BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION SOFTWARE ENGINEERING

DIPLOMA THESIS

Program Comprehension in Embedded Systems using FSM

Supervisor:
Assoc. Prof. Vescan Andreea

Author:
Epure Lucian

2021

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INGINERIE SOFTWARE

LUCRARE DE DISERTAȚIE

Program Comprehension in Embedded Systems using FSM

Conducător științific:
Conf. Dr. Vescan Andreea

Absolvent:
Epure Lucian

2021

ABSTRACT

Embedded systems are everywhere today, from the regular smartphone to the specialised industrial computerised machines. Even though they are very common, development of embedded software is very complex and many times relies on low level programming. One of the hardest parts of software development and enhancement is code comprehension. Developers need to quickly understand the existing systems in order to extend them and doing that with low level syntax may prove complicated

After researching several approaches I decided to investigate the use of Finite State Machines as a mean of increasing the program comprehension of embedded systems. In my thesis I started from an existing research paper and implemented an algorithm capable of extracting Finite State Machines from files containing C source code.

The application receives as input a source file, it parses it to determine if it contains C code, as well as structures that might suggest the existence of Finite State Machines. If a loop without a constant trip count exists and contains a state variable than starting from that state variable, the algorithm is able to extract the states. Knowing the states, the systems then extracts the transitions using a constraint solver and symbolic execution. The result of the application is a set of FSMs which mirror the behaviour of the code. Each FSM consists of basic elements, states, transitions, initial and final state.

In order to validate how helpful Finite State Machines are for program comprehension, an empirical study was performed in which subjects were split into two groups and given a series of tasks. The first group did not have access to the generated Finite State Machines while the second did. Both groups had to solve code comprehension exercises on the source code and the results of both groups were compared.

The study showed that users from the group who had access to the FSM had better results to the exercises and performed the tasks in a slightly better time. These results are extremely relevant because they show that Finite State Machines that can be generated by a software system are helpful enough for embedded systems developers. The fact that Finite State Machines can be automatically extracted from C source code is also very important and can significantly improve the development time.

Contents

1 Introduction **1**
- 1.1 Importance of the research subject . . . 1
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 3
- 1.4 The structure of the thesis . . . 4

2 Theoretical background **5**
- 2.1 General study methodology . . . 5
- 2.2 Systematic literature review . . . 5
  - 2.2.1 Approach Methodology . . . 5
- 2.3 Planing the SLR . . . 7
  - 2.3.1 Necessity of the research . . . 7
  - 2.3.2 Defining the research questions . . . 7
- 2.4 Realization of the SLR . . . 8
  - 2.4.1 Search and selection process . . . 8
  - 2.4.2 Data extraction . . . 8
  - 2.4.3 Data Synthesis . . . 10
- 2.5 Results and analysis . . . 15

3 Application Development **16**
- 3.1 Key Concepts . . . 16
  - 3.1.1 Finite State Machine . . . 17
  - 3.1.2 Symbolic execution . . . 17
- 3.2 Requirements/Specification . . . 18
- 3.3 Analysis and Design . . . 18
  - 3.3.1 Use cases . . . 18
  - 3.3.2 Algorithms . . . 20
  - 3.3.3 Design diagrams . . . 22
- 3.4 Technologies and libraries . . . 25
  - 3.4.1 Programming language . . . 25
  - 3.4.2 wxWidgets . . . 25
  - 3.4.3 Klee . . . 26
- 3.5 Implementation . . . 26
- 3.6 Testing . . . 30
- 3.7 Deployment . . . 31

4 Experimental study **33**
4.1 Study methodology 33
4.1.1 Study participants 33
4.1.2 Study materials 34
4.2 Results 34

5 Conclusions and future work **43**
5.1 Conclusions 43
5.2 Contributions 44
5.3 Future research 44

Bibliography **45**

Chapter 1

Introduction

Embedded systems are surrounding us today and constitute more than 90 percent of the existing computing systems. They are everywhere, from smartphones, to credit cards, industrial computerised machines, and so on.

The development of embedded systems has been and will always be an important branch of engineering. However, compared to regular software systems the embedded technology is more connected to the hardware it is developed on. For embedded systems, the hardware limitations such as memory or power of computing are essential and the software must be written in a tight connection and tailored for the machine that it is designed for.

Program comprehension is an important aspect of the computer science field. Its main purpose is to implement technologies and methods aimed to increase the understanding of software programs and their design, as well as to improve and efficientize the code. There are many methods involved today in order to enhance program comprehension and many new approaches currently studied. One of these approaches suggests the use of Finite State Machines as an alternative, more accessible and easier to understand representation of the embedded code.

1.1 Importance of the research subject

As it was presented earlier, the research subject is one of the highest importance not only because the embedded systems are everywhere today but also because the code of embedded systems tends to be more complex and hard to follow than that of the "regular" software applications.

Even though embedded systems have been around for a while, some of the most promising directions of development include the automotive field [KTS18]. The computers inside the cars have several functionalities which have become essential for the proper functioning of the car and the safety of the passenger. Such functionalities are the assisted driving systems, the fuel and power management, passenger safety systems as well as the carbon emissions monitors and several others. These capabilites of the modern cars help not only the humans but also the environment and ensure the better functioning of the vehicle. In other areas [Oye15] presents

how important embedded systems are in the performance of computers and of energy metering devices. Such types of devices are being used in controlling most of the computing systems and are regulating the electricity and thus the power of the device.

Most of the embedded software is written in C and contains highly specialised functions for certain devices. These functions can become very large and hard to maintain over time. Also because they work in a close relation with the hardware, they have to make the best use of the provided resources. Thus the need for a solution that makes it easier for the developers to understand the code better in order to maintain it and even further improve it becomes crucial. Because there are many embedded systems which are part of life saving devices, it is very important that these systems work properly and are not failing under their own complexity.

Finite state machines are one of the most powerful ways to describe the dynamic way in which systems and their components work and interact. They are easy to observe and understand. An usual FSM (Finite State Machine) consists of states, transitions, initial and final state as well as the multitude of possible inputs.

The problem with FSM is that in general is not easy to obtain them from the source code. In object-oriented environments the existence of special design patterns facilitates the virtualisation of FSMs. In [vGB99], the authors present how trying to obtain a FSM from Object Oriented(OO) code can prove very usefull for understanding and better representing it.

My thesis aims to redirect the powerful capabilities of FSM representation in the direction of Embedded Software and create a tool that allows the generation of FSM from source code as well as a study to observe if the representation is helpful and in what proportion.

1.2 Motivation

My main motivation in researching this subject is the enhancement of embedded systems understanding. In my opinion, FSMs are one of the most easy representation of the flow in a software system. They are very clear and can express all dynamic behaviour, with multiple paths and branches the code can go into.

The embedded software has multiple uses in today's world and it is very important to have good mechanisms to better understand and improve the development of embedded systems. The investigation in [SQK18] presents an interesting approach in the direction of using FSM for embedded software. They propose a study in which they inspect different solutions for creating state machines based on embedded code, observing patterns, and posing different questionaries to test subjects.

Their approach is mostly theoretical and there is no software able to extract the finite state machines from code. In my thesis, I implemented the software able to do that and redo the study in order to observe how the results of a FSM extracting

software can help the regular users of the system.

1.3 Objectives

The main objective of my project is to perform a study to find out how helpful it is for users to understand an embedded software code if they are given a finite state machine representation of it. The paper [CSX+19] presents an algorithm after which a finite state machine extractor can be implemented. After researching different C code snippets, the authors describe some rules with how a finite state machine can be identified and extracted from the code. The paper provided me a good starting point to follow in the development of my application.

- **Study how program comprehension techniques manifest with embedded software**

  During the research done for this project, it is essential to observe how embedded software is most usually analysed and how the finite state machines can help in doing that.

- **Separate the applications which can be FSMs from the reset**

  Not all software (embedded or not) can be reduced to the form of a FSM. This aspect should be considered in the development of the application, not only because it may take a lot of resources to get to a result but also because that result can prove to be misleading for the developer if is not well done. The structure of the code provides some clues for determining if it is a FSM or not and based on these clues the application should stop or continue

- **Obtaining the FSM**

  A FSM is composed of 5 key elements. The states, the possible inputs, transitions, the starting and finishing phase. The application should be able to receive as input the code snippet form an embedded systems, and generate the 5 element tuple as a response for any FSM found in the code piece. The results should be provided relatively fast and with a high precision. Even though the discovered FSMs do not reflect perfectly the input code they are at least guiding in the right direction for a better understanding.

- **Studying the results**

  After the application is done a study is performed. A series of code snippets will be provided and a questionnaire for test subjects will be created. The questionnaire will measure how fast and how well the subjects understand the code in both cases, with or without the help of the FSMs. It will be observed how efficient the helping FSM representation is for the users and how reliable as well.

Remark: The conducted Systematic Literature Review, the proposed approach for FSM extraction and the empirical study are under review at Journal Studia Informatica:

- Epure Lucian, Program Comprehension in Embedded Systems using FSM, Journal Studia Informatica, (under review)

1.4 The structure of the thesis

The thesis is split in several chapters:

- **Chapter 2**
  presents a theoretical study and a Systematic Literature Review(SLR) with the purpose of researching the program comprehension in embedded systems subject. The SLR was performed through a series of steps: Establishing the research questions, Performing the research by doing a database search, Combining the obtained results and Applying the final filtration. During the SLR, 30 relevant articles were found and in the end only 10 were selected and thoroughly analysed for the realisation of this thesis.

- **Chapter 3**
  presents the design and implementation of the system responsible of Finite State Machine extraction. The system consists of several components such as a Code parser, a state extractor and a transition extractor. The application receives as input a source file written and C and extracts Finite State Machines if there exists any.

- **Chapter 4**
  presents an empirical study that aims to determine how helpful finite state machines are for code comprehension. The experiment is described and the results are analysed. The participants were split into 2 groups and both were given a series of tasks meant to test the level of code comprehension. The first group had access only to the source code while the second also had access to the FSMs generated by the system. The results of the study showed that the group that had access to FSM performed both better and slightly faster thus the FSM proved helpful for code comprehension.

- **Chapter 5**
  presents the Conclusions of the thesis. An overview of the discussed research, the implemented application and the empirical study. The contributions are emphasized as well as some of the future directions of development for the Finite State Machine extracting application.

Chapter 2

Theoretical background

My thesis presents the problem of program comprehension in the context of embedded systems. Such systems are very relevant and widespread today thus making it crucial that certain program comprehension systems are in place and can be involved in the development of such software.

2.1 General study methodology

The methodology followed for studying the subject presented in this thesis followed a general pattern. A Systematic Literature Review (SLR) was performed. The process has three main steps. This approach has been presented by Kitchenham [Kit04] and consists of the following phases:

- **The planning phase.** During this phase, the main questions about the thesis's research study are established as well as the principal motivation for the project. With these in plan, the rules needed for performing the SLR are put in place.
- **The review conducting phase.** In the second phase, following the previously established rules, the research studies are searched and the relevant information is extracted.
- **The Reporting phase.** In the last phase, after the materials have been processed, a report is created. This report contains an overview of the studied materials and establishes a connection with the subject of the thesis.

2.2 Systematic literature review

2.2.1 Approach Methodology

The approach methodology of establishing the theoretical base follows a pattern presented in [Kit04] and is very useful for creating an information pool. From all this information a more clear way will be available and a better direction in my thesis as well

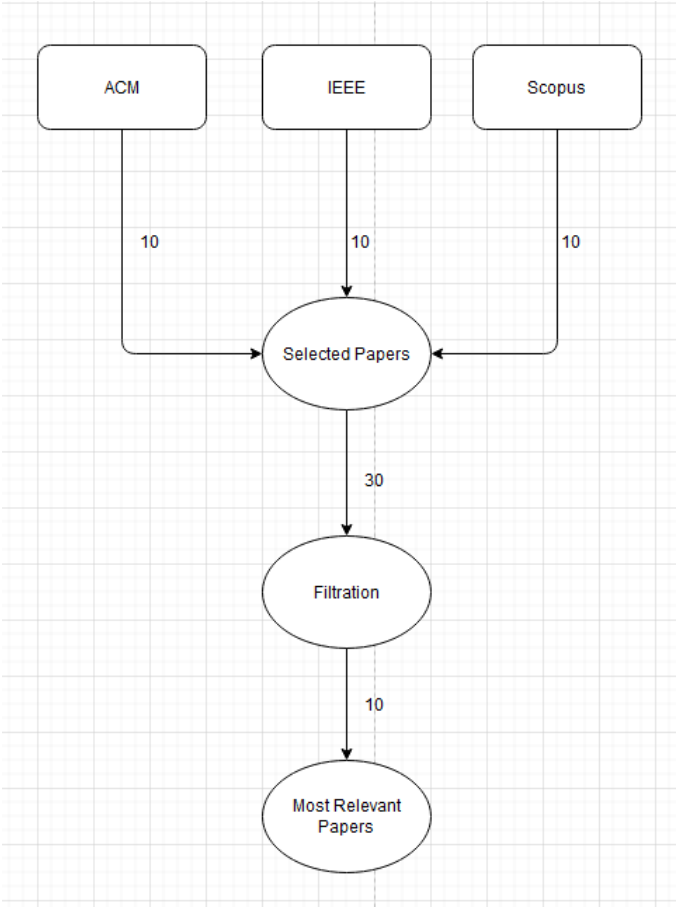

Figure 2.1: SLR overview

2.3 Planing the SLR

2.3.1 Necessity of the research

Embedded systems represent a big part of today computing systems and even though they do not receive such a big share of articles from the research community, they should not be left behind. In areas such as industrial development, there are numerous machines which work with embedded software. The programs created for embedded systems have a high degree of complexity and are developed by big teams and numerous software engineers. Thus, the need for program comprehension mechanisms has become an immediate issue.

Program comprehension needs to be considered every time a software system is built. The complexity to which some software systems can get up to, requires either the developers to plan more on how the software is built or to be able to accommodate specialised systems meant to enhance program comprehension. This will allow the software to be further developed and extended and also other developers to understand and get accustomed to it.

The embedded systems are different from the regular ones and can get up to high levels of complexity while needing special program comprehension tools. These tools should not encumber the running software and have to be a trade between the complexity of the tool and its efficiency.

My research tries to analyse the possible approaches that involve program comprehension and focuses on the implementation of finite state machines as a remodelling of the existing software in order to make the system easier to be understood by the developers and new engineers that have access to the code.

2.3.2 Defining the research questions

After an intense analysis of the thesis's subject, I established a set of questions that will help in a better literature review. The main questions extracted based on the selected subject are:

- How is the complexity of embedded systems different from that of regular systems?
- Which are the most popular program comprehension approaches in the direction of embedded systems study?
- Is the finite state machine representation of an embedded system problem a reliable and efficient method for enhancing the program comprehension?

With these questions in mind, I started to research the domain of my thesis and see different directions of study and development.

2.4 Realization of the SLR

2.4.1 Search and selection process

Database research

Starting from the established research questions in Section 2.3.2, I defined a series of keywords relevant for my thesis subject and began an extensive research through some of the most popular databases containing research studies and papers. Some of the keywords and phrases I used were: embedded systems, program comprehension, task mining, state machines, state extraction, analysis, and dynamic data analysis.

These were just a few phrases. The databases used for my research were IEEE-Explore, ACM and Scopus. They contained several articles which were relevant for my study.

Combining the results

Several of the researched papers were studying the same approaches and the same directions regarding program comprehension. After reading the abstracts, I quickly eliminated the ones containing irrelevant information for my study as well as the ones which duplicated the information and were getting farther from the main point of the thesis.

Applying the filtration

In order to further filtrate the information, the research questions helped in reducing even more the number of papers. After looking for the answer to each question, it was easy to eliminate the irrelevant papers, or to postpone their reading and comprehension. The highest importance was given to the papers trying to answer to the previously presented research questions or to at least be close to the subject of the questions.

2.4.2 Data extraction

I have extracted a total of 30 articles, they were selected from the aforementioned databases. I selected 10 articles from each database depending on my research questions. After studying more carefully the articles, I reduced the number from 30 to 10. The information presented in the abstract of the articles was enough to decide if the presented idea or approach is getting closer or farther from the main subject of my thesis.

Figure 2.2 contains a distribution over time periods of the articles and studies performed on the subject of Program Comprehension in Embedded Software and other related aspects. From the articles that I analysed during the SLR most of them were done after 2006 thus showing that this field of study gained more importance in the recent years.

Now, with a total of 10 articles, presented in Table 2.1, I was able to perform a comprehensive Systematic Literature Review consisting of several relevant approaches for my thesis subject.

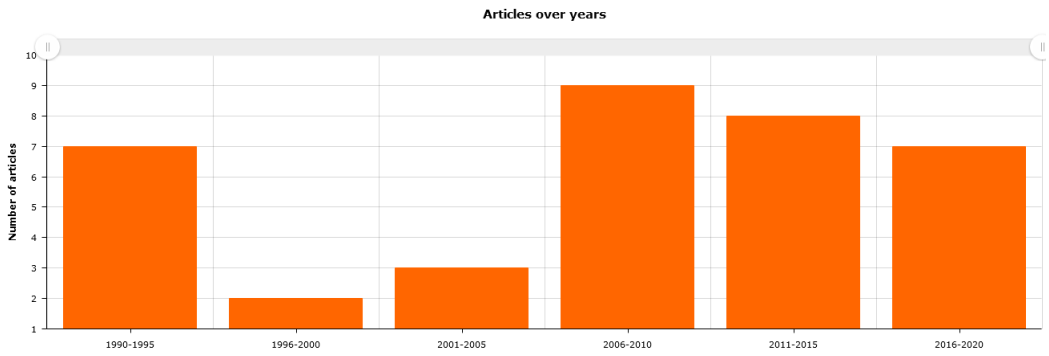

Figure 2.2: Studied articles year distribution

Id	Citation	Title/ Authors/ Publication	Year
P1	[WKTG97]	Facilitating program comprehension via generic components for state machines Johannes Weidl, Rene R. Klosch, Georg Trausmuth, Harald Gall Proceedings Fifth International Workshop on Program Comprehension.	1997
P2	[Kni02]	Dependability of embedded systems J. C. Knight Proceedings of the 24th International Conference on Software Engineering.	2002
P3	[TS02]	On selecting software visualization tools for program understanding in an industrial context S. Tilley, Shihong Huang Proceedings 10th International Workshop on Program Comprehension	2002
P4	[HLL10]	Understanding the complexity embedded in large routine call traces with a focus on program comprehension tasks A. Hamou-Lhadj, Timothy Lethbridge IET Softw.	2010
P5	[TCL+10]	From Applications, to Models and to Embedded System Code H. Tung, C. Chang, C. Lu, W. C. Chu, H. Yang 10th International Conference on Quality Software	2010
P6	[AFF12]	A Systematic Mapping of Architectures for Embedded Software E. A. Antonio, F. C. Ferrari, S. C. P. F. Fabbri Second Brazilian Conference on Critical Embedded Systems	2012
P7	[TVD12]	Supporting Program Comprehension Using Dynamic Analysis J. Trümper, S. Voigt, J. Döllner Second International Workshop on Software Engineering for Embedded Systems (SEES)	2012

P8	[ITF17]	Periodic Task Mining in Embedded Systems O. Iegorov, R. Torres, S. Fischmeister IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)	2017
P9	[IF18]	Mining Task Precedence Graphs from Real-Time Embedded System Traces O. Iegorov, S. Fischmeister IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)	2018
P10	[SQK18]	On State Machine Mining for Embedded Control Software W. Said, J. Quante, R. Koschke IEEE International Conference on Software Maintenance and Evolution (ICSME)	2018

Table 2.1: Article list

2.4.3 Data Synthesis

The efficiency of state machines and its usability in the domain of computer systems has been studied starting from many years ago. In an approach presented in [WKTG97] the problem discussed is that of representing state machines at a high level of abstraction by mapping the state machine concept to generic components, in order to increase software quality and program comprehension. The use of design patterns who were yet at the beginning of study was seen as helpful for the task at hand. Two "state" kind of design patterns were involved: the first is called "Harel State" and based on object oriented, and the second is called „generic Harel State Machine Engine" and is table-based. Using the "Harel State" pattern, machine states are encapsulated in object-oriented components, which leads to an easy representation of states, events, and state transitions in the source code, greatly increasing the understandability and readability of the source code. Using "generic Harel State Machine Engine, the state transition information is kept in a table and is modified by modifying the table entries. These tables actually represent templates that can be used in different implementations. This leads to efficiency in terms of space used and execution time. The nature of the design patterns is highly favourable for an increased re-usability and enhanced comprehension.

In later articles the focus for a better program comprehension was also switched to the efficiency of documentation. The problem of building a legacy software is described as one of the first steps to reuse is to re document the software system to help understand it. The best tools to understand industrial software are visualisation tools, creating a graph representation for example or other visual elements. In the study [TS02] a series of problems regarding the selection of better visualisation tools is discussed. The main issue being the use of only mainstream tools in detriment to academic tools that may be more well-equipped and have a higher performance for the task at hand. Tool selection was directly influenced by a num-

ber of corporation-wide policies: adoption costs, hardware platform, and software integration. In order to satisfy both the requirements related to the nature of the project and the corporate policies of the industrial partner, the evaluation of the candidate tools was made using and existing framework for testing the necessary capabilities, this framework being used for the first time on an real world embedded system. Many tools were rejected because they were academic tools and did not respect corporate policies, even though there are many differences between a tool that work is academically and the one that works in a company. After an experimental selection a visualisation tool, which was augmented to address the constraints imposed by the project's singular requirements, was chosen and tested.

Embedded software has become a necessity in our days and from simple devices like toasters to complex "devices" like humans it must be carefully designed in order not to malfunctions and produce considerable damage. The embedded software is strongly connected to the hardware that it runs on. From the fact that these two parts of the embedded system (software and hardware) influence actively each other, it can be deduced that the term "reliability" used only in relation to the software part of a system is not properly understood. [Kni02] emphasizes that safety-critical systems must be dependable. This feature is divided into six properties: reliability, availability, safety, confidentiality, integrity, and maintainability. In order to avoid possible mistakes in the implementation of the software of an embedded system, the method of formal notation of specification has been proposed, but this method requires a lot of effort and specification in natural language is much more popular. Verification by formal verification and model checking can be another option to avoid mistakes, but testing is the most used method of verification, although even this is difficult. In conclusion, the best way to avoid errors in an embedded system is by ensuring the dependability of that system, even if the process is complex.

The best way to understand the code is by studying its traces, this will provide an insight in the functionality and the structure. Examination of the execution traces can be difficult but also very useful, many trace analysis tools have been developed over time. Even so, the analysis of these can be very complex, which is why [HLL10] proposes a set of metrics that measure the relevant properties of execution traces in order to reveal the effort required to analyze the trace of routine calls. The metrics discussed are: call volume metrics – indicating the complexity of a trace, component's volume metrics – evaluate the number of system components invoked in a trace, comprehension unit volume metrics – measure the content of a trace taking into consideration all kinds of repetition and pattern-related metrics – evaluate the number of patterns in a trace. Four different Java systems were analyzed in this paper in order to apply the mentioned metrics. The results of the analysis showed that when the repetitions are removed, the trace size can be reduced by up to 46 percents compared to the original size. At the same time, the number of comprehension units in the trace was higher than the number of distinct methods invoked, which leads to the conclusion that there is still room to reduce the size of the trace. Reducing the size of the trace is important in order to reduce its complexity, because the lower the complexity is, the better the understanding of the system increases.

Sometimes for a better implementation of the code the starting point can be mod-

eled first and the code generated after. This idea has been presented in [TCL+10] through a system that is able to start from an UML diagram (one of the most widely used and accepted system representations) and generate the code necessary for an embedded system, through an Object Oriented transformation. This is significantly reducing the chances of inserting human errors in the system. Software model components and correctness are essential in embedded systems, thus an approach that uses UML as the system's input model in the software framework proves to be very efficient from the point of view of an enhanced program comprehension and code readability. The case study was done on an ordering system for a restaurant that works with Bluetooth technology. Using the UML modeling method, the paper describes how the system model, process chart, and system category diagrams were created. In the experiment, Miro Samek's quantum programming law was mainly used to execute the code via the UML language. The paper demonstrates that there are major differences between the traditional development process and the method using UML: for example for the traditional method it took 208 lines of code to program a PCB and the time spent on it was 18 hours, while for the method using UML only 128 lines of code were required in just 6 hours. The presented experiment demonstrates that UML modeling increases productivity in developing embedded systems and helps in having a more organized and well planed approach.

For developing good and reliable embedded software systems and specially critical software systems, the use of the representation of software architectures enhances the development process helping in the creation of secure safety critical embedded systems. This paper [AFF12] presents an approach that performs a mapping through the research area aimed to find papers based on a search string. It then applies several criteria for inclusion and exclusion, reducing thus the number of papers. The criteria are defined by the embedded software engineers. After obtaining a set of research papers from multiple databases, the approach applies a series of additional "filters" in order to group the studies: Modelling Techniques (Formal, Informal, Semi-formal), Architecture Type. Some of the most relevant formal techniques obtained were Finite State Machines, Petri Nets and StateCharts. From the semi-formal and informal categories, UML is the most popular and also SysML is mentioned. The architecture type that came up in the majority of the investigation studies is the Domain specific architecture. The paper conclusion states that by using these findings, a development process for critical software embedded systems may be implemented and that this paper is part of a larger study that aims just that.

After the development of a system (and especially an embedded system), the maintenance comes into play and is considered one of the most important cost factors. In order to perform program comprehension properly on embedded systems, different approaches than those for regular systems must be involved. The paper [TVD12] presents such an approach and proves its feasibility while also showing the usage through an experimental study. The software-based tracing concepts differ in embedded systems, systems run on limited resources, tracing must be done in the same process, debugging faces limited connectivity, and the tracing must be independent of the operating system. The presented technique combines a series of different approaches, the final program follows a series of steps: the instrumentation, which prepares the tracing techniques that will be used. The trace process which is

performed while the program is running and manipulates the addresses of the data. The analysis process is responsible for saving the tracing and observing the results through another system. The paper performed a case study on a postage system, multi-threaded software that contained approximately 2 million lines of code. In order to improve the performance of the maintenance, the tracing was done and the results showed what the developers wanted while also how small the impact of the tracing program presented in the paper was on the performance of the system.

In order to involve reverse engineering on such systems in need of better understanding, the mining of tasks seems like the best approach due to the fact that most systems are based on a set of tasks that are performed repeatedly. The paper[ITF17] proposes the use of PeTaMi. The problem is stated in such a way that the time between tasks can be split and obtained with proper tracing. Based on the times that were obtained, the tasks are split into periodic and non-periodic tasks. The times of the intervals between 2 tasks are split into BJI (time difference between the last event of a job and first of the next one) and WJI (time difference between 2 consecutive tasks). The binary classification is performed in order to obtain the best scheduling approach. Tasks are ordered in 2 possible ways usually: run to completion or preemptive. The algorithm can extract the tasks from both these possibilities. Using clustering methods, the tasks are grouped based on their similarities. The paper performs an experiment to observe how useful PeTaMi see the time that it takes and the accuracy of task classification.

Based on the tracing approach and the mining of tasks, [IF18] proposes the usage of graphs for a better representation of the mined tasks. The task precedence graph shows the order of the task execution and is obtained from a set of traces. The algorithm presented in the paper is based on a theorem that states that the mining of the precedence of tasks is non-trivial due to the fact that they can be and most of the time the independent tasks are interleaved. The system's main method obtains a task precedence graph from a set of system traces and obtains a complete set of the occurrences. After the creation of the graph, it will be compared with the other traces that remain and see if there are any anomalies, the anomalies are defined in the paper as events that do not respect the precedence relation. The paper presented two case studies that showed the obtained graphs and how the behaviour of the tasks can be easily represented with the precedence graphs.

The state machines are easy to compute for general purpose software but become harder to model in an embedded software environment. There are multiple approaches for obtaining state machines and most of them follow the pattern of extracting the states and extracting the transitions. In [SQK18] approach the states were extracted from C code based on a series of criteria as well as transitions, the number of transitions obtained by previous methods was too large, with many of them being irrelevant, thus the paper proposes a reduction technique. In embedded software systems, the number of transitions is even higher. The paper observes, categorises the main approaches, and splits the process of state machine mining into the steps of state extraction, transition extraction, and reduction of the transition, and has the main purpose to create a system capable to extract understandable state machines from embedded software.

After this step, we were able to extract relevant information regarding the topic discussed in each article. These information have been summarized in Table 2.2

Article	**Strategy**	**Experiments/Tools**	**Case study**
[WKTG97]	Usage of state design patterns for encapsulating states into object-oriented components or table entities	TV-set control application	academic
[TS02]	Analysis of the problems of choosing good tools for documentation visualisation	Microsoft Visio	academic
[Kni02]	Study how the dependability of the software or hardware can be safety critical and influence the reliability of the system		academic
[HLL10]	Define specific metrics for enhancing the analysis of routine call traces and in order to facilitate the development of analysis tools	Case study on 4 different Java systems/ CheckStyle, Toad, Weka, JHotDraw	academic
[TCL+10]	Shows how using UML modeling of input and code generation helps in creating a more organised and higher performance system	Case study on an ordering system/ Remote Bluetooth communication system model	real life
[AFF12]	Studies a series of architectures on embedded software and observes how high the grade of comprehensiveness for each one of them		academic
[TVD12]	Presents a boundary tracing approach specialised for embedded systems that has a high performance overhead.	Experiment on a postage system with over 2 million lines of code	real life
[ITF17]	Creates a solution that performs task mining and obtains the temporal behaviour of real-time systems. The tool takes advantage of patterns of activity	Two case studies/ PeTaMi	real life
[IF18]	Proposes a solution that creates task precedence graphs on embedded software in order to understand the system, bug finding	PeTaMi	real life
[SQK18]	Presents a series of methods of improving the existing state machine mining techniques. The inefficiency of some of these affects the final results and refinement is necessary	Conducting a series of studies on the efficiency of state machines in program comprehension	academic

Table 2.2: Strategies table

2.5 Results and analysis

In conclusion, after finishing the analysis of the research studies, I got some answers to the research questions that were initially presented.

How is the complexity of embedded systems different from that of regular systems?
Regarding the difference between the comprehension of embedded systems compared to the regular system, it becomes obvious that the complexity of an embedded system and the higher involvement of hardware in its functioning makes it harder to comprehend. Due to this, it has to be given special attention to the development of program comprehension tools regarding embedded systems.

Which are the most popular program comprehension approaches in the direction of embedded systems study?
Through the research, I also observed a few of the methods that are currently used in the program comprehension of embedded systems. Such methods involve task mining or dynamic analysis, but new directions are also studied and further experimented.

Is the finite state machine representation of an embedded system problem a reliable and efficient method for enhancing the program comprehension?
In a recent study I read, the involvement of the finite state machine in the representation of embedded software systems was presented as a reliable and efficient tool. This became the main focus of my thesis and my study will try to confirm the assumption through a series of experiments. Even though it is already believed that state machines will be very helpful for program comprehension of embedded systems, I tend to reinforce that through my master thesis.

Chapter 3

Application Development

As stated previously, the main purpose of my study is to analyse and perform an experiment in order to observe how helpful are finite state machines for program comprehension. The approach presented in this paper started with an observation of different techniques that aim to make easier the understanding of embedded systems and analyse embedded software. After reviewing all the approaches it was reached the conclusion that the usage of a finite state machine can be a viable option that will help the software developers to understand faster and better the code and it will be easier to further develop it.

The study consists of a series of code snippets representing embedded functions from real world applications. These codes are given to the users in order to be analysed and the functionality of the code to be understood. The understanding level is established through a series of measurements such us time taken or level of comprehension. Next to these code fragments the study subjects will be given a finite state machine representation of that code. Using these finite state machines the subjects have a better chance to understand the functionality and after a series of similar questionnaires the level of comprehension is again measured and will be compared to the previous one.

In order to perform this study, the application capable of converting C code into a finite state machine was necessary to be implemented. Following the approach from [CSX+19] the implemented software takes as input the source code given in a text file and generates a finite state machine if possible based on it. Not all functions can be represented in finite state machines, thus if it is not the case, an appropriate message is generated.

In this chapter it will be further presented how the application for obtaining the Finite State Machines was developed and how it works. First we will remind of some of the most crucial concepts required for understanding how the program works. The second part will contain the analysis and design phase of the application and the last part will show the actual implementation, describing the used programming language and the libraries used.

3.1 Key Concepts

In this section, we are presented some of the concepts required to understand how the application was implemented and what this thesis aims to achieve. These con-

cepts will be later referenced a lot through the phases of design and implementation therefor are explained beforehand.

3.1.1 Finite State Machine

The concept of Finite State Machine (FSM) is the main concept used in this thesis. In [BAM18] the general concept of FSM is presented. The elements of a FSM recognised and required by the application are:

- **States**: They are snapshots into the flow of the system.
- **Transitions**: They mark the connections between the states. A transition can be formed between any two elements from the set of states. It can also be from a state and back to it. They have a direction and an input which triggers them.
- **Input Alphabet**: The input alphabet marks the set of possible inputs that the FSM can receive and that can trigger the transitions.
- **Starting State**: The initial state of the FSM.
- **Final State**: The state in which the FSM might finish the execution.

The application is able to identify such Finite State Machines from the given source code if it is the case and return all the elements. The machines returned by the application are used in the experiment for the study of reliability and helpfulness.

3.1.2 Symbolic execution

Symbolic execution [Kin76] is a technique of software testing and bug detection that takes the normal inputs supplied to a program and replaces them with symbolic variables. These variables are arbitrary values and as the execution proceeds instead of normal values the program will run on symbolic formulas with the input symbols.

The testing of programs is a crucial task and cannot be overlooked. The classical testing in which testing input values are given and different scenarios are explored based on some stated use cases is good but not infallible. There can be bugs which are not easy for the tester to get just by running the user cases. In such scenarios, the power of symbolic execution is required. By not giving concrete values to the variables, the program is allowed to run on multiple paths, checking all the possibilities and trying to "break" the program on every path. Thus, by creating the path tree, all the options are explored and any bug is found.

Symbolic execution relies on the use of symbolic states. These states are a set of particular states, not instantiated and concrete. The symbolic state is composed by three important elements. The variables, path conditions and the program counter. All paths obtain in the program will form the execution and the infeasible paths are easy to identify while the variable will find concrete values that will get to fail.

Since the concept of Symbolic Execution has been introduced into the discipline of computer science there have been numerous software systems [CDE08] , [WS17] developed to perform symbolic execution on code written in different programming languages.

3.2 Requirements/Specification

The main purpose of the developed system is to generate FSMs found in a code snippet given as input. The functionality is split into three essential requirements:

- The user uploads a file containing the code.
- The system must search through the code and display to the user how many possible Finite State Machines exist.
- The system will generate for each of the possible machines the 5 element tuple: States, Transitions, Input Alphabet, Start state, Final state

In Table 3.1 it is presented the number of requirements along with the description of the functionality.

Number	Description
1	User uploads a file
2	The system verifies if the file contains Finite State Machines and how many
3	The system generates Finite State Machines based on the source code and presents them to user

Table 3.1: Table of requirements

3.3 Analysis and Design

In this section of the thesis, the design of the application is described. Through the presence of use cases as well as diagrams on the system components and the essential algorithms, a general idea of how the application is working and the logic behind the code can be established.

3.3.1 Use cases

Use case Name: Search for FSMs

Actors: User

Description: The user has opened the application and uploaded the source file. After the system approves that the file has the good format the user presses the *Search FSM* button. The system traverses the file and checks if the conditions for FSM are met. It returns the number of possible FSMS.

Precondition: The user has uploaded the source code file and it was approved by the system.

Postcondition: The file can be parsed

User action	System response
1. Press the Search FSM button	
	2. Parses the file looking if there are existing FSMs
	3. The conditions for existence of a FSM are met and the system stores the information regarding that FSM and increments the number
	4. After parsing all the file displays to the user the number of found FSM and some essential characteristics

Table 3.2: Use case: Search FSM

Exceptions:

- The system has found no possible FSM inside the code and will display to the user an appropriate message.

Use case Name: Generate FSMs
Actors: User
Description: The user has opened the application and uploaded the source file. After the system checks the file to be of good format it checks that there are possible Finite State Machines in the code snippet and informs the user. The user presses the *generate* button and a request for the system is sent to search the code and obtain the Finite State Machines. The system displays the found machines or informs the user if no machine was discovered and the reason for that.

Precondition: The user has uploaded the source code file and performed a scan for Finite State Machines. The scan has shown that there are Finite State Machines in the source code.

Postcondition:
Normal development is presented in Table 3.3.
Exceptions:

- The system is not able to parse the file correctly and it takes the user back to step 1.

- The User decides he does not want to continue in which case the FSMs are not generated anymore.

User action	System response
1. Press the Generate FSM	
	2. Parses the file using the information previously obtained and searches for the number of FSMs
	3. Returns the user the number of possible FSMs and asks if the user wants to proceeds with generating them
4. Confirm the decision	
	5. For each possible FSM it searches for its corresponding 5 element tuple
	6. Generates the state diagram with the elements and displays it to the user together with the list of states, transitions, alphabet, starting and ending state

Table 3.3: Use case: Generate FSM

3.3.2 Algorithms

The application was built with the purpose of extracting information from source code and building Finite State Machines who best mimic the behaviour of the code. In [SQK18] the study performed with several developers on key characteristics of finite state machines that can be observed in source code has shown several approaches to parse that code and obtain satisfactory machines. Those approaches did not really correlate and no application to implement all of them was built.

In [CSX+19] the authors presented a solution to the Finite State Machine extraction problem and developed an algorithm able to do that. The approach presented in this thesis follows the algorithm presented by [CSX+19] and modifies it in some places, trying to achieve the desired results.

The Algorithm 1 (*ExtractingFiniteStateMachines*) is split in more steps:

- After reading the input, the file is checked to be a C source file. It is then parsed by the system and each loop that is found is saved to be later checked if it contains a FSM. Usually a loop can correspond to one or multiple FSMs.
- After obtaining the loops vector, the application traverses this and for each loop checks if the loop respects the condition necessary to be a FSM. The condition states that the loop must not have a constant trip count in order to be a possible FSM. This is verified by: first, identifying the reduction variables of the loop and second, checking if these variables are found to be responsible

Algorithm 1 The *ExtractingFiniteStateMachines* main algorithm

Input:
- File containing source code F ;

Output:
- Number of finite state machines.
- FSM diagram.

Main algorithm *ExtractingFiniteStateMachines (F)* is:
Begin
Let fileEnd := *false*, loops[], possibleFsmCount := 0, loopCount := 0, extractedFSMs[]
while (!fileEnd) **do**
    parse *line*
    if (*line* contains *loop*) **then**
        if (*loop* has constant trip count) **then**
            add *loop* to *loops*
            possibleFsmCount++
        endif
    endif
    if (last line) **then**
        fileEnd := true
    endif
endwhile
while (*loopCount*) **do**
    find state variable for *loop[loopCount]*
    if state variable of *loop[loopCount]* updates **then**
        Create FSM
        Obtain States
        Obtain Transitions
        Obtain Alphabet
        Add FSM to *extractedFSMs[]*
    endif
    loopCount++
endwhile
End.

for any of the exit conditions of the loop. The loops that respect this, are placed in a vector of possible FSMs.

Reduction variables are those variables whose value is increased with a constant value in each iteration of the loop. If these variables are compared to a constant value in any of the exit conditions then the loop has a constant trip count and it does not have a Finite State Machine inside.

- After obtaining the vector with the possible Finite State Machine, the algorithm will try to identify the state variables inside. The state variable is of type int or enumeration type and is responsible for keeping track of the state of the machine. For each finite state machine a state variable is necessary.

  The variables are obtained by a simple mechanism. For each write to a variable of int or enumeration type, the algorithm will look inside the loop and search through all conditional branches that the variable depends on (if, switch) and for each condition it will verify if it is data dependent of the same variable. If it is, the state variable will be saved and starting from it the Finite State Machine is discovered

- Each state variable corresponds to a FSM. Now having the list of all discovered state variables the algorithm will create the FSM for each one. The first elements extracted are the states. The algorithm looks at all assignments of the state variables and will determine the possible states of the machine. If the variable is an enumeration, it will just look at the definition and recover the states from there.

- In order to retrieve transitions the algorithm suggests the use of symbolic execution. For each assignment of the state variable, the algorithm creates a path between its site and another assignment site. During the path traversal, the algorithm will save the constraints got on the way and put them through a constraint solver. The constraint solver will verify that there are no conflicts between them. Another condition is that there are no conflicts between the constraints and the pre-condition of the state variable initial assignment. If all these are satisfied there is a transition between the 2 states. (first and second assignment)

- For establishing the first and last state of the FSM the algorithm will look at the first assignment before the loop is run and at the last assignment of the state variable.

3.3.3 Design diagrams

The are five components used in our approach. In Figure 3.1 it is presented the component diagram of the system.

The diagram consists of 5 main components:

- **Input Interface**

  The input interface was designed using a C++ library and serves as both the entry and the exit point of the application. It provides the user with the ability to give the source file that needs to be analysed as input.

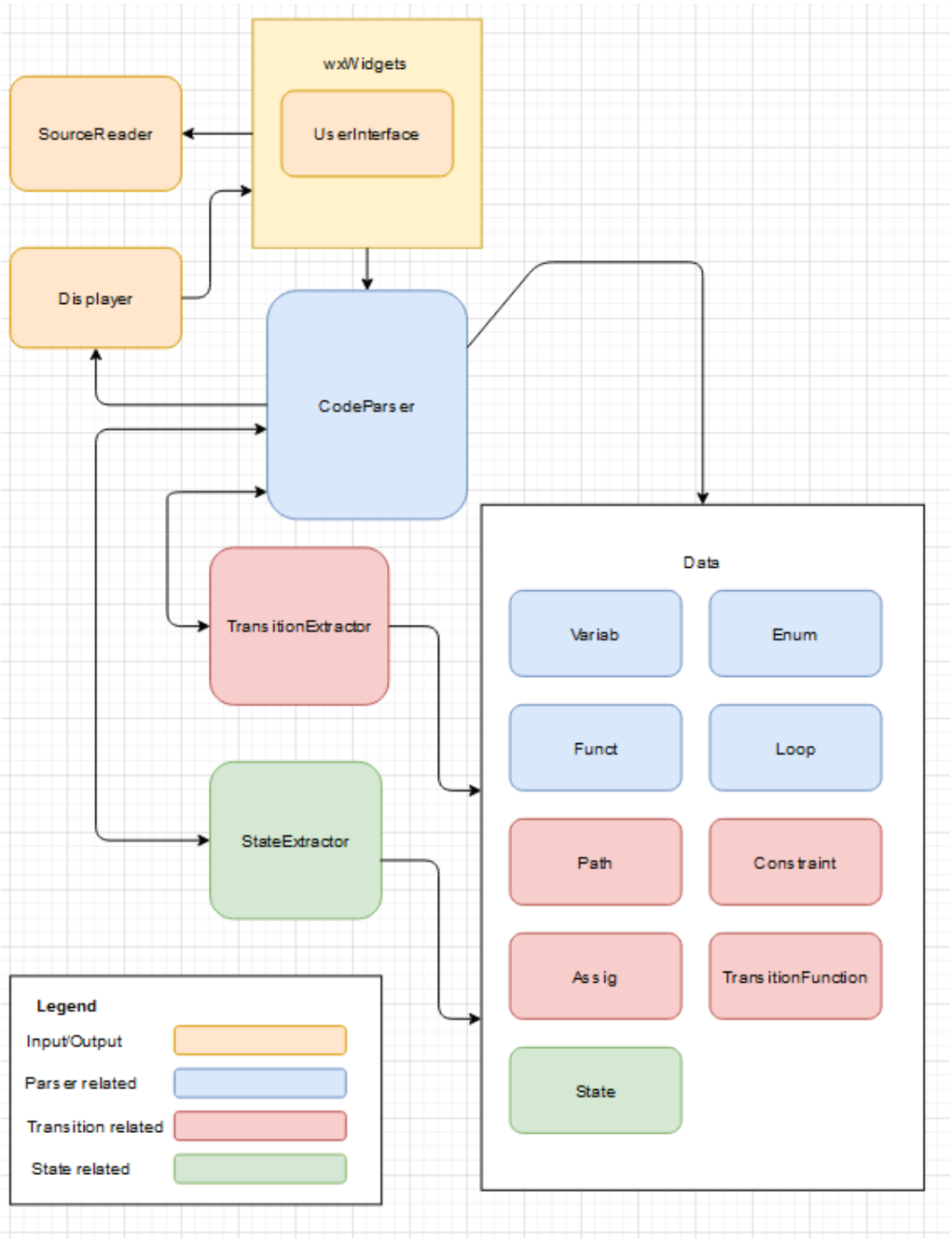

Figure 3.1: Component Diagram

The component is connected to the rest of the code and allows the user to give the input source file. The file is checked to exist and adhere to the source code rules and if that stands, the finite state machines are extracted from the code. The functions from the *CodeParser* are called and after the result is obtained, it is returned to the input interface. There, if there are errors they are displayed and if not, the elements of the discovered finite state machines are displayed: States, Transitions, Initial states and final states.

- **CodeParser**

  The Code Parser is the main "brain" of the operation. The code is written in such a way that the essential elements necessary for the implementation of the algorithm are extracted. The CodeParser component is responsible with taking the input file, and traversing it in order to extract the variables, functions, structures, and other elements presented in the Data packet.

  The CodeParser is started from the input interface. It uses some of the Data components and most importantly triggers the two big parts of the finite state machine generation. The state and transition extraction. The code parser is calling these 2 components to perform their tasks.

- **TransitionExtractor**

  The **TransitionExtractor** component is responsible of determining the transitions that occur between the states. After all the states have been determined the next important step is establishing the transitions. This component will be called from the **CodeParser** and will try to establish paths between assignments of the state variable. Along these paths the constraints are extracted and they are placed in a vector and given to the constraint solver. For each pair of assignments, the constraint solver verifies if they have conflicting constraints in the vector and also check if any of the constraint conflict with the initial constraint of the assignment. If there is no conflict, a transaction function is created between the state of the first assignment and the state of the second assignment.

- **StateExtractor**

  The **StateExtractor** is the component responsible of determining the state variable of the loop and obtain the states of the finite state machine. This component is called from the code parser and implements the algorithm part in which the loop is searched for any state variable and based on the one found the states are obtained and returned to the **CodeParser**.

  For each state variable the **StateExtractor** checks all the assignments that are performed on it and extracts the states.

- **Data**

  The **Data** package is the most simple and has barely close to not at all logic. This component consists of all the necessary data structures for performing

the algorithm. By trying to implement such a cumbersome algorithm and still have a high performance several data structures may be needed. Their main purpose would be storing the variables of the program in different useful forms, in order to reduce the time needed to process them.

This component is used by all the relevant algorithm related components and contains several smaller packages inside, that are more relevant to each component according to the legend's description.

3.4 Technologies and libraries

The algorithm has been explained in both the cited paper and this chapter but its implementation is also extremely relevant. The programming language chosen, as well as the libraries employed into developing an application that extracts Finite State Machines from code are extremely important.

3.4.1 Programming language

The programming language selected for the development of the application is C++. The system performs a straight forward task and does not need to use complex functionalities such as web applications or database management. C++ is useful because of the object oriented capabilities. The algorithm makes use of the structuring and of the classes as well as polymorphism and inheritance.

In the initial approach of the algorithm, the system was required to work with LLVM generated code, and C++, being a system-oriented language would have more low-level access than its object oriented counterparts. One of the main tasks of the system is that of parsing files and because C++ is closer to hardware and very popular for compilers in general, it seemed like the most obvious choice for the project.

Another advantage is the abundance of libraries, created for multiple necessary functionalities, that are easy to use and very accessible.

3.4.2 wxWidgets

wxWidgets [wxW] is a C++ library used for designing user interface for software systems. I chose to use this library for the graphical part of my projects because it is easy to integrate with the C++ code. The design of the application does not have to be very complex and because of that wxWidgets can be quickly configured to create the necessary windows.

This library was used in my application for creating the graphical part and allowing the user to interact with the program. It lets the user upload a source file that wants to be searched and in case any finite state machine is found it will display all elements in a nice manner on the screen.

3.4.3 Klee

Klee [CDE08] is a an engine able to perform symbolic execution, that was built on the top of LLVM compiler. It works on multiple platforms and has several components that allow the user to perform symbolic execution tasks. Klee has several functions that convert the input into symbolic and can be used inside the code to perform the symbolic execution. Besides that, there are also many flags that can be set when running klee command. Flags can influence the output, specify multiple options for algorithms, manage the memory and even debugging.

The main tool for generating the result is *ktest*. The command creates the files with the paths from the symbolic execution tree. And allows the reading of the result.

The tool also has the solver chain, which is responsible for solving the constraints generated along the paths. It can generate Kquery files which will be readable only with another tool on the klee machine, called *kleaver*.

In my project, I used Klee for analysing the symbolic execution paths between assignments in order to establish the transitions between the states. The symbolic execution allows the exploration of the multiple paths and generates the test input cases based on the source code.

3.5 Implementation

The project was implemented in an object oriented manner and consists of several classes. The algorithm is very complex and many steps have been overlooked in the paper [CSX+19]. Those steps have been discovered during development and carefully implemented.

In what follows, listed in Listing 3.1, Listing 3.2, and Listing 3.3. we will present some of the main functionalities:

- In the state extractor the most relevant function is the one obtaining the states. This function uses the state variable and checks if it is an enumeration or an array of integers (it can be only one of these).

  In the case of enumeration it will just retrieve the states from the declaration (elements of the the enumeration). In the other scenario, the function checks the contents of the function in which the FSM's loop is present and searches for all the assignments of the state variable. Each value assigned to the state variable is a state and will be added to the states vector.

```
1 std::vector<State> StateExtractor::obtainStates
2     (Loop loop, Variab stateVar)
3 {
4     std::vector<State> states;
5     Enum enumeration;
6     if(stateVar.isEnum)
```

```
7       {
8           for(int i=0; i<this->enums.size(); i++)
9           {
10              if(this->enums[i].enumName.compare
11                  (stateVar.type)==0)
12              {
13                  enumeration = this->enums[i];
14              }
15          }
16          for(int j=0;
17              j<enumeration.enumElements.size(); j++)
18          {
19              State state;
20              state.stateName =
21                  enumeration.enumElements[j].enumElName;
22              states.push_back(state);
23          }
24      }
25      else
26      {
27          Funct funct = loop.functionContainingLoop;
28          std::vector<std::string> line;
29          for(int i=0; i<funct.functContent.size(); i++)
30          {
31              if (funct.functContent[i].
32                      find(stateVar.name)
33                          != std::string::npos)
34              {
35                  std::istringstream ss
36                      (loop.loopContent[i]);
37                  std::string word;
38                  while (ss >> word)
39                  {
40                      line.push_back(word);
41                  }
42                  for(int j=0; j<line.size(); j++)
43                  {
44                      if((line[j]
45                          .compare(stateVar.name)==0)
46                          &&(line[j+1].compare("=")==0))
47                      {
48                          State state;
49                          state.stateName = line[j];
50                          states.push_back(state);
51                      }
52                  }
53              }
54          }
```

```
55      }
56      return states;
57  }
```

Listing 3.1: Code example for state extraction

- In the transition, extractor component has a function which extracts the constraints for the path between 2 assignments. The extraction is done based on one of the 4 cases. The cases are determined based on the location of the assignments in relation with the finite state machine's loop.

  In the code example, I showed the code for the most common case. When the transitions are both inside the FSM's loop. In this case, the loop entry conditions are added and the loop is traversed from one assignment to the other checking all the branching conditions along the way. If an if is found, the function *extractFromIf* will be called and the constraints are extracted if the assignment is inside it. The same happens when a switch is found and the function *extractFromSwitch* does the same but on the switch statement.

- During the transition extraction, the transition extractor makes use of the constraint solver. The *areConflicts* method searches along the path given and checks for conflicts between constraints. At first, they are converted in order to have the same format. They are split from logical connection elements and are placed in a list one after the other.

  The list is traversed and for each pair of constraints it is first checked if the element in the constraint is a program variable and also if it appears in both sides of the pair. If that happens there is a chance that there is a constraint there.

  A separate function is called to check the conflict between those to constraints and if a conflict was found the return function stating that there are conflicts is called.

```
1  std::vector<Constraint>
2  TransitionExtractor::getConstraints
3      (std::vector<std::string> sourceData, Loop loop,
4          Assig assig1, Assig assig2, bool& noPathPossible)
5  {
6      std::vector<Constraint> ifConstraints;
7      std::vector<Constraint> swConstraints;
8          for (int i=0; i<sourceData.size(); i++)
9          {
10                 std::istringstream ss(sourceData[i]);
11                 std::string word; // for storing each word
12                 std::vector<std::string> line;
13                 while (ss >> word)
14                 {
15                         line.push_back(word);
16                 }
```

```
17          }
18          //case 4 both assignments inside loop
19      if((assig1.insideLoop)&&(assig2.insideLoop))
20      {
21                  Constraint loopConstraint;
22          //obtain correct conditions
23          std::string loopCond =
24              convertCondition(loop.exitCondStatement);
25          //add the loop constraints
26          loopConstraint.constraintString = loopCond;
27                  constraints.push_back(loopConstraint);
28          for (int i = 0; i < loop.loopContent.size(); i++)
29          {
30              //seach for ifs
31              if(((loop.loopContent[i].find("if(")!=0) &&
32                      (loop.loopContent[i].find("else")==0))
33                  ||
34              ((loop.loopContent[i].find("if ")!=0) &&
35                      (loop.loopContent[i].find("else")==0)))
36              {
37                  ifConstraints= extractFromIf
38                              (loop.loopContent, assig1, i);
39                  for(int j=0; j<ifConstraints.size(); j++)
40                  {
41                      constraints.push_back(ifConstraints[j]);
42                  }
43              }
44              if (loop.loopContent[i].find("switch")!=0)
45              {
46              swConstraints = extractFromSwitch
47                              (loop.loopContent, assig1, i);
48                  for(int j=0; j<swConstraints.size(); j++)
49                  {
50                      constraints.push_back(swConstraints[j]);
51                  }
52              }
53          }
54      }
55      return constraints;
56  }
```

Listing 3.2: Extraction of the constraints

```
1   bool ConstraintSolver::areConflicts(Path path)
2   {
3       std::vector<Constraint> constraints;
4       bool constraintsConflict = false;
5       //the constrains are converted to have the same format
```

```
6       constraints = convertConstraints(path.pathConstraints);
7       for(int i=0; i<constraints.size(); i++)
8       {
9           for(int j=0; j<constraints.size(); j++)
10          {
11              if (i != j)
12              {
13                  //check if in the second constraint there is
14                  //any of the elements from the first
15                  //constraint that are also program variables
16                  if(
17                      ((checkIfProgramVariable
18                          (constraints[i].elem1)) &&
19                          (constraints[j].constraintString.
20                              find(constraints[i].elem1)!=0))
21                      ||
22                      ((checkIfProgramVariable
23                          (constraints[i].elem2)) &&
24                          (constraints[j].constraintString.
25                              find(constraints[i].elem2)!=0))
26                      )
27                  {
28                      constraintsConflict =
29                          checkIfContradict
30                              (constraints[i],constraints[j]);
31                              }
32                      if (constraintsConflict)
33                      {
34                          return true;
35                      }
36                  }
37              }
38          }
39      }
40      return false;
41  }
```

Listing 3.3: Checking for conflict between constraints

3.6 Testing

The testing has been done according to the method presented in [Bac00], i.e., Session-Based Test Management. The tests were performed on source data coming from the CGC dataset [CGC] that were given to the algorithm and the results observed. The main steps the application performs are the file upload, the file verification, and the generation of the FSM.

The generation can be split in:

- Determine if there is a FSM;

- Find the state variables;
- Obtain the states;
- Obtain the transitions;
- Obtain the initial and the final state.

Each of these steps was tested separately (See Table 3.4, Table 3.5, and Table 3.6) at first and after the integration of a component the whole system was tested for performance and bugs were fixed.

Session Tester	Epure Lucian
Session Charter	Epure Lucian
Planned session time	5 minutes
Environment Info	Upload file
Area	The file path is given and file is uploaded
Current Active Tag	Bug
Session start	Start adding notes below
Setup	Providing input with file name
Test	Press button to upload file
Note	This can be a bug in finding the file through the system
Bug	The source code file was not found

Table 3.4: Test source code upload

Session Tester	Epure Lucian
Session Charter	Epure Lucian
Planned session time	10 minutes
Environment Info	Find state variable
Area	The system determines which variable is the state variable in the loop
Current Active Tag	Bug
Session start	Start adding notes below
Setup	FSM exists within the loop
Test	Search for the state variable inside the loop
Note	This can be a bug determining the correct state variable
Bug	Variable has not been determined correctly

Table 3.5: Test state variable search

3.7 Deployment

The system is a desktop application and was built in C++. It runs on windows platform and does not require any connection to the internet. The application has

Session Tester	Epure Lucian
Session Charter	Epure Lucian
Planned session time	30 minutes
Environment Info	Extract transitions
Area	The system extract transitions between the states
Current Active Tag	Bug
Session start	Start adding notes below
Setup	States have been found
Test	Search for the state assignments and establish constraints
Note	There is a bug when the constraints along the paths are extracted
Bug	Constraints are not correct

Table 3.6: Test transition determination

an easy to use interface and lets the user access the source files in order to be parsed. The files have to be stored on the user's station and are manually updated. The results are returned in the application user interface as states and transitions and they can be used directly

The application will be integrated in the IDE and will be stored only locally in order to not encumber the functionality and the speed.

Chapter 4

Experimental study

This thesis aimed to establish how helpful finite state machines are in the field of embedded system development. The project analysed several techniques that aim to enhance the programming comprehension and focuses on the use of finite state machines. In order to perform the analysis, a series of experiments establishing the level of comprehension were done.

4.1 Study methodology

The experiments consisted of several software users being given pieces of code and asked to perform tasks which required them to understand the given code. The subjects have used the Finite State Machine Extractor as a helping tool in order to understand better the given code. As presented in [SQK18] the interest in enhancing code comprehension has increased more and more and the authors of the study tried to observe, through a series of experiments, how Finite State Machines can help.

In my thesis I took a similar approach. After developing an application capable of generating Finite State Machine code from source code, I selected several examples and developed an empirical experiment that also uses a questionnaire. The questionnaire containing the examples is given to a the study participants that are asked to answer to some essential questions.

Our research investigation aims to answer to the following research questions(RQ):

- RQ1: Do Finite State Machines reduce the time taken by users to understand the source code ?
- RQ2: Is it easier to extend a software system after having access to a Finite State Machine which represents the current functionality ?

4.1.1 Study participants

The study participants are students in the field of Computer Science and workers in the IT domain. The system parses C source code and all participants are familiar to a certain degree with the syntax of the programming language. All the subjects had contact at a certain point in their academic or professional life with C syntax, but no one uses it currently, thus the subjects have an equal level of understanding

as a starting point.

The number of subjects was 20. All of them were given code examples and were asked to perform a series of tasks. From the subjects, 10 received the Finite State Machine extracted with the Finite State Extractor system as a helping tool.

Each of the users was given the questionnaire in private through a google form and allowed a fixed amount of time (20 minutes) to complete it.

4.1.2 Study materials

To determine the answers to the research questions, an experiment was created. The experiment used a Google Form questionnaire which contains a series of questions and exercises aimed to establish the level of understanding.

The participants are being split in 2 groups and each group received one type of questionnaire. The first type contains only the source code and no helping FSM. They are given 3 examples of code, each one with 3 exercises. The exercises measure how well the subjects understood the code and have been designed for 3 difficulties. First one is easy, the second is medium and the last one is harder. By doing that, it is easier to compare and observe if the group with the FSM performed better and reached a higher level than the ones without.

The second group are given the exact same code snippets but are also provided the obtained FSM after using my application. They have the same questions and the same time as the previous group.

In the questionnaires, there are also other questions that try to determine study circumstances such as the knowledge level in C programming and that of Finite State Machines. The study also considers what approach the subjects took and their emotional state during the form completion. With all these pieces of information, a complete image of the relevance of the studied subject can be established.

The code examples provided to the subjects have been extracted from the CGC dataset and have been simplified in order to better focus on the thesis subject. Thus, for the code in Figure 4.1 and Figure 4.2 (one of the examples), a generated FSM is presented in Figure 4.3. The other examples are very similar to this one.

4.2 Results

After the questionnaire was given to 20 persons the results were gathered and interpreted. The main aspect that was considered were the backgrounds of the subjects. Their backgrounds are important in establishing their connection to the computer science field and how well they are accustomed to C programming language and finite state machines. In Figure 4.4 can be seen how most of the subjects (i.e. 12 are both students and employees in the Computer science field: 6 are just students and 2 are only employee. It is concluded that all the subjects have a connection

```
typedef enum elems {
    mac_start,
    chs_prod,
    ent_mny,
    five_mny,
    ten_mny,
    ffteen_mny,
    twnty_mny,
    check_sum,
    give_rest,
    give_prod,
    cancel,
    error
} sdaOps ;
void sodaMachine ( int *inpData )
{
    sdaOps st ;
    st = mac_start ;
    int totalInpSum = 0 ;
    int prodPrice = 0 ;
    int restToGive = 0 ;
    while ( *inpData && st != cancel && st != error )
    {
        switch ( st ) {
            case mac_start :
                if ( *inpData == 1 )
                {
                    st = chs_prod ;
                }
                else
                {
                    st = cancel ;
                }
                break ;
            case chs_prod :
                if ( *inpData > 0 )
                {
                    prodPrice = *inpData ;
                    st = ent_mny ;
                }
                else
                {
                    st = error ;
                }
                break ;
            case ent_mny :
                if ( *inpData == 5 )
                    st = five_mny ;
                else if ( *inpData == 10 )
                    st = ten_mny ;
                else if ( *inpData == 15 )
                    st = ffteen_mny ;
                else if ( *inpData == 20 )
                    st = twnty_mny ;
                break ;
```

Figure 4.1: Code example

```
            case five_mny :
                totalInpSum += 5 ;
                st = check_sum
                break ;
            case ten_mny :
                totalInpSum += 10 ;
                st = check_sum
                break ;
            case ffteen_mny :
                totalInpSum += 15 ;
                st = check_sum
                break ;
            case twnty_mny :
                totalInpSum += 20 ;
                st = check_sum
                break ;
            case check_sum :
                if ( totalInpSum > prodPrice )
                {
                    st = give_rest ;
                }
                else if ( totalInpSum < prodPrice )
                {
                    st = ent_mny ;
                }
                else
                {
                    st = give_prod ;
                }
                break ;
            case give_rest :
                restToGive = totalInpSum - prodPrice ;
                st = give_prod ;
            break ;
            case give_prod :
                display( "Product was given" ) ;
                st = mac_start ;
            break ;
        }
}
```

Figure 4.2: Code example

```
Finite State Machine:  FSM
State Variable: sda
States: mac_start cancel chs_prod ent_mny five_mny ten_mny
        ffteen_mny twnty_mny check_sum give_rest give_prod error
Transitions:
0 FROM mac_start TO chs_prod
1 FROM mac_start TO cancel
2 FROM chs_prod TO ent_mny
3 FROM chs_prod TO error
4 FROM ent_mny TO five_mny
5 FROM ent_mny TO ten_mny
6 FROM ent_mny TO ffteen_mny
7 FROM ent_mny TO twnty_mny
8 FROM five_mny TO check_sum
9 FROM ten_mny TO check_sum
10 FROM ffteen_mny TO check_sum
11 FROM twnty_mny TO check_sum
12 FROM check_sum TO give_rest
13 FROM check_sum TO give_prod
14 FROM give_rest TO give_prod
15 FROM give_prod TO mac_start
Initial State: mac_start
Final States: error cancel mac_start
```

Figure 4.3: Generated FSM

with the computer science field.

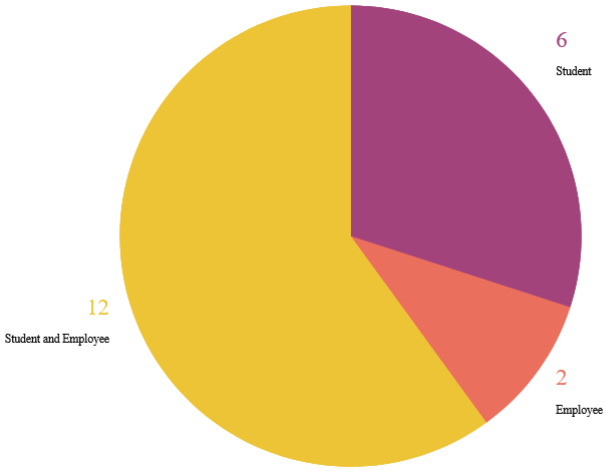

Figure 4.4: Computer science field subjects distribution

Further, it was explored the level of knowledge of both C programming and Finite State Machines by auto estimation. The C knowledge (See Figure4.5) of most users is above average, thus there will be a high level of understanding of the C source code and no one will struggle. On the other hand, the level of FSM knowledge (See Figure 4.6) is lower, with most of the users regarding themselves as beginners and one stating that he/she never even heard about Finite State Machines.

The exercises meant to test the level of comprehension of users in both groups consisted each of 3 questions. In order to better analyse the results for each example,

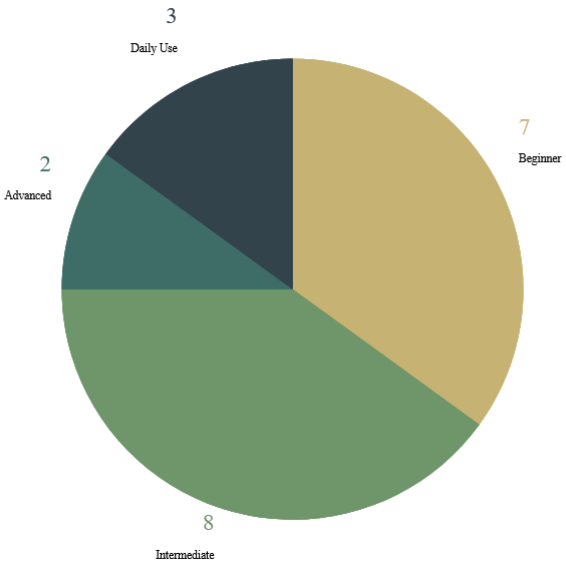

Figure 4.5: C programming knowledge

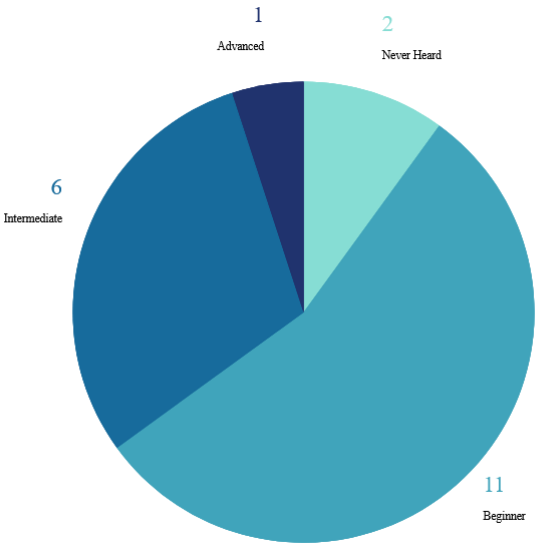

Figure 4.6: Finite State Machines knowledge

we display a graph comparison containing the number of correct and wrong answer for each question.

The first example had the following questions:

- What is the necessary input in order for the function to return "true"?
- Can you explain in natural language what the program does?
- What values will the state have for the following input: — "abc—"d ? (write all the values)

All the questions require the user to provide its written answer, thus no helping choice options are available to aid the user.

As it can be seen in Figure 4.7 the answers are mostly correct for the first question even without the use of the FSM. As well as for the second and third questions, 7/10 subjects got them correct with no help. However, after adding the generated FSM, it can be observed the improvement of the subject's exercise results.

For the first two questions all the users got the correct result as for the last one, which is supposed to be the hardest, it can still be seen an improvement as one more user got the correct response.

From seeing the graphics, it is very obvious the improvement in code understanding when the Finite State Machine generate by the implemented system was presented together with the code.

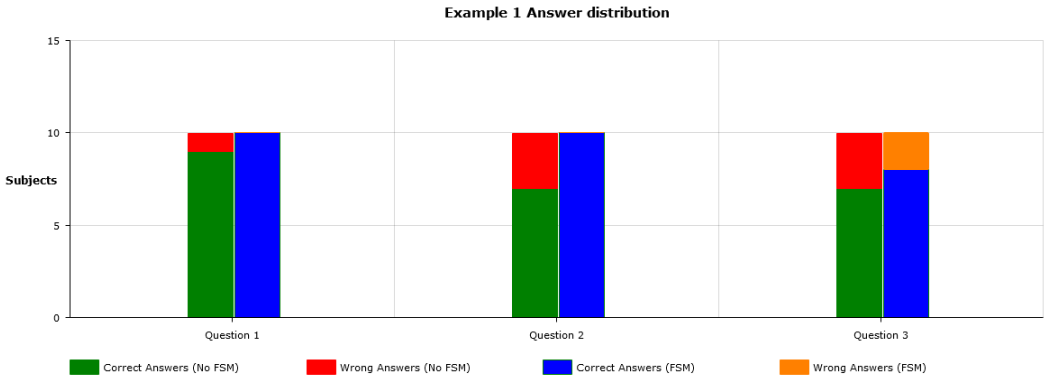

Figure 4.7: Exercise 1, answer distribution

For the second exercise, in Figure 4.8 the questions were somewhat similar, but the code snippet were significantly more difficult to understand.

- What real life system does the code simulate?
- Can you explain in natural language what the program does?
- What value can "inpSeq" have in order for "tmr" to never have a value containing error?

In Fgirue 4.8 it can be quickly observed the increase in difficulty as more users from the group that had only the source code fail to correctly respond to the questions. However, when the FSM is included, the first 2 questions become easier and all the subjects got the correct answers. In the case of the last question, the improvement of understanding is not as big. Only 7 out of 10 users were able to respond correctly even with the help of the given FSM.

Although the questions were more difficult the increase in code comprehension is still visible in the second example.

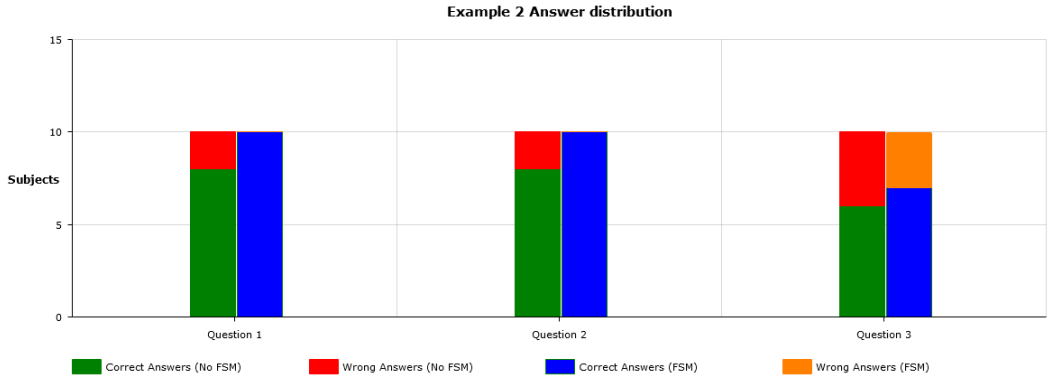

Figure 4.8: Exercise 2, answer distribution

For the last exercise the questions were:

- What real life system does the code simulate?
- Can you explain in natural language what the program does?
- Give a value example for "inpData" in order for "st" to get the value "give_rest" at some point in time.

The results are visible in Figure 4.9 and show that for the first group only a little more of half the participants got the correct response, with the last question having the responses equally split. In the case of the group using the FSM the situation is a little better. For the first 2 questions which were more simple, there is an improvement, but for the last one, only two more users got better results.

By comparing all the graphs it can be concluded that in the context of the presented experiment, the usage of FSM has proven helpful for the code comprehension.

Another direction of the study was the analysis of time efficiency. The questionnaire was timed and based on the duration it can be observed if the existence of Finite State Machines helped. In Figure 4.10 the time taken for each user in the first group to finish their task was measured. The maximum allowed was 20 minutes and from the graphic it can be seen that all users in the first group finished in over 10 minutes and on average around 12 minutes.

In the second group it can be seen that the times taken were somehow similar, see Figure 4.11. Although it was not a big difference, the users from the second

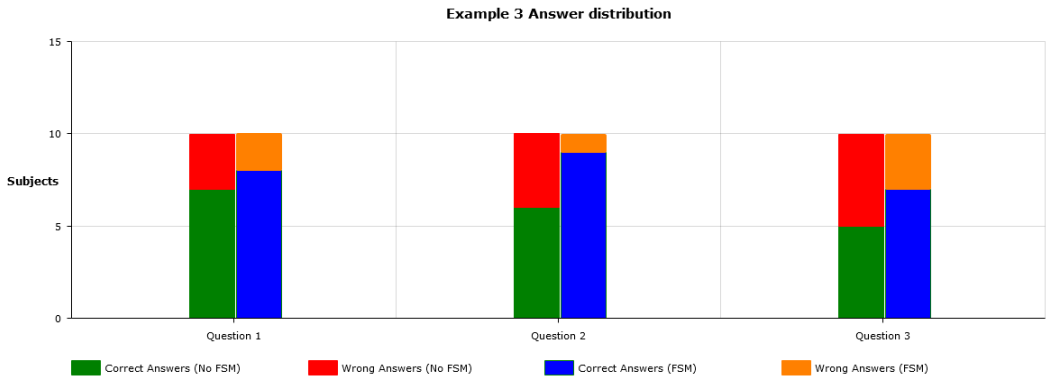

Figure 4.9: Exercise 3, answer distribution

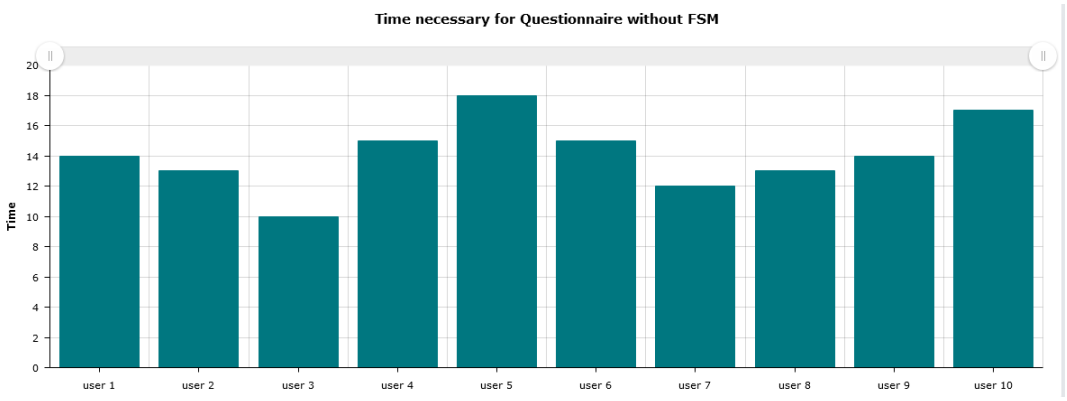

Figure 4.10: Time required to complete the questionnaire with FSM

group performed slightly faster. There are two users who finished under 10 minutes and the average this time would be around 11 minutes.

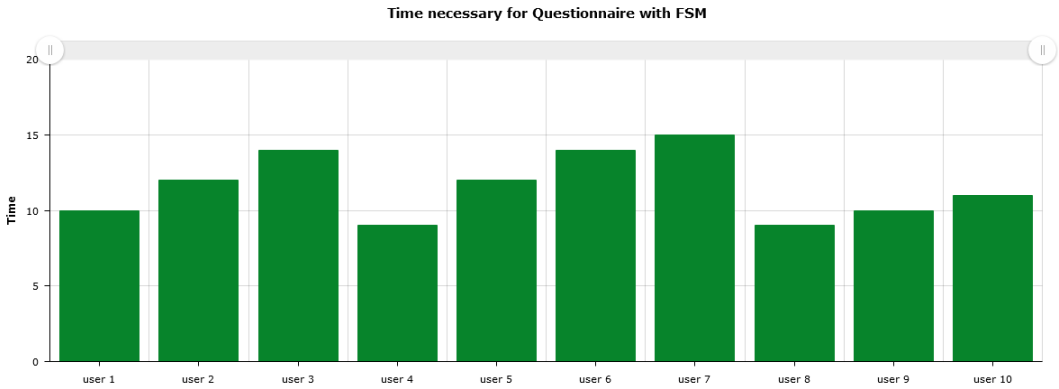

Figure 4.11: Time required to complete the questionnaire with no FSM

Coming back to the questions posed in the beginning, we will next provide the answers based on the empirical conducted experiment:

- RQ1: Do Finite State Machines reduce the time taken by users to understand the source code ?

  The time efficiency was measured during the empirical study and has been proven that even if the users from the second group (who had access to the FSM) didn't finish a lot faster than their counterparts they still gained 1 minute on the average time. Thus, FSM seem to be helpful for increasing the speed of developers who must understand embedded software source code.

- RQ2: Is it easier to extend a software system after having access to a Finite State Machine which represents the current functionality ?

  Yes it is. The three exercises tested comprehension skills that are relevant for allowing the developers to further extend the analysed code. For each experiment, at least one of the questions approach this subject and based on the results and how the existence of the Finite State Machine increased the understand-ability

Chapter 5

Conclusions and future work

The continuous development of embedded systems and the high importance of program comprehension fueled the need for solutions that will help these two elements combine. Thus, several approaches of comprehension enhancement were discussed and were presented in the second chapter of the thesis. Approaches such as the use of UML modelling, Task mining, or the employment of multiple analysis techniques have been useful for code comprehension in general, but in the case of embedded systems the situation changes.

In my thesis I started from an approach presented in [SQK18] in which Finite State Machines are extracted from the source code and are projecting the behaviour of the code into a more simple presentation. Embedded software is used in the "real life" systems and many times the behaviour of such systems is that of a Finite State Machine. Thus, it is more accessible to have the FSMs extracted and because of how intuitive they are, they will help in increasing the comprehension of code.

After analysing [CSX+19] I built a system capable of extracting the Finite State Machines from C source code. The application receives as input the source file and after applying multiple algorithms obtains a FSM if there exists one. The result of the algorithm is a list of states, a list of transitions, and both the initial and the final state. The system is easy to use and creates the helpful FSM quickly.

5.1 Conclusions

To analyse if FSMs really help in embedded code comprehension, an empirical study was conducted. The study aimed to establish if Finite State Machines are helpful for developers in program comprehension. The subjects were split into two groups and were given questionnaires containing a few exercises of code understanding. One group had access to the FSM generated by my application while the other did not. After analysing the results of the exercises, it was concluded that the group with access to FSM performed better at the task and had also a slightly better overall time.

The study showed that Finite State Machines that were extracted from the source code are useful for understanding the code better and faster. This proves that FSMs

are a powerful helping tool for enhancing code comprehension of embedded software systems.

5.2 Contributions

The main contribution of this research was the creation of a software system capable of extracting the FSMs from the source code. Even though the steps of the algorithm were presented in [CSX+19], the implementation was done by me and differs in many places. The system does not use any external libraries for processing the files, only for the graphical interface. This reduces the run time and makes it run smoother.

Another improvement to the algorithm is that the parsing is done directly on the C source code and does not require an intermediate element such as LLVM. Even if the implementation of the parsing is more complex the LLVM conversion libraries and the use of symbolic execution is unnecesairly cumbersome. The application that I implemented is but a prototype and even if it has some capabilities, and can extract simple FSM from C source code, it has a lot of potential and numerous development opportunities.

The biggest contribution of my thesis is the empiric study that I performed. During this study, a group of users were given some exercises on source code and some were helped by the application's extracted FSMs. This study showed how Finite State Machines can increase the comprehension of embedded system C source code and even if it was performed on small-scale source code snippets it had a promising result that will possibly be obtained also on larger scale systems.

5.3 Future research

The current application is only a prototype and can extract Finite State Machines from simple C source code files. There are many future improvements that can be integrated in the application, like:

- **Enhancing the parsing algorithm**

  The current algorithm parses C files and works on a certain structure. Some changes in how the file is written may affect the accuracy of the algorithm.

- **Improving the Transition extraction**

  The transition extraction is currently performed with the help of a constraint solver. The constraint solver was implemented by me and is solving conflicts between simple constraints but will not work well on complex constraints statements. By improving this, the system will be able to discover and extract FSMs from more complex code snippets.

- **Displaying the FSMs**

  The obtained Finite State Machines are currently displayed in textual format. A library should be integrated which will display the results of the algorithm in a easier to understand mode.

Bibliography

[AFF12] E. A. Antonio, F. C. Ferrari, and S. C. P. F. Fabbri. A systematic mapping of architectures for embedded software. In *2012 Second Brazilian Conference on Critical Embedded Systems*, pages 18–23, 2012.

[Bac00] Jonathan Bach. Session-based test management. 01 2000.

[BAM18] Mordechai Ben-Ari and Francesco Mondada. *Finite State Machines*, pages 55–61. Springer International Publishing, Cham, 2018.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[CGC] Cyber grand challenge. https://archive.darpa.mil/CyberGrandChallenge/.

[CSX+19] Yongheng Chen, Linhai Song, Xinyu Xing, Fengyuan Xu, and Wenfei Wu. Automated finite state machine extraction. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 9–15, New York, NY, USA, 2019. Association for Computing Machinery.

[HLL10] A. Hamou-Lhadj and Timothy Lethbridge. Understanding the complexity embedded in large routine call traces with a focus on program comprehension tasks. *IET Softw.*, 4:161–177, 2010.

[IF18] O. Iegorov and S. Fischmeister. Mining task precedence graphs from real-time embedded system traces. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–260, 2018.

[ITF17] O. Iegorov, R. Torres, and S. Fischmeister. Periodic task mining in embedded system traces. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 331–340, 2017.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[Kit04] Barbara Kitchenham. Procedures for performing systematic reviews, keele university tr. Technical report, SE-0401/NICTA, Technical Report, 2004.

[Kni02] J. C. Knight. Dependability of embedded systems. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 685–686, 2002.

[KTS18] Surajkumar Kumbhar, Rachana Thombare, and Amitkumar Salunkhe. Recent advances in automotive embedded systems. 7:232–238, 02 2018.

[Oye15] Oluwole Oyetoke. Embedded systems engineering, the future of our technology world; a look into the design of optimized energy metering devices. *International Journal of Recent Engineering Science (IJRES)*, 12 2015.

[SQK18] W. Said, J. Quante, and R. Koschke. On state machine mining from embedded control software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 138–148, 2018.

[TCL+10] H. Tung, C. Chang, C. Lu, W. C. Chu, and H. Yang. From applications, to models and to embedded system code: A modeling approach in action. In *2010 10th International Conference on Quality Software*, pages 488–494, 2010.

[TS02] S. Tilley and Shihong Huang. On selecting software visualization tools for program understanding in an industrial context. In *Proceedings 10th International Workshop on Program Comprehension*, pages 285–288, 2002.

[TVD12] J. Trümper, S. Voigt, and J. Döllner. Maintenance of embedded systems: Supporting program comprehension using dynamic analysis. In *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*, pages 58–64, 2012.

[vGB99] Jilles van Gurp and Jan Bosch. On the implementation of finite state machines. 01 1999.

[WKTG97] J. Weidl, R. R. Klosch, G. Trausmuth, and H. Gall. Facilitating program comprehension via generic components for state machines. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 118–127, 1997.

[WS17] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, 2017.

[wxW] wxwidgets. https://docs.wxwidgets.org/3.0/.