Arnau Barranco Jansà

Build chain for Automotive Software Applications. Optimization of the Lear SCons Builder.

TREBALL DE FI DE GRAU

dirigit per Ramón Villarino Villarino i Juan Manuel Prieto Martinez

Grau de Enginyeria de Sistemes i Serveis de Telecomunicacions

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

Acknowledgments

Firstly, I would like to thanks to Lear Corporation to brings me the opportunity to participate in the Trainee Program.

To my tutors, Juan Manuel Prieto Martinez, my company tutor, and the person who helped me technically to develop the project.

And my academic tutor, Ramon Villarino Villarino who aided me do the administrative procedures and write the report.

Acronyms

SWC → Software Component.

OOP → Objected-Oriented Programming.

Sgn → Signature.

IR → Intermediate Representation.

BCPDF → Build Chain Project Definition File.

BSO → Boston System Office.

Index

1. Introduction 8

1.1. Planification 8

1.2. Goals 10

1.3. Memory Structure 11

2. Lear 12

3. Build Chain, Python, SCons and Compilers 12

3.1. Build Chain 13

3.2. Python 15

3.3. SCons(fet) 17

3.4. Compilers 18

4. Implementation of the Generic Part of the Core 20

4.1. Definition & Design Core Build Chain V7 21

4.1.1. Conceptual/Overview Diagram 28

4.1.2. Class Diagram 29

4.1.3. Sequence Diagram(fet) 29

4.2. Implementation 30

5. Implement Virtual Builder 30

5.1. Implement Virtual Compiler 31

5.2. Builder Implementation for Virtual Compiler 32

5.2.1. Builder Initialization to adapt to SCons 33

5.3. Project Configuration to use a New Builder and Virtual Compiler (YAML) 33

5.4. Builder Test for Virtual Compiler 34

5.4.1. Test to check Compiler Call (executable) 34

5.4.2. Test to check Compiler Flags Injection and the Order. 34

5.4.3. Test to check Defines and Includes Injection. 34

6. Implement Builder to IAR Compiler 35

6.1. Implementation of Specifics Aspects for this Compiler 35

6.1.1. Builder Initialization to adapt to SCons 37

6.2. Project Configuration to use a IAR Builder and IAR compiler (YAML) 38

6.3. Builder Test for IAR Compiler 38

6.3.1. Test to check Compiler Call (executable) 38

6.3.2. Test to check Compiler Flags Injection and the Order. 38

6.3.3. Test to check Defines and Includes Injection. 39

7. Implement Builder to Tasking Compiler 39
7.1. Implement Specifics Aspects for this Compiler 40
7.1.1. Builder Initialation to adapt to SCons 41
7.2. Project Configuration to use a Tasking Builder and Tasking Compiler (YAML) 41
7.3. Builder Test for Tasking Compiler 42
7.3.1. Test to check Compiler Call (executable) 42
7.3.2. Test to check Compiler Flags Injection and the Order. 42
7.3.3. Test to check Defines and Includes Injection. 42
8. Implement Builder to Hightec Compiler 43
8.1. Implement Specifics Aspects for this Compiler 43
8.1.1. Builder Initialation to adapt to SCons 45
8.2. Project Configuration to use a Hightec Builder and Hightec Compiler (YAML) 45
8.3. Builder Test for Hightec Compiler 45
8.3.1. Test to check Compiler Call (executable) 45
8.3.2. Test to check Compiler Flags Injection and the Order. 46
8.3.3. Test to check Defines and Includes Injection. 46
9. Switch to automatize Process 46
10. Optimization Software 50
11. Test using New Versions of Python/SCons 52
12. Documentation 52
12.1. Manual Develop/Developer Guide LearSAR_v7 (Build Chain) 53
12.2. Code Documentation 53
12.2.1. Automatize Redaction with Doxygen 53
12.2.1.1. Doxygen Manual 53
12.3. Documentation with MarkDown 54
13. Business Values 54
14. Future Steps 55
15. Conclusions 56
16. Webgraphy 57

Figures Index

Figure 1. Gantt Diagram of my Project. 10
Figure 2. Lear Logo 12
Figure 3. General Build Chain Diagram. 13
Figure 4. Build Chain execution phases. 14
Figure 5. Build Chain execution tasks. 14
Figure 6. Python logo. 15
Figure 7. TIOBE Programming Community Index 16
Figure 8. SCons logo. 18
Figure 10. Initialization and Constructor of Compiler_Tool class. 23
Figure 11. Definition and Constructor of CompilerConf class (part 1). 26
Figure 12. Definition and Constructor of CompilerConf class (part 2). 26
Figure 13. SCons Conceptual Diagram. 28
Figure 14. Lear_SCons_Builder Conceptual Diagram. 29
Figure 17. Example of Virtual compilers initialization. 31
Figure 22. Debugger output to check Compiler Call (executable) in Virtual compiler. 34
Figure 23. Debugger output to check Compiler Flags Injection in Virtual compiler. 34
Figure 24. Debugger output to check Defines, Includes and flags injection. 35
Figure 25. Initialization of IAR compiler 36
Figure 26. Constructor of IAR compiler. 36
Figure 27. Method to manage license for IAR compiler. 37
Figure 32. Debugger output to check Compiler Call (executable) in IAR compiler 38
Figure 33. Debugger output to check compiler flags injection in IAR compiler. 38
Figure 34. Debugger output to check Includes injection in IAR compiler. 39
Figure 35. Debugger output to check Defines injection in IAR compiler 39
Figure 36. Inizialization of Tasking Compiler. 40
Figure 37. Constructor of Tasking Compiler. 40
Figure 43. Debugger output Compiler Call (executable) in Tasking compiler. 42
Figure 44. Debugger output to check Compiler Flags Injection in Tasking compiler. 42
Figure 45. Debugger output to check Include injection in Tasking compiler. 42
Figure 46. Debugger output to check Defines injection in Tasking Compiler. 43
Figure 47. Initialization of Hightec Compiler. 44
Figure 48. Constructor of Hightec Compiler 44
Figure 54. Debbuger output Compiler Call (executable) in Hightec compiler. 45
Figure 55. Debugger output compiler flags injection in Hightec compiler. 46
Figure 56. Debugger output to check Includes and Defines in Hightec Compiler. 46
Figure 57. Format version. 47
Figure 58. Initialization of switch_base class. 48
Figure 59. Method to manage a default builder. 48
Figure 60. Main code of Switch_Builder_engine_Dict Class. 49
Figure 62. Result of test versions. 50
Figure 63. Full code of Build_package.cmd file. 52

1. Introduction

The goal of this project is to understand the Lear SCons framework of the Lear Build Chain and analyze/implement/measure optimizations. To perform it, I will use object-oriented programming to combine all individual package in one (currently there is a package for each of the compilers). By making a single package I can separate the generic part of all compilers from the specific part. To do this I will use all the power that object-oriented programming gives us.

The project is necessary for several reasons:

- In recent times support for new compilers has had to be added.

- Minimize possible errors that may appear.

- Facilitate the maintenance service.

1.1. Planification

Firstly, I need to study Python, SCons and Build Chain process to understand my project and then fulfill it efficiently. During the project I will have to train and research to address any doubts.

Secondly, I will implement Build Chain V7. That's why I need a virtual compiler and a SCons Builder for this pseudo compiler to be able to debug and check that the call to the compiler with all the parameters is correctly formed. For this reason, I will make a software that will act as a compiler. This will have the function of collecting all the parameters that are passed to the compilation chain and generating the output file (.o) as if it were a real compiler. If it is a binary file, a text file with the corresponding extension (.o, .obj,...) will be used where all the parameters obtained will be stored, as well as the relevant data for debugging. By using this pseudo-compiler, I make the compilation time shorter thus improving testing and development time. Once completed, I proceed to the configuration, describing the phases that must be carried out, the file that must be executed... Finally, when everything is initialized, I will proceed to visualize the injection of all the variables if they are correct by debugging the code. The main objective of this part of the project is to facilitate the implementation of the SCons wrapper (Builder) to use the compiler (SCons compile Builder).

As can be expected in the design phase, a class diagram will be made to see the relationship between classes, variable inheritances... This diagram is a living document, so it is subject to change throughout its life cycle of the software.

Later, I will develop a SCons Builder to be able to handle this "virtual compiler" in such a way that I can integrate SCons and Lear's Build Chain.

I will continue, with the implementations of a Builder for a real compiler. This part is very similar to the virtual compiler, but now all the initializations and configurations will be from a real compiler (flags, defines, includes, linkers, assemblers...).

To make possible automation, I want to design a switch, so that the new core, which is still a Builder with the features of all the compilers currently supported. Depends on the YAML tag, the switch selects a different Builder compiler.

When I have already completed the automation, I will have a large part of the project finished, now like any software project the optimization phase of the code will come, debugging and seeing how to simplify it.

Finally, if time allows, I would like to consider new versions of Python and SCons, in order to try to reduce the execution times of the Build Chain and further optimize the whole process of adding scalable compilers.

The new versions of Python have optimized the execution time of scrips, reducing this time considerably.

Once the technical part is done, the documentation will be written. I will have to theoretically document elements that I use in the project, in order to make it easier for all those technicians with minimal knowledge of the subject (engineers, teachers, programmers...).

Once I have this description of the theoretical bases, I will proceed to write the Build Chain V7 user manual, with the aim of informing the procedure to be carried out when using Build Chain V7(adding compilers, modifying the configuration).

On the other hand, I will also have to create a developer manual. In order to carry out careful maintenance, in case you want to deal with code, add functions...

I also want to make documentation related to the code (Python), this documentation will not be too explanatory, but it is intended to explain functions and function variables with a small outline of each one. I wish to automate this process with the Doxygen tool. In order to facilitate the use of this tool, I have planned to make a documentation both its installation and its use.

At this point, another very standardized documentation format is MarkDown, and I want to make Build Chain V7 documentation in this format to universalize the documentation.

I would have to make a sequence diagram to make it more visually understandable how Build Chain V7 works, how it is executed, what order it follows, how many phases there are...

This is the diagram Gantt related with my project:

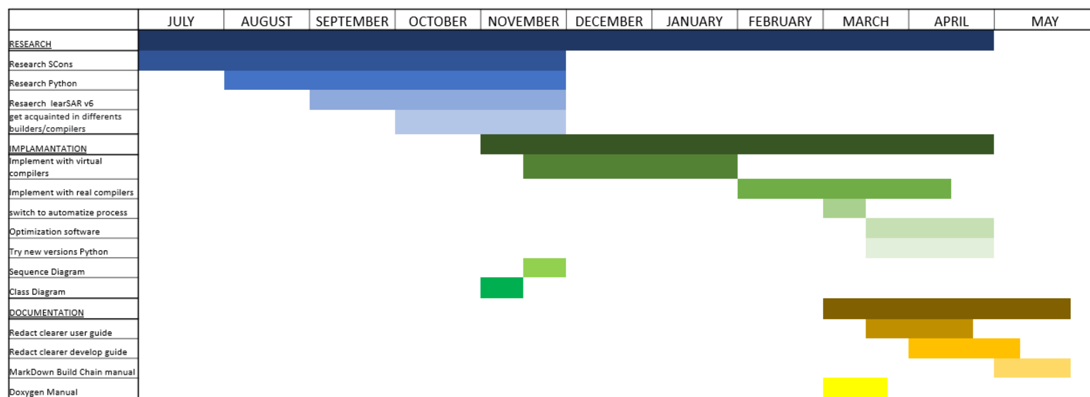

Figure 1. Gantt Diagram of my Project.

1.2. Goals

1) Unify in a packet to increasing the maintenance and reduce the possible errors. It will help us to certificate the build chain.
   It is necessary to validate the good functionality of the tool implemented, to obtain the trust of clients.

2) Configuration compiler in Build Chain more scalable.
   To facilitate the builder configuration and its creation to be more scalable.

3) Build chain with more intern quality.
   To increase the quality of build chain, in terms of intern maintenance, scalability and configuration time.

4) Adapt to Lear software tool process.
   In Lear there are a develop tool process and it is necessary to adapt it.

5) Simplify, clarify, and generalize the code used.
   The purpose is to simplify, clarify, and generalize the code because it will be easier to work with it, to update it, optimize it.

6) Improve the managing license.
   For each compiler is necessary to introduce a license in the build chain to recognize it. Every builder gets the method to manage license to simplify this process.

7) Automatize Build Chain to select a builder.
   In Lear works with more than one compiler, to facilitate and automatize process I'll implement a method to do it.

8) Make up strong and clarify documentation.
   Current documentation is too general and messy… One of the objectives is to try to automate the process as much as possible. Then I would like document how use tools, build chain and automated documentation.

9) Improve performance in terms of build/execution faster.
   The intention is using new Python and SCons versions to verify if the build/execution time can be reduced.

10) Check compatibility with new Python/SCons versions.

1.3. Memory Structure

Introduction: The planning of the project and the main objectives that are intended to be achieved by carrying out the work are defined.

Lear: Company that provides me the opportunity to conduct the project and I explain some relevant information.

Build Chain, Python, SCons and Compilers: Theorical information about the main concepts in my project.

Implementation of the Generic Part of the Core: Structure of my project. I explain how the process has been performed, the evolution/update of Build Chain system.

Implement Virtual Builder: The *Virtual* compiler is described: how adapt the builder initialization to SCons, the implementation of the BCPDF of this compiler and tests performed to verify the functionality.

Implement Builder to IAR compiler: *IAR* compiler is described: how adapt the builder initialization to SCons, the implementation of the BCPDF of this compiler and tests performed to verify the functionality.

Implement Builder to Tasking Compiler: *Tasking* compiler is described: how adapt the builder initialization to SCons, the implementation of the BCPDF of this compiler and tests performed to verify the functionality.

Implement Builder to Hightec Compiler: *Hightec* compiler is described: how adapt the builder initialization to SCons, the implementation of the BCPDF of this compiler and tests performed to verify the functionality.

Switch to automatize process: I've explained how I automatize the selection of builder compiler in each cse.

Optimization Software: As happens in all project's software, there are more than one version, so I must optimize the code and process carried out.

Try New Versions Python/SCons: New Python and SCons versions can improve performance significantly.

Documentation: I must document everything Build Chain V7: process performed, how to document source code automatically and document it with MarkDown format.

Results/Business Values: This part is related to the results according to the business values.

Future Steps: Future lines of work that could contribute to an improvement of the system are presented fulfilled.

Conclusions: I explain the conclusions drawn from the preparation of this project.

Bibliography: Relevant data of source information that I based to redact theorical theory.

Annexes: The documentation generated corresponding to the project, as well as the codes corresponding to the programs developed in accordance with the different parts of the project and which have not been exposed during the explanation of the illative implementation.

2. Lear

Lear is an American global automotive technology leader in Seating and E-Systems delivering intelligent solutions in-vehicle experiences for customers around the world. It is the 186 fortunes company in the world, with 186600 employees, distributed in 265 facilities in 38 countries. It is focused in 2 work lines:

- Seating: Advanced innovation in automotive seats, sustainable solutions for the industry with inhouse electronic and software capabilities. Leads the manufacturing of seats.
- E-Systems: Digital transformation where vehicle architectures will be connected via wired and wireless networks: Electrical distribution systems, connections systems, electronics and connected services connectivity modules.

Figure 2. Lear Logo.

3. Build Chain, Python, SCons and Compilers

My project is based on Python and SCons. SCons is a library of Python, but I'll explain in detail later. My main goal is update actual Build Chain system. That should be more scalable and easier to configure. For this reason, I'll introduce theorical information about Build Chain, Python, SCons and compilers into my project, which is the main concepts of my project.

3.1. Build Chain

Around the world there are Lear engineers working in different project developing software, so, this software must be compiled. Build Chain is in charge to compile this software to then generate a final product software. Build Chain arises from the need to automate this process.

My project is based in actual Build Chain system, which is the Build Chain develop by Lear. It consists of a set of software tools to complete complex software development tasks or to deliver a software product that allows built many files simultaneously and efficiently.

The Build Chain purpose is to provide other features such as redact reports to provide the user with greater control about the construction of the final product.

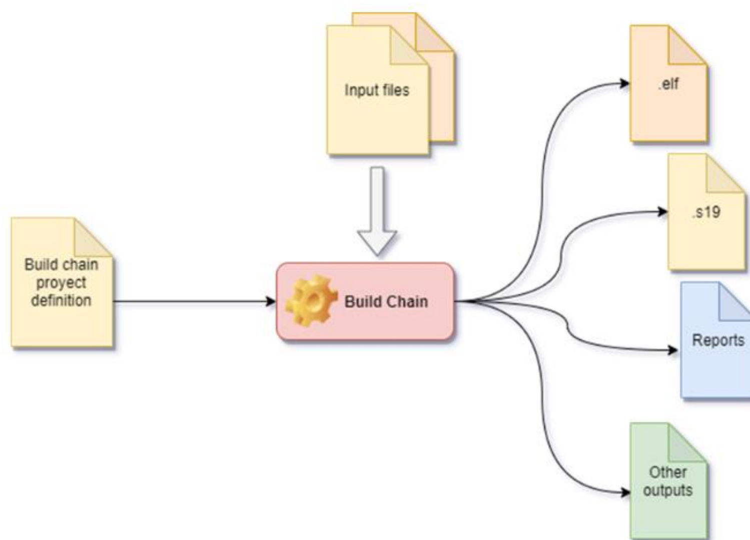

Figure 3. General Build Chain Diagram.

I unify the user-level configuration project in a single site. The Build Chain receives different inputs, like the Build Chain Project Definition File (BCPDF, also named YAML file) or build procedure tasks. The BCPDF defines phases and steps; SCons loads the corresponding tasks for each defined phase. The input files are the source files of our project. In the other hand, output files are all project products and compilation products.

The Build Chain encompass many phases, like the Generation, Compilation or Post Build phases. There are also phases that check some build parameters, as well as before and after the compilation.

Phases must be executed in the correct order, as the output from one phase is the input for the next one.

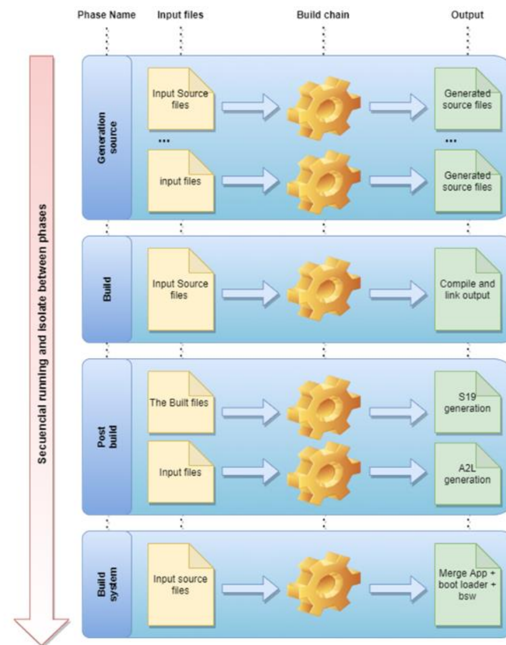

Figure 4. Build Chain execution phases.

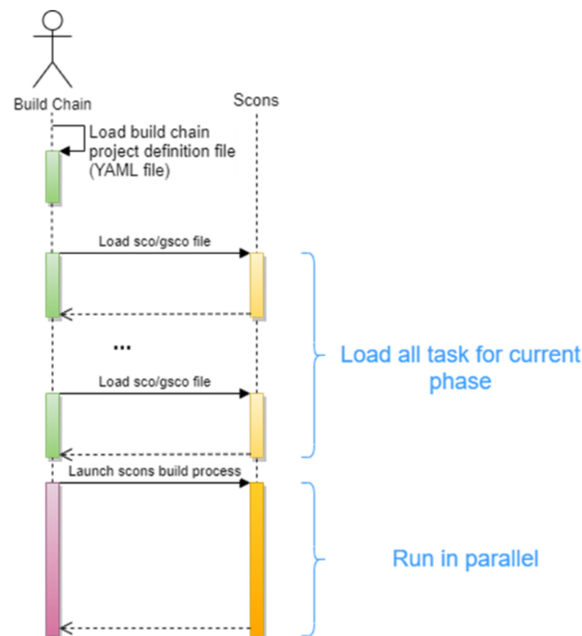

Figure 5. Build Chain execution tasks.

Ex PHASES: phases of code generation and compilation, first code must be generated and then compilate.

Ex TASKS: compilate source files, it can compilate a dozen files at same time.

Tasks are defined in *.sco* and *.gsco files in the BCPDF,* which are Python files that use SCons library.

- *.gsco:* used for generation tasks.
- .sco: used for non-generation tasks.

LearSAR, as a build system, is implemented in Python.

3.2. Python

It was created at late eighties by Guido van Rossum in Centrum Wiskunde & Informatica (CWI), in Netherlands.

The name of the programming language comes to the hobby of the creator fondness about British comedian Monty Python.

Python is an OOP language; I want this language because it allows us to pervert the programming doing classes and subclasses. OOP is a programming paradigm based on the concept of objects, which can contain data, like, attributes or properties, and code, for example methods or procedures. Computer programs based in OOP are designed by making them out of objects that interact with another object.

Figure 6. Python logo.

This programming language is *multi-paradigm.* This forces to programmers to adopt a particular programming style. Can support different paradigms. A programming paradigm is a method to carry out computations, the way how to structure and organize tasks that should realize a computer program. It allows for several styles supports programming-oriented objects (POO), imperative programming and functional programming.

Imperative programming: code directly controls execution flow and state change.

Functional programming: a desired result is declared as the value of a series of function evaluations, so, declares properties of the desired result, but not how to compute it...

In the computer science world, it's named *multi-platform* an attribute conferred an informatic program or methods and computation concepts that are implemented, operate internally in multiples informatic platforms. Multiplatform software can be divided in two big types or classes: one requires an individual compilation to each platform that supports it, and the other can execute directly at any platform, without special preparation, for example, the software implemented in a interpret language or portable precompiled bytecode for which the interpreters or packets in execution time are common or standard of all platforms.

For example, a multi-platform (cross-platform) application can be executed (run) without any problem on both Microsoft Windows on x86 architecture, like in Linux in architecture x86 and Mac OS X, on either PowerPC or Apple Macintosh systems based in x86. In general, a cross-platform application (can run) can be running on all existing platforms or at least two platforms.

Python is also a *structured programming language*, is a programming paradigm aimed to improve the clarity, quality, simplify and reduce development time of computer program by making structured control flow, example: if then, else…; repetition, example: for and while; block structures and subroutines.

It is *dynamic*, a variable can take different type values and is *reference counting*, is a technique to count the times is used a determined resource. That techniques are used to manage memory.

Python is an *interpreted* language, it's no necessary to be built.

A Python advantage is that it is an open-source language. It's free and there is a big community commenting and helping. Another advantage is that it is easy to learn because it's high-level language and it's very similar than Java, C+/C++… Among other advantages is that it is important goal is facility to extend the design of language. It's possible to write easily new modules in C or C++. Python can include in applications that needs a programmable interface.

I can see the growing trend of Python, by TIOBE, (programming community index), I can relate this phenomenon to the emergence of AI. Nowadays is the first programming language used in the world with the 16'33% in May. So, it's compatible with different systems, devices (hardware platforms), … and has the same interface in all of them. But in the future is prevents that it'll be more useful because the IA, data science, the developing applications, machine learning are working in this language, so the difference between the others may be bigger.

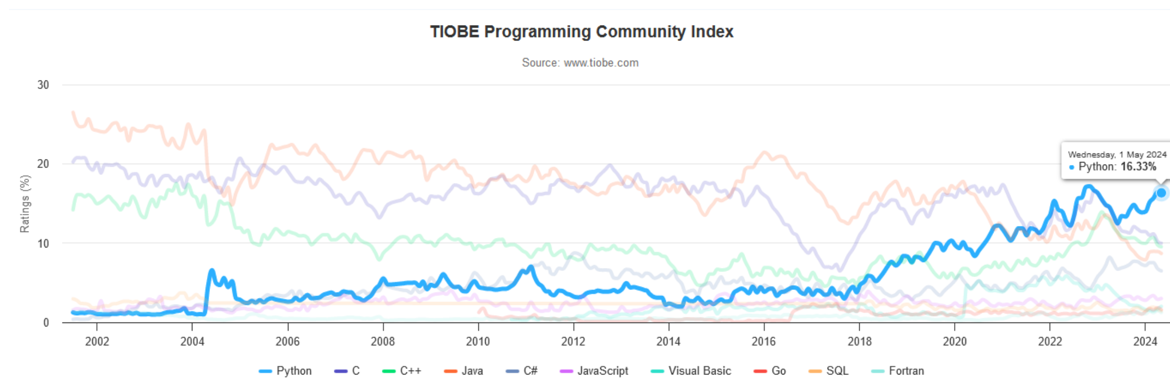

Figure 7. TIOBE Programming Community Index.

The philosophy of Python is described in the Zen of Python (PEP 20), which summarizes principles such as:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

To realize my project, first I use Python version 3.7. but I updated code to support new releases. To check an updated, it I use Python 3.12.2. (Explained in *section 9 and 11*)

3.3. SCons(fet)

SCons is a library of Python. Is the actor responsible to prepare compilation. SCons serves as a software construction tool, parsing source code to identify dependencies and tailoring them for specific operating systems. It then compiles these into executable binaries for installation on the intended platform. Comparable to the conventional GNU build system utilizing *make* and *autoconf* tools, SCons streamlines project configurations and build processes through Python scripts. Tools can be described as Python packages that specify how to create SCons targets using several applications. I can also say that tools are used to tell the environment how to build things.

The main characteristic is that SCons automatically calculates files dependencies in a Project based on MD5 signatures, so it makes a tree of dependencies, the network files which involves every file. If there is a modification in a file, it built only the branch necessary.

The main characteristic of SCons is the incremental compilation, SCons recompiles only modified portions of program to be more effective. When SCons makes a dependency tree, can see modified files and then compiles it. This type of compilation is named Incremental Compilation. Is supported by SCons.

SCons has many advantages to use it like a software construction tool. As SCons is a Python library it's really integrated with projects. Also is Open Source, so, is a next Open-Source generation software construction tool. Another advantage is that SCons is a cross-platform, is an improved, cross-platform substitute for the classic *Make* utility. Is easy to understand and work with. SCons plays a user script that is written in Python and I can use/take advantage the power of Python. Among others, support parallel builds. It has these advantages thanks to Python programming language since they have a lot in common.

Figure 8. SCons logo.

Today there are different Build Chain software. For example:

Gradle → Building application components.

Jenkins → Integrating separate code components.

Selenium → Automated application testing

MakeFile → Manage tree dependencies.

The primary issue with this Build Chain software lies in its reliance on Java. Java software construction tools encounter difficulties when transitioning between different versions of the virtual machine. Compatibility can be compromised when switching versions. The learning curve it's slower because it's complicated and not intuitive. Additionally, due to Oracle's ownership of the Java programming language rights, accessing the latest versions often requires payment for businesses.

I choose SCons, because works in Python. Python is free software programming language, it's easy to use and accepted by community. SCons has unique options and configurations.

3.4. Compilers

In computer science, a compiler is a software tool that converts code written in one programming language (the source language) into another language (the target language). Typically, it's employed to translate high-level programming languages (such as C-like, Python, Java) into low-level languages (like assembly language or machine code) to generate executable programs.

When a compiler works performs some or all of the following operations, often called phases: preprocessing, lexical analysis, parsing, semantic analysis, conversion of input programs to an intermediate representation, code optimization and machine specific code generation:

- Preprocessing: It's a software application that handles incoming data to generate results, which are then utilized as input for another software.
- Lexical analysis: It involves transforming text into predefined categories. For programming languages, these categories encompass identifiers, operators, grouping symbols, and data types.
- Parsing: It's the procedure of analyzing a sequence of symbols across various domains, such as natural language, computer language, or data structures, in accordance with formal grammar rules. It denotes the systematic examination, typically by a computer, of a sentence or string of words, delineating their syntactic interrelation via a parse tree that may include semantic details. This term is also pertinent in computer language analysis, indicating the syntactic dissection of input code into constituent elements to streamline compiler and interpreter development.
- Semantic analysis (syntax-directed translation): It's a step, typically following parsing, aimed at extracting essential semantic details from the source code. This often involves tasks like type checking, ensuring variables are declared before use, which cannot be adequately expressed in the extended Backus–Naur form and are thus challenging to identify during parsing.
- Conversion of input programs to an intermediate representation (IR): It refers to the organizational framework or code utilized by compilers to depict source code. An Intermediate Representation (IR) is crafted to facilitate subsequent operations like optimization and translation. An ideal IR must maintain accuracy, preserving all information from the source code, while remaining agnostic to any specific source or target language. IR can manifest in various formats: as an in-memory data structure, or as a distinct tuple- or stack-based code readable by the program. In the latter scenario, it's also termed as an intermediate language.
- Code optimization: Within computer science, program optimization, also known as code optimization or software optimization, entails refining a software system to enhance its efficiency or resource utilization. Typically, this involves accelerating the execution speed of a computer program, reducing its memory footprint, conserving resources, or minimizing power consumption.
- Machine specific code generation: Within computing, code generation is a crucial step in the compilation process, where it transforms the intermediate representation of source code into a format, such as machine code, that can be directly executed by the target system.

There are many types of compilers which produce outputs in different useful forms.

In the Automotive Industry there are a lot of microprocessors to control different systems like comfort, security, assistance, …

Each microprocessor can be compiled by more than one compiler. Each microprocessor manufacturer recommends on some occasions a compiler, but in other cases not. So, in Lear you can select which compiler is the best or simply is convenient in every case. There are some cases that is the customer the interested party to use the compiler specific, and in many cases, I must configure, introduce and prepare to still work with a new compiler in build chain.

These compilers have got 4 important tools: Compiler C, Assembler, Linker and Archiver. Each tool has a different function:

- Compiler C: In the realm of computer science, a compiler is a specialized software that converts code written in a specific programming language (known as the source language) into a different language (referred to as the target language). The term “compiler” is predominantly used to describe software that transforms source code from a high-level programming language into a low-level programming language (such as assembly language, object code, or machine code), resulting in a runnable program.
- Assembler: Refers to a type of computer program that is responsible for translating a source file written in an assembly language, into an object file that contains machine code, directly executable by the microprocessor.
- Archiver: A file archiver is a software application that amalgamates multiple files into a single archive file (library), or a sequence of archive files, for simplified transfer or storage.
- Linker: Is a program that takes the objects/libraries generated in the early steps of the compilation process, the information of all necessary resources, removes those resources that it does not need, and links the object code with its library or libraries, ultimately producing an executable file or a library.

In the new Lear Build Chain, for each compiler (tasking, iar, hightec…) I want to create a builder with specific information, cause I separate the generic and specific configuration to facilitate the configuration of it.

4. Implementation of the Generic Part of the Core

This project is in continuous development, before that I worked to update the build chain, in Lear was a build chain tool.

To introduce at build chain V7, I should talk about build chain v6. In this version (v6), it’s very entertained add a new compiler, so, all files needed to the building are specific for each compiler, so to implement a new compiler in the Build Chain V6, it’s necessary to create a lot of files, set different variables and methods. That can be a problem because spends a lot of time configuring the builder. So, to solve it I was centered to study the code of version 6, to focus my attention on the parts that are constantly repeated. There are some identical methods for each compiler, so, one solution can be separate it with in generic and specific code, to simplify the process and the tool in general. I grouped all common parts of each compiler in one, called generic part. As a generic part I created any classes to put the information there. This is the core and the most significant part of build chain v7. In the specific part there are only the configuration and generation of each compiler, but in general it is a bit complex.

I see that in every compiler there is a generic part, the code is common, and I unify this code to optimized: This is the key to my project because helps to facilitate many tasks, like maintenance, intern quality of build chain, adaptation to development process tools of Lear, a better managing license, among others.

SCons is really complex build chain tool and I can't explain in detail, in some cases I will call it *Black Box.*

To achieve the proposed objectives, as I commented before, I separate the generic part of this files in generic and specific part. I create Python scripts with the generic methods: *base_init* and *base_user_library*, for the creation builders. The software components files are in common between them, so to easier process the build chain can work with only one *SConstruct* file.

In the *compiler_Builders* folder there is the specific part, but I will comment in the *Section 4.2.*

Then, I create two classes dedicated to the compiler tools, one is to instance a tool, *Compiler_Tool*, and the other is to configures each tool with some parameters, *CompilerConf.* The last one is instanced inside *Compiler_Tool* class as an object. I will explain with more detail in the next section.

Finally, Build Chain V6 works with Python 3.7. and does not contemplate works with more than one Python versions. In new Lear Build Chain, to test both version I must manage the Python versions I've implemented, so I created a folder named *generics_Objects* where there are different files and techniques to control versions.

4.1. Definition & Design Core Build Chain V7

In this section I will explain the structure of my project. It is divided in 6 parts: *lear_scons_builders*, *Python*, *Python_env*, *source*, *test* and *Tools_Env*.

Lear_scons_builders is the part implemented by me. Inside, is the file that starts the build chain, this file is named *__init__*, inside it, is the method *generate(),* to understand better, this class executes the build chain, it is equivalent to the *main* function in other programming languages. The other file in this folder is *about* and it's necessary to create a package, there is all information about fields to generate package, is better explained in *section 10*. In this folder, there are 4 more folders to separate it: *compiler_Base, compiler_Builders, domain and generics_Objects.*

In the class *__init__* I use the switch that automatize the selection of builder. Depends on Python and consequently SCons versions, the switch will create a builder compiler instance different. I would like emphasize that there is a method called *__version_checker()*, this is a method that check the Python version used, then depends on version import a library to use the switch corresponding, for Python 3.12 or older a dictionary and for latter versions a switch.

In this point, *__init__* creates two instances that are based on the build chain, these instances are specific compiler builders that inherits from classes inside *compiler_Base* folder. There are 2 Python scripts with generic compiler information that I talk about them, *base_init* and *base_user_library:*

- *base_user_library*: In this class there are all methods responsible for preparing all source and target files as well as creating a list with files should be to be compiled, to unify the configuration project at level user in only one site for then compile in C programming language, next to transform a binary code with the assembler, once done this phase, it links all necessary files. Finally, it should be using the archiver to store files with post build information. In this class there is defined all build chain processes that intervenes in it.
  To get a general idea is the set of actions to execute for build chain.

- *Base_init*: This class is the father of all compiler builders. There are all specific methods to inject all variables of a different compilers, that are initialized in the son class, for example *tasking_tricore* compiler, *tasking_init*. Also, it detects automatically or manually the path of the different tools, such as compiler C, assembler, linker, archiver... All tools are defined in the build chain project definition (.YAML).

The difference between *base_user_library* and *base_init* is that the former is dedicated integrally in actions of build chain and the latter is created to set/prepare all tools.

Once I have created the instances, I prepare all tools (license, compiler C, assembler, linker and archiver) and next, firstly I used the instance of *nameCompiler_user_library* to add compilation options to the selected environment, finally, to build, compile C, compile ASM, and link.

In the class named previously, there is a rewrite method to overwrite, the method concretely is *_PrepareLicense*, this method is also described in each *nameCompiler_init* where there is the initialization license, but I'll explain more concretely in *section 4.2* and in each compiler section.

Overwrite a method is a feature in most object-oriented programming languages in which two or more methods share the same name but have different parameters. Method overloading allows you to have different functionality for the same method name, reducing complexity and improving code quality. Method overloading can be done by changing the number, data type, and/or order of parameters.

In *compiler_Buiders*¸ there are as many folders as there are compilers, in each folder there is each compiler builder definition with specific code. For each compiler, I need create 2 Python scripts: *nameOfCompiler_init* and *nameOfCompiler_user_library.* I will talk about them in the implementation of specific aspects points, in *section 4.2.*

In *domain*, there are 2 files: *Compiler_Tool*, with the definition of each tool. There are 4 different tools (compiler C, assembler, archiver and linker). For each tool build chain must know information related, so I create an object with the constructor with it and getters and setters. The other class is named C*ompilerConf*, in this class there are SCons variables of each Compiler_Tool, also like *Compiler_Tool,* only there are the constructor and getters and setters related.

I create two classes to configure each compiler tools:

- *Compiler_Tool:* This is the class used to create an instance of each principle/main tools (Compiler C, assembler, archiver, linker). This class creates an instance in its constructor of *CompilerConf* with configuration of every tool.

```
class Compiler_Tool (object):

    def __init__(self,name='', filePath='', extension='', searchPaths=[], compilerConf=CompilerConf()):
        self.__name = name                      # Compiler tool name file.
        self.__path = ''                        # Compiler tool path file.
        self.__extension = extension            # Compiler tool extension file, for example ".exe".
        self.__searchPaths = searchPaths        # list of 2 positions with words: "bin" and "path".
        self.__absPath = None                   # Compiler tool absolute path file.
        self.__filePath = filePath              # Path of compiler tool file.
        self.__absFilePath = None               # Absolute path of compiler tool file.
        # ----- Scons Tools parameters
        self.__CompilerConf = compilerConf      # Compiler Tool configuration.
```

Figure 9. Initialization and Constructor of Compiler_Tool class.

The constructor apart from creating an instance of the class named above, creates the variables that have seen before.

- *Name:* Compiler tool name file.
- *Path:* Compiler tool path file.
- *Extension:* Compiler tool extension file.
- *searchPaths:* List of 2 positions with words: “bin” and “path”.
- *absPath:* Compiler tool absolute path file. (NoneType)
- *filePath:* Path of compiler tool file.
- *absFilePath:* Absolute path of compiler tool file.
- *compilerConf:* Instance of *compilerConf* with Compiler Tool configuration.

Then to initialize these variables inside constructor method follows the getters and setters methods of each variable.

- *CompilerConf:* In this class I create all variables needed to configure the four compilers tools used in the different compilers builders (Compiler C, assembler, archiver and linker). These variables get the name of *SCons Construction Variables*. Every time that I want a compiler tool in the initialization, I use that class to create an instance and set the variable of the tool required. This class is in charge to adapt builder initialization to SCons.

The variables of SCons are these:

Compiler C

- **CPPDEFPREFIX:** The prefix used to specify preprocessor macro definitions on the C compiler command line. This will be prepended to each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.
- **INCPREFIX:** The prefix used to specify an include directory on the C compiler command line. This will be prepended to each directory in the $CPPPATH and $FORTRANPATH construction variables when the $_CPPINCFLAGS and $_FORTRANINCFLAGS variables are automatically generated.
- **OBJSUFFIX:** The suffix used for (static) object file names.
- **CCCOMFLAGS:** General options passed to the compiler C.
- **CCCOM:** The command line used to compile a C source file to a (static) object file. Any options specified in the $CFLAGS, $CCFLAGS and $CPPFLAGS construction variables are included on this command line. See also $SHCCCOM for compiling to shared objects.
- **CXXCOM:** The command line used to compile a C++ source file to an object file. Any options specified in the $CXXFLAGS and $CPPFLAGS construction variables are included on this command line. See also $SHCXXCOM for compiling to shared objects.
- **INCLUDE_COMPILER_PATHS:** include the compiler path.
- **CCCOMSTR:** If set, the string displayed when a C source file is compiled to a (static) object file. If not set, then CCCOM (the command line) is displayed. If not set, then $CCCOM (the command line) is displayed. See also $SHCCCOMSTR for compiling to shared objects.
- **CXXCOMSTR:** If set, the string displayed when a C++ source file is compiled to a (static) object file. If not set, then CXXCOM (the command line) is displayed. If not set, then $CXXCOM (the command line) is displayed. See also $SHCXXCOMSTR for compiling to shared objects.

Assembler

- **ASCOM:** The command line used to generate an object file from an assembly-language source file.
- **PRE_ASCOM:** The command line used to prepare assembler.
- **ASFLAGS:** General options passed to the assembler.
- **ASCOMSTR:** The string displayed when an object file is generated from an assembly-language source file. If this is not set, then $ASCOM (the command line) is displayed.

Linker

- **LINKCOM:** The command line used to link object files into an executable. See also $SHLINKCOM for linking shared objects.
- **MAP_SUFFIX:** The suffix used for the mapping file.
- **PROGSUFFIX:** The suffix used for the binary file.
- **LIBDIRPREFIX:** The prefix used to specify a library directory on the linker command line. This will be prepended to each directory in the $LIBPATH construction variable when the $_LIBDIRFLAGS variable is automatically generated.
- **LINKFLAGS:** General user options passed to the linker. Note that this variable should *not* contain -l (or similar) options for linking with the libraries listed in $LIBS, nor -L (or similar) library search path options that SCons generates automatically from $LIBPATH. See $_LIBFLAGS above, for the variable that expands to library-link options, and $_LIBDIRFLAGS above, for the variable that expands to library search path options. See also $SHLINKFLAGS. for linking shared objects.
- **LIBLINKSUFFIX:** The suffix used to specify a library to link on the linker command line. This will be appended to each library in the $LIBS construction variable when the $_LIBFLAGS variable is automatically generated.
- **LINKERFILESUFFIX:** The suffix of the linker file.
- **LINKCOMSTR:** If set, the string displayed when object files are linked into an executable. If not set, then LINKCOM (the command line) is displayed. See also $SHLINKCOMSTR. for linking shared objects.

Archiver

- **LIBSUFFIX:** The suffix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.
- **LIBPREFIX:** The prefix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.
- **ARFLAGS:** General options passed to the static library archiver.
- **ARCOM:** The command line used to generate a static library from object files.
- **RANLIB:** The archiver indexer.
- **RANLIBCOM:** The command line used to index a static library archive.
- **RANLIBFLAGS:** General options passed to the archiver indexer.
- **ARCOMSTR:** The string displayed when a static library is generated from object files. If this is not set, them $ARCOM (the command line) is displayed.

This is the main code of class *compilerConf*:

```
class CompilerConf (object):

    def __init__(self):
        self.__CFLAGS = []              # C flags
        self.__CPPDEFINES = []          # Defines
        self.__CPPDEFPREFIX = ''        # Prefix fo
        self.__CCFLAGS = []             # C++ Flags
        self.__INCPREFIX = ''           # Prefix fo
        self.__OBJSUFFIX = ''           # Suffix fo
        self.__CCCOMFLAGS = ''          # Build str
        self.__CCCOM = []               # The comma
        self.__CXXCOM = []              # The comma
        self.__INCLUDE_COMPILER_PATHS = []
        self.__CCCOMSTR = ''            # If set, t
        self.__CXXCOMSTR = ''           # If set, t
        # ------- Assembler -------
        self.__ASCOM = ''               # The comma
        self.__ASFLAGS = []             # General o
        self.__ASCOMSTR = ''            # The strin
```

Figure 10. Definition and Constructor of CompilerConf class (part 1).

```
# ------- Linker -------
self.__LINKCOM = ''             #
self.__MAP_SUFFIX = ''          #
self.__LIBPATH = []             #
self.__PROGSUFFIX = ''          #
self.__LIBDIRPREFIX = ''        #
self.__LINKFLAGS = []           #
self.__LIBLINKSUFFIX = ''       #
self.__LIB_LINKER_PATHS = []
self.__LINKERFILESUFFIX = ''
self.__LINKCOMSTR = ''          #
# ------- Archiver -------
self.__LIBSUFFIX = ''           #
self.__LIBPREFIX = ''           #
self.__ARFLAGS = []             #
self.__ARCOM = []               #
self.__RANLIB = ''              #
self.__RANLIBCOM = ''           #
self.__RANLIBFLAGS = ''         #
self.__ARCOMSTR = ''            #
```

Figure 11. Definition and Constructor of CompilerConf class (part 2).

In this class there are getters and setters and the constructor of each class, has all variables initialized.

In *generics_Objects* there are 2 files, in these files are the switches to automatize the selection builders. There are 2 switches, one for each Python version cause since version 3.12 of Python there is a method called *match* like a *switch* in other programming languages like *C*. This method is in *switch_Builder_engine_switch* class. The other class created is *switch_Builder_engine_Dict* and use a dictionary to select the builder corresponding, for older Python versions. Both classes are inherited from *switch_base*, this class is in charge to create an instance of every compiler builder (*compilerName_init* and *compilerName_user_library).* I will explain in *section* 9.

In packages folder, I store the final package for my project. In this package are all class and files that composed the core of Lear Build Chain 7. More detailed explain in *section 10.*

So the build chain is based in *Python* language, so the system need to install all other packages Python that the system need (for example pyYaml …). Here there are all packages needed for build chain operation and more.

And some requirements depend on the Python versions that is executed. The names of files are: *requirements_37.txt* and *requirements_312.txt*

In *Python_env,* is the Python environment with SCons modules, variables, packages and all tools needed for the project. This folder contains a Python environment. This is a copy of Python and all the libraries that have been installed for Python (It is an isolate method for Python it allows have more than one distinct Python for each project). I've created two Python environments to check the different versions of Python and SCons, I'll explain more details in *section 11*. This is the structure of the folder:

- *Lib:* necessary packages to use all methods needed for the execution of project.
- *Scripts:* all executable files related with Python and SCons, for example *pip.exe*, *Python.exe, scons.exe…*

Then, there is *source* folder, inside there are YAML files of each compiler, and a software component for all compilers. Also, there are some *includes* and *defines,* libraries which is necessary to build properly. Finally in this folder there are all files resulting by the build chain. These files are mostly, are binary files (.elf). This file is the more important file for a project so in this file the project is described (BCPDF).

Source folder, must contains user scripts, explain how to generate code or how to compiler (.sco, .gsco). I try to construct the final result, defined in only one place.

Test folders are similar than *source* folder, so the files inside are really similar, because to test project it's necessary all source files to introduce at test and obtains a satisfactory result.

Finally, the *Tools_Env* folder is necessary to create the environment desired like *Python_env*.

4.1.1. Conceptual/Overview Diagram

To make this diagram I use *Visual Studio Code* with the plugin *draw.io*. This plugin is so useful to do any Diagram. All diagrams less class diagram I use this combination of software.

A conceptual model is a representation of a system, made of the composition of concepts that is used to help people know, understand, or simulate a subject represented by the model, includes the important entities and the relationships between them. It is also a set of concepts. The term conceptual model can be used to refer to models that will be formed after a process of conceptualization or generalization.

This diagram is a project overview: SCons and Build Chain.

The first diagram shows the Build Chain Project Definition File (BCPDF), which describes all relevant information. This data is used by BuildBySteps . It is a main scheduler that executes all project phases describes into BCPDF. Then SCons starts running and executes initializations for all the tools (Compiler C, linker, archiver and linker), builders of different compilers. For the specific phase for example to generate code the system calls all the .gsco files for the SWC that are defined. For the compiler and link phase the system will call to .sco files. The .gsco/.sco file is related with each SWC and it is isolate for the compiler that will be used.

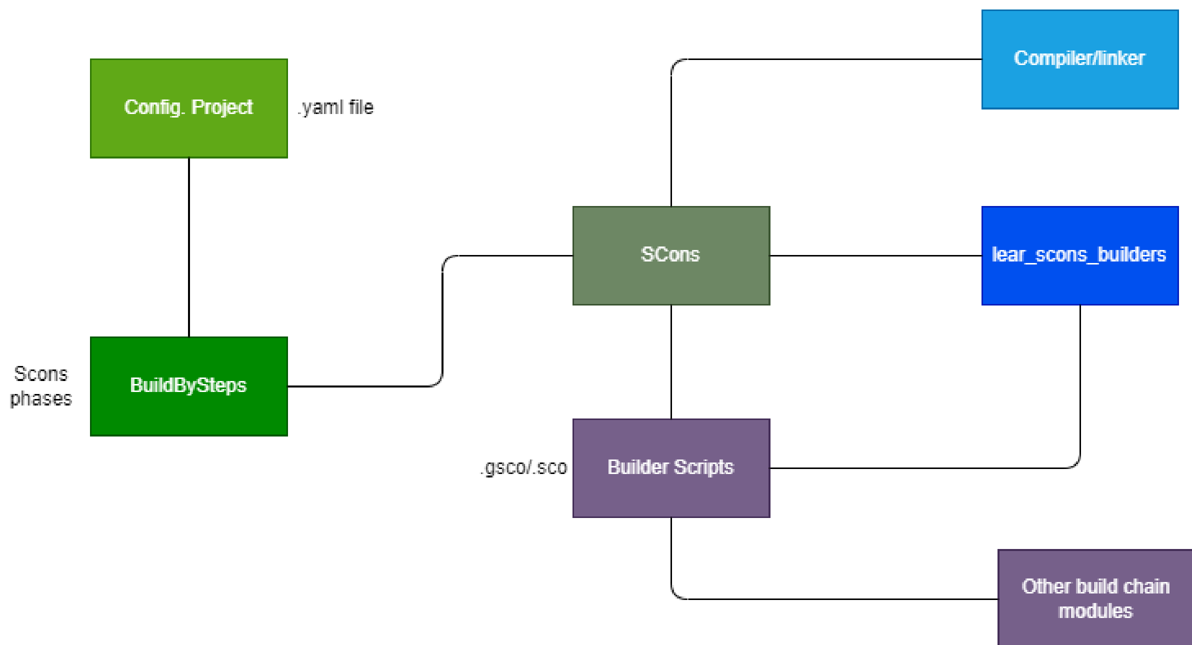

Figure 12. SCons Conceptual Diagram.

In this other diagram it can be seen the new structure of the compiler builder (dark blue box in previous diagram). First, I got a SCons builder that must be executed, when it runs, should be select a different compiler, with specific information, and then create the compiler builder with generic information added.

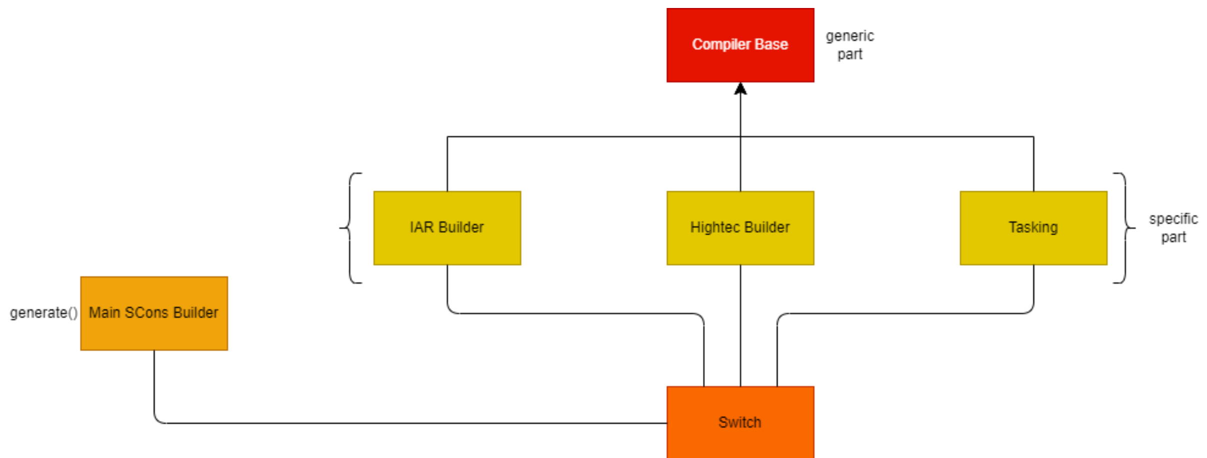

Figure 13. Lear_SCons_Builder Conceptual Diagram.

4.1.2. Class Diagram

To make this diagram I use the *Visual Studio 2022*, which is a different program used for the rest of diagrams. To make this diagram I used C# to program the different boxes.

It is based in POO and Factory type model programming like you can see in the class diagram.

In software engineering, a class diagram is a type of static structure diagram that describes the structure of a system by showing the system classes, their attributes, operations (or methods), and relationships between the objects.

This diagram is useful to see the different classes with all attributes and methods are implemented in.

4.1.3. Sequence Diagram(fet)

To finish with the diagrams, I perform a Sequence Diagram of compilation and how the build chain is composed.

To perform this diagram I've used *Visual Studio Code* with the plugin *draw.io*. This plugin is so useful to do any Diagram.

In the realm of software engineering, a sequence diagram illustrates the chronological flow of process interactions. It visually represents the involved processes and entities, along with the order of messages exchanged to fulfill the desired functionality.

This is Sequence Diagram, I must focus our attention on Compiler Library, the switch and the different builders because are the part that I modified in my project. Compiler/assembler/linker/archiver and externals tools, all tools integrated in my project.

In this Schematic you can see the complexity to build a single file and then the linking of this. You can see intern calls that compose build chain. To understand how works internally the build chain.

Is common in software develop to make diagrams to structure the program to implement. The purpose is to optimize, structure and clarify develop process.

4.2. Implementation

In *section 4* I explain the generic part of Build Chain, the part of builder in common, all methods necessaries and classes. The specific part is in *compiler_Builders* folder where all the information about each compiler is located. In each folder there are 3 files. A *__init__* file is in blanch, but I need for the project architecture. Then there is a *compilerName_user_library* that is a son of *base_user_library* and it's necessary to create an instance to use all public methods inside father class, but I haven't implemented for each compiler, for me it's enough using the father methods (full code is in *sections A.1.2.1.2., A.1.2.2.2., A.1.2.3.2. and A.1.2.4.2.).* Finally, there is the initialization of each builder, with the four main tools used in the building, (Compiler C, compiler ASM, archiver and linker) and a method to manage license named *_prepareLicence()*. The file is named *compilerName_init.* So, all files have the same structure, but the content is different between them. The structure in both is an initialization of attributes with the name, description, path, environment name for each builder. Then, I define the constructure method, and I starts with a *super()* method to reference the father class. Finally, I've needed to initialize the tools with the name, extension, the list called before, and the compiler configuration (*compilerConf* instance). This instance is created in each tool configuration method.

Once constructor method is defined, I need to manage license. So, each builder needs a different preparation license, and in this site is where is introduced all information. The method is named *_PrepareLicense().*

To finish, I define 4 methods to initialize all tools compiler configuration. To do it is necessary in each method create an instance of *compilerConf* class and introduce all information of each tool for the compiler concrete. The methods are: *__CompilerC_Conf()*, *__CompilerASM_Conf()*, *__CompilerLinker_Conf()* and *__CompilerArchiver_Conf().*

For each compiler implementation I will explain in more detail in corresponding sections.

5. Implement Virtual Builder

First, I develop the implementation of Build Chain V7. That's I need a virtual compiler and a SCons Builder for that pseudo-compiler to debug and review that the call to the compiler is correctly formed with all parameters, I debug line per line to check it. For this reason, I have programmed a software which will act as a compiler, this is intended to capture all the parameters passed to it by the build chain and generate the output file (.o) as if it were a real compiler, but instead of being a binary file, it will be a text file with the corresponding extension (.o, .obj,…) where I have stored all the parameters captured, as well as data relevant to our debugging. By using this pseudo-compiler, I make the compilation time lower thus improving the testing and development time.

Secondly I need to test previously the injection of all variables of the different tools in each builder compiler, if the build chain project definition is well-structured to work properly and the software components (.gsco and .sco files) are defined correctly. When I evaluate that previous phases are working properly, the process to implement and configurate a real compiler is simplified because I know is the right process.

I will then develop a SCons builder to be able to handle this virtual compiler in such a way that I can integrate SCons and Lear's Build Chain.

The purpose of implement a virtual compiler is to check code implementation, because the output files are trivial so, are created to avoid some errors like check executable compilers, flag injections, variables values.

5.1. Implement Virtual Compiler

The implementation of virtual compiler is exactly that of the real compiler. The difference between real and virtual is the tools used, virtual compilers executes some executable files as a tools (compiler C, assembler, archiver, linker) but aren't tools, are pseudo-compilers, this files creates object files (.o) and binary files, as a real compilers. The purpose is to check all process, so in real compilers these tools are real. So, first I check it with a Command Line (.cmd file), this step is carried out only for virtual compiler. In this part of process, I don't check if C files or libraries are correctly created. Finally, I change the Command Line file to an executable file with libraries and C files generated properly to check if it correctly creates the object files and binary files.

Virtual compilers do not need a license to work, so I don't implement a method to manage it.

This is an example of virtual compilers:

```
class vcmp_init(base_init):

    builder_name = "vcmp"
    builder_description = "Virtual compiler"
    compilerHome_env_Var_Name = 'VCMP_HOME'
    tool_path = os.path.dirname(os.path.abspath(__file__))
    _verbose = False

    def __init__(self):
        super().__init__(vcmp_init.builder_name, vcmp_init.compilerHome_env_Var_Name)
        searchPaths=['bin','path']
        self._AddCompilerTool(Compiler_Tool(name='vcmp',
                                            extension='.cmd',
                                            searchPaths=searchPaths,
                                            compilerConf=self.__CompilerC_Conf()))
        self._AddAsmCompilerTool(Compiler_Tool(name='vasm',
                                               extension='.cmd',
                                               searchPaths=searchPaths,
                                               compilerConf=self.__CompilerASM_Conf()))
        self._AddArchiverTool(Compiler_Tool(name='varm',
                                            extension='.cmd',
                                            searchPaths=searchPaths,
                                            compilerConf=self.__CompilerArchiver_Conf()))
        self._AddLinkerTool(Compiler_Tool(name='vcmp',
                                          extension='.cmd',
                                          searchPaths=searchPaths,
                                          compilerConf=self.__CompilerLinker_Conf()))
```

Figure 14. Example of Virtual compilers initialization.

5.2. Builder Implementation for Virtual Compiler

A Builder is an object that is used to configurate a compiler with different options.

For each compiler I create a builder with some tools, each tool is used to build and generate files, product files, fruit of Build Chain.

Once a Builder, has configurated a compiler, SCons uses the compiler to compiler the software generated.

To implement virtual compilers, I need to inherit the father/parent class *base_init* to get all properties and methods of this class.

I must debug line a line to check all variables values are correct introduced and to follow the path of execution of my project.

I explain in detail the class *vcmp_init*, that is the class where the builder of virtual compiler is implemented.

To implement this compiler, I focus my attention on the specific part, because I separate it into two parts: general and specific parts. General part is the same for every compiler, and it has been decided not to change it. But the specific part is totally different in content, but not in structure. In this compiler there are 4 main tools: Compiler C, assembler, linker and archiver. Each tool it's necessary to be configured correctly and specifically for different compiler, this is an example:

First of all, I create a class of each compiler, every class since is defined as a class diagram and follows a factory model, is a son of the *base_init* class. I explain variable to variable explicitly:

- *Builder_name*: name of compiler.
- *Builder_description*: a brief description of the compiler.
- *compilerHome_env_Var_Name*: named environment compiler home
- *tool_path*: where is the tool file, the path to follow.
- *_verbose:* Boolean that I put in all compiler builder at *False*, to get more debug information.

Then I implement the constructor method to create a *vcmp_init* object. Firstly, I call *super()* method to reference with the father class (*base_init)* and catch his attributes, I can use all methods and properties of the father/parent class. After use *super()*, I create a list of 2 element to control *bin* and *paths* of each compiler, and finally I initialize in the object class the 4 main tools used in build chain. The name of each executable file, the extension, the path and an instance of *Compiler_Tool* class with information and configuration of each tool, to put inside this object the configuration, so, in this class is instanced an object of *CompilerConf*; like is commented in *Generic part (section 4.1.).* Next, I implement a protected method to prepare license, cause each license compiler is managing different, maybe there are some compilers which does not use a license. Finally, it's necessary to configure each tool depending on needs of the compiler. In all compilers I need to set the 4 main tools, I create a private method for each tool (Compiler C, assembler, linker and archiver). So, in each method, first I create an object of *compilerConf* to insert the configuration suitable. To finish, I return the instance with all configuration set. So, *Compiler_Tool* and *CompilerConf* are an application domain.

Managing license

This compiler hasn't a manage license because is a virtual compiler and it's not necessary. But the other compilers get a method to manage license.

Errors:

In this process there are different errors, so, Python is intelligent and it's not necessary to previously define the type of variable, for example in some cases, I put a coma at the end of the line, and it interprets that the variable was a list type, but I wanted a string, this type of errors is difficult to determine and solve, because for the debugger is not an error. These errors were typical in the injection of variables in the different compiler tools. I solve these problems in this compiler to avoid repeating them in the creation process of other compilers.

5.2.1. Builder Initialization to adapt to SCons

To adapt to SCons, it is necessary initialize SCons variables for each tool. The tools to be initialized are in compiler C, assembler, linker and archiver. The class *compilerConf* is in charge to initialize variables SCons in build chain process.

In *section 4.1.* I explain each SCons variable the meaning and the purpose. So, I only explain in this section the variables related to Virtual compiler with an example:

5.3. Project Configuration to use a New Builder and Virtual Compiler (YAML)

The automation of Build Chain, the definition of all phases, the different tools, defines, includes, flags of each tool and linker options, are defined in a build chain project definition (.YAML).

All the project configuration information is defined in the BCPDF, which is a YAML format file and divided in the following configuration sections.

YAML is a file of data and information naturally read by humans, resulting in human-readable data. It is a data serialization language. This means that can be translated into a format that can be stored or transmitted and then reconstructed later. Data types allowed by YAML are strings, integers, floats, lists, associative arrays like dictionaries, maps or hashes.

The YAML file is backward-compatible with older versions of LearSAR to avoid rewrite files for new build chain.

5.4. Builder Test for Virtual Compiler

To check if this builder/compiler works properly I must do a series of tests. It's important know the functionality of the programed code before delivering the product to the customer. It is only possible to check the compiler call, flags injection, the defines and includes injections. So, it's necessary to certificate the properly functioning, to do that it should be pass a series of test and to then certificate with QKIT and Unit test. These last tests are being executed in Build Chain V6, so thanks to separate general and specific part, only should be test the specific because the other parts is the same and is already tested.

5.4.1. Test to check Compiler Call (executable)

In this section is necessary to check if the command line is correct, so to check it, it is necessary to carefully observe the call of each tool, depending on the file type to be compiled. In the Virtual compiler it should appears the path *vcmp.cmd,* but can be a *vcmp.exe*. The parameter *-o* indicates that the following part is an output, and then the path of the two output files: *main.o* and *main.c*.

```
C:\Projects\Lear_Scons_Builders\trunk\test\vcmp\v1.0\bin\vcmp.cmd -o os\main\_out\main.o os\main\main.c
```

Figure 15. Debugger output to check Compiler Call (executable) in Virtual compiler.

5.4.2. Test to check Compiler Flags Injection and the Order.

In this point, I must solve if the flags and options described in the YAML file is injected properly in the command line. The format is important to the injection, so I check with the output files the result.

The order in which the flags are injected is very important since they directly impact whether the product is a safety complaint. Exist an extra tool by Lear that review this after compilation. Ex: Compiler flag checker

Is checked in the next point, in the image there is the *cflags* (c99).

```
-I. -Ilib -Ios -Ibsw\srv -Isgn -Ibsw\ecual -Ibsw\mcal c99 -DTGT_GMCU_VCMP -DLEAR_OS_USED -DAUTOSAR_OS_NOT_USED -DLEAR_OS_USED
```

Figure 16. Debugger output to check Compiler Flags Injection in Virtual compiler.

5.4.3. Test to check Defines and Includes Injection.

The order is critical only in the compiler flags injection. So, I check if the order is correct and the format. It's possible to check because there is a text file with all information.

In the following picture, it's possible to identify the includes, so, starts with *-I,* the defines because starts with *-D* and flags, only there is one flag (*c99).* This flag describes the format of C files.

```
Virtual compile:
-o "os\main\_out\main.o" "os\main\main.c" -I. -Ilib -Ios -Ibsw\srv -Isgn -Ibsw\ecual -Ibsw\mcal c99 -DTGT_GMCU_VCMP -DLEAR_OS_USED -DAUTOSAR_OS_NOT_USED
```

Figure 17. Debugger output to check Defines, Includes and flags injection.

The Compiler Flags Injection test and Defines/Includes Injection is check to certificate with QKIT method and is correct.

6. Implement Builder to IAR Compiler

IAR, developed by IAR Systems, stands as a genuine compiler. Established in 1983 in Sweden, IAR Systems is a software company specializing in providing development tools tailored for embedded systems. The acronym "IAR" stems from "Ingenjörsfirma Anders Rundgren", translating to "Anders Rundgren Engineering Company."

Specializing in C and C++ programming languages, IAR Systems crafts compilers, debuggers, and assorted tools dedicated to the development and debugging of firmware. This includes binary code, crucial for low-level control, spanning across 8, 16, and 32-bit processors and microcontrollers.

This is the lowest compiler in terms of compiling time. Is the least sophisticated.

6.1. Implementation of Specifics Aspects for this Compiler

First, we create a class of each compiler, every class like this is defined in class diagram and following a factory model is a son of *base_init,* class. Also, I create a class for each compiler to overload the class *base_user_library.*

To implement this compiler, I copy the structure from virtual compiler builder, because all compilers have the same structure, the difference is in the content. When I talk to implement specifics aspects, I refer to the class called *iar_init*. The difference in general terms could be the *builder_name, builder_description, tool_path, compilerHome_env_Var_Name* in the initialization attributes explained in *section 4.1* and the initialization methods for each tool. In this case it's like this:

```
class iar_init (base_init):

    builder_name = "iar_cmp"
    builder_description = "IAR compiler"
    compilerHome_env_Var_Name = 'IAR_CMP_HOME'
    tool_path = os.path.dirname(os.path.abspath(__file__))
    _verbose = False
```

Figure 18. Initialization of IAR compiler.

It should be remembered that the constructor is the same in all compilers, the changes are the different file to be executed for each tool:

```
def __init__(self):
    super().__init__(iar_init.builder_name,
                     iar_init.compilerHome_env_Var_Name)
    searchPaths = ['bin', 'path']
    self._AddCompilerTool(Compiler_Tool(name='iccarm',
                                        extension='.exe',
                                        searchPaths=searchPaths,
                                        compilerConf=self.__CompilerC_Conf()))
    self._AddAsmCompilerTool(Compiler_Tool(name='iasarm',
                                           extension='.exe',
                                           searchPaths=searchPaths,
                                           compilerConf=self.__CompilerASM_Conf()))
    self._AddArchiverTool(Compiler_Tool(name='iarchive',
                                        extension='.exe',
                                        searchPaths=searchPaths,
                                        compilerConf=self.__CompilerArchiver_Conf()))
    self._AddLinkerTool(Compiler_Tool(name='ilinkarm',
                                      extension='.exe',
                                      searchPaths=searchPaths,
                                      compilerConf=self.__CompilerLinker_Conf()))
```

Figure 19. Constructor of IAR compiler.

I focus my attention on the specific part. General part is the same for every compiler, and I don't change it. In this compiler there are 4 main tools: Compiler C, assembler, linker and archiver. Each tool needs to be configured correctly and specifically for different compiler. For each compiler, I need to specify the name of each executable file, the extension, the path and an instance of *Compiler_Tool* class with information and configuration of each tool, to puts inside this object the configuration.

But first I need to manage the license. This compiler has the exception that license is introduced manually.

```
def _PrepareLicense(self, env, compiler_home):
    """Prepare system with the license and then create an environment with the license compiler.
    In this compiler I've introduced license

    :param env:  Construction environment used to prepare license
    :param compiler_home: string with the path where is the compiler.
    :return:
    """
    pass
```

Figure 20. Method to manage license for IAR compiler.

6.1.1. Builder Initialization to adapt to SCons

In SCons there are internal classes that works below that a black box, it's complex. In SCons there are predefined variables that I used in the previous section to define each compiler builder, but these variables are predefined by SCons. These variables got different options. The main difference is that in this compiler I initialize and set the variables: CCCOMSTR, CXXCOMSTR, ASCOMSTR, LINKCOMSTR and ARCOMSTR.

These variables are not necessary to create in case of initializing these others: CCCOM, CXXCOM, ASCOM, LINKCOM and ARCOM. The variables above are set so in this case there is additional info.

To adapt to SCons, it is necessary to initialize SCons variables for each tool. The tools to be initialized are in compiler C, assembler, linker and archiver. it's necessary configures each tool depending on the compiler needs, in all compilers I need to set the 4 main tools, I create a private method for each tool. So, in each method, first I create an object of *compilerConf* to insert the configuration suitable. To finish, I return the instance with all configuration set.

The class *compilerConf* is the domain that store the initialization of variables SCons in build chain process.

In *section 4.1.* I explain each SCons variable the meaning and the purpose.

6.2. Project Configuration to use a IAR Builder and IAR compiler (YAML)

YAML is a readable data serialization language file that is commonly used to configures file and in applications where data is being stored or transmitted.

The YAML is really similar to the virtual compiler. For this reason, it will not be necessary to give further explanations. The only difference are the files to be executed, because are specific for each compiler. These files are: compiler C, compiler asm, archiver and linker (.exe and .ld). Also, the defines and all flags are different between YAML files.

6.3. Builder Test for IAR Compiler

To check if this builder/compiler works properly I must do a series of tests. It’s important to know the functionality of the code programed before give the customer the product. So, it’s necessary to certificate the properly functioning, to do that it should be pass a series of tests and then certificate with QKIT and Unit test, only I can check the compiler call, compiler flags, the defines and includes injections. These last tests are being executed in Build Chain V6, so thanks to separate general and specific part, only should be test the specific because the other parts is the same and is already tested.

6.3.1. Test to check Compiler Call (executable)

In this section it is necessary to check if the command line is correct, so to check so to check this, you must pay attention to the beginning of the command line. In the IAR compiler it should appears the *iccarm.exe.* The parameter *-o* indicates that the following part is an output, and then the path of the two output files: *main.o* and *main.c*.

```
c:\Projects\Learew\tools\iar\s32k3_IAR840FS\arm\bin\iccarm.exe -o os\main\_out\main.o os\main\main.c
```

Figure 21. Debugger output to check Compiler Call (executable) in IAR compiler.

6.3.2. Test to check Compiler Flags Injection and the Order.

In this point, I must solve if the flags and options described in the YAML file is injected properly in the command line, in the correct order. The format is important to the injection, so I check with the output files the result. The output file is a text file with all information.

```
--cpu=Cortex-M7 --cpu_mode=thumb --endian=little --fpu=none -e -Ohz --debug --no_clustering --no_mem_idioms --no_explicit_zero_opt --require_prototypes --no_wrap_diagnostics --diag_suppress=Pa050
```

```
-DS32K3XX -DS32K344 -DIAR -DUSE_SW_VECTOR_MODE -DD_CACHE_ENABLE -DI_CACHE_ENABLE -DMCAL_ENABLE_USER_MODE_SUPPORT
```

Figure 22. Debugger output to check compiler flags injection in IAR compiler.

Compiler flags should be the same than YAML file with the same order that is described in, I check it and it matches.

6.3.3. Test to check Defines and Includes Injection.

Also, it is critical if the includes and defines are putting wrong in the command to get all necessary information for the building. It is possible to check because there is a text file with all information. The includes gets an *-I* before the include path and the defines gets a *-D* before.

Includes

```
-I. -Ilib -Ios -Ibsw\srv -Isgn -Ibsw\ecual -Ibsw\mcal
```

Figure 23. Debugger output to check Includes injection in IAR compiler.

Defines

```
-DTGT_TC38X_IAR -DC_DERIVATIVE_TC397XE -D_GNU_C_TRICORE_=1 -DTRIBOARD_TC38XX -DLEAR_OS_USED -DAUTOSAR_OS_NOT_USED
```

Figure 24. Debugger output to check Defines injection in IAR compiler.

Includes and Defines should be the same than YAML file that is described in, I check it and it matches. There are three includes added by compiler: *sgn*, *bsw/ecual* and *bsw/mcal*, this is a default process that it is necessary to work properly.

7. Implement Builder to Tasking Compiler

The Tasking compiler hails from TASKING GmbH, a German company specializing in embedded software development tools. TASKING introduced its inaugural C compiler in 1986, followed by its first embedded toolset for single-chip microcontrollers in 1988. Expanding its reach, the company entered the U.S. market through a merger with Boston System Office (BSO) in 1989, subsequently unveiling a second-generation compiler.

In 1998, TASKING joined forces with Infineon Technologies to pioneer the inaugural TriCore development software, now used by Lear. Altium acquired TASKING in 2001, initiating work on their third-generation compiler technology. This advanced compiler technology was engineered to enhance both the speed and code efficiency of the TriCore development toolset.

Comprising C/C++ compilers, pin mappers, debuggers, linkers, and assemblers, this comprehensive development package caters to a diverse range of embedded system requirements.

7.1. Implement Specifics Aspects for this Compiler

The structure of this compiler is the same than the others but the files to be executed are not, and the settings as expected neither. For this reason, I must create a new builder for this and each one of them.

First, I create a class of each compiler, every class like this is defined in class diagram and following a factory model is a son of *base_init* class.

To implement this compiler, I copy the structure from virtual compiler builder, because all compilers have the same structure, the difference is in the content. When I talk to implement specifics aspects, I refer to the class called *compilerName_init.* The difference in general terms could be the *builder_name, builder_description, tool_path, compilerHome_env_Var_Name* in the initialization attributes explained in *section 4.1* and the initialization methods for each tool. In this case it's like this:

```
class tasking_init (base_init):

    builder_name = "tasking_cmp"
    builder_description = "Tasking compiler"
    compilerHome_env_Var_Name = 'TASKING_CMP_HOME'
    tool_path = os.path.dirname(os.path.abspath(__file__))
    _verbose = False
```

Figure 25. Inizialization of Tasking Compiler.

It should be remembered that the constructor is the same in all compilers, the changes are the different file to be executed for each tool:

```
def __init__(self):
    super().__init__(tasking_init.builder_name,
                     tasking_init.compilerHome_env_Var_Name)
    searchPaths = ['bin', 'path']
    self._AddCompilerTool(Compiler_Tool(name='cctc',
                                        extension='.exe',
                                        searchPaths=searchPaths,
                                        compilerConf=self.__CompilerC_Conf()))
    self._AddAsmCompilerTool(Compiler_Tool(name='astc',
                                           extension='.exe',
                                           searchPaths=searchPaths,
                                           compilerConf=self.__CompilerASM_Conf()))
    self._AddArchiverTool(Compiler_Tool(name='artc',
                                        extension='.exe',
                                        searchPaths=searchPaths,
                                        compilerConf=self.__CompilerArchiver_Conf()))
    self._AddLinkerTool(Compiler_Tool(name='cctc',
                                      extension='.exe',
                                      searchPaths=searchPaths,
                                      compilerConf=self.__CompilerLinker_Conf()))
```

Figure 26. Constructor of Tasking Compiler.

I focus my attention on the specific part. General part is the same for every compiler, and consequently remains the same. In this compiler there are 4 main tools: Compiler C, assembler, linker and archiver. Each tool needs to be configured correctly and specifically for different compiler. For each compiler, I need to specify the name of each executable file, the extension, the path and an instance of *Compiler_Tool* class with information and configuration of each tool, to puts inside this object the configuration, but first I need to manage the license.

7.1.1. Builder Initialation to adapt to SCons

In *section 4.1.* I explain each SCons variable the meaning and the purpose. So, I only explain in this section the variables related to Tasking compiler. The main difference is that in this compiler I initialize and set the variables: CCCOMSTR, CXXCOMSTR, ASCOMSTR, LINKCOMSTR and ARCOMSTR.

These variables are not necessary to create in case of initializing these others CCCOM, CXXCOM, ASCOM, LINKCOM and ARCOM. The variables above are set so in this case there is additional info.

To adapt to SCons, is necessary initialize SCons variables for each tool. The tools to be initialized are in compiler C, assembler, linker and archiver. it's necessary configures each tool depending on the compiler needs, in all compilers I need to set the 4 main tools, I create a private method for each tool. So, in each method, first I create an object of *compilerConf* to insert the configuration suitable. To finish, I return the instance with all configuration set.

In *section 4.1.* I explain each SCons variable the meaning and the purpose. So, I only explain in this section the variables related to Tasking compiler with an example:

7.2. Project Configuration to use a Tasking Builder and Tasking Compiler (YAML)

All the project configuration information is defined in the BCPDF, which is a YAML format file and divided in the following configuration sections.

The YAML is really similar to the virtual compiler, so, it is not necessary to give more details because it has already been done previously. The only difference are the files to be executed, because are specific for each compiler. These files are: compiler C, compiler ASM, archiver and linker (.exe and .ld). Also, the defines and all flags are different between YAML files.

In asflags, this compiler has the field named, *INFO.* This configuration is described in the Manual of each compiler. The compiler describes the information format.

7.3. Builder Test for Tasking Compiler

To check if this builder/compiler works properly I must do a series of tests. So, I do the same tests than the *IAR* Compilers. The difference between them is the content of the Compiler Call, Compiler flags injection and the defines and includes injections. It is explained in *sections 6.3*.

7.3.1. Test to check Compiler Call (executable)

In this section is necessary to check if the command line is correct, so to check it, is necessary carefully observe the beginning of the command line. In the Tasking compiler it should appears the *cctc.exe*. The parameter *-o* indicates that the following part is an output, and then the path of the two output files: *main.o* and *main.c*.

```
C:\Projects\Learew\tools\Tasking\TriCore_v6.3r1\ctc\bin\cctc.exe -o os\main\_out\main.o os\main\main.c
```

Figure 27. Debugger output Compiler Call (executable) in Tasking compiler.

7.3.2. Test to check Compiler Flags Injection and the Order.

In this point, I must solve if the flags and options described in the YAML file is injected properly in the command line, in the correct order. The format is important to the injection, so I check with the output files the result. The output file is a text file with all information.

```
--create=object --core=tc1.6.2 --iso=99 -O2 --eabi=BCFHNSW -AGKpvX --tradeoff=2 --fp-model=1 --switch=auto
```

```
--integer-enumeration --default-near-size=0 --global-type-checking --debug-info -Wa--list-format=1 -Wa--optimize=gs
```

Figure 28. Debugger output to check Compiler Flags Injection in Tasking compiler.

Compiler flags should be the same than YAML file with the same order that is described in, I check it and it matches.

7.3.3. Test to check Defines and Includes Injection.

Also is critical if the includes and defines are putting wrong in the command to get all necessary information for the building. It is possible to check because there is a text file with all information. As IAR compiler, includes starts with *-I* and defines with *-D.*

<u>Includes</u>

```
-I. -Ilib -Ios -Ibsw\srv -Isgn -Ibsw\ecual -Ibsw\mcal
```

Figure 29. Debugger output to check Include injection in Tasking compiler.

Defines

```
-DTGT_S32K3_IAR840 -DC_DERIVATIVE_TC397XE -D_GNU_C_TRICORE_=1 -DTRIBOARD_TC38XX -DLEAR_OS_USED -DAUTOSAR_OS_NOT_USED
```

Figure 30. Debugger output to check Defines injection in Tasking Compiler.

Includes and Defines should be the same than YAML file that is described in, I check it and it matches.

There are three includes added by compiler, are predefined includes: *sgn*, *bsw/ecual* and *bsw/mcal*, this is a default process that it is necessary to work properly.

8. Implement Builder to Hightec Compiler

HighTec EDV System, a privately held company since its inception in 1982, stands as the world's largest commercial vendor of open-source compilers. Presently, HighTec is part of Infineon, the same company than Tasking compiler.

With an unwavering commitment to independence and the provision of dependable, secure tools for embedded software development, HighTec guarantees cutting-edge solutions for future endeavors. Their C/C++ compiler boasts portability and timely availability for the latest chip revisions across supported architectures, often preceding general release.

HighTec's certified multi-core real-time operating system (RTOS) delivers unparalleled levels of data protection, functional safety, and robustness. The company offers comprehensive development, training, and consulting services covering areas such as performance optimization, functional safety, porting from single core to multicore systems, and integration into the AUTOSAR environment.

Backed by an innovative and dedicated team committed to quality, HighTec fosters a passion for their work, striving to empower customers in sustaining and advancing their market positions. Notably, within Lear, HighTec plays a pivotal role in compiling Tricore Family microprocessors, showcasing its significance in embedded systems development.

8.1. Implement Specifics Aspects for this Compiler

The structure of this compiler is the same than the *Tasking* Compiler, but the files to be executed are not, and the settings as expected neither. For this reason, I must create a new builder for this and each one of them. So, I do not repeat the content. For more information, go to *section 7.1.*

This is an example of *Hightec* initialization:

```
class hightec_init (base_init):

    builder_name = "hightec_cmp"
    builder_description = "Hightec compiler"
    compilerHome_env_Var_Name = 'HIGHTEC_HOME'
    tool_path = os.path.dirname(os.path.abspath(__file__))
    _verbose = False
```

Figure 31. Initialization of Hightec Compiler.

It should be remembered that the constructor is the same in all compilers, the changes are the different file to be executed for each tool:

```
def __init__(self):
    super().__init__(hightec_init.builder_name,
                     hightec_init.compilerHome_env_Var_Name)
    searchPaths = ['bin', 'path']
    self._AddCompilerTool(Compiler_Tool(name='tricore-gcc',
                                        extension='.exe',
                                        searchPaths=searchPaths,
                                        compilerConf=self.__CompilerC_Conf()))
    self._AddAsmCompilerTool(Compiler_Tool(name='tricore-as',
                                           extension='.exe',
                                           searchPaths=searchPaths,
                                           compilerConf=self.__CompilerASM_Conf()))
    self._AddArchiverTool(Compiler_Tool(name='tricore-ar',
                                        extension='.exe',
                                        searchPaths=searchPaths,
                                        compilerConf=self.__CompilerArchiver_Conf()))
    self._AddLinkerTool(Compiler_Tool(name='tricore-gcc',
                                      extension='.exe',
                                      searchPaths=searchPaths,
                                      compilerConf=self.__CompilerLinker_Conf()))
```

Figure 32. Constructor of Hightec Compiler.

I focus my attention on the specific part. General part is the same for every compiler, and I don't change it. In this compiler there are 4 main tools: Compiler C, assembler, linker and archiver. Each tool needs to be configured correctly and specifically for different compiler. For each compiler, I need to specify the name of each executable file, the extension, the path and an instance of Compiler_Tool class with information and configuration of each tool, to puts inside this object the configuration, but first I need to manage the license.

8.1.1. *Builder Initialation to adapt to SCons*

To adapt to SCons, it is necessary initialize SCons variables for each tool. The tools to be initialized are in compiler C, assembler, linker and archiver. It is necessary configures each tool depending on the needs of the compiler. In all compilers I need to set the 4 main tools, I create a private method for each tool. So, in each method, first I create an object of *compilerConf* to insert the configuration suitable. To finish, I return the instance with all configuration set.

In *section 4.1.* I explain each SCons variable the meaning and the purpose. So, I only explain in this section the variables related to Hightec compiler. In this compiler I don't initialize the variables named previously in the other compilers to check if it works properly. As result I can confirm it, is extra info. This is an example of Hightec tools initialization:

8.2. Project Configuration to use a Hightec Builder and Hightec Compiler (YAML)

The YAML is really similar to the others. For this reason, it is not explained in more detail. The only difference are the files to be executed, because they are specific for each compiler. These files are: compiler C, compiler asm, archiver and linker (.exe and .ld). Also, the defines and all flags are different between YAML files.

Hightec is the only compiler with object copy executable and in *cflags,* (compilation options, for a safety compilation) has two default options. The reason is because is described in the compiler manual.

8.3. Builder Test for Hightec Compiler

To check if this builder/compiler works properly I must do a series of tests. So, I do the same tests than the *IAR* Compilers. The difference between them is the content of the Compiler Call, Compiler flags injection and the defines and includes injections. It is explained in *sections 6.3.*

8.3.1. *Test to check Compiler Call (executable)*

In this section is necessary to check if the command line is correct, so to check it, it is necessary carefully observe the beginning of the command line. In the Hightec compiler it should appears the *tricore-gcc.exe.*

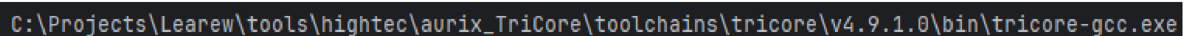

Figure 33. Debbuger output Compiler Call (executable) in Hightec compiler.

8.3.2. Test to check Compiler Flags Injection and the Order.

In this point, I must solve if the flags and options described in the YAML file is injected properly in the command line, in the correct order. The format is important to the injection, so I check with the output files the result.

```
-c -ansi -ffreestanding -fno-short-enums -fpeel-loops -falign-functions=4 -funsigned-bitfields -ffunction-sections
```

```
-fno-ivopts -fno-peephole2 -mtc162 -std=iso9899:1999 -nostartfiles -Wundef -W -Wall -Wuninitialized -mcpu=tc39xx -g2 -O1
```

Figure 34. Debugger output compiler flags injection in Hightec compiler.

Compiler flags should be the same than YAML file with the same order that is described in, I check it and it matches.

8.3.3. Test to check Defines and Includes Injection.

Also is critical if the includes and defines are putting wrong in the command to get all necessary information for the building. It is possible to check because there is a text file with all information.

<u>Defines</u>

```
-DTGT_TC39X_HT491 -DC_DERIVATIVE_TC397XE -D_GNU_C_TRICORE_=1 -DTRIBOARD_TC39XX -DLEAR_OS_USED -DAUTOSAR_OS_NOT_USED
```

<u>Include</u>

```
-I. -Ilib -Ios -Ibsw\srv -Isgn -Ibsw\ecual -Ibsw\mcal
```

Figure 35. Debugger output to check Includes and Defines in Hightec Compiler.

Includes and Defines should be the same than YAML file that is described in, I check it and it matches. There are three includes added by compiler: *sgn*, *bsw/ecual* and *bsw/mcal*, this is a default process that it is necessary to work properly.

9. Switch to automatize Process

Each project uses a different compiler so the system must by switch for the correct builder

I implement a switch to automatize the selection builder process, because the new core of build chain has all builders for each compiler inside. I need a method to switch to a specific object. It is interesting to speed up it.

In each build chain project definition file (.yaml), there is a field named *builder_type*, in this variable there is the name of each compiler and is the field used to know which compiler is needed depending on the case. In Python there is a variable type named dictionary, which is like a list in the sense that it is a collection of objects. Dictionary is a data structure, generally known as an associative array, that store data values in key:value pairs, dictionary is ordered, changeable and do not allow duplicates, can be referred to by using the key name, and the value/data type can be anyone. Dictionaries are mutable, dynamic, can grow and shrink as needed and can be nested, a dictionary can contain another dictionary, list or the type that its preferred. Dictionary elements are accessed via keys. In these keys I put the name of each compiler, and the elements are methods. For older Python versions, build chain uses a dictionary implemented in the *switch_Builder_engine_Dict* class. Depends on Python versions or SCons version I can use different methods, since Python 3.12 there is a switch method called *match*, but in 3.7. versions there isn't and I've implemented it with dictionary.

To implement the switch, I've based on Major Version, Minor Version and Patch Version format, this format is called Semantic Versioning:

Figure 36. Format version.

To implement the switch using this formatting version I have used the library *Version* from *version_parser*. This library helps me to separate version number in Major, Minor and Patch, to then check the version used. It is implemented in the *__init__*.

These are the classes used to switching:

Switch_base: This class is the father of the other ones. The constructor is empty because I'm not interested to create any object of this class. But inside, there are methods to create instance of each compiler created and inject to the build chain.

This is the main code:

```
class switch_base():

    def __int__(self):
        pass

    # -------------------------------- PROTECTED ----------------------------------

    def _iar_instance(self):
        """
            Function to create an instance or IAR compiler

            :return: iar_init, iar_user_library
            :rtype Any, Any
        """
        return iar_init.iar_init(), iar_user_library.iar_user_library()

    def _tasking_instance(self):
        """
                Function to create an instance or IAR compiler

                :return: tasking_init, tasking_user_library
                :rtype Any, Any
            """
        return tasking_init.tasking_init(), tasking_user_library.tasking_user_library()
```

Figure 37. Initialization of switch_base class.

```
def _Default_Builder(self):
    """this function pause the execution."""

    raise Exception('ERROR: Does not exists this Builder!!!')
```

Figure 38. Method to manage a default builder.

_Default_Builder() is used to treat some possible error if doesn't found some builder type in build chain.

Switch in Python 3.7.

switch_Builder_engine_Dict: This class is the son of *switch_base*, the constructor is also empty, because I'm not really interested to create any object of this class, like *switch_Builder_engine_switch*. The purpose is to create a method to initialize each compiler since a dictionary that selects it depending on the *builder_type* in the YAML file. The method is also called *switch_Builder()*.

```python
class switch_Builder_engine_Dict(switch_base):

    def __init__(self):
        pass

    def switch_Builder(self,builderType):
        """
            Function to switch different builder tools.

            :param String builderType: builder tool name.
            :return: the compiler initialized and its user_library to be builded.
            :rtype Any
        """
        builder_init = {'vcmp': super()._vcmp_instance,
                        'hightec': super()._hightec_instance,
                        'iar': super()._iar_instance,
                        'tasking': super()._tasking_instance,
                        'DEFAULT': super()._Default_Builder}

        builder_callback = builder_init.get(builderType, super()._Default_Builder)

        return builder_callback()
```

Figure 39. Main code of Switch_Builder_engine_Dict Class.

<u>Switch in Python 3.12.</u>

switch_Builder_engine_switch: At newest Python versions there is a function called *match* and is method like the switch in C programming language, more efficient, and I will modify this method to updated version.

This class is the son of *switch_base,* the constructor is empty. I'm really interested to create a method to select depending on the *builder_type* in YAML file. The method is called *switch_Builder().*

Each method checks if *builderType* matches a key, then, call constructor method of this matched builder compiler initialization and user initialization, following returns these instances created previously. These instances are injected to a list of 2 positions (tuple) and the variable *builder_init* get the list created. Is the same than dictionary, but in a dictionary, I must call the *get* method.

This is the method to automatize the process to select a compiler builder depending on options in the YAML file introduced in the command line.

10.Optimization Software

To update the Build Chain V6 to a Build Chain V7 I must consider simplify all build chain process: since configuration till depuration of errors.

I want to improve performance of Build Chain, in terms of scalability, maintainability, compilation time, execution time…

I do not optimize as such since I do not reduce the code and simplify it.

I generate an efficient architecture so that it is scalable.

Finally, I create a package using the script *build_package.cmd*, *setup.py* and *about.py*, in this package there are all files/classes/methods that I created for Build Chain V7.

<u>testVersions.cmd</u>

In this file I create the same lines that executes the debugger for each compiler and Python version, then creates a text file with the result. This file is a Command line script (.cmd) named *testVersions.cmd* to automatize process and check all Python and Scons versions in all compilers simplified form, only executing this file, so without that I had change the interpreter in PyCharm and it's slower and inefficient. The test result is writing in a text document (.txt) named *report_test.txt* to simplify and clearer. This is really important when I was developing Build Chain V7 because this allows me to reduce the testing and developing time, than, executes and compile each compiler for each Python versions.

Result:

```
OK: python37 with Virtual Compiler
OK: python312 with Virtual Compiler
OK: python37 with Hightec Compiler
OK: python312 with Hightec Compiler
OK: python37 with Tasking Compiler
OK: python312 with Tasking Compiler
OK: python37 with IAR Compiler
OK: python312 with IAR Compiler
```

Figure 40. Result of test versions.

About.py

In this file I put information about the project, it is the source where it should be packaged. Name, version, authors…

This file is used to create the final Python package.

Setup.py

This file is used to create the final Python package. It shows to Python how must be created the package. Here used *about* file to request the information about the project. This is a full example:

```
import os.path
from setuptools import find_packages, setup


about = {}
package = 'lear_scons_builders'
here = os.path.abspath(os.path.dirname(__file__))
with open(os.path.join(here, package, 'about.py')) as fp:
    exec(fp.read(), about)

# Get the long description from the README file
# with io.open(os.path.join(here, 'README.rst'), encoding='utf-8') as fp:
#     long_description = fp.read()

long_description = r"Build Chain V7.x package"
name = about['__name__']
version = about['__version__']
setup(
    name=name,
    version=version,
    description=about['__description__'],
    long_description=long_description,
    url=about['__url__'],
    author=about['__author__'],
    author_email=about['__mail__'],
    packages=find_packages(),
    include_package_data=True,
    #packages=[package,],
    package_data={
        '':['./Tests/*.*']
    },
    install_requires=[
    ],
    Python_requires=">=3.7.0,<4",
)
```

Build_package.cmd

This file depending on the file *about* and *setup* creates a compressed folder with all information packaged. In my case I compressed the file *lear_scons_builders.*

This is a full example:

```
@echo off
set PYTHON_EXE=%~dp0\python_env37\Scripts\python.exe
pushd "%~dp0"
del dist\*.gz 2>nul 1>nul
%PYTHON_EXE% setup.py sdist --formats=gztar
popd
if NOT [%1]==[no_pause] timeout 60
```

Figure 41. Full code of Build_package.cmd file.

11.Test using New Versions of Python/SCons

I do the project with Python3.7 but to know how it works and if there is any notable improvement I test it with version 3.12 due to the lack of support from the Python organization.

In Build Chain V6.x SCons now a days is working the version 4.1.0 but to try Python3.12 I've updated to the 4.7.0 version. In the process to try new Python and SCons versions, the best option is to automatize the testing process, to be more efficient in terms to develop process. In this line, I've created two Python environment with the Python versions 3.7 and 3.12. In each one of them there is a different SCons versions, the 4.1.0 and 4.7.0 respectively. I've updated some packages to be compatible with new SCons and Python versions.

To be compatible with new Python/Scons versions, I have updated third party packages:

Package	Old version	New version	Reason/Description
Scons	4.1.*	4.7.0	To improve performance.
Pywin32	224	306	It's necessary to work properly, be compatible in SCons 4.7.X.
Lxml	4.3.3	5.1.0	It must be updated to create a Python environment in new Python/SCons versions.
PyYAML	3.11	6.0.1	To read YAML files correctly and avoid format errors.
Parser_version	-	1.0.1	I must install this packet to compare Python versions to automatize the process to select a switch.

In the future, it will be unnecessary automatize process because there is running in only one Python and SCons versions.

12. Documentation

The software projects are required in a lot of cases document the code used. In my case that has not been an exception and I have done different manuals and formats. I must redact a Develop Manual, Doxygen Manual and MarkDown format.

12.1. Manual Develop/Developer Guide LearSAR_v7 (Build Chain)

A software developer manual is a valuable resource that provides guidance, instructions, and reference material for software developers. I've done a manual to give some information to the Lear developer that is working in build chain department, so this manual is more explicit than user manual, because theoretically the engineer who deals with this process knows how it works internally a build chain. So, I explain step by step all process to add a new compiler builder and then start the build chain with all parameters and variables needed.

Also, in this manual I've specifically explained which files need to be modify, to add a new compiler builder in the build chain and then configure the debugger.

This manual is an internal document for Lear Company, and therefore confidential.

12.2. Code Documentation

In my project has been necessary to document the source code, because to do a software project, there is a team of computer engineers who works together to achieve commons objectives, the purpose it is that all members in team must understand code, additionally, it needs to be understood by users who are not originally involved in the project. The best option is to document code, and clearly explain methods, class, properties… I found a way to document source code automatically, to facilitate the process that for engineer can be bulky and heavy.

12.2.1. Automatize Redaction with Doxygen

I use a program named Doxygen that can make the source documentation for different programming languages (C, C+, C++, C#, Objective-C, Python, PHP, Java, IDL, D, Fortran and VHDL). Documentation is written in different formats like HTML, LATEX, Man pages, RTF (only looks nice with Microsoft's Word), XML and DocBook. Indirectly supports Compiled HTML Help (a.k.a. Windows 98 help), Qt Compressed Help (.qch), Ecplise Help, XCode DocSets, PostScript and PDF.
This program is useful because it generates documentation automatically, you only must adjust some parameters in the program interface. Also, can make class diagrams.
This is the reason why I write a manual, so, could be important for the software department. It is necessary to set it up properly.

12.2.1.1. Doxygen Manual

Doxygen is a tool that generates documentation from source code. It is useful because it generates automatically in different formats like, MarkDown, HTML, PDF… Also supports different programming languages like, C, C#, Python…

So for my project and different projects in the company can be really interesting.

I've done a manual to explain how download, install and configures doxygen to create automated documentation about source code.

12.3. Documentation with MarkDown

Markdown is a streamlined markup language designed for crafting formatted text via a basic text editor. As a lightweight markup language (LML), it boasts a straightforward syntax, making it easier to write and read in any standard text editor. LMLs are particularly useful in scenarios where unprocessed documents need to be readable, as opposed to only the final rendered output.

Markdown's primary objective is to optimize readability and ease of publication in both its raw and rendered forms. It draws inspiration from existing conventions for marking up emails in plain text. Markdown is distributed under a BSD license and can also be found as a plugin in various content management systems (CMS).

Originally implemented in Perl, Markdown has since been translated into a wide array of programming languages, including PHP, Python, Ruby, Java, and Common Lisp.

To convert to MarkDown format, I use the tool: https://pdf2md.morethan.io , it is easy and quickly.

There are different translators that catch the document and converts it to MarkDown format, in this converter is necessary a PDF file.

13. Business Values

This project has direct repercussions for the company so improve performance in terms of configuration time, compilation time or maintainability, is important. I check Build Chain V7.x in a real project, named: Lear AutoSAR CLAS. In Lear there isn't any project that works with Python 3.12. so, I have checked with Python 3.11. The result is encouraging, cause the build time is reduced approximately a 5%:

- Python 3.7. → Compilation Time = 1303,7 seconds = 21,7283 minutes,
- Python 3.11. → Compilation Time = 1240,41 seconds = 20,6735 minutes

To obtain the reduction compilation time % you should do this operation:

$$Reduction\ Compilation\ Time\ (\%) = \left(1 - \frac{1240{,}11}{1303{,}71}\right) \cdot 100 = 4{,}855\ \% \cong 5\ \% \quad (1)$$

The Reduction Compilation Time in terms of time is 1'06 minutes, so if there is the needed to compile 60 times in a day, Build Chain V7.x reduces Compilation Time 63'6 minutes, so, could be interesting by Lear to be more efficient to do some project:

$$Reduction\ Build\ Time\ (min) = (21{,}73 - 20{,}67) \cdot 60 = 63{,}6\ min \quad (2)$$

There is another notable improvement, the configuration time. To add and configure a new compiler in the Build Chain V6.x more less spent 40 hours, 1 work week. The key to reduce time it is separate code in general and specific part. But, in Build Chain V7.x to do the same process, I sent 8 hours, 1 workday. So, I reduce the configuration time 80 %.

- Build Chain V6.x → Configuration Time = 40 hours
- Build Chain V7.x → Configuration Time = 8 hours

To obtain the reduction configuration time % you should do this operation:

$$Reduction\ Configuration\ Time\ (\%) = \left(1 - \frac{8}{40}\right) \cdot 100 = 80\ \% \tag{3}$$

These parameters are not more relevant because Lear introduce a new compiler one time every year, so in long-term is more attractive the improvement of compilation time.

But the most important improvement it is the Build Chain maintainability, this means that is easier to check and solve errors. Thanks to separate code in two big groups: common and specific, I should only review the specific part, and this is simpler to maintain.

14. Future Steps

In the software world, one tool, application, program, process… is in continuous development, and this software tool Generation is not an exception. Can be implemented some improvements in the future and there are some steps to do:

- Continuous Maintenance: This is the main work to do in the future because this process needs to be maintained, there are problems to solve cause the bad operation of user.
- Add news compiler in build chain: This tool is created to support more than one compiler, this version is an updated that reduce configuration time considerably, so maybe there is a need to create and add a new compiler in build chain.
- Fix an issue found for work mates: Sometimes can generate some error and it should be solved by build team.
- Teach to my work mates about Build Chain V7: Even though/Although I do a user and develop guide to instruct people about build chain. Maybe they can have a doubt because I didn't take that into account. The best option is solving the problem one member of the team that works on the update.
- Research other ways to improve it like automatize and simplify more Build Chain with switches used in Python 3.12: One of the goals of an engineer is think about how to improve situations, systems, process… and this is not an exception, so can be upgraded to a better new version. For example: new Python versions, new SCons versions, automatize the process to generate some documents…
- Certificate Build Chain V7.x: This process should be performed by my tutor, because is necessary experience in sector and needs responsible to credit the good functionality of software to then sell it at customers.
- Add new features: New features are pending to be added to the build chain, like package management or carlines compilations, among others.

15. Conclusions

To conclude, I can say that this project was a challenging job for me, because I have learned new programming language and computer science concepts that I haven't studied in the degree. But it was really, gratifying finish the project a see that the works that I do, it's a big learning, for example: Python programming language, so I didn't use it during my university studies, but it's really similar than Java, to debug Python I use PyCharm software as IDE; to us SVN software, it's a control versions really similar at GitHub, Visual Studio Code to do diagrams with plugin *drawio* like Sequence diagram or Conceptual/Overview diagram; to finish, Visual Studio to do Class diagram. Following, I like to talk about initial goals and if I achieve and how:

1. In terms of scalability, it's done properly, so I separate generic and specific code, thus facilitating the configuration compiler, in terms of scalability is perfected and improved.
2. The intern quality in build chain, for the same cause, separating the generic and specific parts, the intern quality increase considerably and the maintenance is easier, so you only have focused on the specific part, in the generic is always the same for each compiler.
3. To adapt at development process tools of Lear. This job I've developed with my teammates, because they must teach me how is these tasks: reports, process…, since to know it, I can adapt.
4. Build Chain team wants to automatize the selection of builders, to make it possible I do first in Python 3.7. a dictionary with the methods and the key word expressed in the YAML file. In the Python 3.12 I can implement a switch function, so it's in the newest Python versions. this switch is responsible for automating the process. It's explained in detail at *Point 7*.
5. I proposed simplify, do clearer and to generalize code. These goals are achieved by separating the code in the two parts named previously.
6. To manage the license better, the secret it's create a method inside the builder class, in the specific part that is responsible for finding it and executing it. It's important because without license the compiler cannot build the program.
7. It's necessary perform a strong documentation, because the changes and updates must be written to be consulted in case there is a doubt for user or developer. Also, I document how can create source code documentation automatically with *Doxygen* software.
8. I check if the building or execution is faster and I can improve it. So, I've calculated the execution/build time and I can conclude that the Build chain does not take excessive time else it is the compiler. The compiler is the one that consumes time.
9. Build Chain V7 is compatible with new Python and SCons versions, so I can conclude that is a reason to increase the performance of build chain.
10. I unified to a packet all source code generated but I haven't certificated yet.

I would like comment that all the proposed goals were satisfactorily met.

Finally, I have verified that the project is functioning as expected and the code is accurate. Subsequently, I consolidated all the code into a package to then, be certificated by my technical tutor.

The Build Chain V7 has improved significantly, in terms of configuration time, build time and maintainability. I can verify the improvement.

16. Webgraphy

Doxygen: Home page. Link: https://www.doxygen.nl/

Hightech Compiler: Home Official Hightec compiler. Link: https://hightec-rt.com/

Wikipedia: Home page. Link: https://www.wikipedia.org/

Python: Home page. Link https://www.Python.org/

SCons: Official Home page. Link: https://scons.org/

Tiobe: Official Home page. Link: https://www.tiobe.com/tiobe-index/