-FR-37
Result:	✓ Validated
Description:	In this test we evaluate the compiler performing the proper behaviour of an executable with *ensures* contract verifying the following conditions: • Using a proper syntax for ensures. • The code transformation into a condition. • The code aborting if condition is not fulfilled. • The code continuing if condition is fulfilled. • The code supplying four assertion levels. • The generation of *axiom* contracts. • The generation of *always* contracts. • The generation for different build levels and its support. • The execution of contracts according to the build level. • The evaluation of direct values. • The usage of the continuation flag to provide support to the test suite. • The information of the violation is given to the user. • The ability to generate a custom contract violation handler. • The warning in the usage on lambda functions. • The warning on the usage on function pointers. • The access to forbidden variables. • The use of default assertion level. • The usage of templates variables of any kind. • The generation of a default handler.

Table 6.7: *Test-06*

6.1.4 Traceability matrices

	Compiler Behaviour Tests															Executable Behaviour Tests											
	assert					expects					ensures					assert				expects				ensures			
	nothing	**always**	**axiom**	**default**	**audit**	**nothing**	**always**	**axiom**	**default**	**audit**	**nothing**	**always**	**axiom**	**default**	**audit**	**test**	**test_off**	**test_default**	**test_audit**	**test**	**test_off**	**test_default**	**test_audit**	**test**	**test_off**	**test_default**	**test_audit**
SR-F-01	✓	✓	✓	✓	✓											✓	✓	✓	✓								
SR-F-02						✓	✓	✓	✓	✓										✓	✓	✓	✓				
SR-F-03											✓	✓	✓	✓	✓									✓	✓	✓	✓
SR-F-04	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-05																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-06																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-07						✓	✓	✓	✓	✓																	
SR-F-08											✓	✓	✓	✓	✓												
SR-F-09	✓	✓	✓	✓	✓																						
SR-F-10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-11						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
SR-F-12	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-13			✓					✓					✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-14		✓					✓					✓				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-15																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-16																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-17																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-18																	✓				✓				✓		

Table 6.8: *Traceability Matrix Software Requirements (SR-FR-01 to SR-FR-18) to Tests*

	Compiler Behaviour Tests															Executable Behaviour Tests												
	assert					expects					ensures					assert				expects				ensures				
	nothing	**always**	**axiom**	**default**	**audit**	**nothing**	**always**	**axiom**	**default**	**audit**	**nothing**	**always**	**axiom**	**default**	**audit**	**test**	**test_off**	**test_default**	**test_audit**	**test**	**test_off**	**test_default**	**test_audit**	**test**	**test_off**	**test_default**	**test_audit**
SR-F-19																		✓				✓				✓	
SR-F-20	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-21																			✓				✓				✓
SR-F-22																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-23																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-24																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-25																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-26																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-27																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-28																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-29																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-30																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-31						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓														
SR-F-32						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓														
SR-F-33						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓														
SR-F-34																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-35						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓														
SR-F-36	✓					✓					✓					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SR-F-37																✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.9: *Traceability Matrix Software Requirements (SR-FR-19 to SR-FR-37) to Tests*

6.2 Performance Tests

This section defines the use case that has been developed for proving the performance and validity of the implementation. These tests provide an overview of what is going to be the impact on performance. Specifically, in Section 6.2.1 we describe the objective of the use case. Later on, in Section 6.2.2 we describe how modifications to an existing library were performed. In Section 6.2.4 we describe the device where the tests have been performed. Afterwards, in Section 6.2.3 we define the different tests that are performed in the benchmark. Finally in 6.2.5, we explain the results that we have obtained from executing this benchmark.

6.2.1 Overview

In this section an overview of the process followed to evaluate the implementation with a benchmark is depicted.

The objective is to make an evaluation of the impact on performance. For this purpose, a use case is designed. It consists of the modification of the STL basic_string and its annotation with contracts. The class *basic_string* was modified by means of an automatic script that was also developed as a part of this evaluation. The script substituted assertions in GNU notation to the new C++ notation. Then a total of 6 test benchmarks are developed in which the strings from the library are intensively used. These test benchmarks are used for the performance evaluation.

It is important to remark that in order to imitate the behaviour that the STL basic_string, the violation handler specified for this tests will be set to `std::terminate` and no custom handler will be assesed. This is because a custom handler can include any code the user decides to introduce. However the relative impact of its usage can be evaluated on further work.

6.2.2 Modification of basic_string

In this section, we detail which were the steps that were carried out for the modification of the `basic_string` class in an automated way. The modification was done by means of an automatic script in Perl programming language. Perl is a programming language which is mainly used for text parsing and modification. It provides a lot of tools to perform changes easily on files and different elements. For this reasons is why we chose Perl for this modifications.

The script analyzes a source code file. It iterates over all the lines of the code trying to find any of the assertions. When it finds one matching the regular expression, the line is modified to change the syntax to the new one. For that first, the condition is extracted from the GNU assertion and then it is introduced into a template of an assert or expects clause. In case that the assertion needs to be transformed to a

expects clause backtracking is performed to place it where it corresponds. This is caused because all GNU assertions are equivalent to assert but we are interested in testing all.

At the end of this section, we already have two versions of basic_string, the annotated version and the original version. With these libraries, we perform the benchmark in next section.

6.2.3 Test Benchmark

In this section, we describe the set of benchmarks used for the performance evaluation of the proposed C++ contracts implementation. As stated in the previous section, we modified the `basic_string` class of the STL library. Since the `std::basic_string` is the underlying template class of `std::string`, we propose a set of six tests that perform intensive string manipulation:

Benchmark 1 generates a vector of n random strings of a length which may vary between 1 and n. This vector is the main work unit. The vector of strings is ordered according to the bubble sort algorithm. Right after that they the vector is again randomly ordered. Next the vector is ordered by inverse size. Finally the strings of the vector are reordered again randomly. This process of ordering, disordering, ordering, disordering is repeated 100 times times for each n. With regard to the value of n, it goes from 100 to 6000 with a increment of 100 each time.

Benchmark 2 generates a random string of length n. It accesses random positions of the string swapping their values to randomize more the strings. This whole process is repeated 10000 times for each n. The size of the string goes increasing from 100 to 6000 with increments of 100.

Benchmark 3 generates a vector of n random strings of a length n. This vector is the main work unit. The vector is ordered alphabetically according to the bubble sort algorithm. Then the vector gets its positions randomized. Later on it is ordered inverse alphabetically. Finally the vector is randomized again. This process is repeated 100 times for each n for n going from 100 to 6000 in intervals of 100.

Benchmark 4 generates a vector of n random strings of length n. This strings are translated to lower case and then to upper case. This whole process is repeated 100 times for each n for values of n starting at 100 and growing by increments of 100 up to 6000.

Benchmark 5 generates a vector of random strings of length n. The strings will be reformated to order the characters of those strings alphabetically. This process will be done for each of the strings of the vector. The process is repeated for values of n starting at 100 going up to 6000 in intervals of 100.

Benchmark 6 is in charge of performing a substring substitution. A random string of size $n^2 * 10$ will be generated, and the operation of finding the substring "abc" and substitute it by "def". It iterates for values of n from 100 to 6000 in intervals of 100.

6.2.4 Benchmarking Environment

In this section we describe the benchmarking environment that was used for the evaluation of the tests. This includes both the description of hardware and software used for the tests.

With regard to the hardware a node from the Tucan Cluster that the ARCOS (Computer Structure and Architecture Group) was used for this experiment. A cluster is a set of computing nodes which are coordinated from a front-end node through which tasks can be launched to all of them. For this benchmark we only used a node. This node accounted with an *Intel(R) Xeon(R) CPU E5-2630 v3* processor with *8* cores running at *2.4GHz* each, *378 GB of RAM*.

With regard to the software, the node was running *Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-79-generic x86_64)*. The version of the Clang compiler used is version 6.0.0. The compiler with which the source code of Clang was compiled was GCC 7.2.0.

6.2.5 Evaluation

In this section, we evaluate the results that were obtained. The results correspond to the execution of the programs both with and without the basic_string library. We will be comparing the versions compiled with the optimization flags *-O2* and *-O3* since are the versions that are used in release versions. In addition, not using any of this optimization flags would always result in the contracts version since it would imply the verification of more conditions than not using it.

Benchmark 1

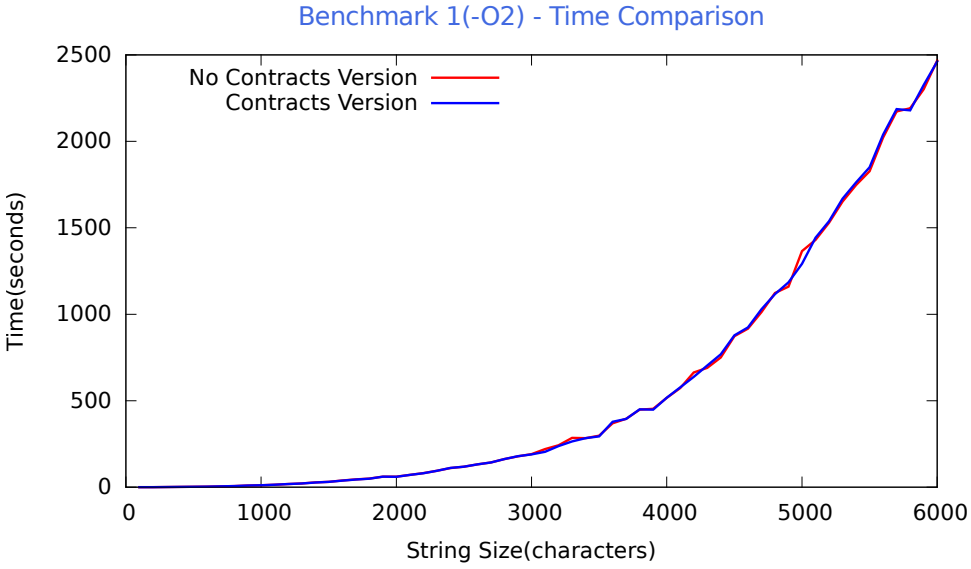

Figure 6-1: *Benchmark 1: Time comparison of Contracts and No Contracts -O2.*

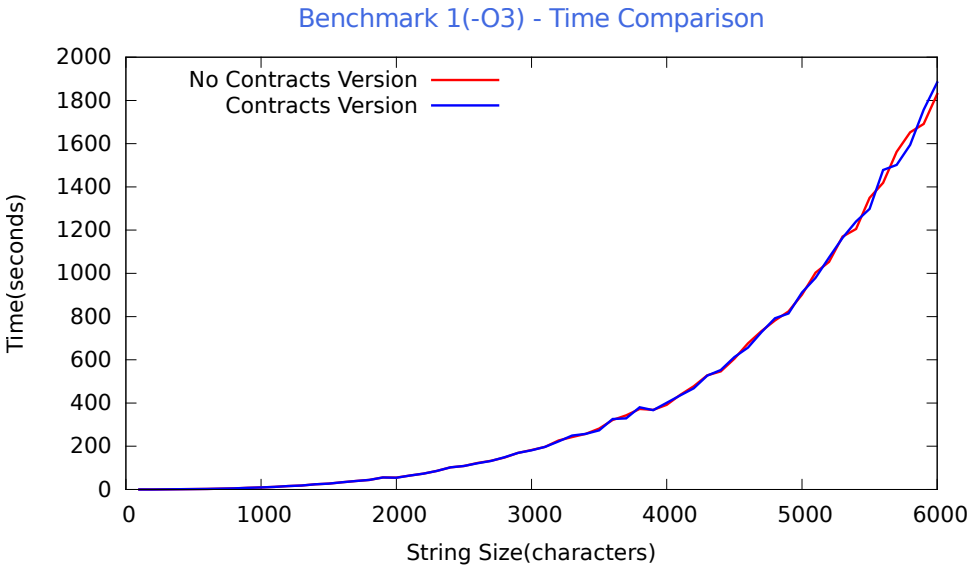

Figure 6-2: *Benchmark 1: Time comparison of Contracts and No Contracts -O3.*

Figure 6-1 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O2*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. Hence, the graph does not provide a lot of information since it is very tied in both executions. The only observ-

able thing is that one line or another remains on top or below depending on the specific test case.

Figure 6-2 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O3*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. Hence, the graph does not provide a lot of information since it is very tied in both executions. The only observable thing is that one line or another remains on top or below depending on the specific test case.

Figure 6-3 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O2*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. In this case, we observe that for lower values of the string size, contracts are an advantage in performance, whereas for larger string sizes the no contracts alternative get better results. The improvement in performance reaches an average of 2% for this version.

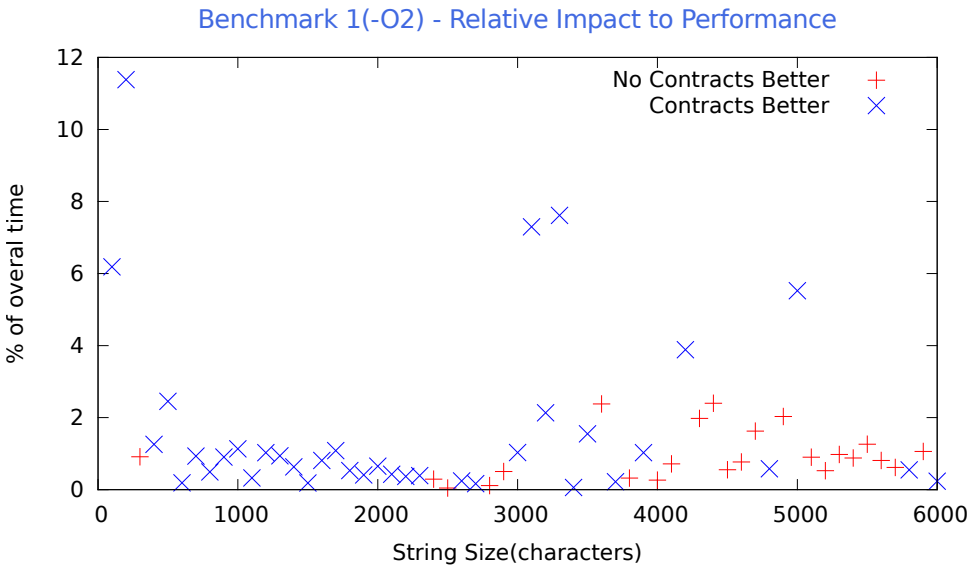

Figure 6-3: *Benchmark 1: Relative Impact in Performance of Contracts vs No Contracts -O2.*

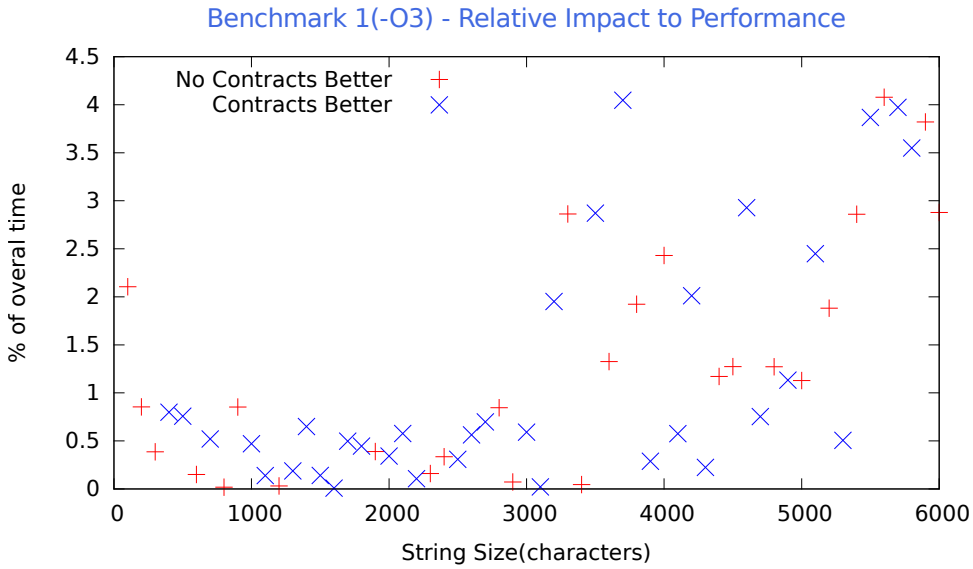

Figure 6-4: *Benchmark 1: Relative Impact in Performance of Contracts vs No Contracts -O3.*

Figure 6-4 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O3*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. In this case, we observe that for lower values of the string size, contracts tend to suppose an improvement around 1% of the time, whereas for larger string sizes the improvement reaches 3-4%.

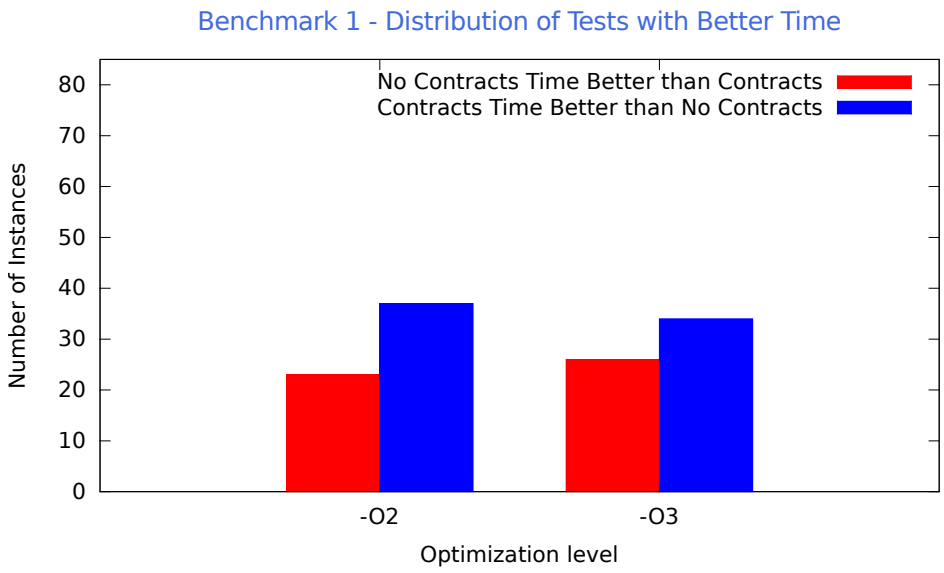

Figure 6-5: *Benchmark 1: Distribution of tests with better times -O3.*

Figure 6-5 represents the number of tests going to each side per optimization flag *-O2 and -O3*. The *x axis* represents the optimization flag that is used and the *y axis* the number of tests that go in favour of each alternative. The blue column represents the number of tests where contracts are better and the red column the number of tests where contracts are worse. In this case, we can specifically see that contracts seem to be a better alternative for both compilation flags supposing roughly a difference of 10 test in which results are better.

Benchmark 2

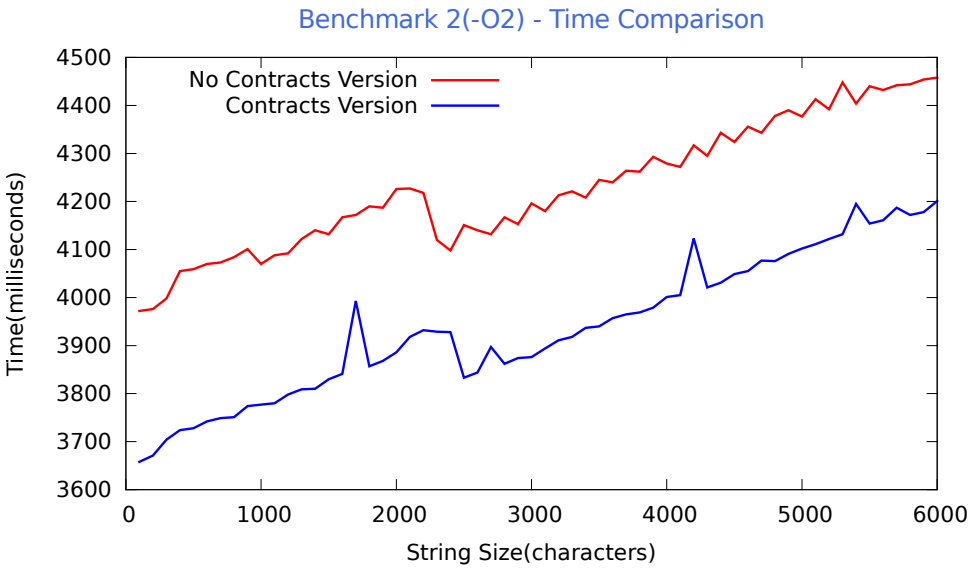

Figure 6-6: *Benchmark 2: Time comparison of Contracts and No Contracts -O2*

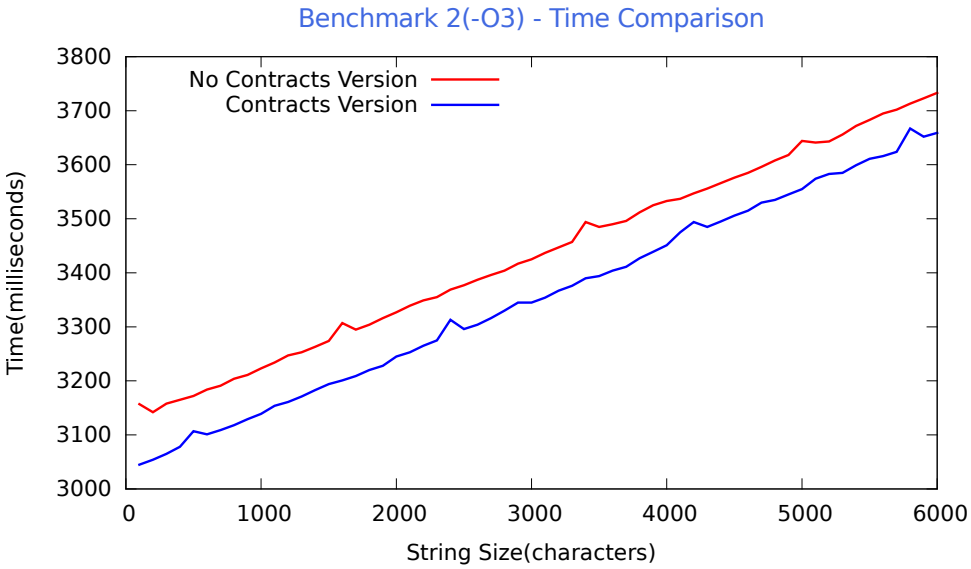

Figure 6-7: *Benchmark 2: Time comparison of Contracts and No Contracts -O3*

Figure 6-6 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O2*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. Hence, the graph clearly shows an advantage of the contracts alternative with respect to the no contracts alternative, supposing a rough advantage of around 400 milliseconds in all executions.

Figure 6-7 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O3*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. Thus, the graph shows a clear advantage of the implementation with contracts. The difference between the two alternatives is smaller, however, it is still around 200 milliseconds.

Figure 6-8 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O2*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. We can observe that the improvement in these tests supposes roughly a 7% of improvement, which implies a clear advantage on the side of contracts.

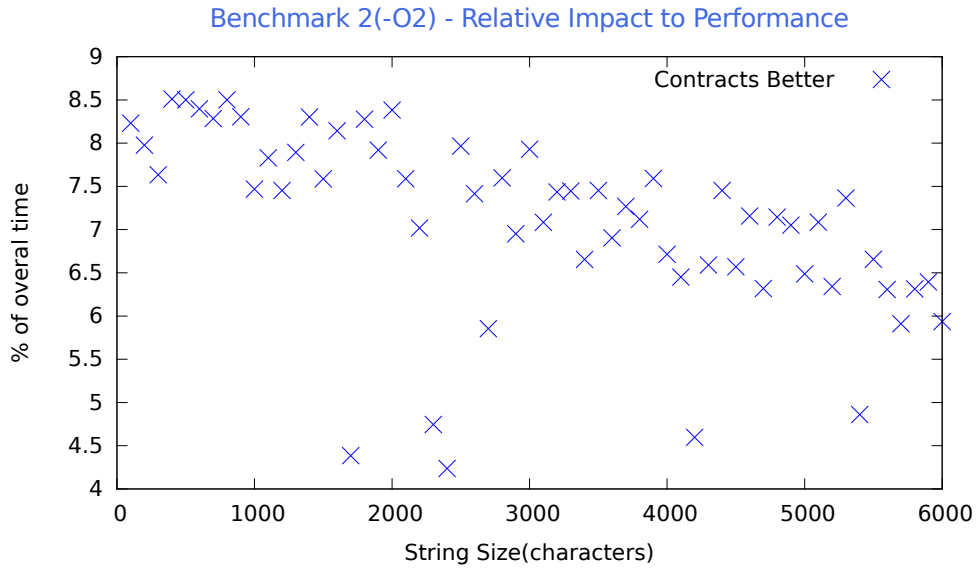

Figure 6-8: *Benchmark 2: Relative Impact in Performance of Contracts vs No Contracts -O2*

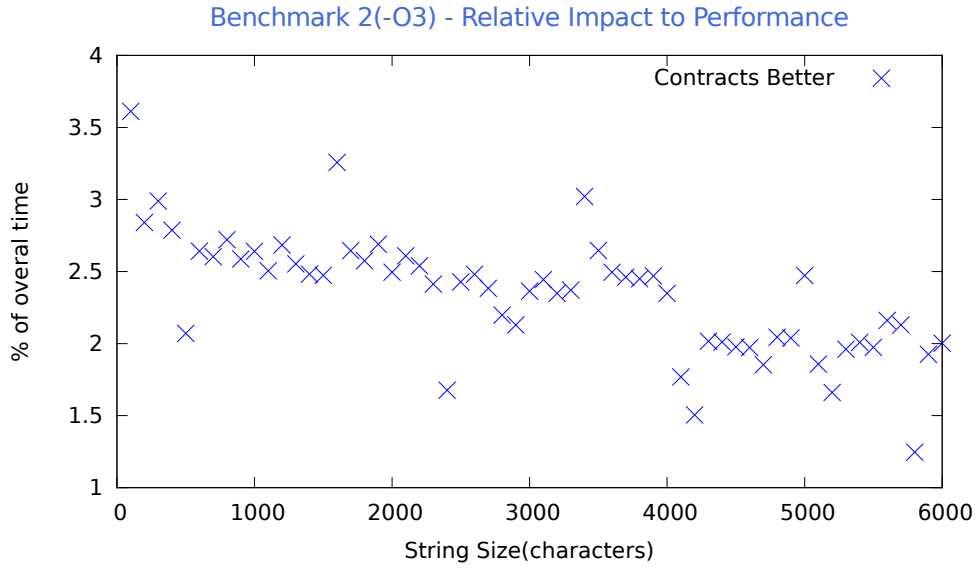

Figure 6-9: *Benchmark 2: Relative Impact in Performance of Contracts vs No Contracts -O3*

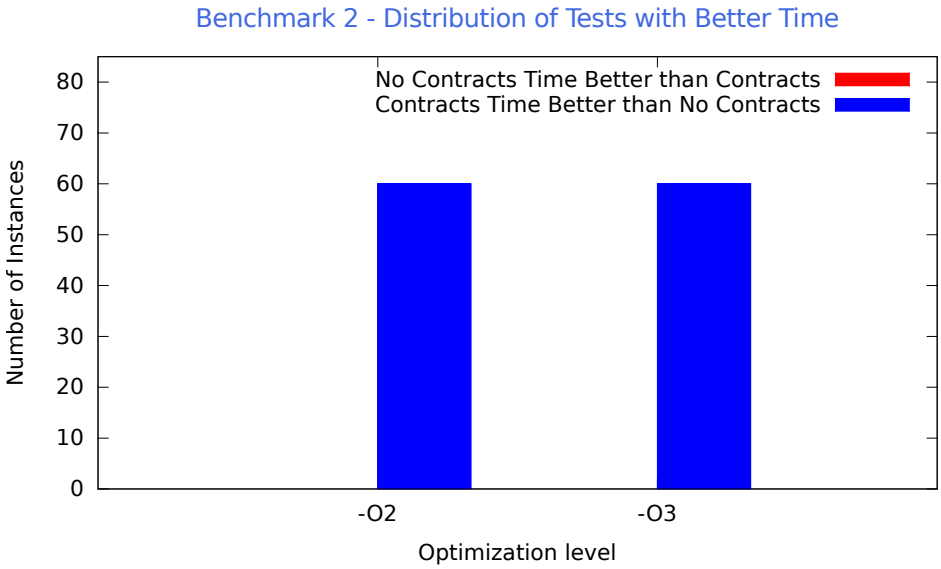

Figure 6-10: *Benchmark 2: Distribution of tests with better times -O3*

Figure 6-9 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O3*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where contracts version was worse in time and supposed a delay. The advantage in relative performance of the implementation with contracts is around 2.75%. Again, it is a great improvement, although the difference with respect to the non-annotated version is smaller.

Figure 6-10 represents the amount of tests going to each side per optimization flag *-O2 and -O3*. The *x axis* represents the optimization flag that is used and the *y axis* the number of tests that go in favour of each alternative. The blue column represents the number of tests where contracts are better and the red column the number of tests where contracts are worse. We clearly observe that the contracts are better in all cases.

Benchmark 3

Figure 6-11 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O2*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. The graph grows exponentially in both cases, however, differences are light and difficult to perceive.

Figure 6-12 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O3*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. This graph shows little differences between the two alternatives. In the higher string sizes, we can observe that both lines cross among them more times. However, differences are difficult to depict.

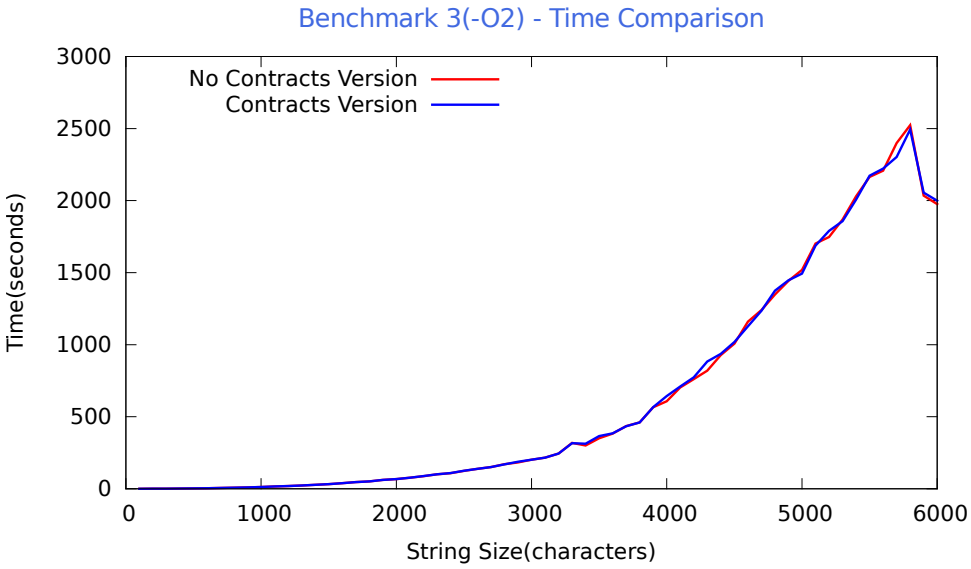

Figure 6-11: *Benchmark 3: Time comparison of Contracts and No Contracts -O2*

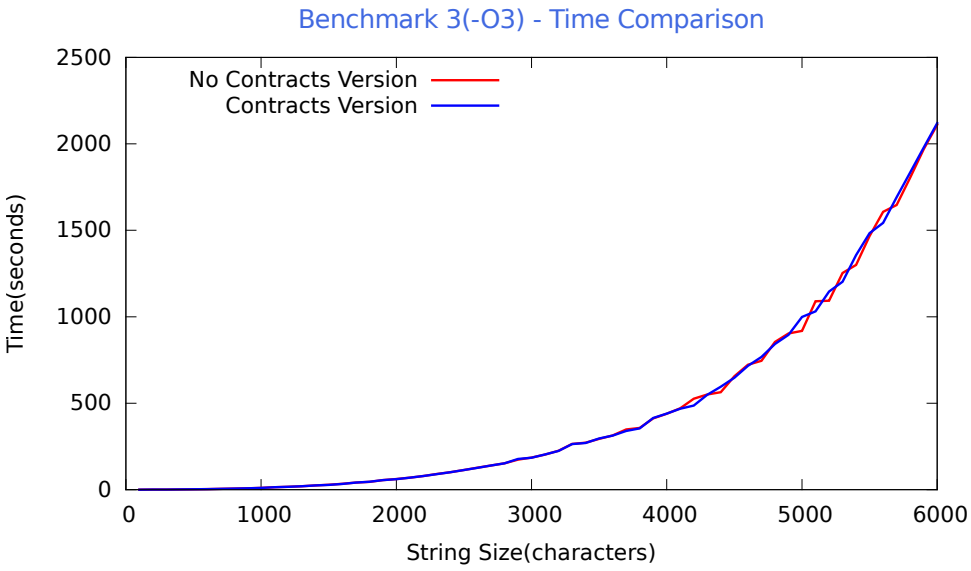

Figure 6-12: *Benchmark 3: Time comparison of Contracts and No Contracts -O3*

Figure 6-13 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O2*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. In this case, the differences are very small and neither of the alternatives seems to provide advantages. In addition to that, we can observe that the difference in execution time is around 1% improvement for both.

Figure 6-14 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O3*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts where better in that case and they supposed an improvement. A red cross represents a case where contracts version was worse in time and supposed a delay. This figure shows that there are more blue crosses than red ones but all of them have homogeneously distributed around an improvement of 1-5%.

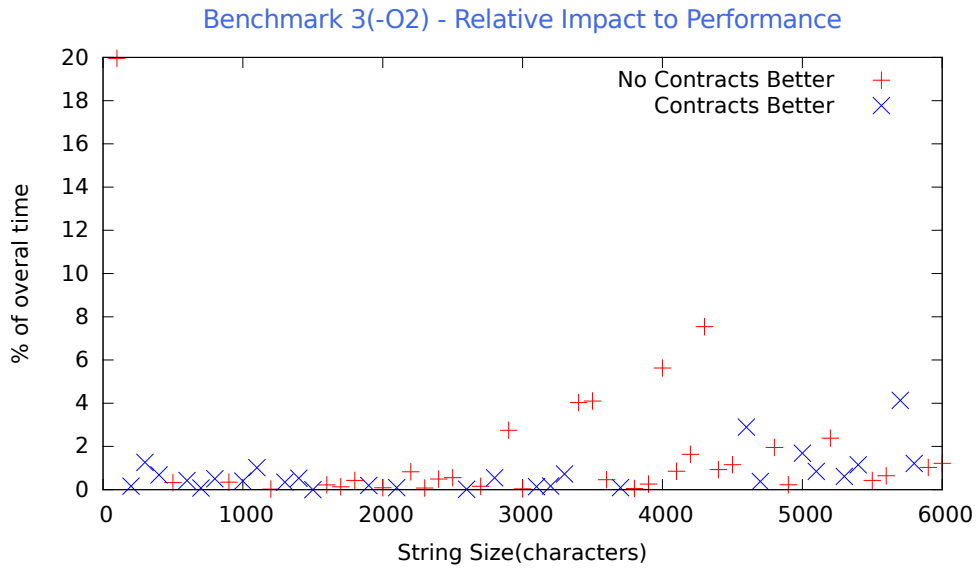

Figure 6-13: *Benchmark 3: Relative Impact in Performance of Contracts vs No Contracts -O2*

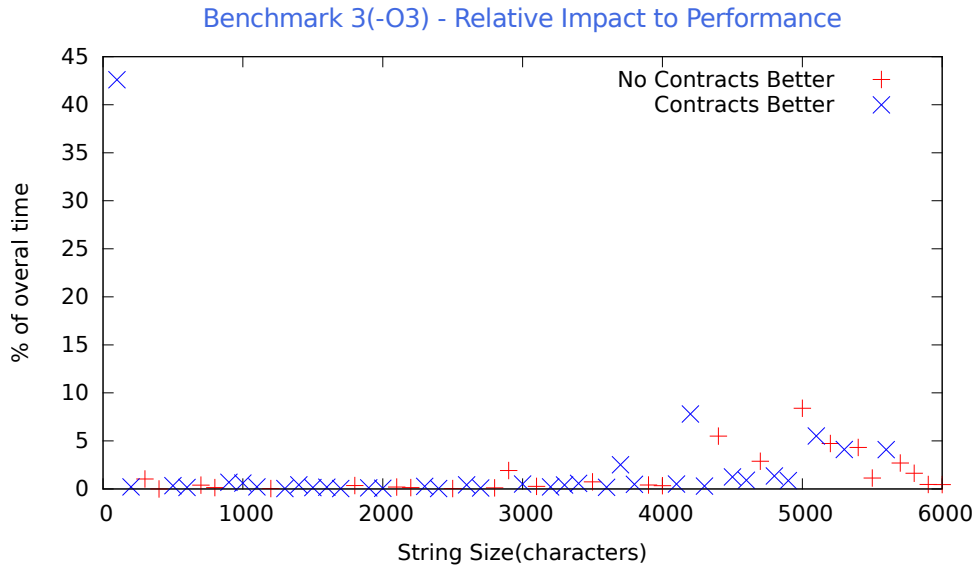

Figure 6-14: *Benchmark 3: Relative Impact in Performance of Contracts vs No Contracts -O3*

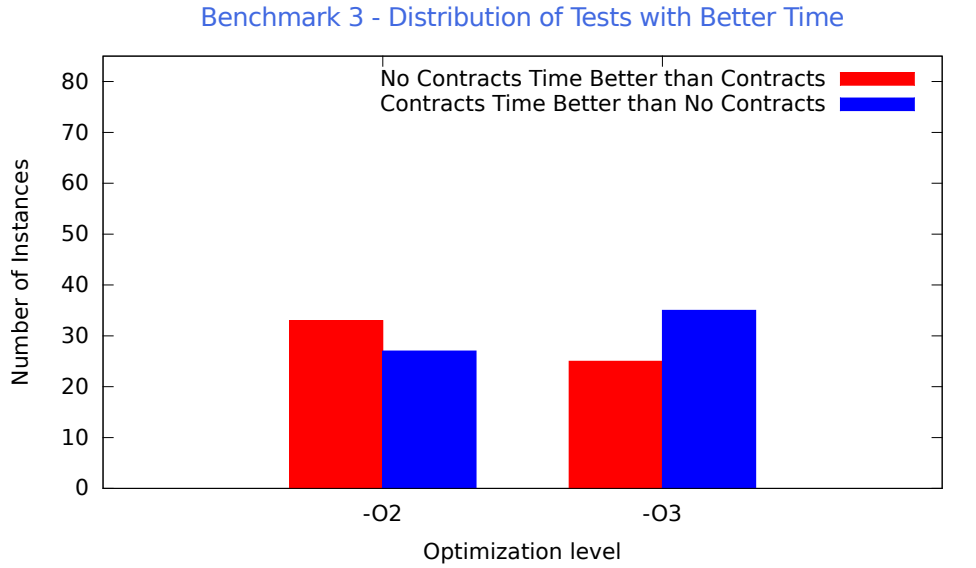

Figure 6-15: *Benchmark 3: Distribution of tests with better times -O3*

Figure 6-15 represents the number of tests going to each side per optimization flag *-O2 and -O3*. The *x axis* represents the optimization flag that is used and the *y axis* the number of tests that go in favour of each alternative. As it was depicted by previous graphs, the *-O2* optimization flag does not seem to be enough to get an improvement. However, when we switch to *-O3* flag, the number of tests which work better with contracts is higher than the number of tests that works with no contracts.

Benchmark 4

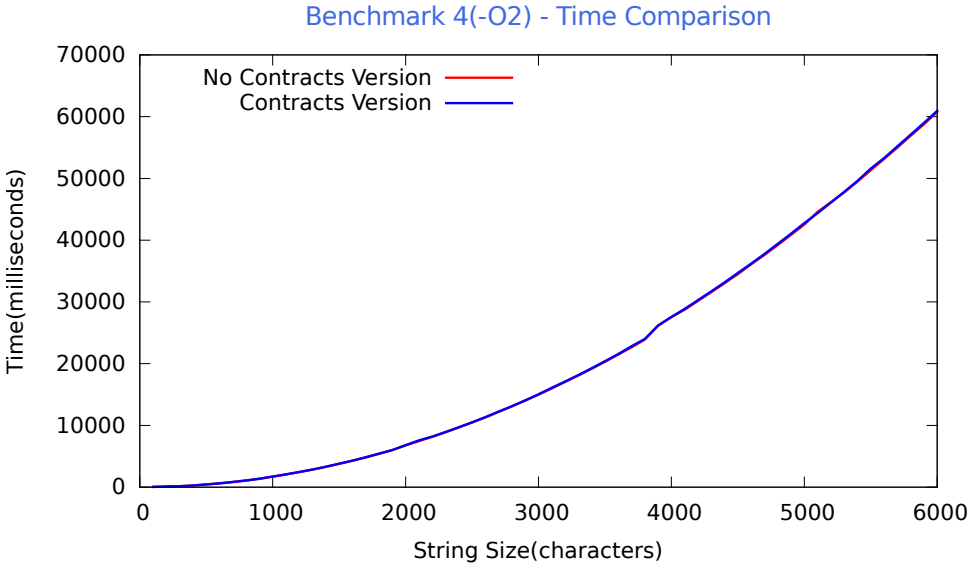

Figure 6-16: *Benchmark 4: Time comparison of Contracts and No Contracts -O2*

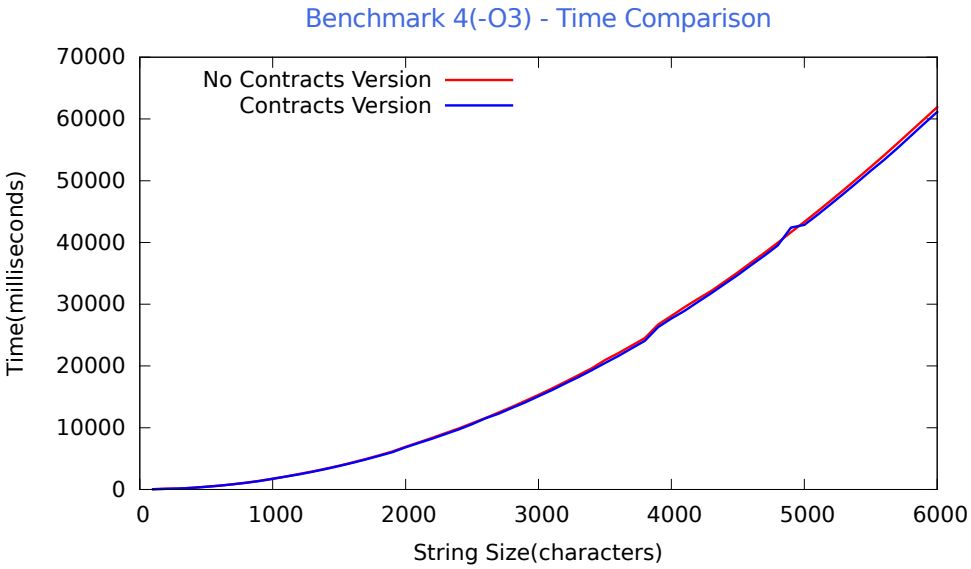

Figure 6-17: *Benchmark 4: Time comparison of Contracts and No Contracts -O3*

Figure 6-16 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O2*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. The graph grows exponentially and at the same time in both cases. Consequently, we cannot appreciate any

differences.

Figure 6-17 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O3*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. This graph shows little differences between the two alternatives. In the higher string sizes, we can observe that the blue line is underneath. However, differences are difficult to depict.

Figure 6-18 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O2*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. In this case, the differences are very small however the usage of contracts seems to affect performance in less than 1%, which is minimal.

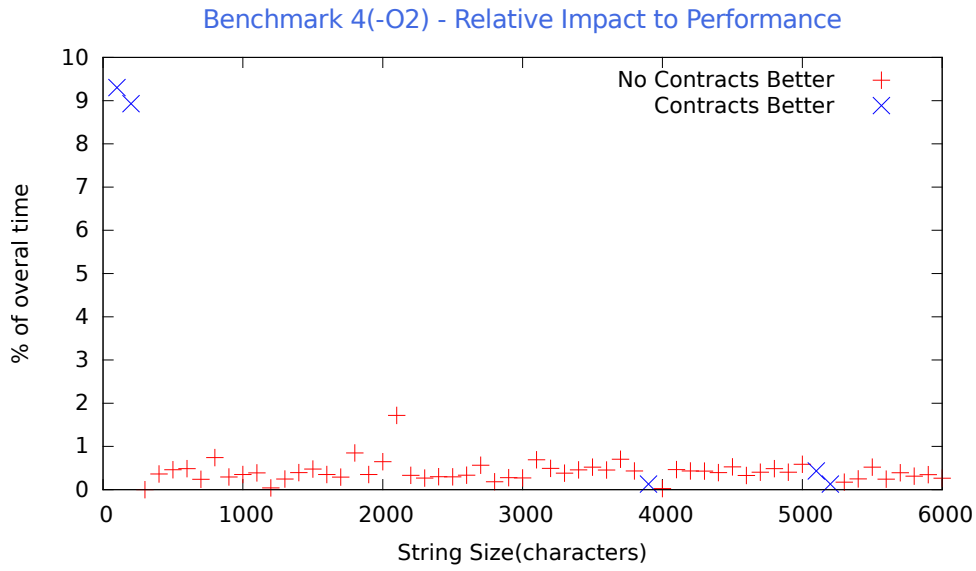

Figure 6-18: *Benchmark 4: Relative Impact in Performance of Contracts vs No Contracts -O2*

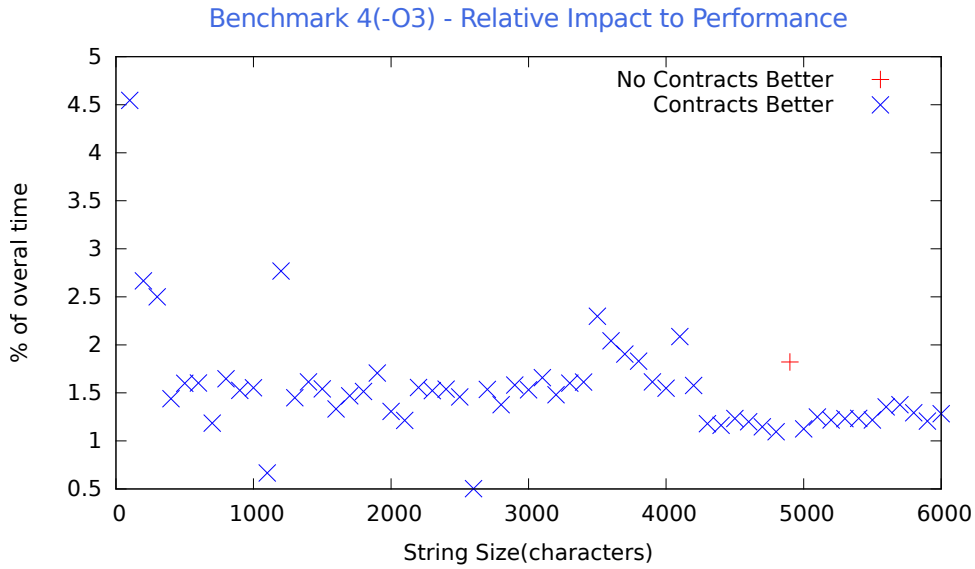

Figure 6-19: *Benchmark 4: Relative Impact in Performance of Contracts vs No Contracts -O3*

Figure 6-19 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O3*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where contracts version was worse in time and supposed a delay. In this graph, we can identify a clear advantage of the usage of contracts with around a 1% of improvement.

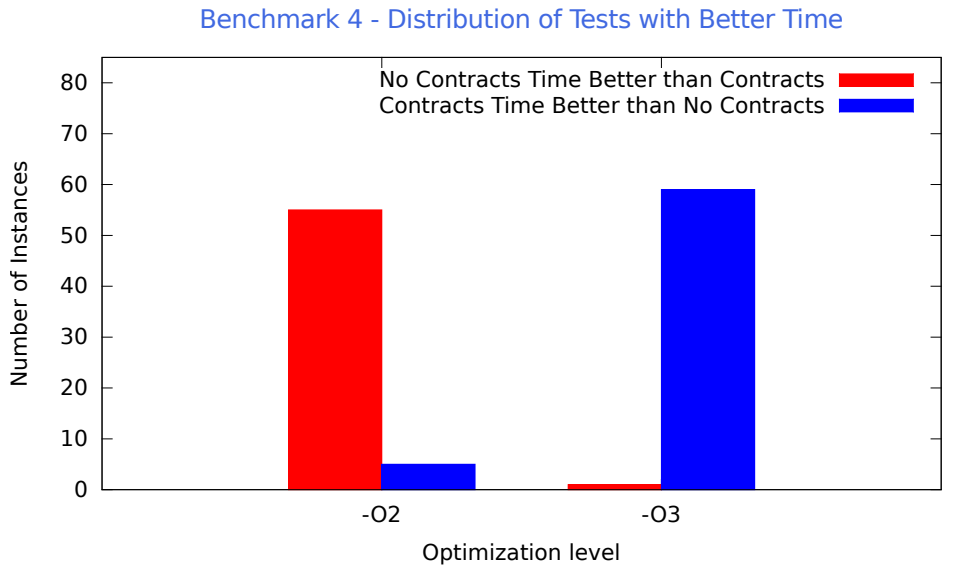

Figure 6-20: *Benchmark 4: Distribution of tests with better times -O3*

Figure 6-20 represents the number of tests going to each side per optimization flag *-O2 and -O3*. The *x axis* represents the optimization flag that is used and the *y axis* the number of tests that go in favour of each alternative. As it was depicted by previous graphs, the *-O2* optimization flag does not seem to be enough to get an improvement. However, when we switch to *-O3* flag, the number of tests which work better with contracts is much higher and worthy for the implementation.

Benchmark 5

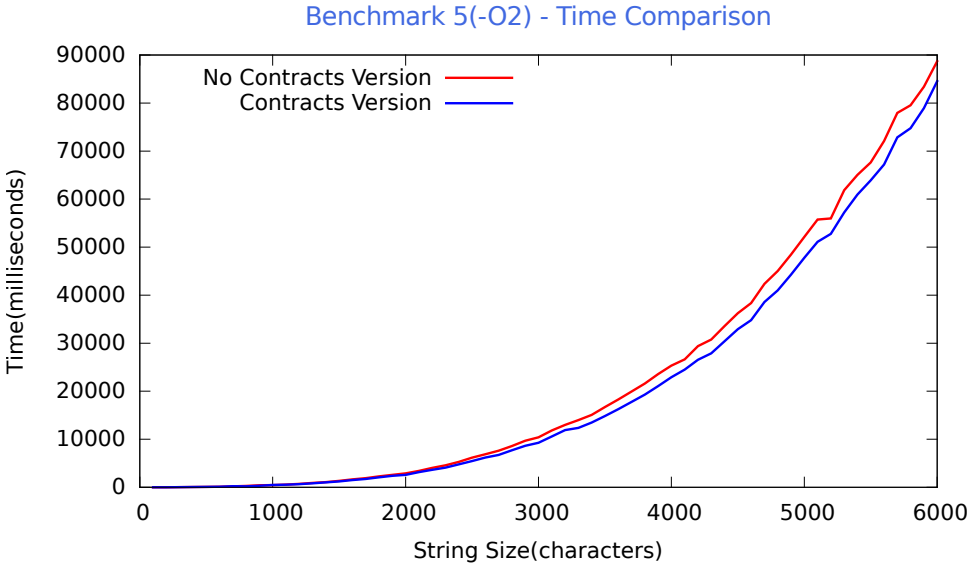

Figure 6-21: *Benchmark 5: Time comparison of Contracts and No Contracts -O2*

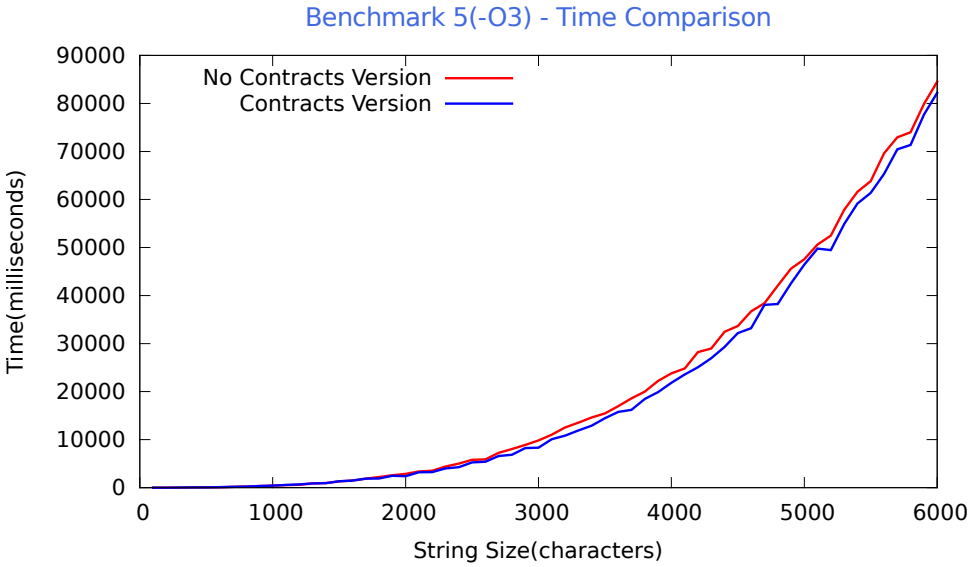

Figure 6-22: *Benchmark 5: Time comparison of Contracts and No Contracts -O3*

Figure 6-21 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O2*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. We can clearly see that the times of the contract version are underneath the red line, meaning we are getting better times.

Figure 6-22 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O3*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line is the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. In this graph, we can clearly observe that the blue line is under the red line. This means that we have better times for the modified version.

Figure 6-23 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O2*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. We can observe that the percentage of improvement is on the side of contracts. The relative impact on performance is around 10% which is a great improvement.

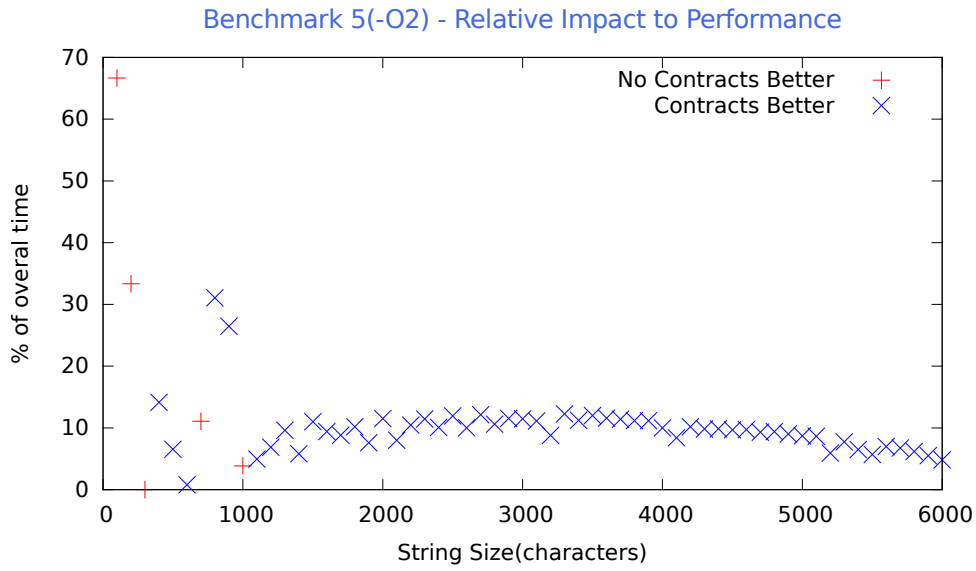

Figure 6-23: *Benchmark 5: Relative Impact in Performance of Contracts vs No Contracts -O2*

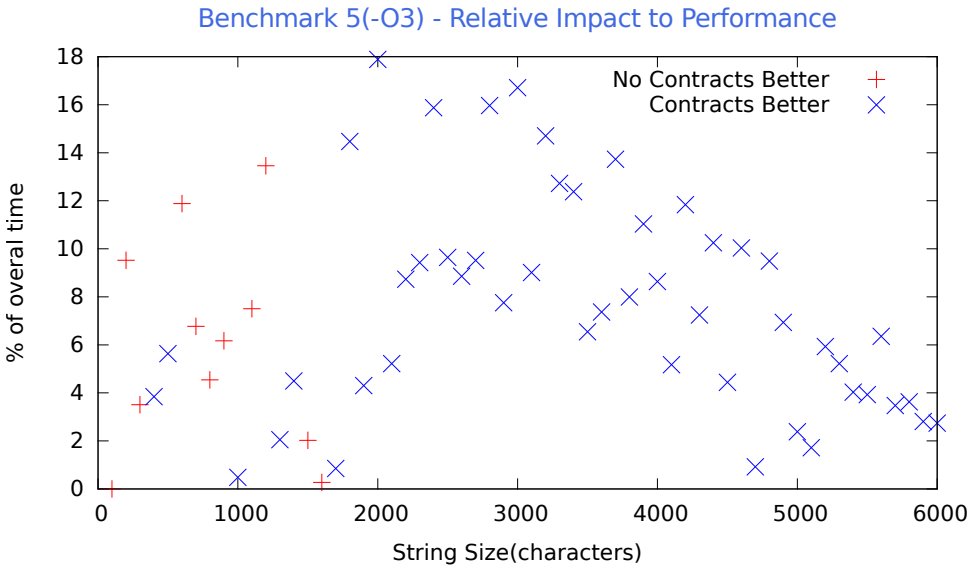

Figure 6-24: *Benchmark 5: Relative Impact in Performance of Contracts vs No Contracts -O3*

Figure 6-24 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O3*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where contracts version was worse in time and supposed a delay. This graph shows a very dispersed representation of the improvement in favour of the contracts version. It varies from values close to 4% up to improvements of 14%. Nevertheless, even the smallest improvement implies something good.

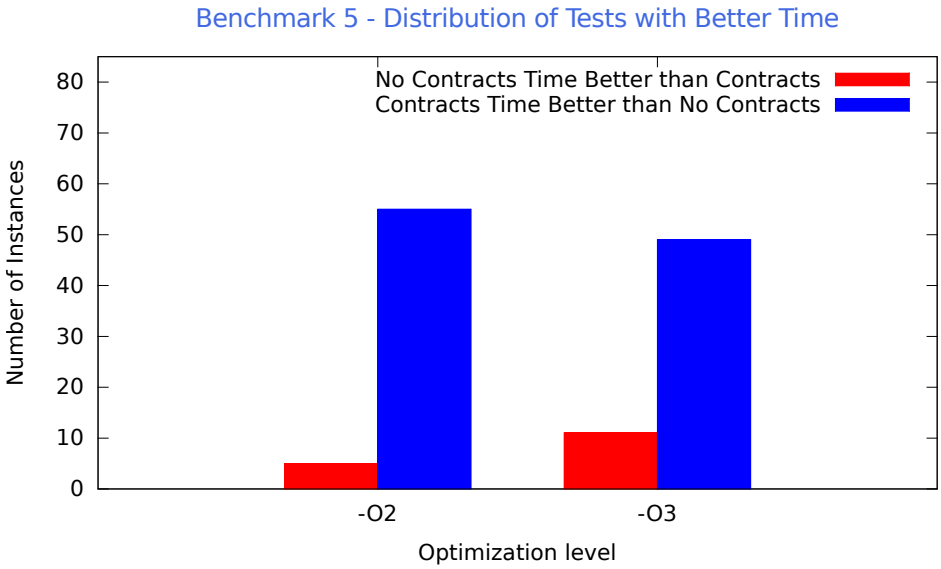

Figure 6-25: *Benchmark 5: Distribution of tests with better times -O3*

Figure 6-25 represents the number of tests going to each side per optimization flag *-O2 and -O3*. The *x axis* represents the optimization flag that is used and the *y axis* the number of tests that go in favour of each alternative. We can clearly depict that the usage of contracts in both optimization flags supposes an improvement to performance. This is a clear proof that using contracts improves the coding efficiency.

Benchmark 6

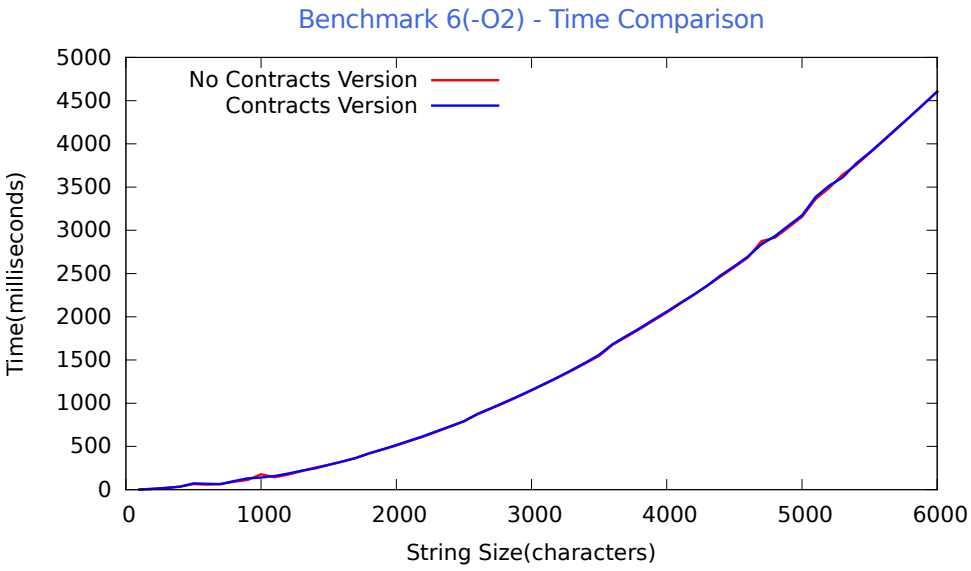

Figure 6-26: *Benchmark 6: Time comparison of Contracts and No Contracts -O2*

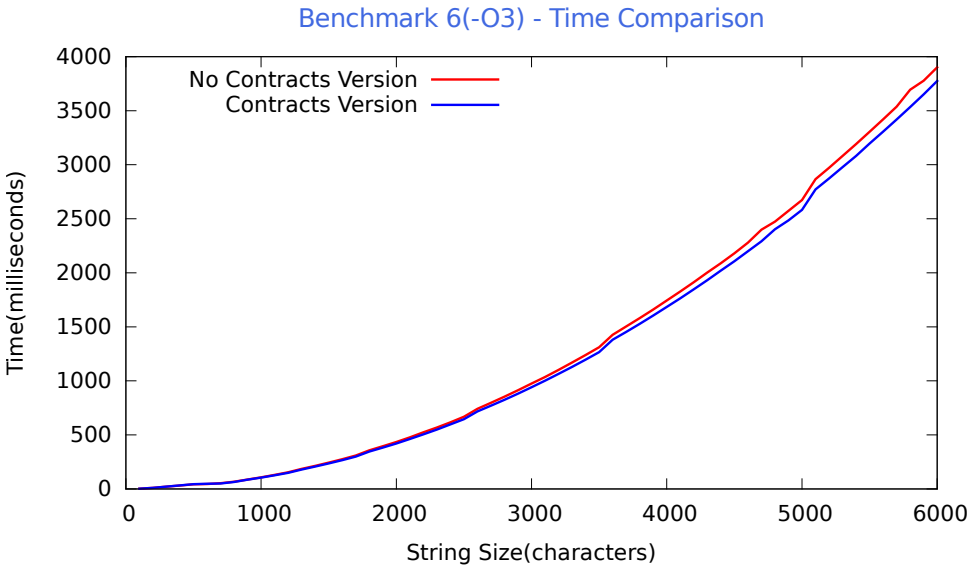

Figure 6-27: *Benchmark 6: Time comparison of Contracts and No Contracts -O3*

Figure 6-26 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O2*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. The graph grows exponentially and at the same time in both cases. Consequently, we cannot appreciate any differences.

Figure 6-27 shows the time in seconds of the execution with contracts and without contracts for the optimization flag *-O3*. The *x axis* represents the string size in characters and the *y axis* the time that the evaluation took for that string size. The blue line represents the time that the contract version took to execute the tests and the red line represents the time of the version which did not use contracts. In this graph, we can clearly observe that the blue line is under the red line. This means that we have better times for the modified version.

Figure 6-28 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O2*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where the contracts were worse in time and supposed a delay. In this case, the differences are very small however the usage of contracts seems to affect performance in less than 1%, which is minimal.

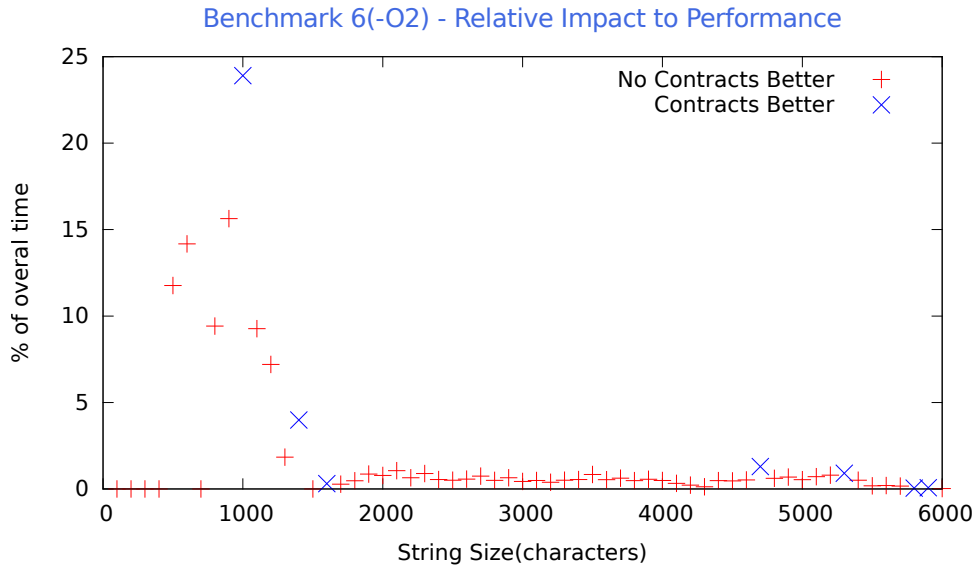

Figure 6-28: *Benchmark 6: Relative Impact in Performance of Contracts vs No Contracts -O2*

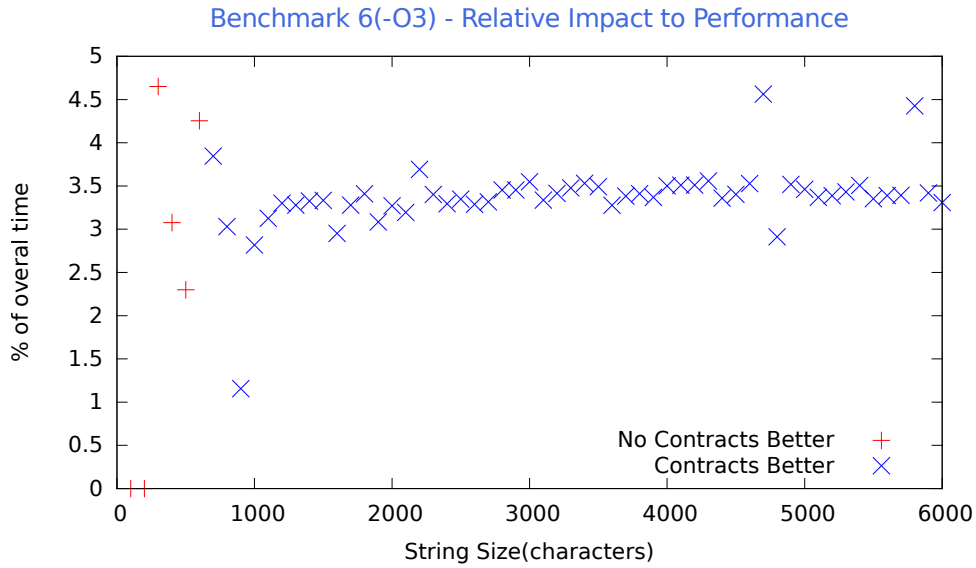

Figure 6-29: *Benchmark 6: Relative Impact in Performance of Contracts vs No Contracts -O3*

Figure 6-29 represents the relative time impact that the overhead of any of the alternatives supposed to the whole execution for flag *-O3*. The *x axis* represents the string size in characters. The *y axis* details the percentage of the overall time that the difference in the execution of the benchmark supposed to the whole execution. The blue cross represents that the contracts were better in that case and they supposed an improvement. A red cross represents a case where contracts version was worse in time and supposed a delay. In this case, we can observe a clear improvement of the performance of about 3.5% with almost all test being better.

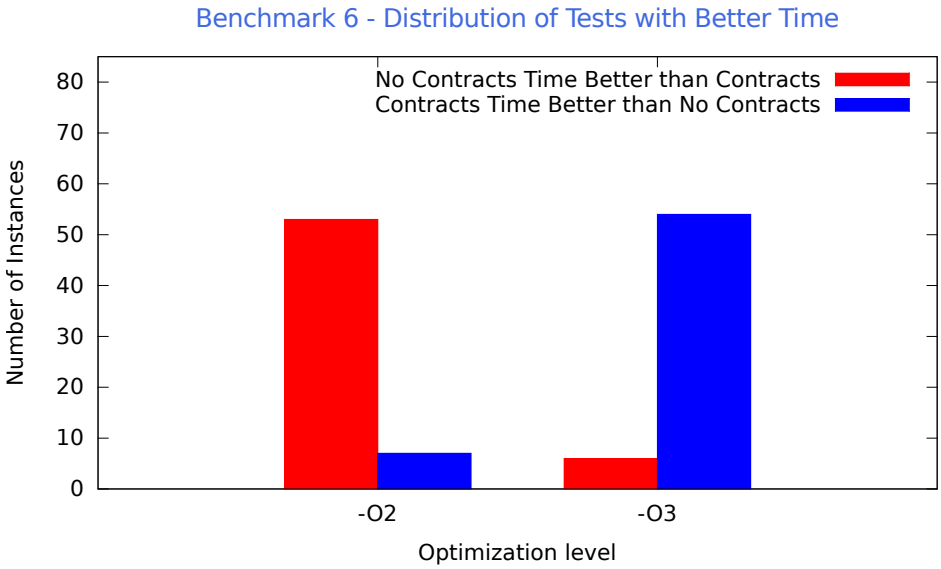

Figure 6-30: *Benchmark 6: Distribution of tests with better times -O3*

Figure 6-30 represents the number of tests going to each side per optimization flag *-O2 and -O3*. The *x axis* represents the optimization flag that is used and the *y axis* the number of tests that go in favour of each alternative. As it was depicted by previous graphs, the *-O2* optimization flag does not seem to be enough to get an improvement. However, the *-O3* flag shows a clear improvement in performance.

6.2.6 Conclusions on Performance

Before starting the development of this section, the hypothesis that we had was that contracts would induce no improvement but some overhead. Actually, an optimistic thought was observing that the performance was not being affected at all. This was the a priori thought that we were managing before performing any tests. This reasoning was backed-up by the fact that we were introducing some branch instructions in the code. This directly implied that we were creating overhead and this might imply a direct reduction of performance. But after seeing the results, all these thoughts changed since it seemed that contracts, in fact, were positive to the executables.

Even though, we can observe some cases where the performance of C++ contracts is not performing better than the usage of the not modified basic_string class, the relative decrease of performance decreases only by less than 1%. On the other hand, the cases where contracts are better is the improvement reaches in some cases the 15%. According to the number of tests that go in favour of using C++ contracts, the possibility of dealing with a case in which contracts are better is around 70%.

The only reason that can explain this, is the existence of an optimizer in the compiler. The code is

read and the optimizer changes code automatically so that it is more efficient. In fact, the optimizer that Clang compiler provides is pretty good in this sense. The optimizer builds a branch graph and traverses all the possibilities, assuming conditions already traversed to be true. With that in mind, it is able to detect that a branch is not feasible, and therefore, it traduces the code to remove that branch. This removal of branches and expressions that are not feasible have two direct improvements, first, the size of the executable, and second the performance.

That is the main idea that supports this improvement in performance. Contracts are providing extra information that without them would not be used. This extra information is good for the optimizer, which uses it to remove branches. Since this code has changed to provide more information with the new implementation of basic_string, the improvements in performance are due to the assumptions that these new conditions allow the optimizer to make.

The information that optimizer is probably dealing with, is the usage of `std::terminate` which induces the compiler to know that if that code is reached, there is no way of continuing through that branch and allows the elimination of that possible branch.

That is why, after looking at all these graphs, we can say that the information that contracts provide to the optimizer improves the code performance in a significant factor. The reason behind this improvement is the existence of a branch optimizer. With these affirmations, we conclude that giving information to the optimizer in some way, for example, by introducing contracts that are not translated to code but just taken into account by the optimizer, would definitely improve performance in the code.

This, as a conclusion, favours the usage of DBC, since it not only is used to give reliability and correctness to the code but also as a tool to improve the performance. The code is significantly better if we can help the compiler to guess which is the expected result of some branches. The more useful information we can give a compiler, the better the resulting executable will be.

Chapter 7

Project Plan

In this section, a description of the project plan will be carried out. Initially, in Section 7.2, an evaluation of different methodologies will be carried out justifying which is the chosen one. In Section 7.3, an overview of how the methodology was implemented for the duration of the project will be reviewed. This chapter aims to outline the steps that must be carried out for the completion of the project on time.

7.1 Justification of Methodology

In this section, we discuss which methodology to use for the project.

As it was mentioned in chapter 5, the software development process followed on the modification of a compiler is not the usual software development process. The main reason behind it is that the modification of a compiler cannot be done in a very modular way since interfaces are multiple and take all into account is complex. It directly implies the modification of the compiler and inclusion of new features within the already existing modules. Apart from that, the project is joined to a standardized proposal. This aims for the implementation of a standard that has already been defined and whose requirements will not change over time.

From that three different methodologies will be compared and analyzed namely Waterfall, Spiral and Prototyping. The Waterfall methodology refers to the sequential execution of all the software development phases one after the another. It is considered a traditional and usually rigid approach for systems development. This methodology would fit our project since the requirements and design are not going to evolve over time. However, the main disadvantage of using this methodology in this project remains on testing phase. The creation of a project within an already existing software implies the validation of each development so that the whole system does not break apart. And testing the whole system at once

would imply a more difficult resolution of bugs since they should be hardly targetable.

Spiral methodology is a great alternative for usual software projects since it allows the creation of the project step by step. It is a more costly alternative since it implies doing each phase multiple times but it allows a successful development of the project. Nevertheless, this methodology does not fit well the project. Repeating the requirement or the design phase is not going to provide anything to the project since they are all known from the beginning of the project.

The prototyping alternative is more centred on repeating phases several times by implementing a little functionality and evaluating it with the client to improve it. However, the requirements are already set and we want to test the functionality many times.

For the reasons explored above the chosen methodology is **Incremental Build Model**. This methodology aims to reduce the requirements and design phase and it is more centred on the implementation and evaluation of it. It combines the initial phase of *Waterfall*, where requirements and design are elaborated one after the other, and the last phase of *Prototyping*, where new elements are built over a basic functionality. The process would imply the creation of a first prototype with a reduced functionality of the final and evaluate it. And over it build the rest of the functionalities. In this project, the followed approach is similar to this methodology. The basic functionality was developed and tested with the basic contracts. Once it was fully functional, newer features such as the compilation flags and the contract violation handler were built over it.

7.2 Methodology

In this section, an overview of the proposed methodology is depicted.

The methodology chosen to carry out this project is Incremental Build Model. In this methodology, the requirement set is divided into several incremental builds. In this way, the requirements are developed in a more manageable way all the steps. In the incremental model, the system is built by pieces but ensuring that each piece is complete once each of them is finished. In the case of this project the Incremental Build Model will be applied in the following way:

1. **Requirement Phase**. All the requirements were extracted from the specification that was spread among many papers to come up with a uniform base to be developed.

2. **Division of Tasks**. The requirements were divided into several build projects. A first build with the basic functionality was created and from it, three more builds were created. One including the

build level, another including the contract violation handler and another including the support for templates.

3. **Development**. The functionality of each of this builds is designed for each of the specific requirements of this phase. When the functionality has been implemented, a testing phase is carried out to test that the specific requirements of this build are properly implemented. This was done specifically for each of the builds and in a general way, at the end of the project.

Figure 7-1 shows the overview of the Incremental Development Software methodology lifecycle.

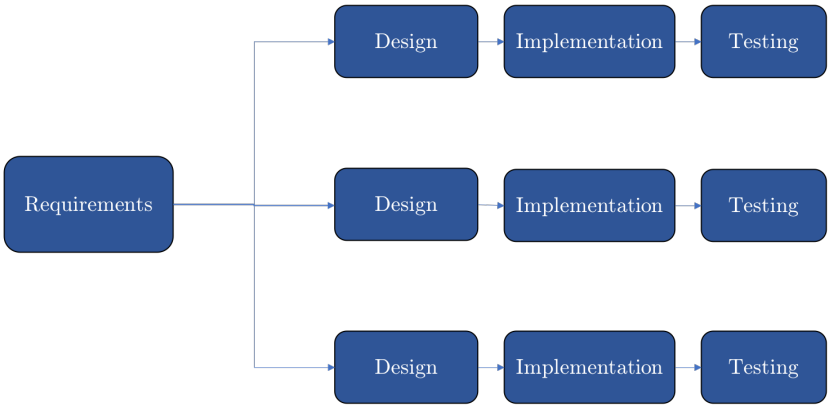

Figure 7-1: *Incremental Software Development Lifecycle*

The main advantages that this software development model provides us is that generates software fast and speeds up the software lifecycle. It also gives feedback to the consumer since it can be giving small builds when they are being ready. But the most important fact is that it allows testing by batches that make debugging easier.

7.3 Subdivision of Tasks

In this section, it is detailed how the task division of the project is carried out.

The main objective of this section is to detail how the project time was divided for the conclusion of the former according to specified. As it was detailed in the implementation section. The project and the learning curve of a compiler are hard, therefore there are three main sections on the project. During the first phase of the project, the basic functionality implementation was carried out. During this phase, the work consisted of the proper learning of the compiler and the implementation of the basic functionality. During the second phase, a new flag was included. The last section is deeper and more difficult and required a bigger amount of time since it was more complex. During the last weeks of the project, the

task was devoted to creating a test interface which ensured the proper functionality of the elements together with some benchmarks that allowed the evaluation of the project.

The project was developed in 42 weeks of work starting the 1st of September of 2017 and ending on 12th of June of 2018 with a total of 9 months of work. The project plan has taken into account holidays since part of the work has been developed in spare time. The Gantt diagram in Figure 7-2 shows the planned development of the project whereas Figure 7-3 represents the final depiction of time. Finally, Figure 7.1 depicts the starting and end times.

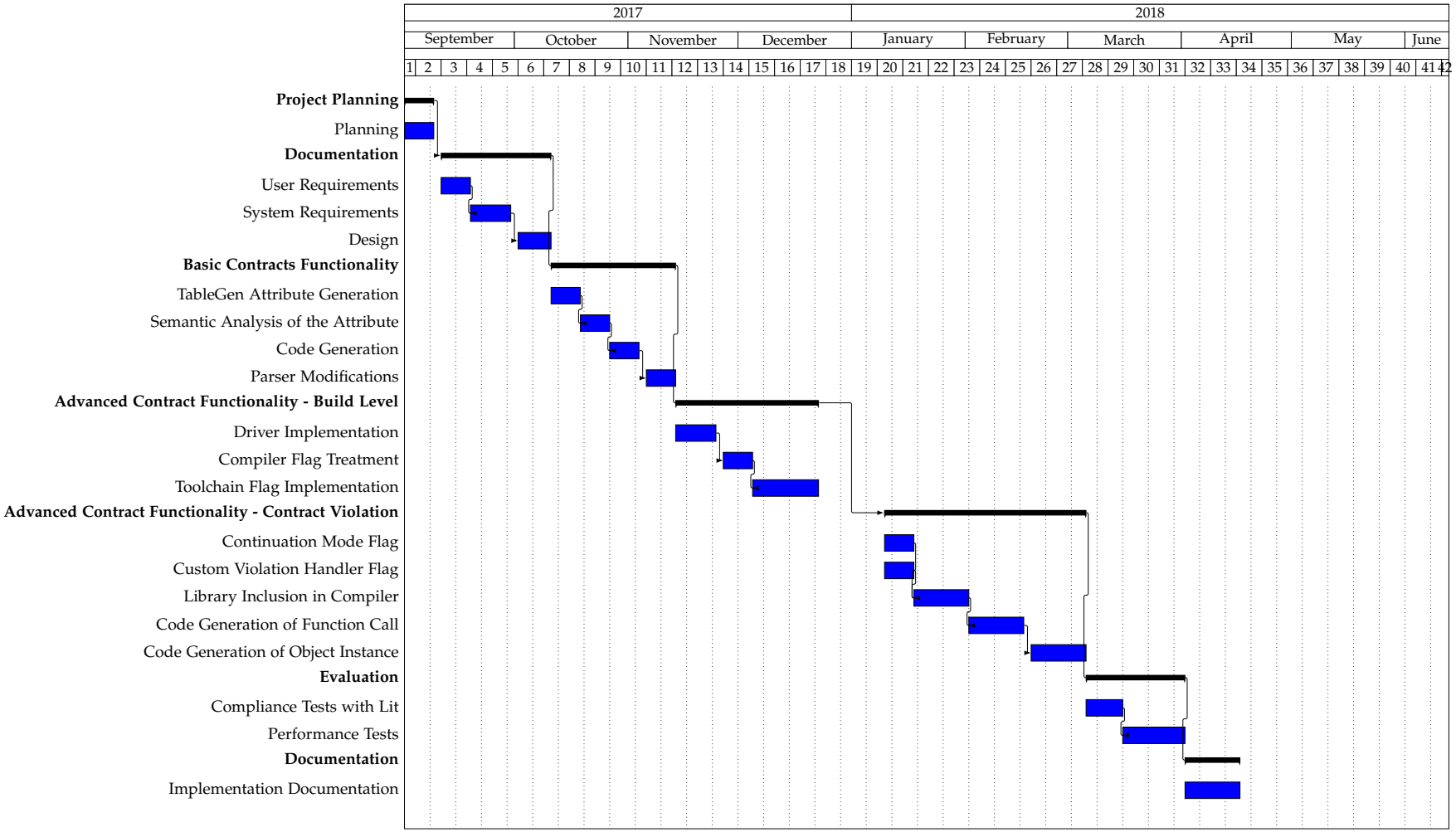

Figure 7-2: *Gantt Chart Expected Time Expenditure*

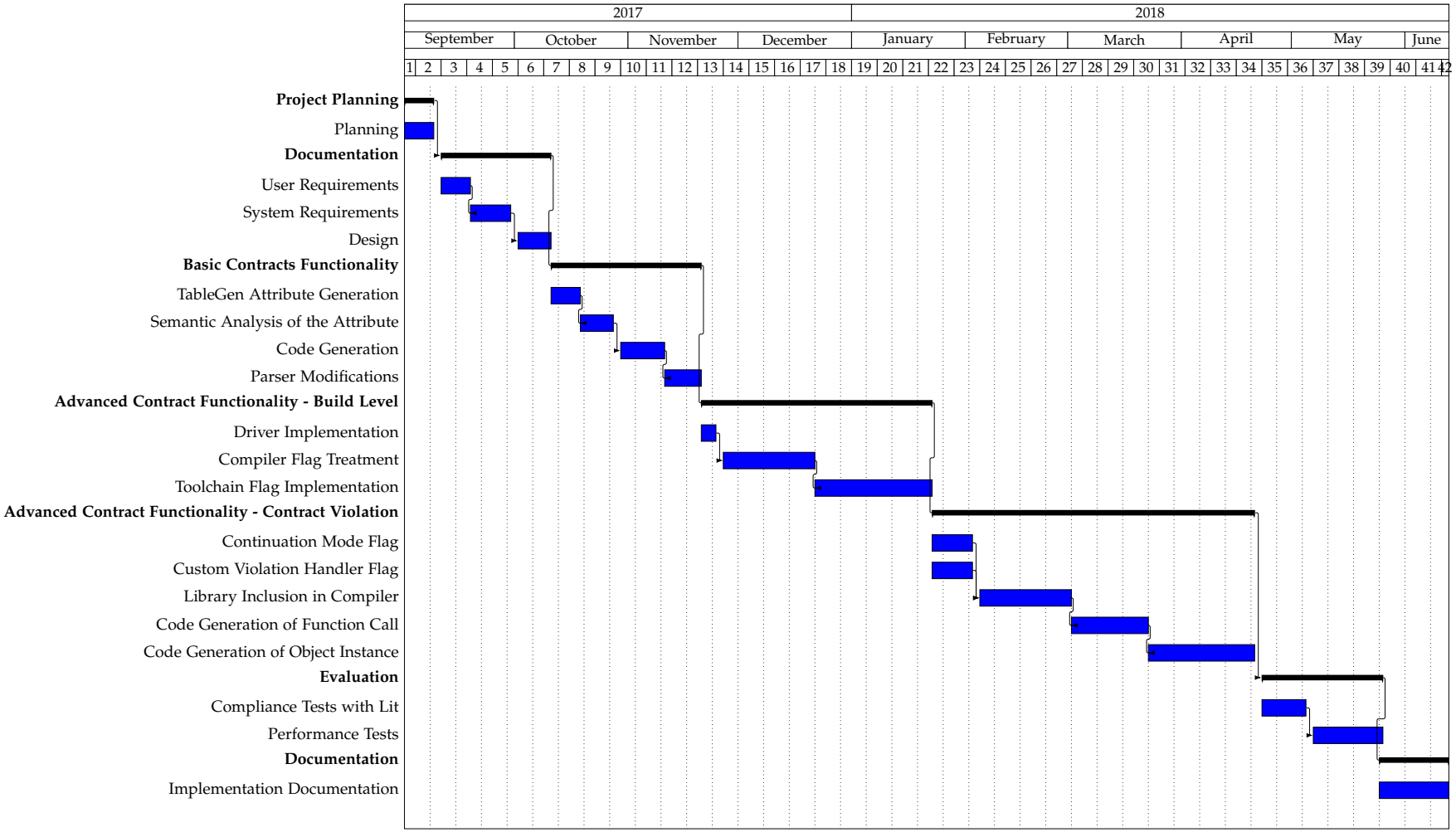

Figure 7-3: *Gantt Chart Actual Time Expenditure*

Phase	Expected Start	Expected End	Actual Start	Actual End
		Project Planning		
Planning	2017-09-01	2017-09-08	2017-09-01	2017-09-08
		Documentation		
User Requirements	2017-09-11	2017-09-18	2017-09-11	2017-09-18
System Requirements	2017-09-19	2017-09-29	2017-09-19	2017-09-29
Design	2017-10-02	2017-10-10	2017-10-02	2017-10-10
		Basic Contract Functionality		
TableGen Attribute Generation	2017-10-11	2017-10-18	2017-10-11	2017-10-18
Semantic Analysis of the Attribute	2017-10-19	2017-10-27	2017-10-19	2017-10-26
Code Generation	2017-10-30	2017-11-10	2017-10-27	2017-11-03
Parser Modifications	2017-11-11	2017-11-20	2017-11-06	2017-11-13
		Advanced Contract Functionality - Build Level		
Driver Implementation	2017-11-21	2017-11-24	2017-11-14	2017-11-24
Compiler Flag Treatment	2017-11-27	2017-12-21	2017-11-27	2017-12-04
Toolchain Flag Implementation	2017-12-22	2018-01-22	2017-12-05	2017-12-22
		Advanced Contract Functionality - Contract Violation		
Continuation Mode Flag	2018-01-23	2018-02-02	2018-01-10	2018-01-17
Custom Violation Handler Flag	2018-01-23	2018-02-02	2018-01-10	2018-01-17
Library Inclusion in Compiler	2018-02-05	2018-03-01	2018-01-18	2018-02-01
Code Generation of Function Call	2018-03-02	2018-03-22	2018-02-02	2018-02-16
Code Generation of Object Instance	2018-03-23	2018-04-20	2018-02-19	2018-03-05
		Evaluation		
Compliance Tests with Lit	2018-04-23	2018-05-04	2018-03-06	2018-03-15
Performance Tests	2018-05-07	2018-05-25	2018-03-16	2018-04-01
		Documentation		
Implementation Documentation	2018-05-25	2018-06-12	2018-04-02	2018-04-16

Table 7.1: *Gantt Diagram Dates Comparison*

Chapter 8

Socio-economic Environment

This chapter aims for the analysis of the project budget. This chapter is divided into two sections. In section 8.1 a breakdown of the costs that this project has accounted for is detailed. In section 8.2, a description of the possible socio-economic impact of this project is depicted.

8.1 Project Budget

In this section we depict the breakdown of the different costs that the project accounted for. In Section 8.1.1, the human resources cost is depicted. In Section 8.1.2, the total cost of the equipment is overviewed. In Section 8.1.3, we consider the cost of the software licences that we used. In Section 8.1.4, the cost of consumables is reviewed. Afterwards, in Section 8.1.5, the travel resources are depicted. Later on, in Section 8.1.6, other special resources are computed. Finally, Section 8.1.7 the computation of all resources is depicted.

For this section, a total of 10 months of work will be assumed starting in September 2017 until May 2018 both included. A total of 22 labour days are considered per month and a work of 4 hours per day is also considered. Figure 8.1 represents the breakdown of the days and hours worked.

Total Project Time	
Months	10 months
Days	220 days
Hours per day	4 hours per day
Total project hours:	880 hours

Table 8.1: *Total time considered*

8.1.1 Human Resources

In this section the cost of human resources is described.

The cost of human resources is obtained from the 2018 Spanish Government official site [48]. It takes into account the average salary for a technical engineer. We take into account the average salary between the maximum salary and the minimum salary considered in [48].Figure 8.2 represents the breakdown of the human resources costs.

Human Resources				
Role	Analyst	Designer	Programmer	Tester
Hours	44 hours	44 hours	616 hours	176 hours
Maximum salary/hour	21.31 €	21.40 €	21.28 €	21.32 €
Minimum salary/hour	6.58 €	6.81 €	5.64 €	5.85 €
Base/hour	13.95 €	14.06 €	13.48 €	13.58 €
Total:	11,920.69 €			

Table 8.2: *Human Resources Costs*

8.1.2 Equipment Resources

In this section the cost of the equipment used is described

For this section the amortization period of the equipment is considered. The amortization is calculated over the time expected to generate a benefit that covers the expenses of any equipment. As we already said it is calcualated over a period of 10 months. It follows the following formula:

$$Cost = \frac{TotalPrice}{AmortizationTime} * UsageTime \tag{8.1}$$

Figure 8.3 represents the detailed description of the equipment costs.

Equipment Costs			
Equipment	**Total price per unit**	**Cost per month (Amortization)**	**Cost (10 months)**
Personal Computer	600.00 €	10.00 €/month	100.00 €
Tucan Cluster Node	5,000.00 €	83.33 €/month	833.33 €
Total:	933.33 €		

Table 8.3: *Equipment Costs*

8.1.3 Software Resources

In this section, we compute the cost of software resources.

As in the previous section the cost of the software acquired and used is considered. It is also depicted the prices of the different free licences used.Figure 8.4 represents the breakdown of the software licences costs.

Software Resources		
Software	**Licence price**	**Amortization Costs(10 months)**
Ubuntu 16.04 Desktop Edition	0.00 €	0.00 €
Visual Paradigm Community Edition 15.0	0.00 €	0.00 €
Microsoft Office Student Licence	150.00 €	2.50 €
GitLab Community Edition 10.1.4	0.00 €	0.00 €
GNUPlot	0.00 €	0.00 €
ShareLatex Community Edition	0.00 €	0.00 €
Total:	2.50 €	

Table 8.4: *Software Costs*

8.1.4 Consumables Costs

In this section the price of the office material is considered. Figure 8.5 represents the total cost fo the consumables.

Consumables		
Consumable	**Price per unit**	**Total Cost**
Pens	2.00 €	20.00 €
Paper	0.05 €	5.00 €
Folders	1.50 €	4.50 €
Printer Toner	-	20.00 €
Total:	49.50 €	

Table 8.5: *Consumables Costs*

8.1.5 Travel Expenses

This section computes the cost of moving from home to the working place during the period of the work. Figure 8.6 represents the overview of the transportation costs.

Transportation Expenses		
Item	**Monthly Price**	**Total**
Madrid Transport Card	20.00 €	200.00 €
Total:	200.00 €	

Table 8.6: *Transportation Costs*

8.1.6 Other Costs

This section computes the cost of using a facility or any equipment that is left such as the electricity, maintenance or building degradation costs. It is considered to be 20% of the total cost of the project. Figure 8.7 represents the breakdown of other costs.

Other Expenses	
Concept	**Cost**
Indirect Costs	2,625.70 €
Total:	2,614.27 €

Table 8.7: *Other Costs*

8.1.7 Total Cost

In this section, it is considered the total cost before taxes and the total cost applying the taxes applicable in Spain which is *21%* of the total cost. In the end the total cost of the project for the contraction of the engineer and all the equipment and material needed sums up to a total of *18,979.59 €*. Figure 8.8 represents the breakdown of the total cost of the project.

Total Costs	
Concept	**Cost**
Human Resources	11,927.51 €
Equipment Resources	933.33 €
Software Resources	2.50 €
Consumable Costs	49.50 €
Transportation Expenses	200.00 €
Indirect Costs	2,625.70 €
Margin of Benefit (15%)	1,969.27 €
Total before taxes:	*17,723.51 €*
Taxes (21%)	3,723.87 €
Total with taxes:	**21,445.45 €**

Table 8.8: *Total Costs*

8.2 Socioeconomic Impact

In this section, we will try to identify the target market and analyze the impact that this project might have both economically and socially. This project has a huge factor that makes it reach a wide variety of targets and that makes it very useful. Traditionally, C++ has been a language where users could act freely and do almost everything they wanted dealing directly with the internals of the computer. This implies a direct impact on efficiency and performance. And that has been the target audience that C++ has reached over years. This means that this project will already have a target audience on Software projects that already use C++. This implementation can be used from that point on so that it gives the new modules and implementations over the existing functionality correctness and reliability. Besides, we have software projects that are about to start and in whose design phase performance and efficiency are requirements. This means that those project will also be able to find an advantage of using C++ as the programming language for their project.

However, as we said traditionally, C++ has been a synonym of efficiency and performance. With the inclusion of Design By Contract, C++ will immediately start implying reliability and correctness in addition to those other features. This means that it is expanding the target audience that it initially had. It is clearly a case of horizontal expansion, with a product that is trying to fulfil the needs of a group of users that might already have a solution, but that with the appearance of C++ might find a competitive advantage and switch to it.

With that in mind, we can clearly say that the target market is Software projects where the need is either performance, efficiency, reliability or correctness. And that is a huge advantage that Design By Contract includes a programming language. With regard to the economic impact that this social impact might have, it will directly imply more investment in C++ programming language as a technology. Enterprises whose projects are based directly in C++ will favour investment in projects that are built over C++ and will imply benefits from enterprises of this kind.

In the end, it is a programming alternative that gives a lot of benefits and the way in which it is implemented allows the programming language not to be affected in the performance or efficiency terms. In fact, the availability of the custom violation handler will directly imply the availability to implement certain callback functions that can be used for either an IDS, further analysis of the code or direct interaction with the Operating System. Finally, a hidden advantage of using Design By Contract is that it will ease programming since it allows the location of errors and unexpected behaviours since the violations give information on the issue that has happened.

Chapter 9

Legal Framework

In this section 9.1 an evaluation of the applicability of the current legislation for the protection information systems will be carried out. Among the main applicable legislation we can find the LOPD and the recently applied GDPR. Later on, in section 9.2 we will be depicting the licences with which the project will be distributed.

9.1 Applicable Legislation

The legislation of information systems is nowadays applied to the protection of personal data. Personal data is any kind of data that can be used to identify an individual. Everyone should have the right to manage its personal data, however, anyone wishes. Otherwise, it will make society more controllable and could target attacks against an individual. In this section, we will start by depicting the Organic Law for Data Protection (LOPD) since it is a more complete law and we will end up evaluating the General Data Protection Regulation (GDPR).

9.1.1 Organic Law For Data Protection (LOPD)

An organic law is a legal order that derives straight from the Constitution and that will help the application of the Constitution. The Spanish Constitution guarantees to any citizen the right to the Honor, the Personal and Family Privacy, the Self Image, the inviolability of the home and the secrecy of communications. This implies that the law will limit the use of information technology to ensure any of these rights. In general, there are several concepts to be taken into account in LOPD:

- **Personal Data**. Personal data is any information able of identifying uniquely a person. It can be of any type such as name, DNI, religion, etc...
- **File**. It is named to any structured container which contains personal data.

- **Data Controller**. It is the responsibility for keeping the data file safe according to the measures that are determined by the law. It is also responsible for the usage under the conditions that were exposed to the owner of the data.
- **Data Processor**. It is an entity that processes the data on behalf of the data controller. It is usually a third party that uses the data from a data controller in order to provide some service. The data processor does have to keep the data under the same conditions as the data controller.

When we speak of the LOPD, we usually take into account the rights that anyone which delivers any personal data has. Those are the **ARCO** rights that come from:

- *Right to **A**ccess*. The ability to access the personal data that is being kept.
- *Right of **R**ectification*. The ability to change any data that is being kept.
- *Right of **C**ancellation*. The ability to eliminate the possession of those data from a third party.
- *Right of **O**position*. The negation to a party to the store the personal data.

In addition to these basic rights, the LOPD regulates what data should be extracted. It establishes that it should be the data shall be adequate, relevant and non-excessive. It rules the obligation of informing of the purpose of the collection of data whenever the data discloser claims it. In addition to that, it rules the obligation informing of the recollection of data and that data cannot be obtained without explicit permission of the discloser.

In its last part, it establishes the security measures that shall be established over data. It classifies data among several levels depending on the sensibility of the content. There are three different levels of security basic, intermediate and high. The higher the category, the more security measures are taken over it.

With regard to the *applicability* of the LOPD to our project, we can say that it does not apply. According to LOPD, we would be data processors, if we had to deal with any kind of data. However, this data is at no point stored in our program. This means that we are in no moment storing sensitive data and the data depends on it to be written in a source file. This means that the security measures shall be applied to that file and are outside our responsibility.

9.1.2 General Data Protection Regulation (GDPR)

The General Data Protection Regulation (GDPR) is the new law that has been established at European level for the data protection. These rules cover the different aspects of how any subject has to deal with data protection. Especially in Spain, it is ruled by the Spanish Association for Data Protection (AEPD). The GDPR establishes the following [49]:

- Any entity holding data from a public shall notify the corresponding authority (AEPD) of the possession of that data.
- As in the LOPD, the GDPR establishes three different classifications+ depending on the sensibility of data Risk Factor 1 (LOPD Basic), Risk Factor 2 (LOPD Intermediate) and Risk Factor 3 (LOPD High). These levels have different obligations in what data protection refers.
- The need for the explicit and unequivocal consent of the data discloser. This affirmative consent is understood if and only if the discloser agrees, in any other case, it is considered disagreement.
- The conditions under which the transfer of data can be done to third parties. This concession of services can be done under specific conditions and usually under the agreement of the data discloser.
- The data discloser has the right to *access* the data, to *rectify* the data, to *suppress* the data, to *oppose* on the obtention of data, the *limitation* of the treatment and the *portability* of data. It is similar to the ARCO rights in LOPD.

As in the case of the LOPD we are not dealing at any point with personal data directly and a compiler is not intended to deal with identifiable data. In addition to that, it is not storing at any point the personal data of any of the user that is why this legislation does not apply to this project.

9.2 Licences

Clang is distributed under a University of Illinois/NCSA Open Source License[1]. This licence leaves the program free of use for anyone that wants to contribute to it. It ensures that the changes and contributions will remain attached to the person that creates them and that the changes to the licences of any contributor have to be approved by him.

Clang and LLVM are intended to be kept open-source. The University of Illinois/NCSA Open Source License establishes mainly the following principles:

- Anyone is free to distribute LLVM and to use it freely.

[1] `https://opensource.org/licenses/UoI-NCSA.php`

- The copyright that the program includes in the source code must be retained.
- The binaries that are produced from any of this products shall also retain the legal notice.
- The name of LLVM or Clang cannot be used to promote any LLVM-derived or Clang-derived product.
- The licence does not provide the warranty on the functioning of any of the products.

The intention is that the product of this project[2] is also delivered under the same licence as a Free/Libre and Open Source Software (FLOSS) project so that anyone can access these contents and so that they can be freely used. As in the case of the NCSA Open Source Licence, any redistribution of this project must retain in the copyright the authors, the institution where they were created as well as the link to the author's page.

[2] `http://llvm.org/docs/DeveloperPolicy.html#license`

Chapter 10

Conclusion

This chapter concludes the document and enumerates future works. Section 10.1 gives an overview of the objectives that were achieved as part of this work. Section 10.2 includes some personal remarks, such as the main problems found during the project development. Finally, in Section 10.3, we elaborate on future works.

10.1 Project Retrospective

In this section, we analyze the achievement degree of the objectives described in Chapter 1. The main objective of our proposal was to develop a complete and native implementation of Design By Contract in C++. From that main objective, the following four secondary objectives were extracted:

1. **O1. Basic Functionality**, i.e. implement a set of directives that worked similarly to what other programming languages provide. As discussed earlier in this document, the Clang compiler was modified to include all the required functionality. Its behaviour is validated in Section 6.1.

2. **O2. Advanced Functionality.** We provide an advanced functionality that allow developers to activate and deactivate the compilation of contracts or the inclusion of features such as the custom contract violation handler.

3. **O3. Efficiency.** We have implemented the basic and advanced functionality without incurring in significant overheads on the application execution time. As demonstrated during the evaluation, the performance achieved using the proposed C++ contract implementation equals or even increases to that obtained by an equivalent implementation using assert statements. This is probably due to the fact that contracts give additional information to the optimizer.

4. **O4. Efficiency of the compiler.** That is, the compilation time shall not be significantly affected by the contract code generation. Because Clang was written focusing on performance, we tried to

reuse Clang code whenever possible.

10.2 Personal Conclusions

This section includes some personal remarks worth mentioning. It is important to note that it describes the personal view of the author.

This project is probably one of the most difficult tasks that I had to deal with during the bachelors degree. The author had no prior knowledge on compilers at the time this project started. The author barely understood C++ and had never dealt with such an amount of code. This made the inherently steep learning curve of a compiler even worse.

Also, in order to modify a compiler, a good understanding of the small details of the programming language is needed. The fact that Clang/LLVM is around 2 million lines of code is added to the difficulty and forced the author to make an extra effort. A list of common problems that were found follows:

- Added code interfering with any of the existing tasks of the compiler.
- The control flow is hard to guess.
- New code causes the compiler to fail for no apparent reason.

After the completion of this work, the author managed to build strategies to deal with huge code-bases. The effort is worth the acquired knowledge.

10.3 Future Work

In this section we describe future work that may be accomplished on top of this project.

- Contract inheritance, i.e. make a subclass inherit contract specified as part of the superclass, was one of the features that were not finally implemented. It is, nevertheless, useful in OOP to be able to write common pre/post-conditions at the highest level in the class hierarchy.
- Also, it would be interesting to analyze the impact that a custom violation handler has to the size of the binary and its execution time.
- Finally, contracts may be useful to interfere in the optimization phase of the compiler. We proved that using contracts may, depending on the user code, improve performance. As a matter of fact, the optimizer may be forced to assume that expressions that are part of contracts are always true, even if no code is being generated. As a consequence, it should further improve execution times for most cases.

Appendix A

Implementation

In this section, we will be dealing with the implementation of the functionality within the Clang compiler. The learning process of a compiler is not easy and has an exponential curve. So it is very difficult to start doing things from scratch without previous background. This project is therefore divided into two phases of development.

During this implementation, the changes that were necessary to be made to the compiler are described.

A.1 Practical Background on Clang

The first step of the implementation was to download the source code of Clang. This was done through their webpage and by means of the Subversion system, it was downloaded. The steps are available in [50]. Following this steps, we ended up having a running copy of Clang with its source code. However, Clang needs of LLVM and many other elements behind to make it work. The only step that we changed from the installation guide was the path of the **build** directory which was placed within the tree structure as it is usually done with source code programs.

Now we mention some of the most important directories to take into account when dealing with Clang compiler. These directories are be employed during the implementation phase and are important to be taken into account. We assume that the root path (/) is where Clang is downloaded. **/tools/clang/** is the root of Clang tree structure. Clang is placed as a tool that works over LLVM, therefore we need to place it here. In this directory, we can find a tree structure where we find all the code that makes Clang run. **/tools/clang/include/** is the directory where we can find all the header files of Clang. They are usually classified by categories and permit to observe a high-level interface of what the code

finally is. Apart from that it sometimes includes implementations of different functions in case that the implementation is very short. Finally, it includes *.td* files, which are TableGen files. **/tools/clang/lib/** is the directory where we find the implementation of Clang. In this directories is where the main implementation will be written. **/tools/clang/test/** is the directory that is used in the evaluation section for the implementation of the unit tests.

A.2 First Phase

In this section, we describe the first phase of the implementation. During the first phase of the implementation, the basic building blocks of the specification had to be developed. It was needed to create a basic set of functions on which the later implementation could be supported. Implementing something in a compiler requires to change the way of thinking.

The first element to be implemented was the *assert* attribute. If we remember from the requirements section 4.8.1, we needed to follow the following syntax for attributes *[[assert assertion-level: condition]]*. If we look at this syntax, it reminds a lot of the C++ attribute syntax *[[attribute]]*. An attribute is a specifier that is applied over any statement of the code and whose purpose is to give further information than the available on the sentence of the code. With that in mind, the first intention was to develop this as an attribute. In fact, from the requirements phase, we already had the clue that it had to be done this way since they had to be applied to the specific type of statements and the only way to check that was associating them directly to that.

With that in mind, we had to find where attributes were generated and how they were known by the system. This process is done automatically by a program called TableGen which is a program included within the LLVM infrastructure. The program is able to generate the implementation of the class from a file including the definition of some attributes of the class. The definition of the elements follows a specific syntax that simplifies the implementation avoiding having to include specific details. When building the compiler, it will create from it the different classes and implementation of those. Understanding how it worked was difficult since no previous knowledge of the compiler was attained. Once it is understood that the code is not generated in a usual class, the process was to define an entry in the TableGen file. TableGen files use the extension *.td* and are huge lists of definitions of classes and their code omitting a huge part of it.

Since no previous knowledge of how attributes were assumed, we had to look at what attributes did and to their fields. Those fields are the ones in charge of specifying what is the behaviour that they will attain. The first process consisted of investigating whether any attribute already existed with a similar

functionality in Clang. In such case, we could even be reusing some of its functionality. While analyzing the full list of attributes in [51], we realised there was an attribute which performed a very similar function to the *assert* attribute that we were desiring. In this case the attribute is called `diagnose_if`. It holds a condition, a message and optionally a warning level. This attribute works at compile-time and evaluates the condition. In case that is false, it will warn the user. For that, the condition must be evaluable at compile-time. However, there are two main differences between this attribute and ours. The first is that it is a GNU attribute. GNU attributes are the syntax that GCC attributes follow and affect usually this type of compiler. Since Clang aims to be interoperable, it also supports this syntax. The syntax used is *__attribute__((diagnose_if(a >= 0, "a is not bigger than 0", "warning")))*. The second main difference apart from the syntax is the compile-time evaluation. We are interested in this fact, however, we are also interested in the fact of evaluating the condition at runtime in case that it is not possible at compile-time.

This implies that modifications had to be made to the attribute, first, because we needed to change the syntax to be the C++ syntax, and later on to support the runtime evaluation of the condition. Then we had to look for the implementation of the *diagnose_if* attribute in order to analyze how it received the arguments and how they were later on used for the implementation. If we look at the code below, we can observe most of the characteristics of the attribute and see how it is generated by the reception of its arguments. The *diagnose_if* attribute can be found in the directory */tools/clang/include/clang/Basic/Attr.td*. This is important to be remarked because even finding the generation of this is not as trivial and easy as it might seem. The most important ones are the following three:

- **Spellings**. It determines which is the keyword that is going to identify the attribute. Apart from that, it includes the set of attributes within which it is included. In the case of diagnose_if, it is included in the GNU set. This implies that we have to switch it to start belonging to C++.

- **Subjects**. They determine which is going to be the nodes that are able to be affected by the attribute. In the case of the `diagnose_if`, we see that mainly we see functions are what affects it. Other than that, we see Objective-C nodes that are not of our interest. This means that our attribute should be affecting only *null statements*.

- **Args**. The args is the set of arguments that our attribute receives. In this case, we can easily identify an expression a string argument and an enum argument. The expression is going to be the condition of the attribute, the string argument is going to be the message and the enum is going to have either the value of "warning" or "error" in case that it fails to be marked in either of this two ways.

Figure A.1 shows the diagnose_if code in *Attr.td*.

Listing A.1: *Diagnose_if attribute definition in Attr.td*

```
def DiagnoseIf : InheritableAttr {
  //The spelling determines which is the keyword identifying the attribute
  let Spellings = [GNU<"diagnose_if">];
  //The subjects establish which nodes this can be associated to functions and methods.
  let Subjects = SubjectList<[Function, ObjCMethod, ObjCProperty]>;
  //The arguments that our code is going to receive.
  let Args = [ExprArgument<"Cond">, StringArgument<"Message">,
              EnumArgument<"DiagnosticType",
                           "DiagnosticType",
                           ["error", "warning"],
                           ["DT_Error", "DT_Warning"]>,
              BoolArgument<"ArgDependent", 0, /*fake*/ 1>,
              NamedArgument<"Parent", 0, /*fake*/ 1>];
  let DuplicatesAllowedWhileMerging = 1;
  let LateParsed = 1;
  let AdditionalMembers = [{
    bool isError() const { return diagnosticType == DT_Error; }
    bool isWarning() const { return diagnosticType == DT_Warning; }
  }];
  let TemplateDependent = 1;
  let Documentation = [DiagnoseIfDocs];
  }
```

With that in mind, we can already try to define our new attribute. The attribute that is defined is going to be called *assert*. We only want it to receive two arguments, the condition and the assertion level, being this last one optional. With regard to the subject, it has to be associated to a *null statement* however since this check is usually performed in Sema it can be left apart and will be checked later on. With regard to the syntax that is going to hold is not going to be the final one that we want. That is not a responsibility of this section. The syntax that is going to admit is going to be `[[assert(assertion-level, condition)]]`, but we will deal with this in a later step. The resulting attribute is going to be of this shape:

Listing A.2: *Assert attribute definition in Attr.td*

```
def Assert : StmtAttr {
  //The spelling is according to what is specified.
  let Spellings = [CXX11<"", "assert", 201603>];
  //The set of arguments, the order is important.
```

```
5   let Args = [IdentifierArgument<"Level">,
6               ExprArgument<"Cond">];
7   let Documentation = [Undocumented];
8 }
```

Figure A.2 represents the syntax of the new attribute.

It is important to remark the order that it is accepting. If it does not take this order, the attribute will be accepted later when we modify the syntax an incorrect order, and in this case, it is important for the further evaluation of the changes in the grammar.

A.2.1 Sema Modifications

In this section we describe the changes in the Semantic Analysis Since making the changes in the grammar is one of the most difficult changes, we are going to leave it for the end, and we are going to first assume the syntax is the correct one. If we recap from what we proposed in the explanation, the Sema component in Clang receives the pieces from the attribute in order for it to be able to analyze it. With this pieces, it is able to create the node of the specific element that is needed. In what is applied to our case, the Sema code shall verify that the pieces that form our attribute are correct. Then it will have to create a node for the AST and return it.

For the evaluation of the attribute, we perform several steps. First, it has to take a look at the condition and shall evaluate whether what we are putting is convertible to a boolean or a condition. Many elements are convertible to conditions in C++, either way, there are certain expressions which are difficult to evaluate, or even not possible to evaluate, such as *null*. Regarding the statement, we have to consider that the attribute is associated with a *null statement*, all this can be checked with *Sema* since its functions receive this evaluation. The evaluation of the attributes in Sema is done hierarchically. There is a bigger function which is calling all the evaluations and the evaluations are later divided by categories. In this case, since our attribute is associated with a statement, we need to use the function `ProcessStmtAttribute`. This function receives as parameters a statement (*Stmt*), the attribute list, which is the set of arguments that a compiler receives, *Sema* which is a bigger object which contains the state of the program and allows us to obtain its functions, and the range (*SourceRange*) where the attribute is. The following code can be found in */tools/clang/lib/Sema/SemaStmtAttr.cpp*.

Listing A.3: *Assert attribute node generation in SemaStmtAttr.cpp*

```
1 static Attr *handleAssertAttr(Sema &S, Stmt *St, const AttributeList &A,
2                                SourceRange Range) {
```

```
3   //Evaluating if we are getting the desired number of attributes or not
4   if (A.getNumArgs() < 2) {
5     S.Diag(A.getLoc(), diag::err_attribute_too_few_arguments) << A.getName() << 2;
6     return nullptr;
7   }
8   //We contextually convert the expression to a bool to evaluate if it is a correct condition
9   Expr *E = A.getArgAsExpr(1);
10  if (!E->isTypeDependent()) {
11    ExprResult Converted = S.PerformContextuallyConvertToBool(E);
12    if (Converted.isInvalid())
13      return nullptr;
14    //If everything is correct we get the correct condition as an expression
15    E = Converted.get();
16  }
17  //At this point the attribute is created with the parameters received.
18  AssertAttr Attr(A.getRange(), S.Context, A.getArgAsIdent(0)->Ident, E,
19                    A.getAttributeSpellingListIndex());
20  //We evaluate whether the statement to which it is associated is a NullStmt
21  if (!isa<NullStmt>(St)) {
22    S.Diag(A.getRange().getBegin(), diag::err_fallthrough_attr_wrong_target)
23        << Attr.getSpelling() << St->getLocStart();
24    return nullptr;
25  }
26  // If this is spelled as the standard C++1z attribute, but not in C++1z, warn
27  // about using it as an extension.
28  if (!S.getLangOpts().CPlusPlus1z && A.isCXX11Attribute() &&
29      !A.getScopeName())
30    S.Diag(A.getLoc(), diag::ext_cxx17_attr) << A.getName();
31
32  return ::new (S.Context) auto(Attr);
33 }
```

If we take a look at the code in Listing A.3, we can observe that we are performing the check of the different conditions that we said before. The main difficulty is finding the element that you need in order to perform the transformation of the elements. As we can see, at the end of this method the newly created node is returned. It is done through the *ASTContext*, a class that is going to deal with

the different elements of the AST. This class has a memory allocator, that is why all the elements are created this way since it allows that we are later able to free this memory without leaving an unoccupied memory. It is a pretty interesting implementation since it hides all the memory management task so that anyone implementing anything new has only to care about generating the new node and allocating it in the *ASTContext*. It is important to remark that *ASTContext* is one of the core classes that we are dealing with repeated times during this implementation. Another important class is *Sema* which is the core of the Semantic Analysis, it is probably one of the most powerful classes in Clang since it provides mechanisms for almost anything. However, Sema is only present in this phase of the analysis.

Once we have dealt with the generation of the node, we test whether the node is being properly generated. For this purpose, we use a tool that Clang provides and allows the user to print the AST that a code is generating. Since we are already generating code for this kind of statement, we should be able to see how the node is generated. The following image represents the subsection of the AST related to the *assert attribute*. Figure A-1 shows the generation of the node in the AST.

```
-AttributedStmt 0x22a57b0 <line:3:9, col:26>
 |-AssertAttr 0x22a5790 <col:11, col:24> default
 | `-BinaryOperator 0x22a5748 <col:19, col:23> '_Bool' '>'
 |   |-ImplicitCastExpr 0x22a5730 <col:19> 'int' <LValueToRValue>
 |   | `-DeclRefExpr 0x22a56e8 <col:19> 'int' lvalue ParmVar 0x22a5530 'num' 'int'
 |   `-IntegerLiteral 0x22a5710 <col:23> 'int' 0
 `-NullStmt 0x22a5780 <col:26>
```

Figure A-1: *Assert Attribute AST Generation for [[assert: num > 0]]*

A.2.2 Code Gen Modification

If we remember from what we commented on the explanation, once the node in the AST is generated, we just need to implement the code generation that we want it to be equivalent to. This implementation is then part of the CodeGen implementation. In CodeGen there are several alternatives on how we can act with a node. Usually when we want to implement a complex behaviour, we have to deal with creating our new generic LLVM code and make its generation. However, if the behaviour can be represented with the abstraction that the AST provides that is not necessary and although it is also complex to represent the code by means of objects, we can use it to generate just small parts of the AST. In our case, the code that we want to implement the pseudo-code that we want to implement for any of the contracts is the following:

Listing A.4: *Assert attribute code*

```
1 ...
2 [[assert: x > 0]];
3 ...
```

Listing A.5: *Assert attribute equivalence*

```
1 ...
2 if (!(x > 0)){
3     //std::terminate() == abort()
4     std::terminate();
5 }
6 ...
```

The important points to take into account at this point is the behaviour. As we see in Figure A.4 and Figure A.5, we need to perform a function call to `std::terminate`, which interrupts the execution of the code. In C++ it the usual way of interrupting the execution is by means of *std::terminate()* but it is equivalent to the C expression *abort()*. And we will have to switch the direction of the condition, meaning that if we need to perform this call to abort when the condition is not fulfilled.

All this code is implemented in CodeGen. As in the rest of the elements, CodeGen is divided into several archives depending on the element that it deals with. In the case of the assert, we are dealing with the generation of a statement, and what we have to modify is that whenever we deal with an assert statement, it is substituted by the code on the right side of Figure A.5. The way of doing this is by creating a new function in *CGStmt.cpp* which is the code file that is in charge of dealing with the statements. What we have to modify is the generation of *AttributedStmt* nodes. When we try to generate something related to that, we have to check if the attribute that is associated is an *assert* attribute. This is implemented, however, we have to add an entry to a list of checks that is verified when we deal with an attribute of this type.

In Listing A.6,we are operating with the expression contained in the code. The expression needs to be extracted from its container which is the attribute, and once it is extracted, we negate it by applying the *unary operator not* (!) and then we are going to build an *if statement*. For the if statement, we already have the condition, the only thing that we have to modify is the *then statement*. For this first version, it is going to be just the `std::terminate` call. To do so, we create a function that later on will be generalized. This function searches over the symbols table for a declaration with that name and that type. Once it finds it we will generate a *call expression* to that function declaration. That is all done within the function *GetRuntimeFunctionDecl*. This function will be modified in a newer version of the code to be generalized for any behaviour. With that in mind, we will already have the unary operator and the call expression which is what we need to generate the if statement. Once the if statement is generated, we already have a function to generate the code equivalent to an if statement. This code will be generated at an equivalent point as the assert attribute. Meaning that by putting the attribute we will already be generating this

behaviour that we desired. This code can be found in */tools/clang/lib/CodeGen/CGStmt.cpp*

Listing A.6: *Assert attribute code generation*

```
1  void CodeGenFunction::EmitAssertAttr(const AssertAttr *_Attr,
2                                       SourceLocation Loc) {
3     //We extract the Call Expression to abort
4     CallExpr *CE = SynthesizeCallToFunctionDecl(&getContext(),
5           const_cast<FunctionDecl *>(CGM.GetRuntimeFunctionDecl(getContext(),
6                                       "terminate")), {});
7     //We extract the condition into an expression
8     ParenExpr *PE = new (getContext()) ParenExpr(
9            SourceLocation(), SourceLocation(), _Attr->getCond());
10    UnaryOperator *UO = new (getContext()) UnaryOperator(
11           PE, UO_LNot, PE->getType(), VK_RValue, OK_Ordinary,
12           SourceLocation());
13    //Generating the IfStmt
14    auto _S = new (getContext()) IfStmt(getContext(),
15           Loc, false, nullptr, nullptr, UO, CE);
16    //Emitting the IfStmt
17    EmitIfStmt(*_S);
18 }
```

A.2.3 Parser Modifications

Once we reach this point, the functionality of the assert attribute is almost fulfilled, we are already creating an attribute that is able to evaluate the condition. But we still need to perform some changes on the code, to adopt the syntax that we desired. What we need now is to deal with the grammar that we desire in order to evaluate what is the syntax to be admitted. To do so, we have to reach the point of the parser where the attributes are analyzed and permit that the attributes that we chose are used with a different syntax. This code can be found in */tools/clang/lib/Parse/ParseDeclCXX.cpp*

Listing A.7: *Assert attribute parser code*

```
1  unsigned Parser::ParseContractAttrArgs(IdentifierInfo *AttrName, SourceLocation AttrNameLoc,
2                                         ParsedAttributes &Attrs, SourceLocation *EndLoc) {
3    AttributeList::Kind AttrKind =
4       AttributeList::getKind(AttrName, nullptr, AttributeList::AS_CXX11);
5    ArgsVector ArgExprs;
```

```
6   ExprResult ArgExpr(static_cast<Expr*>(nullptr));
7   if (!Ident_axiom) {
8     Ident_axiom = PP.getIdentifierInfo("axiom");
9     Ident_default = PP.getIdentifierInfo("default");
10    Ident_audit = PP.getIdentifierInfo("audit");
11    Ident_always = PP.getIdentifierInfo("always");
12  }
13  IdentifierInfo *II1 = Ident_default;
14  SourceLocation Loc1;
15  if (Tok.isOneOf(tok::identifier, tok::kw_default)) {
16    II1 = Tok.getIdentifierInfo();
17    Loc1 = ConsumeToken();
18  }
19  if (ExpectAndConsume(tok::colon))
20    goto out;
21  ArgExpr = Actions.CorrectDelayedTyposInExpr(ParseExpression());
22  if (ArgExpr.isInvalid() || !Tok.is(tok::r_square)) {
23    Diag(Tok.getLocation(), diag::err_expected) << tok::r_square;
24    goto out;
25  }
26  ArgExprs.push_back(IdentifierLoc::create(Actions.Context, Loc1, II1));
27  ArgExprs.push_back(ArgExpr.get());
28  Attrs.addNew(AttrName, SourceRange(AttrNameLoc, Tok.getLocation()), nullptr, SourceLocation(),
29               ArgExprs.data(), ArgExprs.size(), AttributeList::AS_CXX11);
30  if (EndLoc) *EndLoc = Tok.getLocation();
31 out:
32  SkipUntil(tok::r_square, StopAtSemi | StopBeforeMatch);
33  return static_cast<unsigned>(ArgExprs.size());
34 }
```

Before Listing A.7, a little fragment of code is executed. This fragment of code is going to receive the first two square brackets of the attribute and it is going to discard them since they are of no use. Right after that, the program is going to identify the attribute with its name, this name will be an identifier as a variable or function name and will be stored in the symbols table although as a generic identifier. This identifier name is what we receive in this function. We use the identifier to extract the *attribute kind*. The attribute kind is generated automatically from the TableGen that we mentioned previously. This

is important since we are going to verify that our contract, in fact, is an *assert attribute*. Otherwise, we would be dealing and return since we are not dealing with a correct expression.

Once we verify it, we are able to go for the next element in the specification. This element is the *assertion-level*. We did not deal with its implementation yet, but with regard to the parsing, it is important to take it into account because it will be later used for the generation of contracts. If we recap, there were 4 possible assertion levels *axiom, always, default and audit*. These assertion levels determine which are the contracts that are activated at each point. This forces that our field has one of the four values. For that, we extract the identifiers from the *symbols table* and cache it in the parser so that they do not have to be extracted each time we deal with them. Then we are going to deal with other of the requirements which is that the default assertion level if none was specified, shall be *default*. For that purpose, we generate a variable which is going to hold the identifier for the attribute. This identifier is going to have by default, the *default* assertion level. Then the function *isOneOf()* allows the extraction of the identifier Token in case it is of this kind. In such case, we extract it and assign it to the identifier variable. In any other case, the default value will remain. Then, we go for the next element in the structure, the colon. We look for it and otherwise interrupt the execution.

Finally, after the colon, we just have the condition. For this purpose, a function provided by the parser is used. This function is *ParseExpression* and returns an expression from the tokens that it will analyze. Since we can have mistakes in the expression because the user can commit them, we use the function *isInvalid* to evaluate it. Finally, we check whether we are in a right side square bracket. If that is the case it means we have correctly parsed an expression and then we are able to generate the arguments of the Attribute and push them back. Those arguments will be pushed into the attributes list that we mentioned in the Semantic Analyzer part. With that in mind, we already have the possibility of using this new syntax. In fact, the inclusion of this new function excludes from now on the old syntax of an attribute (*[[attribute(arg1, arg2)]]*) for the contracts making a difference between the concept of contract and attribute.

A.3 Second phase

In this section, we are going to deal with the advanced implementation of the project. This advanced implementation deals with further functionality related to contracts.

A.3.1 Expects and Ensures

At the end of the first phase, the *assert* attribute was already developed. However, there were still contracts to be developed. However, the development of those was more difficult than it would seem. The

evaluation of the assert attribute could be just the substitution of it by an *if statement*, however, there were still some major changes that needed to be made for the evaluation of the contracts that were assigned to the function type as it was in the specification.

In order to solve this problem of the assignation to the function type, the solution found was the creation of two versions of the function for the argument check. Before talking more about this, we are going to explain a technique which is going to be used in this section. This technique is inlining. In C and C++, there are arguments that support inlining. Inlining consists of substituting a call to a function by the code inside it. This has several results, on the one hand, it improves efficiency since it implies neither movements of data nor branch instructions, on the other hand, it increases noticeably the size of the code. For the application of this technique in our problem, only the side effects of this were taken into account.

The implementation consisted of the creation of an unchecked version of a function. The unchecked version of the function performed inside it first the checks of the contracts and then a call to the main function was performed. The great thing about this is that the calls to the version with the contracts were substituted by this function meaning that all end up working at the points they are needed. It has a lot of advantages for the user. The first is the reduction of the size of the executable. Another alternative would have been performing all the checks of the preconditions on the first line of the function, and all the checks of the postconditions before any return call. This, of course, would have ended up generating a bigger amount of code. In fact, it loses no performance with respect to the original functions because the original function is marked as inline. This means that the body of the function will be substituted on its calls, which will be just one. What is more, since all these changes are performed artificially, and the calls to the original function are substituted, the original function is not used, and at the end is removed leaving the executable with only one version of the function, which is the function with contracts.

Listing A.8: *expects and ensures code*

```
1  void square(int x)
2  [[expects: x > 0]]
3  [[ensures ret: ret == x * x]]
4  {
5      return x * x;
6  }
7  int main(int argc, char* argv[]){
8      int x = square(10);
9      return 0;
10 }
```

Listing A.9: *expects and ensures equivalence*

```
1  int square_unchecked(int x)
2  [[expects: x > 0]]
3  [[ensures ret: ret == x * x]]
4   __attribute__((always_inline)) //Inline
5   {
6      return x * x;
7  }
8  //New unchecked function
9  int square(int x)
10 {
11     //Preconditions evaluation
12     if(!(x > 0)){ std::terminate();}
13     //Function execution
14     int ret = square(x);
15     //Postconditions evaluation
16     if(!(ret == x * x)){ std::terminate();}
17     return ret;
18 }
19 int main(int argc, char* argv[]){
20     //Modified function call
21     int x = square_unchecked(10);
22     return 0;
23 }
```

Figure A.8 shows how this idea provided a solution both efficient and at the same time good for the implementation of preconditions and postconditions. The code that had to be developed for this phase included the following changes:

- **Variable Creation for *ensures***. The ensures contract implied that we needed a variable to link the return value. This variable was declared thanks to the support that C and C++ have for declaring functions. It is called the Kernighan and Ritchie (K&R) style. In this style, some variables are declared in the function declaration. With this mechanism when the identifier is parsed, an identifier is created in the symbols table. This involved later problems regarding the resolution of template variables. Because of the generation of the prototype dependent variable.

- **Attr.td and TableGen**. The new attributes required a new entry in the TableGen as usual, however,

both required to include new fields, since in functions we can have a separate declaration and definition of it. In order to solve that a new field had to be implemented and since it was not available, the TableGen generation of classes had to be modified.

- **Parser**. The parser required a more complex implementation since, in the case of the *ensures*, we had the possibility of adding the return variable. This implied that a common representation of the variable had to be taken into account and even at that point, the possibility of choosing different names was possible. In fact, this also had to include code for a possible delayed generation in case it is used with generic programming.

- **Sema**. In the case of Sema, the new attributes, *expects* and *ensures*, had to be related to the a *declaration* instead of an *statement*. This causes the implementation to be present in *SemaDeclAttr.cpp* instead. The checks performed in this section were pretty similar to the ones made in the case of assert with the difference that we check that it is associated to a function declaration (a *FunctionDecl* object).

In the following sections, we will be dealing with several features that were implemented in order to complete the features that we have in the contract's specification.

A.3.2 Build Level

One of the first elements we had to deal with was the implementation of the *build level*. If we recap from the specification, the build level is a flag that had to be implemented in order to deactivate and activate contracts as the user desired. There are three main build levels: *off, default and audit*. Before the generation of the code verification, the build level is compared to the assertion level and depending on both the validation is generated or not. First of all, we have to take into account that *axiom* contracts are annotations. They are present in the source code to give the programmer information. That means that they never generate code. With regard to *always* contracts, they are always verified, no matter what the build level is. For the remaining two assertion levels it depends on the build level. With the *default* build level, *always and default* assertion level attributes are taken into account. With the *audit* build level, *always, default and audit* assertion level attributes are taken into account. Figure A-2

For the implementation of the functionality, there were two parts missing. First of all, we needed to establish a mechanism for the user to specify the build level. Then we have to implement how the code is going to behave depending on the build level. The first part is the most difficult one. It is a process which is easily repeatable but that the first time that it has to be done requires a lot of patience since no result was found. I have to note that the information of how the Driver was working was unknown until the moment in which this document was written. While I was trying to develop the functionality it was like a blind search since nothing seem to be able to help me find the solution.

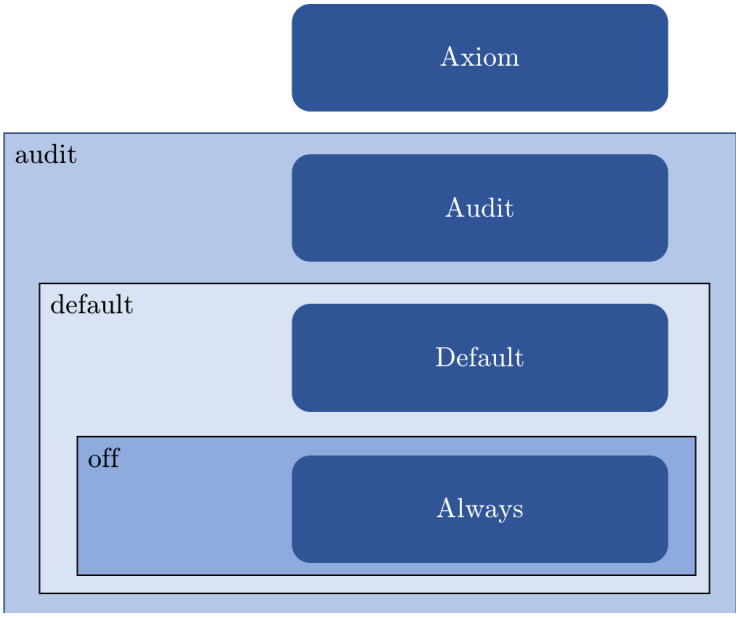

Figure A-2: *Assertion Levels vs Build Levels*

The implementation of a compilation flag is a similar process to the attribute implementation. Flags are placed in some directories where we are later able to generate some information about them. The first step to carry out is the inclusion of the new flag with its syntax so that the compiler recognizes it. There are several lists in Clang where attributes can be placed and where they will, later on, be treated by similarly as TableGen files for the generation of the compilation options list. The list in which we are interested in firstly is the list that the Driver receives. Placing our argument in any other list would not make the user able to introduce it as a compilation option since the compiler would not recognise it. The list that we are interested in is *Options.td*. This list follows a specific format of attributes that is needed to follow in order to include a new element. In our case we are interested in supporting two syntaxes *–build-level=<build_level>* and *-build-level=<build_level>*. This code can be found in */tools/clang/include/clang/Driver/Options.td*.

Listing A.10: *–build-level flag inclusion in Driver*

```
def build_level_EQ : Joined<["-", "--"], "build-level=">,
                     Flags<[CC1Option, CC1AsOption, CoreOption]>,
                     HelpText<"Build level for this translation unit">,
                     Values<"off,default,audit">;
```

Followingly, we will explain what the code means. Firstly it identifies the attribute. The name given before to the attribute is not trivial. It has to coincide with the name later on given in the syntax and it does allow only certain syntaxes. In this case the "_" is translated later on to "-" and the "_EQ" is translated to "=". The clause *Joined* is used to express that we are going to give more than one syntax. The Flags clause is used to classify the option within the different options of the system. In this case, CC1 refers to the compiler tool. There are flags which are destined for different other purposes. However, as far as our implementation has reached, choosing different values for this field does not alter the behaviour. Its functionality is only classification of the different compilation flags. Finally, we determine a help text which serves as documentation for this flag, and the possible values that it takes, which are the ones mentioned before, *off, default and audit*. All these fields affect the Driver, meaning that are useful for the Driver in terms of generating representative error reports for the user. However, with just this, the flag is just recognized in the system and no behaviour is produced.

By this point, this flag is only persistent in the Clang Driver, this means that after that, if the flag was not preserved somehow, the flag was going to be lost. As we already mentioned, each Clang tools and even subcomponents has its own flags. These flags are expressed by means of different *.def* files. In our case, the decision to be made was to choose to keep the flag either in *CodeGenOptions.def*, which is destined for options affecting the generation of code or in *LangOptions.def* which is destined to hold the language options. In this case our option is going to affect whether the code for a contract is generated or not, however, we are a lot more interested in holding it inside the LangOptions since it grants it to be visible to the whole compiler in case that we need to perform any further modification to the flag. The file *LangOptions.def* can be found in *tools/clang/include/clang/Basic/LangOptions.def*. The code included is the following:

Listing A.11: *build level flag in Lang Options*

```
VALUE_LANGOPT(BuildLevel, 2, 1, "C++ contracts build level")///build-level=off,default,audit
```

The first field is the name that is going to be given to the variable in the code. The second field determines the number of bits that are needed. In our case since we have 3 possible values, we need at least 2 bits. The third field represents the default value. In this the representation given to those values is 0 for *off* build level , 1 for *default* build level and 2 for *audit* build level. This representation is created

and followed in the implementation. In this case, that is why the third value is 1 because, as it is said in the specification, the default build level in case that the user does not specify it is going to be *default*.

Taking that into account, we have to find how the information is moved from the first list of compilation flags to the second of LangOptions. However, and it is here where we found our main problem. The list from which we were trying to move the arguments from was not the Driver flag list. This meant that there was a hidden interaction between both actions that was in charge of moving the arguments. Realising that this was the problem was not trivial at that point. Neither was finding where the elements were translated. We found that the Driver gives it control to the specific Toolchain and it is in charge of determining which arguments are passed to each of the things. The main reason to follow this approach is to be similar to what GCC does. The file where the files can be moved from the Driver to the CC1 argument list is Clang.cpp, which is a generic file expressing the toolchain of the compiler. This file can be found in */tools/clang/Driver/Toolchains/Clang.cpp*. The changes that were performed in this file were the following:

Listing A.12: *Argument Management in Clang Toolchain*

```
1  void Clang::ConstructJob(Compilation &C, const JobAction &JA,
2                           const InputInfo &Output, const InputInfoList &Inputs,
3                           const ArgList &Args, const char *LinkingOutput) const {
4                             const Driver &D = getToolChain().getDriver();
5    const Driver &D = getToolChain().getDriver();
6    ArgStringList CmdArgs; //Arguments that are passed to the job
7    ...
8    if (Arg *A = Args.getLastArg(options::OPT_build_level_EQ))
9      A->render(Args, CmdArgs);
10   ...
11   getToolChain().addClangWarningOptions(CmdArgs);
12   ...
13 }
```

The code in Figure A.12 takes the argument from the Driver Flag List checking if it is present or not. If it is present, it moves it from the Driver list to the Compiler Option List. Finally, the compiler arguments are moved to the next place.

After some search following the flow of the program, we found that in the *front-end* library was where most of the information traversed from the list compiler argument list to the language options. Specifically, it was performed in the *CompilerInvocation.cpp*. In that point arguments received by the compiler

were processed and moved specifically to each of the places. At this point in the implementation, we have to relate things as we said we would do it. We are going to store in the language options the bit corresponding to the build level that we received. The code that is going to perform this translation is:

Listing A.13: *Build Level handling in CompilerInvocation.cpp*

```
1  static void ParseLangArgs(LangOptions &Opts, ArgList &Args, InputKind IK,
2                            const TargetOptions &TargetOpts,
3                            PreprocessorOptions &PPOpts,
4                            DiagnosticsEngine &Diags) {
5   ...
6  // Handle -build-level= option.
7   if (Arg *A = Args.getLastArg(OPT_build_level_EQ)) {
8     unsigned Val = llvm::StringSwitch<unsigned>(A->getValue())
9         .Case("off", 0)
10        .Case("default", 1)
11        .Case("audit", 2)
12        .Default(~0U);
13    if (Val == ~0U)
14      Diags.Report(diag::err_drv_invalid_value) << A->getAsString(Args) << A->getValue();
15    else
16      Opts.BuildLevel = Val;
17  }
18 ...
```

With the code in A.13, we put the value inside the *BuildLevel* attribute included in the LangOptions file. That means that the comparison among the build levels is already possible. In this code, a templatized class is used. This class is StringSwitch, which performs a switch with strings, something that is not usually offered with the classical C++ switch. In the case that we are putting an invalid value, we are going to launch an error with the diagnostics engine.

With that in mind, the only thing, that we need to evaluate is whether the code has to be generated in the CodeGen section. For this purpose, a identification is generated for each assertion level. The identifications assigned are 0 for *always*, 1 for *default*, 2 for *audit* and 4 for *axiom*. We will only be generating the code in case that the assertion level is smaller or equal to the build level. This will enforce the *always* level to always be generated and taken into account. And the *axiom* level as it is always bigger than the build level is never going to be generated. This code is generated together with the code that was in charge of generating the code in */tools/clang/lib/CodeGen/CGStmt.cpp*.

Listing A.14: *CodeGen Assert Emission with Build Level*

```
void CodeGenFunction::EmitAssertAttr(const AssertAttr *_Attr,
                                     SourceLocation Loc) {
   unsigned Level = llvm::StringSwitch<unsigned>(_Attr->getLevel()->getName())
       .Case("always", 0) // Assert is emitted even if -build-level=off
       .Case("default", 1)
       .Case("audit", 2)
       .Default(~0U);
   //If Level > BuildLevel we stop execution.
   if (CGM.getLangOpts().BuildLevel < Level)
   return;
   //We extract the Call Expression to abort
   CallExpr *CE = SynthesizeCallToFunctionDecl(&getContext(),
         const_cast<FunctionDecl *>(CGM.GetRuntimeFunctionDecl(getContext(),
                                    "terminate")), {});
   //We extract the condition into an expression
   ParenExpr *PE = new (getContext()) ParenExpr(
          SourceLocation(), SourceLocation(), _Attr->getCond());
   UnaryOperator *UO = new (getContext()) UnaryOperator(
          PE, UO_LNot, PE->getType(), VK_RValue, OK_Ordinary,
          SourceLocation());
   //Generating the IfStmt
   auto _S = new (getContext()) IfStmt(getContext(),
          Loc, false, nullptr, nullptr, UO, CE);
   //Emitting the IfStmt
   EmitIfStmt(*_S);
}
```

Figure A.14 shows the new version of the Code Generation with the build level. With that in mind, everything regarding the assertion levels and build levels would be implemented and the functionality provided.

A.3.3 Contract Violation Handler Generation

In this section, we explain how the functionality of the custom violation handler was developed. When a contract is not fulfilled, a violation handler is executed. The violation handler is a function that by default will do nothing and at the end will abort by executing `std::terminate()`. However, it exists

the possibility that the user defines its own defined contract violation handlers. Those handlers will be defined by the user and will determine which is the behaviour that the program has once the violation happens. In our case, the violation carries out some consequences. The violation handler receives by parameter an object which holds all the data about the violation. The object that is created, is represented in the Figure A.15. This is a high-level interface that represents the public attributes, if for our implementation we need anything different, we can consider changing the private attributes.

Listing A.15: *Contract Violation Object*

```
1  namespace std {
2    class contract_violation {
3    public:
4      int line_number() const noexcept;
5      const char * file_name() const noexcept;
6      const char * function_name() const noexcept;
7      const char * comment() const noexcept;
8      const char * contract_violation() const noexcept;
9    };
10 }
```

In order to solve the generation, we had to first be generating a call to this function handler that is a function that the user provided and then passing as a compilation flag. The first step consisted of generating a function that could be executed, for that we needed to find first the declaration. Later on, we would make generate the call expression to that function declaration. What we received from the user was a function name. This means that we needed to be able to generate a call to a function with just that name. This was the first objective, for that we created a function that searched among all the identifiers and returned as an element corresponding to the type and with the same name. A first approach to this function was obtained from the function *GetRuntimeFunctionDecl*. This function was specifically created for FunctionDecl objects. However, as it will be proposed later, the approach was generalized in a clever implementation to show a generic behaviour and finding any kind of object. The generation of the call was made by another function, *SynthesizeCallToFunctionDecl*. This function was able to create a function call (*CallExpr*) from the function declaration. With a *CallExpr*, CodeGen is going to generate the equivalent IR of a function call. The code for that is divided into two parts. We are going first on the basics of *GetRuntimeFuncitionDecl*:

Listing A.16: *Obtaining Function Declaration from Name*

```
1  const FunctionDecl *
2  CodeGenModule::GetRuntimeFunctionDecl(ASTContext &C, StringRef Name) {
3    TranslationUnitDecl *TUDecl = C.getTranslationUnitDecl();
```

```
4   DeclContext *DC = TranslationUnitDecl::castToDeclContext(TUDecl);
5
6   IdentifierInfo &CII = C.Idents.get(Name);
7   for (const auto &Result : DC->lookup(&CII))
8     if (const auto FD = dyn_cast<FunctionDecl>(Result))
9       return FD;
10
11  if (!C.getLangOpts().CPlusPlus)
12    return nullptr;
13
14  // Demangle the premangled name from getTerminateFn()
15  IdentifierInfo &CXXII =
16      (Name == "_ZSt9terminatev" || Name == "\01?terminate@@YAXXZ")
17          ? C.Idents.get("terminate")
18          : C.Idents.get(Name);
19
20  for (const auto &N : {"__cxxabiv1", "std"}) {
21    IdentifierInfo &NS = C.Idents.get(N);
22    for (const auto &Result : DC->lookup(&NS)) {
23      NamespaceDecl *ND = dyn_cast<NamespaceDecl>(Result);
24      if (auto LSD = dyn_cast<LinkageSpecDecl>(Result))
25        for (const auto &Result : LSD->lookup(&NS))
26          if ((ND = dyn_cast<NamespaceDecl>(Result)))
27            break;
28
29      if (ND)
30        for (const auto &Result : ND->lookup(&CXXII))
31          if (const auto *FD = dyn_cast<FunctionDecl>(Result))
32            return FD;
33    }
34  }
35  return nullptr;
36 }
```

What this code in Figure A.16 does is, first assuming that we are going to look up for an identifier that has a type. It will look for the identifier name in the ASTContext identifiers table (symbols table) to

look whether it is there or not. If it finds it with that name, it means we were right and it returns the cast to the function declaration. In any other case, it might happen that we are in C++ mode. In C++, a process is performed to functions so that artificial calls cannot be created easily. This process is called name mangling. It consists of adding certain characters at the beginning of the function. With that in mind, what we will do is trying to find the proper arguments for that function. Another option that might be happening is that we are looking for a function within a namespace. Namespaces are a mechanism that C++ provides and that allows the declaration of functions with the same name in different namespaces. For that purpose, the namespace identifier is also considered together with the function name. In this step, the function will be looking inside each namespace for the function name. If it is found there it will be returned, otherwise, the function will return a *nullptr* which is something invalid.

Listing A.17: *Synthesizing Call Expression from Function Declaration*

```
1  CallExpr *CodeGenModule::SynthesizeCallToFunctionDecl(ASTContext *Context,
2                            FunctionDecl *FD, ArrayRef<Expr *> Args,
3                            SourceLocation Loc) {
4    //Type of what we are searching.
5    QualType _Type = FD->getType();
6    //Get Function Type of the Function Declaration
7    const FunctionType *FT = _Type->getAs<FunctionType>();
8    //Call Expression that we are going to generate
9    CallExpr *CE;
10   //We pick this branch if we are in a C++ Class Member Method
11   if (auto *MD = dyn_cast<CXXMethodDecl>(FD)) {
12     CXXThisExpr *TE = new (Context) CXXThisExpr(Loc, MD->getThisType(*Context),
13                                                 false);
14     MemberExpr *ME = new (Context) MemberExpr(TE, true, Loc, FD,
15                                               FD->getNameInfo(), _Type,
16                                               VK_RValue, OK_Ordinary);
17     CE = new (Context) CXXMemberCallExpr(*Context, ME, Args,
18                                          FT->getCallResultType(*Context),
19                                          VK_RValue, Loc);
20   } else { //We pick this branch if we are not dealing with C++.
21     DeclRefExpr *DRE = new (Context) DeclRefExpr(FD, false, _Type,
22                                                  VK_LValue, SourceLocation());
23     QualType pToFunc = Context->getPointerType(_Type);
24     ImplicitCastExpr *ICE =
```

```
25        ImplicitCastExpr::Create(*Context, pToFunc, CK_FunctionToPointerDecay,
26                                 DRE, nullptr, VK_RValue);
27      CE = new (Context) CallExpr(*Context, ICE, Args,
28                                  FT->getCallResultType(*Context),
29                                  VK_RValue, Loc);
30    }
31    return CE;
32  }
```

With regard to this second function represented in Listing A.17, it generates a *CallExpr* object from a function declaration. In Clang class hierarchy, *FunctionDecl* objects are in the upper hierarchy and cover *CXXMethodDecl* and the rest of functions. This implies that we have to differentiate that. In this case, what the function does is creating and extracting from the function the different features that are necessary to create a *CallExpr*. It verifies differently whether it is a C++ method (*CXXMethodDecl*) or a normal function declaration (*FunctionDecl*). Finally, if everything goes fine, it returns the *CallExpr*, which will be translated to that.

However, the functionality does not end here. We needed to pass some arguments to that function. And related to that, several decisions had to be made. The first decision regards the fact of having to include the header file with *#include <contract>*. As a first approach, the idea was to modify the compiler so that the *Preprocessor* took automatically this file and generated it if we found any contract. However, after a lot of time trying to replicate the behaviour and forcing the *Preprocessor* to parse that file, the approach was discarded. The main problem underlying was that *CodeGen* generates the code by *TopLevelDeclarations*, and we were not able to generate the implementation of the code. What we need to do is artificially generating an object of the class *contract_violation* that we already showed, filling the different fields, and passing it as a parameter. First, let us take a look at what is the new equivalence in the code of a contract violation.

Listing A.18: *assert attribute code*

```
#include <contract>
int main(int argc, char* argv[]){
    int x = -1;
    //This will execute the handler.
    [[assert: x > 0]];
    return 0;
}
```

Listing A.19: *assert attribute equivalence*

```

void contract_violation_handler
(contract_violation &v) const{
    //Do whatever we want.
}

int main(){
    int x = -1;
    if (!(x > 0)){
        //We initialize a contract violation
        // with the information of the file.
        contract_violation v{4, "main",
                             "test.cpp",
                             "x > 0",
                             "default"};
        //We pass the object as argument.
        contract_violation_handler(v);
        //We invoke to std::terminate to abort.
        std::terminate();
        }
    return 0;
}
```

With the code in Listing A.18, what we observed is what we needed to do in order for our function to work properly and according to the specification. The solution to implement this implied two main design decisions. The first of them was modifying the *contract_violation* class and including inner attributes inside that allowed to store the information. Apart the creation of a constructor that received pointers to basic types was needed. The main problem with that is that the types used in *contract_violation* are all complex types. Those types are unknown to the compiler and finding them supposes a lot of overhead and it exists the possibility of them not existing. Therefore, we shall try as most as possible to use basic types. Luckily, it exists a *StringView* constructor which accepts a pointer to a character (*char**) which is a string in the end. With that in mind, we just needed to perform the proper conversions in order to avoid how this worked. Then the prototype class that was specified changed to the following code in Listing A.20. This was the first step on how to solve the problem.

Listing A.20: *assert attribute code*

```
namespace std {
  class contract_violation {
  private:
    int line_number_;
    experimental::string_view file_name_, function_name_, comment_, assertion_level_;
  public:
    contract_violation(int line_number,
                   const char * file_name,
                   const char * function_name,
                   const char * comment,
                   const char * assertion_level):
               line_number_{line_number},
               file_name_{file_name},
               function_name_{function_name },
               comment_{comment},
               assertion_level_{assertion_level}
           {};
    int line_number() const noexcept{
        return line_number_;
    };
    experimental::string_view file_name() const noexcept{
        return file_name_;
    };
    experimental::string_view function_name() const noexcept{
        return function_name_;
    };
    experimental::string_view comment() const noexcept{
        return comment_;
    };
    experimental::string_view assertion_level() const noexcept{
        return assertion_level_;
    };
  };
}
```

With that, we already had something from which we were able to start. The first step was to support the creation of a *contract_violation* object is generating a constructor for that class. Since it is complicated to guess which nodes of the AST have to be created, the first step was to compile to generate the AST dump of that code and the objective is trying to replicate it with as much fidelity as possible. The AST dump of the code we are interested in replicating is the relative to a declaration statement. Figure A-3 shows the AST dump of what we needed to generate.

```
-DeclStmt 0x17fc0f8 <line:35:2, col:51>
 `-VarDecl 0x17fbeb0 <col:2, col:50> col:26 used cv 'std::contract_violation':'class std::contract_violation' callinit
   `-CXXConstructExpr 0x17fc0a0 <col:26, col:50> 'std::contract_violation':'class std::contract_violation' 'void (int, const char *, const char *, const char *, const char *)'
     |-IntegerLiteral 0x17fbf10 <col:29> 'int' 0
     |-ImplicitCastExpr 0x17fc040 <col:32> 'const char *' <ArrayToPointerDecay>
     | `-StringLiteral 0x17fbf30 <col:32> 'const char [2]' lvalue "1"
     |-ImplicitCastExpr 0x17fc058 <col:37> 'const char *' <ArrayToPointerDecay>
     | `-StringLiteral 0x17fbf60 <col:37> 'const char [2]' lvalue "2"
     |-ImplicitCastExpr 0x17fc070 <col:42> 'const char *' <ArrayToPointerDecay>
     | `-StringLiteral 0x17fbf90 <col:42> 'const char [2]' lvalue "3"
     `-ImplicitCastExpr 0x17fc088 <col:47> 'const char *' <ArrayToPointerDecay>
       `-StringLiteral 0x17fbfc0 <col:47> 'const char [2]' lvalue "4"
```

Figure A-3: *Assertion Levels vs Build Levels*

In order to be able to properly develop this task, we have to start from the bottom of the tree and go generating upwards. The first element to be generated are the leaf nodes. For the leaf nodes, we needed to generate the string and integer literals. These literals are going to include the information of the contract violation, therefore, we previously need to be able to extract this information from the compiler. The code for the extraction of the different attributes is going to be exposed in A.21.

Listing A.21: *Information Extraction For Violation Information*

```
1  //File name extraction
2  const std::string file_name_ = CGM.getDiags().getSourceManager().getFilename(Loc).str();
3  //We extract the line number
4  std::string location_ = Loc.printToString(CGM.getDiags().getSourceManager());
5  std::size_t a_ = location_.find(":") + 1;
6  std::size_t b_ = location_.find(":", a_);
7  int line_number_ = std::stoi(location_.substr(a_,b_- a_));
8  //We extract the function name.
9  const FunctionDecl* EFD = cast<FunctionDecl>(CurFuncDecl);
10 std::string function_name_ = EFD->getDeclName().getAsString();
11 //We extract the assertion level.
12 std::string assertion_level_ = _Attr->getLevel()->getName();
13 //We extract the condition of the contract violation.
14 const char *start_ =
15     CGM.getDiags().getSourceManager().getCharacterData(_Attr->getCond()->getLocStart());
16 const char *end_ =
17     CGM.getDiags().getSourceManager().getCharacterData(_Attr->getCond()->getLocEnd());
```

```
18 std::string comment_(start_, (end_ - start_) + 1);
```

As we can see in the code we have to perform several computations in order to be able to generate the conditions that we need in order to give the information to the contract violation. The easiest element to extract is the file name. The file name is contained in the *SourceManager*. If we remember, we mentioned it within the preprocessor. It is one of the key elements in charge of performing an abstraction from the file system. We use it to obtain the file name associated with a source location. The *SourceLocation* object is in charge of storing the locations of each of the elements in the AST.

With regard to the line number, it can be extracted from the source location. However, it has a strange format that we have to be able to interpret it to extract the information we want. For that purpose, we have to reformat the string to extract it specifically. The functions find and the generation of a string is used in that case to extract the information.

The function name can be extracted from *CodeGenFunction*. Inside these objects, there is a record that holds the function declaration object that is being emitted. That object can be cast to a *FunctionDecl* object and then it can be used for the extraction of the function name.

With regard to the condition, the strategy used to extract the condition is to obtain the data from the source buffer. The source buffer works as a big string of which pointers are created on each of the AST elements. In this case, two pointers are obtained which point to the beginning and the end. With those two pointers a string can be created and this string will be holding the condition of the contract. This process named here is applicable to any contract clause, including *assert*, *expects* and *ensures*. With this step all the data needed for the creation of the literals are ready. The creation of the literals is the next step. The code in Listing A.22 represents how a literal is created.

Listing A.22: *Literals generation*

```
1  //String literal for the line number.
2  llvm::APInt IL1_NUM = llvm::APInt(32,line_number_, false);
3  IntegerLiteral* IL1_line_number = IntegerLiteral::Create(Ctx, IL1_NUM,
4                                                           Ctx.IntTy, Loc);
5  //String literal for the file name
6  llvm::APInt SL1_QT_EXPR_NUM =
7      llvm::APInt(32, file_name_.length() + 1/*String size*/, false);
8  QualType SL1_QT = Ctx.getConstantArrayType(Ctx.getConstType(Ctx.CharTy),
9                                             SL1_QT_EXPR_NUM,
10                                            ArrayType::ArraySizeModifier::Normal,
```

```
11                                          0);
12 StringLiteral* SL1_file_name =
13     clang::StringLiteral::Create (Ctx, file_name_,
14                                   clang::StringLiteral::StringKind::Ascii,
15                                   false, SL1_QT, Loc);
16 //String literal for the function name
17 llvm::APInt SL2_QT_EXPR_NUM = llvm::APInt(32,
18                                function_name_.length() + 1/*String size*/,
19                                false);
20 QualType SL2_QT =
21     Ctx.getConstantArrayType(Ctx.getConstType(Ctx.CharTy),
22                              SL2_QT_EXPR_NUM,
23                              ArrayType::ArraySizeModifier::Normal,
24                              0);
25 StringLiteral* SL2_function_name =
26     clang::StringLiteral::Create (Ctx, function_name_,
27                                   clang::StringLiteral::StringKind::Ascii,
28                                   false, SL2_QT, Loc);
29 //String literal for the commentm in this case the condition
30 llvm::APInt SL3_QT_EXPR_NUM =
31     llvm::APInt(32, comment_.length() + 1/*String size*/, false);
32 QualType SL3_QT =
33     Ctx.getConstantArrayType(Ctx.getConstType(Ctx.CharTy),
34                              SL3_QT_EXPR_NUM,
35                              ArrayType::ArraySizeModifier::Normal, 0);
36 StringLiteral* SL3_comment =
37     clang::StringLiteral::Create (Ctx, comment_,
38                                   clang::StringLiteral::StringKind::Ascii,
39                                   false, SL3_QT, Loc);
40 //String literal for the assertion level of the contract
41 llvm::APInt SL4_QT_EXPR_NUM =
42     llvm::APInt(32,
43                 assertion_level_.length() + 1/*String size*/,
44                 false);
45 QualType SL4_QT =
46     Ctx.getConstantArrayType(Ctx.getConstType(Ctx.CharTy),
```

```
47                          SL4_QT_EXPR_NUM,
48                          ArrayType::ArraySizeModifier::Normal, 0);
49 StringLiteral* SL4_assertion_level =
50    clang::StringLiteral::Create (Ctx, assertion_level_,
51                                  clang::StringLiteral::StringKind::Ascii,
52                                  false, SL4_QT, Loc);
```

For the creation of an integer literal needed for the line number only one element is needed and it is an APInt with storing the number that it has inside. The *APInt* is a LLVM object which represents a integer of any size. In our case it is initialized to 32 bits since it is the default size of an integer on many OS and it is enough for this representation. The integer literal receives the *ASTContext* where it will be created, the integer that we just created, the type of the literal and the source location that we fake using the same as in the assert attribute.

In the case of the strings, it becomes more difficult because of the internal representation of strings in memory of C like programs. The string is represented as an array of characters and it is what it is needed to represent. For each of the strings, we will also need an *APInt* object that will store the size of the array. We will use the strings with the values to extract the length. Apart from that a *QualType* is needed. A *QualType* is the object used to represent any type in Clang. The *ASTContext* implements several functions in order to interact with basic types and provides the possibility of creating complex types as well as modifications of the former. A *QualType* associated with an array type will be created for a character array representation. Since the strings will not have modifications in their size it is enough with a constant sized array. The *ASTContext* function, *getConstantArrayType* is a good alternative to obtain this type and is the one used in this project. With this type, the string literal can be generated. For the creation of the string literal, we will also need the *ASTContext*, the string value that will use, the string encoding that in this case is ASCII, the QualType that was generated and a source location that is again faked assuming the assert contract value. The false element is used for a Pascal String Literal representation since it is not interesting it is marked as false.

Internally the compiler transforms the string literals to arrays of characters of type *const char**. The reason for that is passing the argument that the function requires. Since this behaviour shall be replicated the implicit cast expressions are created. The implicit cast expressions need of the output type. For that reason, the first task is the artificial generation of the *const char** type. The code in Listing A.23 represents the generation of it.

Listing A.23: *const char * type generation*

```
1 QualType IC_QT = Ctx.getPointerType(Ctx.getConstType(Ctx.CharTy));
```

Now it just needs the generation of the different *ImplicitCastExpr*. This objects receive as parameters the *ASTContext* that is used as allocator, the *QualType* to which the conversion is made, then a type of conversion is used (array to pointer conversion), the string literal and what is the consideration of the literal being passed. The consideration of a literal can be two, either RValue or LValue. This is a consideration that C++ introduced with movement semantics. In this case it implies that the value is not extracted from a variable and therefore it can be passed straightly. With regard to the *nullptr* it refers to calls to C++ member functions that in this case is not interested in and is not useful.

Listing A.24: *const char * type generation*

```
1 ImplicitCastExpr* IC1= ImplicitCastExpr::Create(Ctx, IC_QT,
2   CK_ArrayToPointerDecay, SL1_file_name, nullptr, VK_RValue);
3 ImplicitCastExpr* IC2= ImplicitCastExpr::Create(Ctx, IC_QT,
4   CK_ArrayToPointerDecay, SL2_function_name, nullptr, VK_RValue);
5 ImplicitCastExpr* IC3= ImplicitCastExpr::Create(Ctx, IC_QT,
6   CK_ArrayToPointerDecay, SL3_comment, nullptr, VK_RValue);
7 ImplicitCastExpr* IC4= ImplicitCastExpr::Create(Ctx, IC_QT,
8   CK_ArrayToPointerDecay, SL4_assertion_level, nullptr, VK_RValue);
```

Only the generation of the call to the constructor is remaining in now. However, that is one of the most difficult steps. The constructor generation requires from the class and that is something that is not obtainable directly. For that, the extraction needs to be performed from the identifiers table. In this step, a generic and very powerful function is created. This function is able to obtain from the identifiers table (symbols table) any element with a name given its type. It is important to match both the name and the type since it is what the function will try to match. Apart from that, the function allows looking first either to the std namespace or to other namespaces first. The reason for that is that the first alternative included a default violation handler inside the *std* namespace, and in order to avoid any user overwriting the handler the search over std was first performed. The generic function is listed in Listing A.25.

Listing A.25: *Generic function for any type extraction*

```
1 template<typename T>
2 T* getDecl(ASTContext& C, StringRef Name, bool SearchOnStd) {
3   T* invalid_ret = nullptr;
4   TranslationUnitDecl *TUDecl = C.getTranslationUnitDecl();
5   DeclContext *DC = TranslationUnitDecl::castToDeclContext(TUDecl);
6   auto lambda1 = [&](){
7     IdentifierInfo &CII = C.Idents.get(Name);
8     for (const auto &Result : DC->lookup(&CII))
9       if (const auto TD = dyn_cast<T>(Result))
```

```
10          return TD;
11        if (!C.getLangOpts().CPlusPlus)
12          return invalid_ret;
13        return invalid_ret;
14    };
15   auto lambda2 = [&](){
16      IdentifierInfo &CXXII =
17        (Name == "_ZSt9terminatev" || Name == "\01?terminate@@YAXXZ")
18        ? C.Idents.get("terminate")
19        : C.Idents.get(Name);
20        for (const auto &N : {"__cxxabiv1", "std"}) {
21          IdentifierInfo &NS = C.Idents.get(N);
22          for (const auto &Result : DC->lookup(&NS)) {
23            NamespaceDecl *ND = dyn_cast<NamespaceDecl>(Result);
24            if (auto LSD = dyn_cast<LinkageSpecDecl>(Result))
25              for (const auto &Result : LSD->lookup(&NS))
26                if ((ND = dyn_cast<NamespaceDecl>(Result)))
27                  break;
28                if (ND)
29                  for (const auto &Result : ND->lookup(&CXXII))
30                    if (const auto TD = dyn_cast<T>(Result))
31                      return TD;
32                  }
33                }
34                return invalid_ret;
35      };
36      if(SearchOnStd) return lambda2();
37      else return (lambda1() != nullptr)? lambda1() : lambda2();
38  }
```

The code in Listing A.25 shows the execution is divided into two lambda functions, this allows inverting the orders of both functions easily. As we see in the last line the steps carried out are returning in different orders depending on the function. With regard to the first lambda, it takes the identifier by the name and checks whether it casts to the type of the function created. Since the type of the function must be given the function will be working for any type that the user wants to search. The second lambda will be doing the same but within the different namespaces in order to be able to find an equally

named function within different namespaces. The important element that is needed for the project is the contract violation class declaration. With the class declaration, we are able to extract some elements that are contained in it such as the constructor, something that is directly needed to create an instance of it. The obtention of the class declaration is reduced to the code below.

Listing A.26: *Contract violation extraction*

```
1 ASTContext& Ctx = getContext();
2 auto CRD = getDecl<CXXRecordDecl>(Ctx, "contract_violation",false);
```

With this code, the program looks for a pointer to a *CXXRecordDecl* that is called *"contract_violation"* within all namespaces. A *CXXRecordDecl* is the class used to represent a C++ class declaration in Clang. A class is sought since having the class gives the ability of obtaining the constructor, and also the *QualType* of the class, which is the type of objects it generates. However, if the search would have looked for the constructor, there would not be a way to obtain the *QualType* since the return type of the constructors is *void*.

Listing A.27: *Finding the Constructor in the CXXRecordDecl*

```
1  //Extraction of the constructor for the std::contract_violation class
2  CXXConstructorDecl* CCD;
3  //Finding the correct constructor in our case
4  for(auto C = CRD->ctor_begin(); C != CRD->ctor_end(); C++){
5      auto P = cast<CXXConstructorDecl>(*C);
6      //Number of initializers is bigger than 1 and is not a copy or move constructor
7      if(P->getNumCtorInitializers() >1 && !P->isCopyOrMoveConstructor()){
8              CCD = P;
9       }
10 }
```

The code in Listing A.27 looks along the members of the class for the constructor. However, something has to be noticed and it is that C++ does have class constructors, copy constructors and movement constructors. And those are generated at compile time if nothing is specified. In such case, the code has to be able to differentiate which of them it is. What is used is the fact that the constructor that is represented in the class definition has to receive more than an argument. That is the cause of the first condition and the second condition is related to eliminating the possibility of it being a copy or move constructor. As it is demonstrated, it is more useful to look for the constructor from the class because it allows the acquisition of more information with fewer computations.

Next, the instance of the class has to be created and it has to be assigned to a variable. The steps to

be performed are first the generation of a constructor and then the assignation of the constructor to the initializer of a variable declaration. For the generation of the constructor, as it was already mentioned, the main elements needed are the type of the class and the constructor declaration. As it is observed in the AST frame, a *CXXRecordExpr* has to be generated. The expression is the equivalent the use of the *CXXConstructorDecl*. The code below shows the elements needed for the creation of the former.

Listing A.28: *Generating a Class Instance*

```
1 CXXConstructExpr* CCE1 = CXXConstructExpr::Create(Ctx, Ctx.getRecordType(CRD),Loc,CCD,
2               false, {IL1_line_number, IC1, IC2, IC3, IC4, }, true, true, false, false,
3               CXXConstructExpr::ConstructionKind::CK_Complete, _Attr->getRange());
```

As we see, many elements are needed for the creation of the constructor call. In this case, they will be exposed with detail since they provide a lot of information to what the overall meaning of the expression and it is important for future steps.

1. The **ASTContext** acts as the allocator for the memory.
2. **QualType** determines the type of the constructor. It is obtained from the CXXRecordDecl that was extracted on the first step.
3. The **SourceLocation** is assumed to be the same as the *assert attribute*.
4. **CXXConstructorDecl** it is obtained in the loop described above.
5. **Elidable**. The elidability refers to the ability to be compared. In this case, it is established to false.
6. **ArrayRef< Expr *>**. The list of arguments is passed to the constructor. The arguments are the implicit cast expressions created during a previous step.
7. **HadMultipleCandidates**. It establishes whether there are multiple candidates to construct a class of this type. It is true since the default constructor might be automatically generated and there are copy and move constructors.
8. **ListInitialization**. It establishes whether the initialization is done with a list. It is true in this case.
9. **StdInitListInitialization**. It establishes whether the initialization is done with a standard initialization list. It is false in this case.
10. **ZeroInitialization**. It establishes whether there is a zero-argument initialization. It is not interesting so it is established to false.
11. The **ConstructionKind** determines the elements that are going to be generated. In this case, all arguments are initialized so it is established to a complete initialization.

12. **SourceRange**. As with the source location it is obtained from the *assert attribute*.

What comes next is the creation of the variable. The variable will hold this value and will be created in the memory of the program. For its creation a *VarDecl* object is needed. And the use of its member method *setInit* will allow to establish the constructor as the initializer of the function.

Listing A.29: *Creating the Variable and associating the Class to it*

```
1  VarDecl* VD = VarDecl::Create (Ctx, IIDC, Loc, Loc, &II,
2                    Ctx.getRecordType(CRD), Ctx.getTrivialTypeSourceInfo(Ctx.getRecordType(CRD)),
3                    clang::StorageClass::SC_None);
4  //We make the var decl implicit since it has been declared outside the code of the user
5  VD->setImplicit();
6  //We establish the initialization style as a list initialization.
7  VD->setInitStyle(clang::VarDecl::InitializationStyle::ListInit);
8  //We establish which is going to be the Init of the constructor expression.
9  VD->setInit(CCE1);
10 //We establish which is the Declaration context of the VarDecl a second time.
11 VD->setLexicalDeclContext(IIDC);
12 //We make the declaration to be added to the declaration context
13 IIDC->addHiddenDecl(VD);
14 //We mark the variable declaration as used.
15 VD->markUsed(Ctx);
```

This code in Listing A.29 refers to several characteristics that allow the compiler not to crash when dealing with this new variable. An important fact is that it shall exist, but the compiler shall not take it into account. That is the main reason to declare it implicit and reset the lexical context because it is important to be taken into account only there. Apart from that, the variable is declared as used so that it is not automatically removed and the user is not warned.

To conclude with the declaration of the variable a declaration statement has to be generated to include all the elements. The declaration statement uses a generic declaration and for this purpose the *VarDecl* object has to be transformed to a generic *Decl* reference. With that, the declaration statement would be ready. At this point the code is similar to the one explained in previous sections in which the function call was generated from the name and an *if statement* was generated. There are two main modifications to that code. The default function call to *std::terminate()* is replaced to be a call to the function that is chosen by the user. This function call will receive the argument with the information on the violation. Another flag will allow switching between aborting on a violation or not. Depending on that the call to abort is generated or not. Anyways a compound statement is generated to contain all the information, and that

compound statement is used as the "then" condition of the if statement that evaluates the condition of the contract. By default, the violation handler will be aborting as it was stated on the requirements.

Listing A.30: *Generation of the If Stmt*

```
DeclStmt* DS = new (Ctx) DeclStmt(DeclGroupRef::Create(Ctx,DA, 1),
               _Attr->getRange().getBegin(),_Attr->getRange().getEnd());
//We create a Reference Expression to a Declaration in case that we need
DeclRefExpr* Arg = new (Ctx) DeclRefExpr(cast<ValueDecl>(VD),
                             false, Ctx.getRecordType(CRD),
                             clang::ExprValueKind::VK_LValue, Loc);
auto FD = (!isDefault)?
        const_cast<FunctionDecl *>(CGM.GetRuntimeFunctionDecl(Ctx, HandlerName)) :
        const_cast<FunctionDecl *>(getDecl<FunctionDecl>(Ctx, "default_handler", true)) ;
//Call Expression to the Function Declaration obtained
CallExpr *CE = CGM.SynthesizeCallToFunctionDecl(&Ctx, FD, {Arg});
CompoundStmt* CS;
if(abort){
    auto FD1 = const_cast<FunctionDecl *>(CGM.GetRuntimeFunctionDecl(getContext(),
    "abort"));
    CallExpr *CE1 = CGM.SynthesizeCallToFunctionDecl(&getContext(), FD1, {});
    CS = new (Ctx) CompoundStmt(Ctx, {DS, CE, CE1}, Loc, Loc);
}
else
    CS = new (Ctx) CompoundStmt(Ctx, {DS, CE}, Loc, Loc);
// negate expression
ParenExpr *PE = new (Ctx) ParenExpr(
  SourceLocation(), SourceLocation(), _Attr->getCond());
UnaryOperator *UO = new (Ctx) UnaryOperator(
  PE, UO_LNot, PE->getType(), VK_RValue, OK_Ordinary,
  SourceLocation());
//We create the If Stmt for the class.
auto _S = new (getContext()) IfStmt(getContext(),
 Loc, false, nullptr, nullptr, UO, CS);
EmitIfStmt(*_S);
```

A.3.4 Continuation Mode and Custom Handler Flags

In the last fragment of code, two variables were used to generate the custom handler and switching between aborting or not. The value of this variables comes from some compilation flags defined by the user. In this section, we will be describing the main changes needed to perform this changes. The two flags will enable continuation and will allow the customization of the function handler. There are few changes from the steps carried out to create the build level flag.

The first step consists on creating a entry so that the Driver recognises the new flags and allows us to manage them. The structure that the violation handler flag is going to have is *–contract-violation-handler=<functionname>* and the structure that the abort flag is going to have is *–enable-continue-after-violation* being activated only when we want the continuation mode. If we remember the entry was included in *Options.td* file.

Listing A.31: *Flag Declaration*

```
def contract_violation_handler_EQ : Joined<["-", "--"], "contract-violation-handler=">,
    Flags<[CC1Option, CC1AsOption, CoreOption]>,
    HelpText<"Name of the handler function to be called if a contract is violated">;
def enable_continue_after_violation : Joined<["-", "--"], "enable-continue-after-violation">,
    Flags<[CC1Option, CC1AsOption, CoreOption]>,
    HelpText<"Enable continuation after violation of a contract">;
```

Right after that, we need to define the movement from the Driver argument list to the toolchain argument list. This movement is done in Clang.cpp.

Listing A.32: *Movement of Flags*

```
if (Arg *A = Args.getLastArg(options::OPT_contract_violation_handler_EQ))
    A->render(Args, CmdArgs);
if (Arg *A = Args.getLastArg(options::OPT_enable_continue_after_violation))
    A->render(Args, CmdArgs);
```

This code is going to move the flags from the Driver list to the Toolchain list. In this way, we will be able to operate with it in the CompilerInvocation.cpp. The violation handler flag requires a different kind of language option. As it was seen with the build level flag, to determine a value in the *LangOptions.def*, a number of bits shall be determined. However, a string has an undetermined length. In order to solve this problem, the definition of the LangOptions object was modified. In it, a string was added which would hold the value of the violation handler instead of using the usual way of obtaining the value. With regard to the continuation mode flag, it can be represented with one bit and it is enough for this purpose.

Listing A.33: *Lang Option definition in LangOptions class*

```
/// \brief The name of the handler function to be called if a contract
/// is violated (C++ D0542R2).
///
/// If none is specified, std::terminate()
std::string ContractViolationHandler;
```

Listing A.34: *Lang Options flag definition*

```
VALUE_LANGOPT(EnableContinueAfterViolation , 1,
              0, "Set the violation continuation mode to on (D0542R2)")
```

Finally, with the code in the CompilerInvocation.cpp, we manage those flags and transfer it to the final destination from which they are later received in the CodeGen.

Listing A.35: *Build Level handling in CompilerInvocation.cpp*

```
 static void ParseLangArgs(LangOptions &Opts, ArgList &Args, InputKind IK,
                           const TargetOptions &TargetOpts,
                           PreprocessorOptions &PPOpts,
                           DiagnosticsEngine &Diags) {
...
// Handle -contract-violation-handler= option.
if (Arg *A = Args.getLastArg(OPT_contract_violation_handler_EQ))
    Opts.ContractViolationHandler = A->getValue();
// Handle -enable-continue-after-violation option.
if (Arg *A = Args.getLastArg(OPT_enable_continue_after_violation))
    Opts.EnableContinueAfterViolation = 1;
 ...
```

With this, all the implementation remains complete. This section has described in depth all the implementation details of the contracts in C++, emphasizing the section that I have developed.

Bibliography

[1] Eiffel team, "Two-Minute fact sheet." https://www.eiffel.org/doc/eiffel/Two-Minute%20fact%20sheet. Last visited 27th April 2018, 2018.

[2] Eiffel Software team, "Building bug-free O-O software: An Introduction to Design by Contractt." https://www.eiffel.com/values/design-by-contract/introduction/. Last visited 27th April 2018, 2018.

[3] Ada Information Clearing House, "Ada Advantages - The time-tested, safe and secure programming language." http://www.adaic.org/advantages/. Last visited 3rd May 2018, 2018.

[4] Ada - Europe, "Ada Reference Manual - Introduction." http://www.adaic.org/resources/add_content/standards/05rm/html/RM-0-3.html. Last visited 3rd May 2018, 2018.

[5] ISO/IEC team, "ISO/IEC 8652:2012 Information technology – Programming languages – Ada." https://www.iso.org/standard/61507.html. Last visited 19th May 2018, 2018.

[6] Ada team, "Ada Benefits and Features." https://www.adacore.com/about-ada/benefits-and-features. Last visited 3rd May 2018, 2018.

[7] Margaret Rouse, "Structured programming (modular programming)." https://searchsoftwarequality.techtarget.com/definition/structured-programming-modular-programming. Last visited 3rd May 2018, 2018.

[8] C. Hope, "Imperative programming."

[9] Edaqa Mortoray, "What is imperative programming?." https://mortoray.com/2017/05/10/what-is-imperative-programming/. Last visited 3rd May 2018, 2018.

[10] F. Bauer, M. Broy, W. Dosch, R. Gnatz, B. Krieg-Brückner, A. Laut, M. Luckmann, T. Matzner, B. Möller, H. Partsch, P. Pepper, K. Samelson, R. Steinbrüggen, M. Wirsing, and H. Wössner, "Programming in a wide spectrum language: a collection of examples," *Science of Computer Programming*, vol. 1, no. 1, pp. 73 – 114, 1981.

[11] Ada Team, "About Ada." https://www.adacore.com/about-ada. Last visited 7th May 2018, May 2018.

[12] Jacob Sparre Andersen, "Contract Based Programming in Ada 2012." http://www.jacob-sparre.dk/reliability/Contract-based-programming-2014.pdf. Last visited 7th May 2018, 2014.

[13] Ada Team, "Subtype Predicates." http://www.ada-auth.org/standards/12rm/html/RM-3-2-4.html. Last visited 8th May 2018, 2014.

[14] Community, "What are the main weaknesses of Ada as a programming language." https://www.quora.com/What-are-the-main-weaknesses-of-Ada-as-a-programming-language. Last visited 19th May 2018, 2016.

[15] Jack Ganssle, "Why aren't developers interested in Ada?." https://www.embedded.com/electronics-blogs/break-points/4008214/Why-aren-t-developers-interested-in-Ada-. Last visited 19th May 2018, 2009.

[16] Spark Team, "About Spark." http://www.spark-2014.org/about. Last visited 7th May 2018, May 2018.

[17] J. Barnes, *High Integrity Software - The Spark Approach to Safety and Security*. Pearson Education Limited, 2003.

[18] Martin Becker, "Ada/Spark 2014 - Mini Cheat Sheet." https://github.com/mbeckersys/spark2014-cheat-sheet/blob/master/cheatsheet.pdf. Last visited 19th May 2018, 2017.

[19] Benjamin M. Brosgol, AdaCore, "Developing secure code using SPARK: Part 2 – How to use it." https://www.embedded.com/design/safety-and-security/4419247/Developing-secure-code-using-SPARK--Part-2---How-to-use-it. Last visited 19th May 2018, 2013.

[20] Claire Dross, "Contracts of Functions in SPARK 2014." http://www.spark-2014.org/entries/detail/contracts-of-functions-in-spark-2014. Last visited 19th May 2018, 2015.

[21] Claire Dross, "Contracts of Functions in SPARK 2014." https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573(v=vs.85).aspx. Last visited 19th May 2018, 2015.

[22] Maira Wenzel, Mike Jones, Erik Lydersen, Luke Latham, Tom Pratt, "Code Contracts." https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts. Last visited 19th May 2018, 2015.

[23] Sunil Mohan, ".NET and Its Advantages / Disadvantages - All You Need to Know," 2016.

[24] D Team, "Overview." https://dlang.org/overview.html. Last visited 24th May 2018, May 2018.

[25] D Team, "Major Features of D." https://dlang.org/overview.html#major_featuresl. Last visited 24th May 2018, May 2018.

[26] D Team, "Contract Programming." https://dlang.org/spec/contracts.html. Last visited 24th May 2018, May 2018.

[27] Shachar Shemesh, "What are the advantages of using D language?." https://www.quora.com/What-are-the-advantages-of-using-D-language. Last visited 24th May 2018, May 2018.

[28] Laeeth Isharc, "Did D fail to become "the better C++"? If so, why? If not how is it better? What's the experience of people replacing C++ with D?." https://www.quora.com/Did-D-fail-to-become-the-better-C-\If-so-why-If-not-how-is-it-better-Whats-the-experience-of-people-replacing-C-with-D. Last visited 24th May 2018, May 2018.

[29] Mike Blome, Gordon Hogenson, Colin Robertson, Mike Jones, Genevieve Warren, Saisang Cai , "Understanding SAL." https://docs.microsoft.com/en-us/visualstudio/code-quality/understanding-sal?view=xamarinmac-3.0. Last visited 24th May 2018, May 2018.

[30] Mike Blome, Gordon Hogenson, Mike Jones, Genevieve Warren, Saisang Cai , "Annotating Function Behavior." https://docs.microsoft.com/en-us/visualstudio/code-quality/annotating-function-behavior?view=xamarinmac-3.0. Last visited 24th May 2018, May 2018.

[31] Mike Blome, Gordon Hogenson, Colin Robertson, Mike Jones, Genevieve Warren, Saisang Cai , "Annotating Structs and Classes." https://docs.microsoft.com/en-us/visualstudio/code-quality/annotating-structs-and-classes?view=xamarinmac-3.0. Last visited 24th May 2018, May 2018.

[32] Eli Bendersky, "The context sensitivity of C's grammar." https://eli.thegreenplace.net/2007/11/24/the-context-sensitivity-of-cs-grammar/. Last visited 29th May 2018, 2007.

[33] Eli Bendersky, "How Clang handles the type / variable name ambiguity of C/C++." https://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc/. Last visited 29th May 2018, 2012.

[34] Gloria Inés Alvarez V., "Compiladores Análisis Semántico." http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:compi:comp_sesion18_2008-1.pdf. Last visited 29th May 2018, May 2018.

[35] Tutorials Point Team, "Compiler Design - Semantic Analysis." https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm. Last visited 29th May 2018, May 2018.

[36] Tutorials Point Team, "Compiler Design - Code Generation," May 2018.

[37] Jason Merril, "GENERIC and GIMPLE: A New Tree Representation for Entire Functions," *Red Hat, Inc.*

[38] The Clang Team, "Clang CFE Internals Manual." https://clang.llvm.org/docs/InternalsManual.html. Last visited 5th June 2018, 2018.

[39] The Clang Team, "Driver Design & Internals." https://clang.llvm.org/docs/DriverInternals.html. Last visited 5th June 2018, 2018.

[40] The Clang Team, "Clang vs Other Open Source Compilers." https://clang.llvm.org/comparison.html. Last visited 5th June 2018, 2018.

[41] Michael Larabel, "LLVM/Clang 3.2 Compiler competing with GCC." https://www.phoronix.com/scan.php?page=article&item=llvm_clang32_final. Last visited 3rd June 2018, 2012.

[42] V. Makarov, "SPEC2000: Comparison of LLVM-2.9 and GCC4.6.1 on x86_64." https://vmakarov.fedorapeople.org/spec/2011/llvmgcc64.html. Last visited 3rd June 2018, 2012.

[43] Michael Larabel, "GCC 8 vs. LLVM Clang 6 Performance At End Of Year 2017." https://www.phoronix.com/scan.php?page=article&item=gcc-clang-eoy2017&num=1. Last visited 3rd June 2018, 2017.

[44] Colfax Team, "A Performance-Based Comparison of C/C++ Compilers." https://colfaxresearch.com/compiler-comparison/. Last visited 3rd June 2018, 2017.

[45] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup, "A Contract Design." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0380r1.pdf. Last visited 18th June 2018, 2018.

[46] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup, "Support for contract based programming in C++." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0542r1.html. Last visited 18th June 2018, 2018.

[47] LLVM Team, "lit - LLVM Integrated Tester." https://llvm.org/docs/CommandGuide/lit.html. Last visited 18th June 2018, 2018.

[48] Spanish Government, "Bases y tipos de cotización 2018." http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm. Last visited 18th June 2018, 2018.

[49] Marc Gallardo, "GDPR." http://www.expansion.com/especiales/2018/GDPR/identificacion-y-documentacion.html. Last visited 13th June 2018, 2018.

[50] The Clang Team, "Getting Started: Building and Running Clang." https://clang.llvm.org/get_started.html. Last visited 5th June 2018, 2018.

[51] The Clang Team, "Attributes in Clang." https://clang.llvm.org/docs/AttributeReference.html#diagnose-if. Last visited 5th June 2018, 2018.