Vaasan yliopisto
UNIVERSITY OF VAASA

Zeya Zheng

Automatic synthesis of fuzzy systems in C from high-level specifications

School of Technology and Innovations
Master's thesis in Discipline
Industrial system analysis

Vaasa 2025

UNIVERSITY OF VAASA	
School of …	School of Technology and Innovations
Author:	Zeya Zheng
Title of the Thesis:	Automatic synthesis of fuzzy systems in C from high-level specifications
Degree:	Master of Science in Technology
Programme:	Industrial System Analysis
Supervisor:	Ville Tuomi, Luca Ferranti
Year:	2025 **Pages:** 73

ABSTRACT:

This thesis focuses on extending a C code generator for an existing fuzzy logic system library. It explains the background of fuzzy logic systems and the motivation behind this work. A literature review on Mamdani and Sugeno fuzzy systems is included.

The Design Science Research methodology was applied to this topic, which is important for the field of this thesis, industrial system analysis. Design Science Research helped the author define the research question: "How to synthesize a fuzzy logic system into optimized C code from high-level specifications and improve and verify the code through implementation." It also guided the author through the research process, starting from awareness of the problem and then moving to suggestions, development, evaluation, and conclusion. This approach proved to be very useful.

The implementation is the core focus of this thesis. It provides a detailed explanation of how the C code generator was created based on an understanding of the FuzzyLogic.jl library, Julia language features, and the Julia programming environment. The fuzzy system's C code is generated as strings in Julia, derived from variables and values input by the user in FuzzyLogic.jl. This section also includes code comments, a user manual, and contributions to the open-source community.

To evaluate the C code generator, the author applied it to a real-world case study: a traffic light control system. This example is appropriate because fuzzy logic is effective for managing traffic control, which involves uncertain inputs and outputs. Additionally, most traffic control systems use C-based embedded devices, making this an ideal test case.

Finally, the limitations, achievements, and future possibilities of this research are analyzed. This study contributes to the ongoing development of fuzzy logic systems and offers valuable insights for future research in the field.

KEYWORDS: Fuzzy Logic, Fuzzy System, C code, Julia

Contents

Pictures

Figures

Tables

Abbreviations

FCL : Fuzzy Control Language
FML: Fuzzy Makeup Language
DSR: Design Science Research
FIS: Fussy Inference System
REPL: Read-Eval-Print Loop

1 Introduction

This chapter introduces the background, motivation, and objectives of this thesis. The reader will understand the basic concept of this research, the reason why it was conducted, and the specific goal of this thesis.

1.1 Background

The main topic of this thesis is related to Fuzzy systems; it utilizes fuzzy logic to process imprecise inputs and outputs. Nowadays, it is becoming increasingly significant for solving real-world complex problems. Unlike traditional binary logic, fuzzy logic allows using degrees of truth (Zadeh, 2008). That's also why it's called fuzzy. When dealing with problems, it mirrors human reasoning. This capability makes fuzzy systems more useful when conditions and outcomes cannot be clearly defined as true or false.

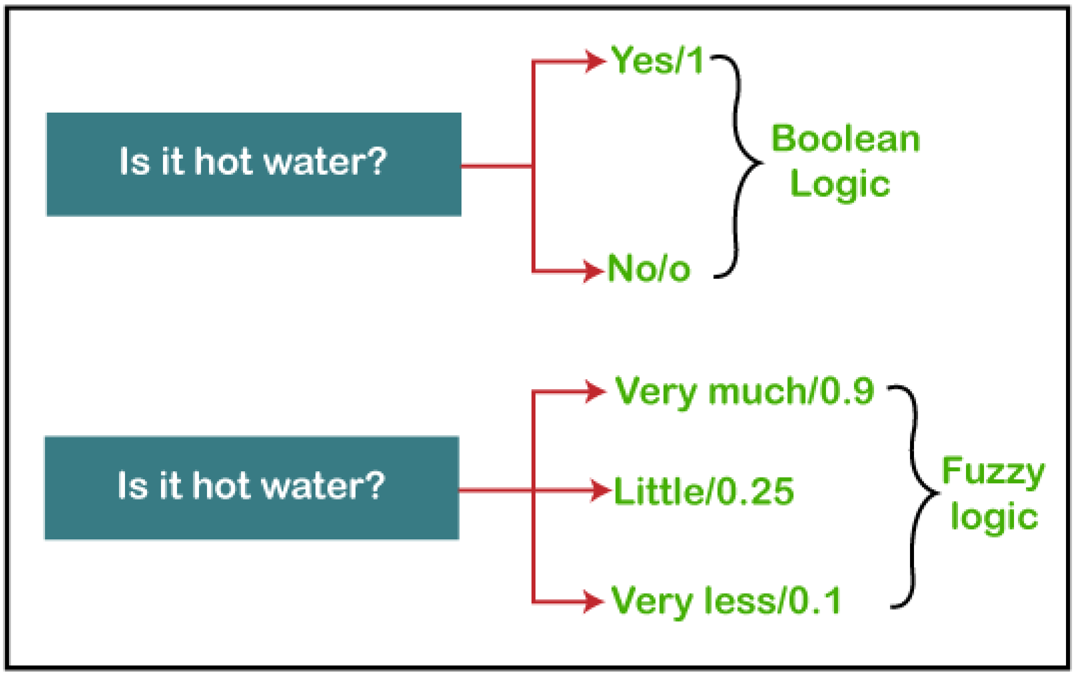


Figure 1. Example of Fuzzy logic compared to Boolean Logic (Fuzzy, 2024)

In the book Fuzzy Logic with Engineering Applications, Ross (2010) uses examples to show that Fuzzy systems cover a variety of industries, including control systems, automotive engineering, image processing, and decision-making processes, where the ability to deal with ambiguity and uncertainty is crucial. The application range of fuzzy logic includes household appliances such as washing machines and air conditioners to advanced automotive control systems.

In recent years, significant progress has been made in developing and applying fuzzy systems. Researchers have explored various ways to improve efficiency and effectiveness, resulting in innovative solutions to contemporary challenges. The growing interest in this field highlights its importance and potential impact on future technological development.

A research paper titled 'FuzzyLogic.jl: A Flexible Library for Efficient and Productive Fuzzy Inference' written by Luca Ferranti and Jani Boutellier (2023) from the University of Vaasa demonstrates the validity of the Julia Library for fuzzy logic. It also highlights how this library can increase efficiency and productivity for fuzzy system designers. The whole topic of my thesis is highly based on the tool mentioned in this research paper. The author will also develop a new function that can let the existing library be able to generate C code from the existing fuzzy logic model.

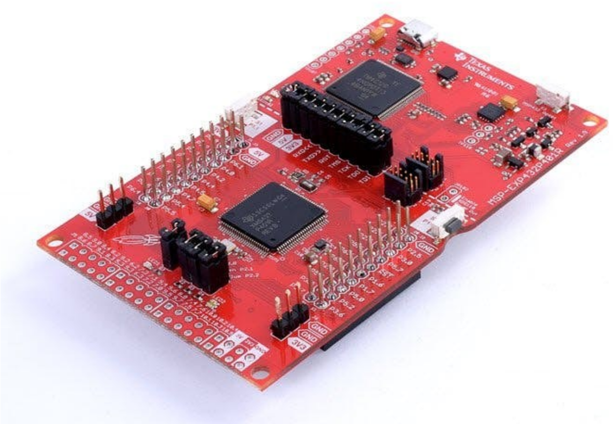
1.2 Motivation

Creating a C code generator for the Julia fuzzy logic library is necessary for several reasons. One key reason is to reduce the gap between the library's high-level abstraction and efficient implementation in C code. Another benefit is that it increases the efficiency of low-level implementation to better support research and industry needs.

In Zadeh's paper and many other research, fuzzy logic has grown to be a helpful instrument because of its capacity to offer a flexible framework for handling imprecision and uncertainty (Zadeh, 2015). It is especially useful in systems such as control systems, complex system modeling, and decision-making under uncertainty, where typical binary logic is insufficient (Yen, 1999). The FuzzyLogic.jl library offers an expressive domain-specific language (DSL) in Julia, along with a wealth of features that have greatly improved the usability and expressiveness of fuzzy systems.

The FuzzyLogic.jl library also offers a powerful way to define and manipulate fuzzy inference systems using high-level abstractions. Its domain-specific language (DSL) lets users create fuzzy systems in a manner that aligns with their conceptual understanding, avoiding low-level complexities. This approach boosts productivity and allows for intuitive modeling and experimentation. However, while Julia is excellent for high-level abstractions, challenges arise when deploying these models in environments where Julia is not preferred due to system requirements, performance needs, or ecosystem compatibility (Ferranti & Boutellier, 2023).

Many real-world fuzzy logic applications need to be deployed in environments where efficiency and performance both need to be considered. These environments often include embedded systems, real-time applications, and legacy systems. C is the dominant language due to its CPU architecture, efficiency, and widespread use in system-level programming. Therefore, it is essential to translate high-level fuzzy models into efficient implementations running in these performance-critical environments.



Picture 1. Ti MSP 432, a typical C code embedded device.

Fuzzy logic is supposed to preserve expressiveness and flexibility and leverage efficiency and performance of C. By automatically synthesizing C code from FuzzyLogic.jl specifications, we preserve the expressiveness and flexibility that the high-level DSL provides. By translating these specifications into C, we leverage the language's efficiency and performance, especially in low-resource and real-time environments

Fuzzy logic is supposed to enhance interoperability and usability as well as facilitate rapid prototyping and deployment. The automatic synthesis approach enhances interoperability between Julia and C environments and increases the usability of fuzzy logic systems in traditional engineering domains. The fuzzy logic enables rapid prototyping and testing in Julia, combined with fast deployment in C, accelerating the development cycle and bringing fuzzy logic applications to market faster. (Perkel, 2019)

This work is both academically significant and highly relevant to industry needs. By enabling the easy and automatic translation of fuzzy logic systems into C, this thesis supports the broader adoption of advanced fuzzy logic models across various sectors, including automotive, robotics, consumer electronics, and more.

1.3 Aim of the Thesis

This paper is mainly supported by several clear goals, each dedicated to creating a framework model to transform fuzzy logic systems from high-level environments into more controlled and efficient low-level implementations.

Based on the research method Design Science Research which will be explained in the chapter 3, the corresponding research question is also introduced to clarify the direction and purpose of the study.

The central research question guiding this thesis is:

"How can fuzzy logic systems be automatically synthesized into efficient C code, with enhanced performance, and how can this synthesized code be verified and benchmarked against existing implementations and manually written C programs?"

This question points out the main challenges. It is hard to change fuzzy logic systems from high-level code to low-level C code. It is also important to make the C code faster and test if it works well. To answer this question, we need to build a tool. This tool must keep the original logic from Julia and also make sure the C code runs well on real devices like embedded systems.

Keeping this research question in mind, the next section presents the core objectives of the thesis. These objectives define the steps needs to be developed a robust and user-friendly C code generator and to ensure its successful deployment and validation in practical.

The primary objective of this thesis is to develop an automatic translation that can convert fuzzy logic systems written in Julia's FuzzyLogic.jl into optimized C code. This tool will help deploy complex fuzzy logic algorithms in environments that require C programming because the C language has advantages in terms of efficiency and hardware control. Such transformations allow high-level language systems to be effectively used in low-level environments with limited resources.

The key to this is to keep the generated C code consistent with the original model developed in Julia. This means maintaining the logical structure of the fuzzy system during the translation process. Maintaining the integrity of these translations is critical to ensuring that the efficiency and performance of the original system is accurately replicated in the new implementation.

Another important objective is to improve the performance of the generated C code by implementing optimizations that can improve computational efficiency and reduce memory usage. These optimizations are particularly important for deploying these systems in embedded or real-time applications because these applications have very limited resources.

In addition, synthesis tools should be designed to be user-friendly and easy to integrate. Providing clear documentation and strong support so that various parameters can be debugged in generated code, or integrated generated C code into larger systems, an easy-to-use tool will promote its widespread adoption of it in science and industry, expanding the impact of research.

The third objective of this thesis is verifying and benchmarking synthesized C code through rigorous testing and comparison with existing implementations and manual coding practices. This testing showed that the automatic code generator works correctly and has good performance. Running benchmarks will give real-world proof that the tool is effective and useful compared to older methods.

The tool should be able to work with many types of fuzzy systems, especially Mamdani and Sugeno, which are the most commonly used. Supporting different types makes the tool more useful in many areas and for different fuzzy logic applications. This improves its overall value and practicality

Successful completion of these goals will lead to 3 key outcomes:

1. Comprehensive C code generator: Develop a full-featured tool to seamlessly transform the fuzzy logic system defined in FuzzyLogic.jl into highly optimized C code. The tool will encapsulate the complexity of fuzzy logic transformations while providing an efficient and deployable solution.

2. Documentation and user guides: Create comprehensive documentation and detailed user guides. These materials will support users in operating the tool and demonstrate its usefulness through practical examples and use cases.

3. Contribution to the open-source community: Ideally, this tool will be available as an open-source resource. This openness facilitates further development and collaboration within the fuzzy logic and broader computing community, enhancing the collective toolset for researchers and practitioners.

Achieving these goals and results will significantly advance the field of fuzzy logic applications, especially in transforming high-level conceptual designs into practical, deployable C implementations. This advancement will bridge current gaps in deploying fuzzy logic systems and promote their widespread adoption and more innovative applications in various industries.

2 Literature Review

This chapter provides an overview of fuzzy logic, introduces the Mamdani and Sugeno inference systems, explains how fuzzy logic and related libraries functions, and briefly discusses automatic code generation. By the end of this chapter, the reader will have a foundational understanding of fuzzy logic systems and how the FuzzyLogic.jl C code generator works.

2.1 Fuzzy Logic’s history and applications

Fuzzy systems have been connected to various practices and studies in the field of industrial engineering. Fuzzy Inference Systems have been used to calculate dimensions of sustainability (Mastrocinque et. al. 2022) as well as decision making (Azadnia 2021). Fuzzy logic has been connected in the decision making related to the selection of lean tools in manufacturing (Manickam 2022), choosing of technology (Maretto 2024) as well as control system of production (Huo 2020).,

Fuzzy systems use fuzzy logic to process inputs and generate outputs. Unlike traditional logic, it sticks to strict rules like "If A is more than 3, then B must be 5," fuzzy logic is more flexible. It deals with situations where conditions are not so clear-cut. For instance, a fuzzy rule might say, "If the food is bad, then the tip should be lower." This way, the system mimics human thinking, making it better at handling uncertainty and vague information. Because of this, fuzzy systems are advantageous when inputs and outputs are not precise or predictable (Wang, 1997)

The concept of fuzzy systems was first introduced in 1965 by Zadeh, who introduced fuzzy sets (Zadeh, 1965). Since then, fuzzy logic has grown into a strong theory. One of the most important parts of fuzzy logic is fuzzy sets, and they are the key in fuzzy systems. Instead of just putting things into simple "yes" or "no" groups, fuzzy sets let things belong to different groups simultaneously but in different amounts.

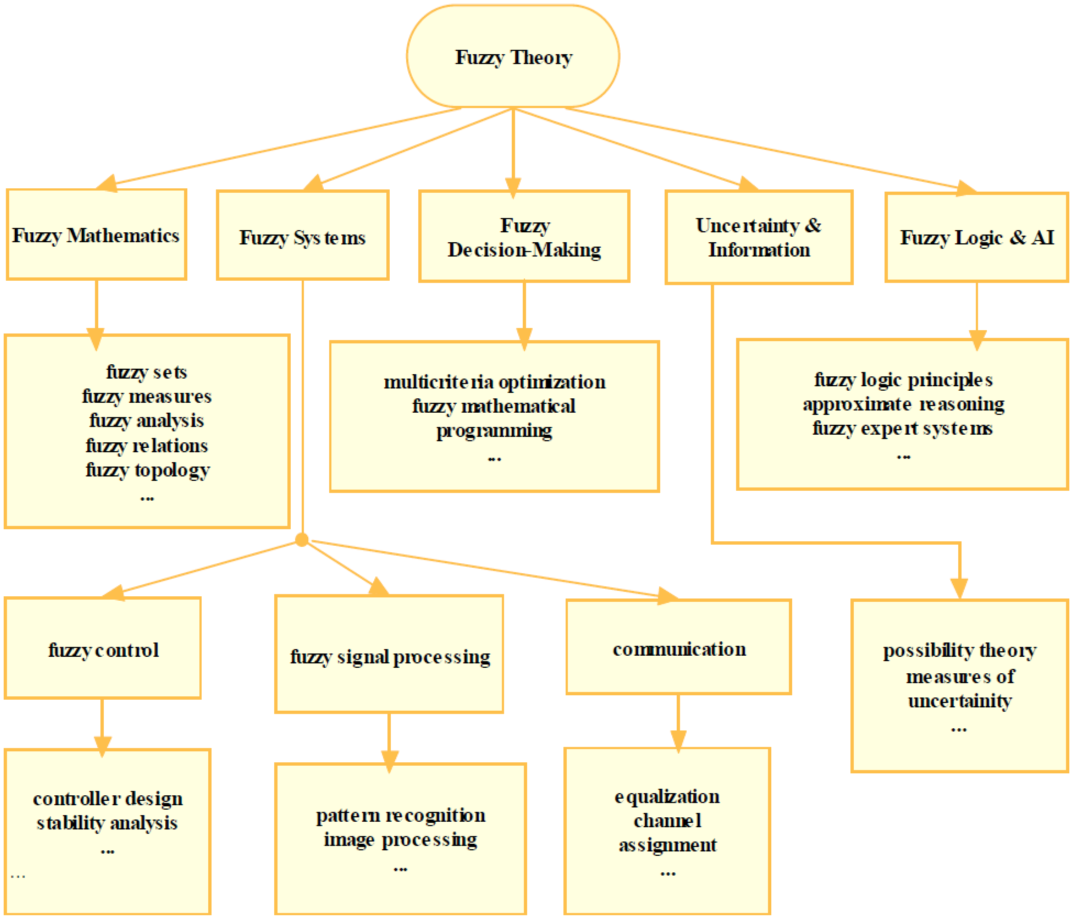


Figure 2. Fuzzy theory structure and development

Fuzzy logic methods handle situations that can be partially true or false, making them particularly useful in real-world applications where conditions are often not entirely true or false. Unlike traditional logic, which requires states to be either true or false (represented by 1 and 0 in computer systems), fuzzy logic describes states using degrees of membership. This allows a status to be expressed as a certain percentage of truth or falsity. (Ferranti, 2024)

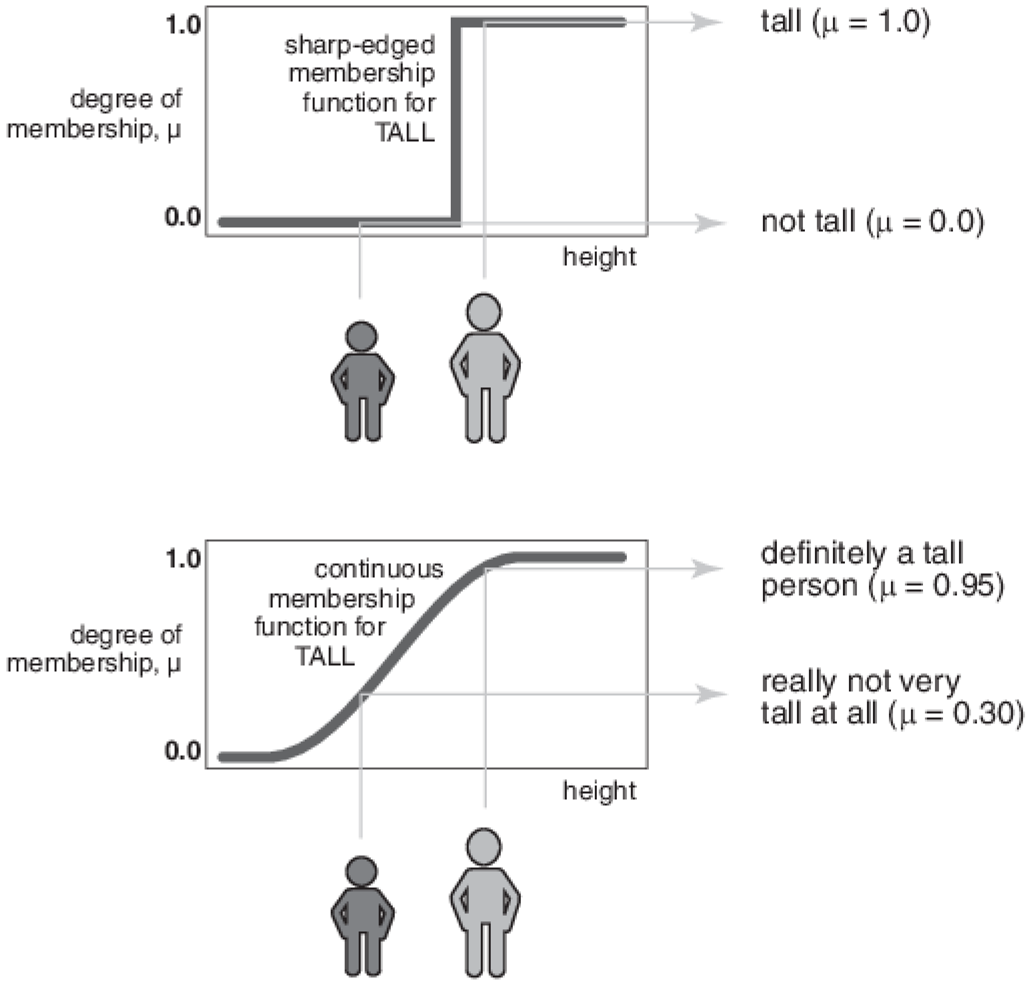


Figure 3. An example of fuzzy membership compared (MathWorks n.d).

In human natural logic, people also don’t think and treat things mathematically. In human language, we have many words like hot, tall, and young, which describe things very vaguely but play an important role in human logic. In human thinking patterns, people do not perceive issues in a purely black-and-white manner. But in the computer world, it only judges things as 0 or 1. These two Boolean numbers To get computers to solve problems in a human-like way, fuzzy logic has been introduced to handle more complicated problems, which are hard to create with the classic mathematical model.

Fuzzy logic systems are important and useful in people's daily lives. They have been widely used in control systems, decision-making processes, artificial intelligence, and other fields.

Picture 2. Rice cooker using fuzzy system (Zojirushi, n.d)

Fuzzy logic extends traditional binary logic by allowing for degrees of truth between absolute true and false, thus providing a more nuanced approach to reasoning about uncertain or imprecise data. Fuzzy logic uses fuzzy sets, where elements have different degrees of belonging to a group. This makes fuzzy logic really helpful for complicated systems where normal logic don't work well because real-world situations is often vague. The process usually has few steps: fuzzification, rule checking, aggregation, and defuzzification. These steps helps the system to understand unclear information and make decisions even when not all data is complete. Fuzzy logic is now very important in many areas like control systems, decision making, and AI, because it gives strong solutions to uncertainty problems (Ferranti, 2024).

Definition and History

Lotfi Zadeh talked about fuzzy logic for the first time in his very famous 1965 paper "Fuzzy Sets." This logic system is different because it uses true or false logic, like classical binary logic, and can handle half-true things. In fuzzy set theory, things belong to a group by different levels, not only yes or no, but can be between 0 and 1. This level is called the membership function, which tells how much something belongs to a set (Zadeh, 1965). This was a very new idea at that time because it let people use math to explain real-world things that were not always clear.

Zadeh wants to make fuzzy logic because he thinks regular math is insufficient to deal with complex and unclear things. The first-time people used fuzzy logic is in the control system, and it works very well. One good example is the train system. Before, train control used the traditional control method, and the movement was not so smooth. However, the train can accelerate and stop more gently after using a fuzzy controller (Zadeh, 1965).

During the 1970s and 1980s, fuzzy logic becomes more and more better. Especially in Japan, researchers use fuzzy logic in factory automation, home electronics, and car control systems (Lin, 2009). In the early 1990s, Japanese companies make the first product that used fuzzy logic: a washing machine. This washing machine can change washing time and water level by checking how dirty the clothes are. This shows that fuzzy logic is not only for science but also can make life easier.

Many scientists realize that fuzzy logic can do much more. It started to be used in artificial intelligence, where it helps make expert systems and decision-support systems. These systems work like a human brain because they can deal with unclear or missing information (Sakunthala, Kiranmayi, and Mandadi 2017). In economy and finance, fuzzy logic is also instrumental. People use it to predict the stock market and calculate risk because financial data always have many uncertain things.

Later, fuzzy logic became even stronger. One example is Type-2 fuzzy logic. This new fuzzy logic can handle more uncertainty because it not only uses a simple membership function but also makes the membership function fuzzy. This means it can understand even more complicated and unclear data.

Because computer software is powerful now, using fuzzy logic becomes easier. Many tools, like MATLAB's Fuzzy Logic Toolbox and Julia's FuzzyLogic.jl, make designing and testing fuzzy logic systems possible without writing too much code. These tools help many people use fuzzy logic in many areas, including science and industry.

Applications of Fuzzy Logic

Fuzzy logic is instrumental because it can deal with uncertainty and work similarly to human thinking. This is why it has been used in many industries. One of the most early and famous uses of fuzzy logic is in control systems. For example, in the car industry, fuzzy logic controllers have been used to control automatic transmission. It helps the car choose the best gear by considering different factors like speed and engine load. Also, in home appliances, Kumar and Haider (2013) explain that fuzzy logic lets washing machines change washing time and water amount by checking how big and dirty the clothes are. This can make washing machines more smart and save energy.

Fuzzy logic also plays an important role in environmental control. HVAC systems help control temperature and humidity by using sensor data and fuzzy rules. This way, it can make air conditioning more comfortable and save more energy than the traditional way. Also, in water treatment, fuzzy logic controls how much chemical to put in water, so even if water quality changes, it still can keep good standard.

Another big area where fuzzy logic is used is in robotics. Robots use fuzzy logic to make decisions and control movement because it can deal with uncertain and changing environments. This is very important for self-driving cars. Fuzzy logic helps process data from different sensors to understand the road and decide how to drive in a safe way.

Fuzzy logic is also very useful in the medical field. It is used in medical decision support and diagnosis systems. These systems use fuzzy logic to analyze patient information, even when data is not clear or missing some parts. In this way, fuzzy logic can help doctors and medical staff take care of patients, suggest the best treatment plan, and find out what illness they have. This makes medical decisions more accurate and more reliable because it thinks about patient problems in a similar way as human experts do.

Fuzzy logic also has a big influence on finance and the economy. It predicts stock market changes, checks credit risk, and manages investment. Because financial data are always uncertain and unclear, fuzzy logic is better than traditional methods. It can give more correct and stable results for financial decision-making.

In artificial intelligence, fuzzy logic also has many contributions. It is used to make innovative systems that need to think like humans. Expert systems, which try to copy human expert’s decision-making, usually use fuzzy logic to deal with information that is not clear. This method is often used in things like natural language processing, pattern recognition, and machine learning.

In general, fuzzy logic is very powerful because it can be used in many fields (Ross 2010). It is good at dealing with uncertainty and making logical decisions, which traditional binary logic cannot do well. As technology keeps improving, fuzzy logic will continue to be used more in different industries, bringing new innovation and higher efficiency.

2.2 Mamdani and Sugeno Fuzzy Inference Systems

As an effective method to deal with uncertainty and fuzzy information, fuzzy logic systems are widely used in modern control and decision-making systems. Mamdani fuzzy system is one of the most classic and commonly used. In 1975, the Mamdani Fuzzy Inference System (Mamdani FIS) proposed by Ebrahim Mamdani used fuzzy set theory and fuzzy rules to infer, providing a feasible control and decision-making framework for complex systems. This paper will detail the Mamdani fuzzy logic system's principles, structure, and applications, aiming to reveal its advantages and limitations in practical problems.

The core of the Mamdani fuzzy system is to encode the knowledge of human experts in the form of fuzzy rules, thereby realizing control and decision-making of complex systems (Izquierdo & Izquierdo, 2017). Its basic structure includes four main steps: fuzzification, rule reasoning, aggregation, and defuzzification. The fuzzification step converts the precise input value into a fuzzy set and expresses the degree of fuzziness of the input variables through a membership function. Rule reasoning uses the "if-then" rule for fuzzy reasoning and generates fuzzy output based on the fuzzy values of the input. The aggregation step combines the fuzzy outputs of all rules into a comprehensive fuzzy set. Finally, defuzzification converts the comprehensive fuzzy set back to precise values and outputs the system's final decision or control instructions.

Implementing Mamdani fuzzy systems usually involves multiple input and output variables, each represented by multiple fuzzy sets. In practical applications, such a system can be implemented through the following steps: first, define fuzzy sets and membership functions of input and output variables; then, establish a fuzzy rule base based on expert knowledge or experimental data (MathWorks, n.d.). Next, the output fuzzy sets are calculated through a fuzzy reasoning mechanism. Finally, a de-fuzzification method is used to obtain accurate output values. In this process, the selection of membership functions and the construction of rule bases are the keys to system design, which directly affect the performance and accuracy of the system.

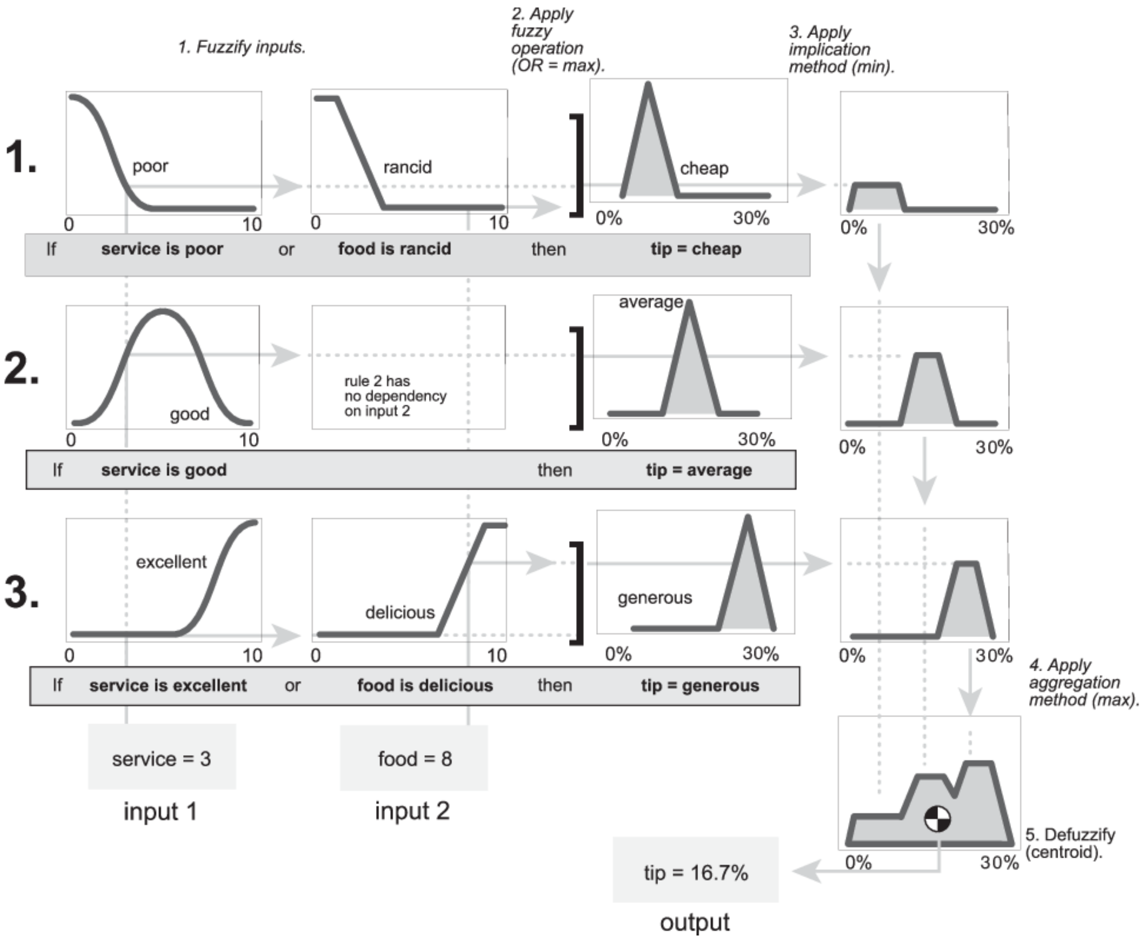


Figure 4. The inference process of a Mamdani system (MathWorks, n.d.)

Because Mamdani fuzzy systems are simple to comprehend and apply, they are widely used in many different industries. Mamdani fuzzy systems, for instance, are employed in industrial control to regulate intricate nonlinear processes including chemical reaction control, temperature control, and pressure management. Medical diagnosis and financial investment decisions are two examples of multi-criterion decision-making problems that are handled by decision support systems. Furthermore, the Mamdani fuzzy system has demonstrated good practical results and high application potential in the domains of robot control, intelligent transportation systems, and pattern recognition.

The main advantage of the Mamdani fuzzy system is that it is easy to understand and implement, and is particularly suitable for formalizing the knowledge of human experts. At the same time, it is highly robust and can handle uncertainty and fuzzy information.

However, as the number of input variables and rules increases, the computational complexity of Mamdani fuzzy systems increases significantly, which may lead to real-time problems. In addition, the defuzzification process requires high computing resources, especially when precision requirements are high. Therefore, in practical applications, system complexity and computational efficiency need to be weighed off to achieve optimal performance.

As an effective tool for processing fuzzy information, Mamdani fuzzy logic system has been successfully applied in many fields. Its reasoning through fuzzy rules makes the control and decision-making process of complex systems more intuitive and interpretable. Although there are challenges in terms of computing complexity and real-time, system performance can be effectively improved by optimizing algorithms and hardware acceleration. In the future, with the development of artificial intelligence and machine learning technology, Mamdani fuzzy systems are expected to play a greater role in more fields, providing new methods and ideas for solving complex practical problems.

Fuzzy Inference System	Advantages
Mamdani	Intuitive
	Well-suited to human input
	More interpretable rule base
	Have widespread acceptance
Sugeno	Computationally efficient
	Work well with linear techniques, such as PID control
	Work well with optimization and adaptive techniques
	Guarantee output surface continuity
	Well-suited to mathematical analysis

Table 1. Mamdani and Sugeno Fuzzy Inference System advantages.

The Sugeno fuzzy system is another widely used fuzzy reasoning method proposed by Takagi and Sugeno in 1985. Unlike Mamdani fuzzy systems, the output of Sugeno fuzzy systems is usually a linear function or constant, which makes them simpler and more efficient in mathematical processing and optimization (Bergsten, Palm, & Driankov, 2002). This article will introduce Sugeno's fuzzy system's basic principles, structure, and practical applications and discuss its advantages and limitations.

The reasoning process of Sugeno's fuzzy system is similar to that of the Mamdani system, but its output part is processed differently. The rule for Sugeno's fuzzy system is of the form: "If x is A and y is B, then z=f(x,y)", where f(x,y) is usually a linear function or constant. The purpose of this design is to simplify the calculation process of the output and make the system more efficient when handling complex control tasks. Specific steps include: fuzzification, rule reasoning, aggregation and output calculation. Fuzzification converts inputs into fuzzy sets, rule reasoning is processed based on fuzzy rules, aggregation combines the results of multiple rules, and finally calculates accurate output through methods such as weighted average.

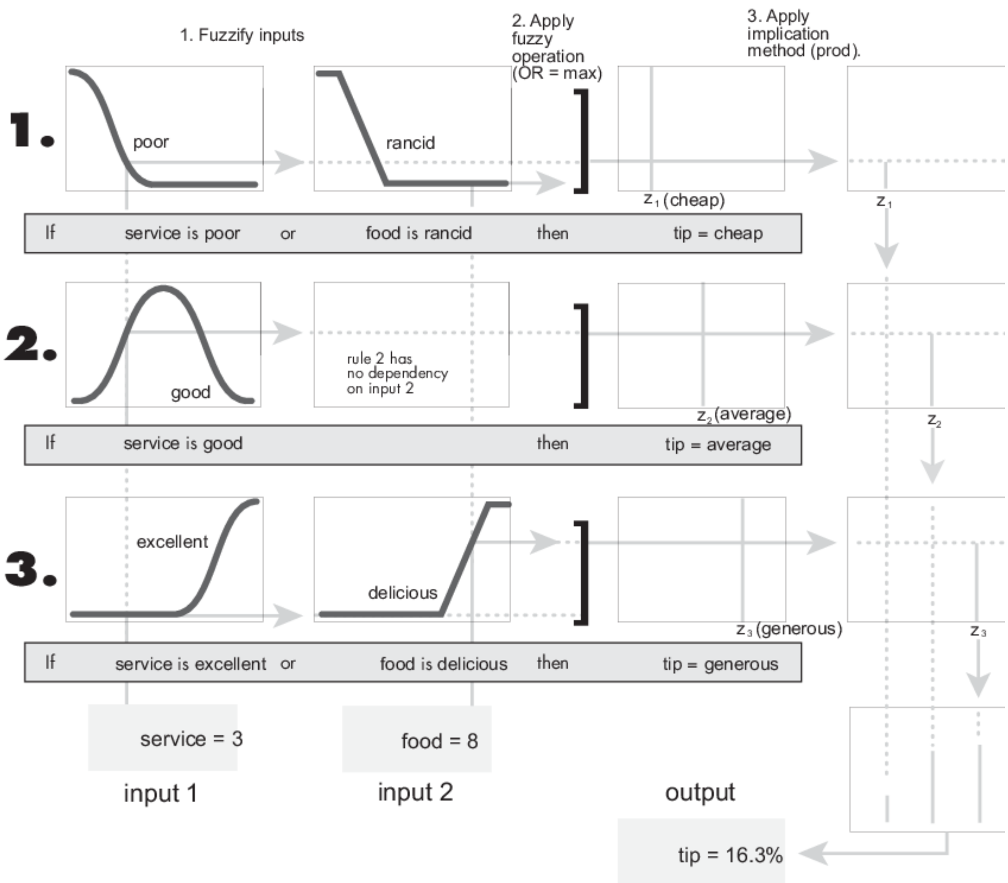


Figure 5. The inference process of a Sugeno system (MathWorks, n.d.)

The structure of Sugeno's fuzzy system includes input variables, fuzzy rules, and output functions. The implementation process including:

- Fuzzification of input variables: Convert the precise input value into a fuzzy set, and use a membership function to express the fuzzy degree of input variables.
- Definition of fuzzy rules: Build a fuzzy rule library based on expert knowledge or data, and the output part of each rule is a function.
- Rule reasoning: Calculate the applicability of each rule through fuzzy logic, and calculate the fuzzy output based on the output function of the rule.
- Aggregation and output calculation: Weighted average the output results of all rules to get the final output of the system.

For example, for a system with two input variables, the rule can be expressed as:
If x is A and y is B, then z = px + qy + r, where p, q, and r are constants.

Sugeno fuzzy systems are widely used in fields such as automatic control and pattern recognition, while the Mamdani system has been used more than decision-making. because of the Sugeno system’s simple output calculation and suitability for optimization. So it works better in real-time process system. For example, In automatic control, Sugeno fuzzy systems are used to design adaptive controllers that can dynamically adjust control parameters based on the state of the system. For example, Sugeno fuzzy systems in self-driving cars can be used for path planning and speed control to ensure the vehicle travels safely and efficiently in complex environments (Onieva, Alonso, Pérez, Milanés, & De Pedro, 2009). In the field of pattern recognition, the Sugeno fuzzy system is used for image recognition and speech recognition, processing input signals through fuzzy rules to achieve high-precision recognition results. In addition, Sugeno fuzzy systems also have wide application prospects in financial forecasting, medical diagnosis, and other fields.

A typical application example is using the Sugeno fuzzy logic system to design an autonomous driving control system. Traditional driving control systems make it difficult to cope with complex and changeable driving environments. However, the Sugeno fuzzy system can dynamically adjust control strategies based on real-time road conditions and vehicle conditions to improve driving safety and comfort.

- Input variables: The input variables of the autonomous driving control system include vehicle speed, distance from the vehicle in front, and road curvature.
- Output variables: Output variables include acceleration and steering angle.
- Membership function: Define a fuzzy set and membership function for each input variable. For example, the vehicle speed can be defined as low, medium, and high; the distance from the vehicle in front can be defined as near, medium, and far.
- Fuzzy rule base: Build a fuzzy rule base based on expert knowledge and experimental data, such as:If the vehicle speed is low and the distance from the vehicle in front is close, then the acceleration is negative and the steering angle is small.If the vehicle speed is high and the road curvature is large, then the acceleration is negative and the steering angle is large.
- Output function: Define the corresponding linear output function according to the conditional part of the fuzzy rule.

Through the above design, the system can analyze the driving environment in real time and dynamically adjust the control parameters of the vehicle according to fuzzy rules to ensure the safe driving of the vehicle under complex road conditions.

The main advantages of Sugeno fuzzy systems are their simple calculation and accurate output, making them particularly suitable for optimization and adaptive control. At the same time, the Sugeno system is easy to combine with other mathematical tools, such as neural networks and genetic algorithms, to further improve system performance. However, Sugeno fuzzy systems rely on expert knowledge for rule definition and membership function selection, which may lead to complex system design. In addition, for systems with high-dimensional input variables, the number of fuzzy rules will increase sharply, increasing the difficulty of calculation and management (Terano, Asai, & Sugeno, 2014).

As an efficient fuzzy reasoning method, Sugeno fuzzy logic system has wide application prospects in fields such as automatic control, pattern recognition and decision support. It uses a simple linear output function to make the system more efficient and accurate in the calculation and optimization process. Although there are certain challenges in system design and rule management, system performance can be further improved through the combination of optimization algorithms and intelligent technology. In the future, with the development of artificial intelligence and automation technology, Sugeno fuzzy systems are expected to play an important role in more fields, providing new methods and ideas for solving complex practical problems.

2.3 Fuzzy Logic Tools and Libraries

Many tools have been built and developed to create a fuzzy logic system faster. These tools let users build fuzzy systems with features like membership functions, rule-making, and decision-making. This chapter talks about popular fuzzy logic tools and what they can do. It especially looks at FuzzyLogic.jl, a library built in Julia, along with others like MATLAB fuzzy logic toolbox, fuzzylogic for python, Fuzzylite, Jfuzzylogic, and eFLL,

In recent years, fuzzy systems software (FSS) has been developed with more focus on making fuzzy systems more straightforward to use and more uncomplicated for implementation. Some tools, for example, Fuzzy Logic Toolbox for MATLAB and Fuzzy Logic 2 add-on for Mathematica, are provided through commercial distribution. However, many FSSs come from the efforts of the scientific community. Many of these projects are from open-source development, like FisPro and KEEL, which are often freely available. Because these tools are open-source, the way fuzzy systems are developed has changed. Before, many barriers made it difficult to improve FSS, but now, these barriers is reduced. This open-source approach make it possible to find errors more quickly, encourage new creative ways of using fuzzy logic, and speed up the process of applying fuzzy systems in many academic fields and industries (Alcalá-Fdez & Alonso, 2015).

FuzzyLogic.jl is a modern approach to implementing fuzzy logic. Developed in Julia, this library combines the language's high-performance capabilities with intuitive syntax, making it a powerful tool for researchers and developers. Key features of FuzzyLogic.jl include Ease of Use, Modular Design, and High Performance.

This library has a simple syntax, which makes it easier to create a fuzzy inference system. For example, a fuzzy system can be started with expressions like fis(service = 2, food = 3), so the design process becomes more simple. It also supports many types of membership functions, rule definitions, and inference methods, so users can change the system based on their needs. Because Julia is very fast, FuzzyLogic.jl is a good choice for the applications that need real-time processing, like robots and control systems (Ferranti & Boutellier, 2023).

The library is very flexible and also efficient in computation, which makes it very suitable for advanced fuzzy logic use. It also connects well with Julia's bigger system, which makes it even more attractive to use.

MATLAB's Fuzzy Logic Toolbox has a large user base, and it is a particularly popular tool for fuzzy logic. It is intended to support the design, simulation, and implementation of fuzzy logic systems. Numerous helpful functions in the toolbox enable users to quickly construct fuzzy inference systems. Users may quickly define fuzzy sets, develop rule bases, and perform fuzzy inference because to its user-friendly graphical user interface (GUI). Users can create their own membership functions and rule sets, and it supports both the Mamdani and Sugeno fuzzy inference systems. This toolbox is highly helpful for control system design, data analysis, and optimization activities since it can be used in conjunction with other MATLAB tools such as Simulink and Optimization Toolbox (MathWorks, n.d.).

It is worth noting that several fuzzy libraries written in other programming languages, such as Python, Java, and C++, are available online. However, the majority of these libraries currently lack associated academic publications.

Fuzzy Logic for Python 3 is a fuzzy logic library developed by Anselm based on an MIT license. This fuzzy logic library in Python simplifies implementing fuzzy systems within the Python ecosystem. Its user-friendly design makes it highly suitable for prototyping and educational purposes, allowing users to experiment with fuzzy logic concepts without requiring extensive programming knowledge (PyPI, n.d.).

jFuzzyLogic is a Java-based fuzzy logic library that provide strong and rich functions for designing, testing and using fuzzy systems. Because it deeply connect with Java enterprise system, developers can easily use fuzzy logic in real-time applications, such as automatic monitoring, fault finding, and predictive analysis. Also, this library have powerful debugging and simulation functions, which make it very helpful for big projects that need high precision in control and making decisions. Since it is open-source, it keep getting updates and support from global developers, ensuring it can be used for long time (Cingolani & Alcalá-Fdez, 2013).

eFLL (Embedded Fuzzy Logic Library) Designed specifically for embedded systems, **eFLL** is a lightweight and highly optimized library tailored for resource-constrained environments. Its small memory footprint and efficient computational capabilities make it indispensable in the development of Internet of Things (IoT) devices, robotics, and sensor-based systems. The library's flexibility allows developers to implement complex fuzzy systems directly on microcontrollers, enabling real-time decision-making and control. Its extensive use in robotics highlights its practical utility in dynamic and adaptive environments (Alves, n.d.).

The different kinds of fuzzy logic tools and libraries show that fuzzy logic is very useful in many areas. Each tool has its own special advantage, and they are made for different needs—some for high-speed computing, some for embedded systems, and some for fast prototyping. As fuzzy logic keeps developing, these libraries will become more and more important to connect theory with real-world use. In the future, it is expected that development will focus on making tools easier for users, improving computing speed, and making them work better with other technologies.

2.4 Automatic Code Generation

Automatic code generation is essential in developing fuzzy logic systems. It enables efficient and accurate conversion of high-level models into executable code. This section discusses the role of FCL (Fuzzy Control Language) and FML (Fuzzy Markup Language) in this process, explores the transition from high-level languages to C, and reviews related work in the automatic synthesis of fuzzy systems.

Fuzzy Control Language has been defined by IEC 61131-7 as the standard programming language, which is specifically made for creating fuzzy logic controllers. Developers can define fuzzy variables, membership functions, and inference rules in an organized and comprehensible manner with FCL. Conversely, the XML-based Fuzzy Markup Language (FML) facilitates the modeling and sharing of fuzzy systems. FML is useful for modular and flexible applications since it defines fuzzy variables, rules, and inference techniques using a hierarchical framework (Cingolani & Alcalá-Fdez, 2012). Because they bridge the gap between machine-executable code that can execute fuzzy system designs and human-readable fuzzy system designs, FCL and FML are crucial.

Converting high-level languages such as Julia, Python, or MATLAB into low-level languages like C is very important for deploying fuzzy systems in real-time environments. Some tools can generate fuzzy logic systems by using FCL and FML methods (Ferranti & Boutellier, 2023). This kind of tool can be improved by adding an extension function that allows automatic conversion of fuzzy systems into low-level programming languages like C. Such extension would provide an easy-to-use interface for defining fuzzy systems and at the same time generate efficient C code that is optimized for embedded systems. The process usually includes parsing high-level descriptions, optimizing rule structures, and generating executable code that fit to the target platform.

Researchers have studied the automatic synthesis of fuzzy systems for many years. In the past, fuzzy rule-based systems were written manually. It was time-consuming and easy to make errors. In recent years, several tools and libraries have been developed to automate this process, such as MATLAB's Fuzzy Logic Toolbox, FCL parsers, and FML-based frameworks (Kozlov & Kondratenko, 2023). These tools support various inference methods including Mamdani, Sugeno, and Type-2 fuzzy systems.

Comparisons between modern tools have shown that libraries such as FuzzyLogic.jl have much better performance than traditional platforms like MATLAB, while still being easy to use and extend. This makes them very good for both academic research and industry applications.

The combination of FCL and FML with automatic code generation tools has made deploying fuzzy systems much easier. By connecting high-level modeling languages with low-level implementation, these technologies improve the efficiency, reliability, and scalability of fuzzy logic applications. Moreover, research in automatic fuzzy system synthesis is still ongoing, helping these systems adapt better to real-world challenges.

3 Methodology

In this chapter, design science research will be discussed and explained. Because it is the main method which directs this research

3.1 Design science research

Design Science Research (DSR) is a methodology for solving practical problem through scientific approach. It focus on creating and evaluating artifacts, like systems or models, to improve real-world situations. Unlike traditional research, DSR combine both practice and theory, making it very unique and effective for innovation (Hevner et al., 2004)

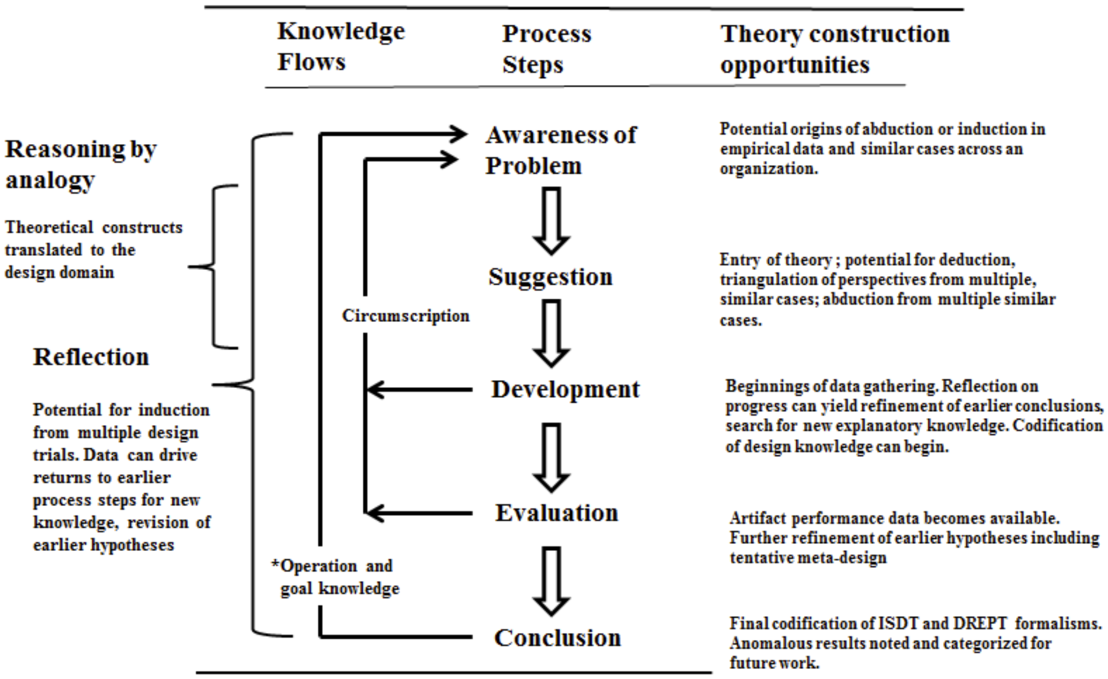


Figure 6. High-Level Design Research Process (Kuechler & Vaishnavi 2008)

In Design Science Research (DSR), awareness of the problem is the first and most important step. This is because, without understanding what problem need to be solved, it is impossible to make good solutions (Kuechler & Vaishnavi 2008). Researchers must carefully study the situation to know why the problem happen and who is being affected.

Sometimes, people only see the problem on the surface, but deeper analysis can show that the real issue is more complex. For example, if a company has low sales, the reason might not just be a bad product but may also be poor customer service or marketing.

To understand the problem, the researcher also should look at existing research and solutions. If they ignore this, they might create something that already exist or not useful for the situation. Also, they should connect the problem with scientific theory. This makes sure the solution is not only practical but also contribute to knowledge. However, this part is not easy because real-world problems are often messy, and researcher might not have clear answers right away.

Another important part is to work with stakeholders who face the problem in daily life. These people can give important insights, but sometimes they explain problem based on feelings instead of facts. Researcher need to carefully analyze their input to find true causes. In conclusion, awareness of the problem is very critical, and researcher must take time to understand it fully before moving forward.

The suggestion phase in Design Science Research (DSR) is about proposing solution to the problem that was identified earlier. This step is where researcher design new artifact, such as a model, method, or system, that can solve the problem. In this stage, creativity is very important because the solution must be innovative and fit the real-world context. However, some researcher might focus too much on technical detail and forget about the usability of the artifact. It is important to balance technical and practical aspects of the design (Hevner et al., 2004).

When making suggestion, the researcher also needs to consider existing knowledge and theory. If the new solution has no scientific base, it might not be accepted by academic community. At same time, the solution must be evaluated by the stakeholders to make sure it is useful for their needs. Sometimes, researcher might face problem where stakeholders want very specific solution, but it is not possible to achieve due to resource limitation. In this case, researcher need to find compromise.

In Design Science Research (DSR), the development phase is when the artifact is build to solve the defined problem. This artifact can be a model, tool, or even framework that aim to improve practical issues. Researcher must focus not only on technical design but also usability to make sure it works well in real-world. Often, the development process is iterative, needing multiple revisions to address feedbacks and limitation. The success of this phase depends on how well the artifact aligns with the problem and research goals (Hevner et al., 2004).

In Design Science Research (DSR), evaluation is very important to check if the artifact really solve the problem identified earlier. This phase uses methods like experiments, simulations, or case studies to test the artifact. Researcher must compare the performance of the artifact with the goals set in the beginning. Sometimes, results show that the artifact is not perfect or has unexpected issues. In such cases, researcher may need to make changes and test again. Evaluation also include feedbacks from users or stakeholders, which is helpful to understand how practical and useful the artifact is in real-world situations.

After evaluation, the conclusion is where the researcher summarize the main findings of the study. This part explains how the artifact contributes to solving the problem and also how it adds to existing knowledge. It is also important to talk about the limitations of the research, like if the artifact only works in specific situations or has certain constraints. The conclusion should suggest future research directions, so others can continue improving on the work. A good conclusion ties the entire research together, showing both its strengths and its areas for improvement.

3.2 Overview of design science approaches

Because Design Science Research (DSR) is a practical research method that focuses on solving real-world problems. It connects practical uses with theoretical ideas by creating and testing new tools or systems (O'Keefe, 2014). So In this chapter I will explains how to use design science research to make a tool that changes fuzzy logic systems from high-level programming languages, like Julia, into optimized C code. The process of design science research has several repeated steps: understanding the problem, suggesting solutions, creating the tool, testing it, and drawing conclusions. Each step helps move from finding the problem to solving it step by step.

Awareness of Problem

As the first step of the DSR methodology, we need to be clear of the problem. AI Stratagems (2023) reports that there are over 50,000 patents involving fuzzy logic, with more than $10 billion in product sales and profits estimated in the billions. However, there are big challenges with current fuzzy logic libraries. For example, MATLAB and its Fuzzy Logic Toolbox require expensive licenses. The toolbox also uses its own .fis format to store the implemented models and does not support reading/writing FCL nor FML (Ferranti & Boutellier, 2023,). Octave's Fuzzy Logic Toolkit also has strict licensing rules. FuzzyLogic.jl only works in the Julia programming environment. Because of this, it will be usefull to have an open-source fuzzy logic toolbox that can work across different platforms to meet the needs of many applications

In addition, many fuzzy logic libraries are restricted to specific environments. For instance, jFuzzyLite and jFuzzyLogic, which are Java-based fuzzy logic libraries, can only operate within the Java environment. (Cingolani & Alcalá-Fdez, 2012). Similarly, Python-based libraries like PyFuzzyLite, PyFuzzy, and Fuzzy Lab are limited to the Python environment. However, in real-world applications, there is a significant demand for fuzzy logic in C-based embedded devices (Alcalá-Fdez & Alonso, 2015).

Suggestion

To solve the problem, one suggestion could be that using powerful and convenient fuzzy logic library, FuzzyLogic.jl. It has been developed by using the Julia programming language, offers significant advantages for fuzzy logic modeling. These advantages mainly include high performance, ease of use, and a robust ecosystem.

A high-performance programming language that blends the advantages of compiled and interpreted languages is Julia, to start. Julia can produce effective machine code with Just-In-Time (JIT) compilation, which significantly speeds up execution. For fuzzy logic systems, which frequently require analyzing big information and carrying out intricate computations, this property is very important. Because of its great performance, Julia can efficiently manage large-scale data processing and real-time systems, cutting down on calculation time without sacrificing accuracy.

Second, developers can write and maintain code more easily and effectively because to Julia's straightforward and understandable syntax. Julia contains a lot of built-in arithmetic functions and data types because it was initially designed to make scientific computing easier. High-level mathematical computations and matrix operations are also natively supported. Because fuzzy logic systems frequently require complex mathematical calculations and logical reasoning, these functions are particularly beneficial.

Julia offers strong metaprogramming capabilities, enabling developers to create efficient macros and generate code automatically. This not only improves development efficiency but also enhances code reuse (Ferranti & Boutellier, 2023). The combination of flexibility and ease of use makes Julia an excellent choice for developing fuzzy logic systems.

Julia's ecosystem is another significant advantage. It offers a wide range of libraries and packages that cover areas such as data processing, numerical calculations, and machine learning. In the field of scientific computing, Julia's ecosystem is particularly well-developed, allowing users to leverage existing libraries to build complex fuzzy logic systems easily. For instance, JuMP is used for mathematical optimization, Plots for data visualization, and DataFrames for data processing. These tools enable developers to construct and validate fuzzy logic models efficiently.

Julia is also quite good at collaborating with other programming languages. Its External Function Interface (FFI) makes it simple for developers to call Python and C code. This gives fuzzy logic systems greater flexibility and improves their usability by making it easier to link and grow them using various tools and platforms.

This situation reflects the well-known "Two-Language Problem" in software engineering. It’s a term that refers to a problem: high-level languages being used for logic, and low-level languages being needed for performance and hardware control (Perkel, 2019). For example, using Julia to build fuzzy logic systems and using C to handle low-level operations is a typical case. While this approach combines the strengths of both languages, it also brings challenges, such as increased complexity in integration, difficulty in maintenance, and the need for performance tuning across different platforms.

Development

To enhance the solution, it is essential to incorporate a C code converter into Julia's fuzzy logic library. This converter would allow developers to translate fuzzy logic code written in Julia into C, effectively addressing the gap between high-level logic and low-level system control. By implementing this solution, development workflows could be significantly simplified, leading to improved efficiency and usability of fuzzy logic systems.

Fuzzy logic, as an effective method for handling uncertainty and fuzziness, is a natural fit for application in Julia. However, the implementation of fuzzy logic systems often requires direct hardware control or high-performance computations, tasks that are typically handled using the C language. As a result, this combination presents developers with dual-language challenges, including difficulties in language integration, data conversion, and error handling.

As an extension, a C code converter can be added to Julia's fuzzy logic library. This approach takes advantage of Julia's strong extensibility to enable efficient cross-language interaction by automatically generating C code and integrating it into the Julia library. The process begins with the design of a C code generator that converts the high-level logic in the fuzzy logic library into corresponding C code. This generator parses Julia code, identifies essential control logic and computational components, and produces equivalent C code, streamlining the development process and ensuring compatibility across languages.

Evaluation and conclusion

We can examine a real-world scenario to better illustrate how this approach functions. Consider an industrial automation project in which engineers construct a fuzzy logic control system using Julia. This system must efficiently control machinery and process sensor data in real time. The issue is that embedded devices, which are frequently utilized in actual industrial systems, cannot be run directly by Julia. Therefore, in order to construct the most crucial control logic and computations, C must be used.

First, the engineers write the fuzzy logic system in Julia because Julia is easy to use and very good for scientific computing. But for making the system work on embedded devices, a C code generator is used. This generator automatically converts Julia code into C code, so developers do not need to write C code manually. This saves a lot of time and also avoids mistakes.

After generating the C code, it is directly used in the embedded system. Julia still plays an important role in the development stage, where engineers use it to test and design the fuzzy logic model. However, the final implementation needs to be in C, because embedded devices usually do not support Julia.

The C code handles the most important control logic and computational tasks because C is more suitable for embedded systems. To make sure everything works correctly, a data conversion module is created. This module manages the exchange of sensor data and control signals, making sure all information is processed accurately in the embedded system.

This approach makes the system more practical by dividing work logically between Julia and C. Julia is used for high-level design and testing, while C ensures that the system can actually run on embedded devices. With this approach, the fuzzy logic control system can be deployed in real-world industrial applications and work effectively in automation environments.

4 Implementation

This chapter describes the development process of the C code generator implemented in the Julia library. As this project falls under standard software development practices, established software design principles and implementation techniques are carefully combined with the design science research framework discussed in the previous chapter.

The development process is organized into three key stages: the system architecture design, the iterative coding implementation, and functional validation. Special attention is paid to maintaining compatibility with existing library functions while ensuring code readability.

Finally, the generator's practical value is tested through fuzzy logic operations. These experiments are designed to verify whether the generated C code produces accurate results while maintaining computational efficiency. By linking theoretical research methods with practical development requirements, the tool's real-world applicability is systematically evaluated.

4.1 Software design principles

When designing and implementing a c code generation to the FuzzyLogic.jl library, understanding the software design principles and applying them to this case, it important to save time and ensure quality.

General Principles of Software Design

Design patterns have become a fundamental concept in software engineering (Gamma et al., 1994). Software design follows a set of principles to ensure reliability, maintainability, and efficiency. One key principle is modularity, which breaks the system into smaller, self-contained units. It makes the system easier to develop, test, and maintain. Another essential principle is scalability, meaning the software should be designed to handle future expansions and additional features. Readability and clarity are also crucial, as clear and well-structured code improves collaboration and reduces debugging time. Additionally, reusability allows existing components to be leveraged in new contexts, reducing redundant work. Lastly, compatibility ensures that the software works well with other tools or systems. By following these principles, developers can create efficient and user-friendly software that meets the needs of its intended audience.

Applying These Principles to the C Code Generator

The C code generator follows these principles in its design. Modularity is achieved by breaking the generator into separate functions for parsing input, processing data, and generating C code. This structure allows easier debugging and future modifications. Scalability is considered by designing the system to support additional fuzzy logic features in the future. The generator also prioritizes readability and clarity, ensuring that both the generated C code and the Julia implementation are well-documented and easy to understand. Reusability is another key aspect, as the code generator is built on existing data structures from FuzzyLogic.jl, minimizing unnecessary duplication of logic (Ferranti & Boutellier 2023). Finally, compatibility is maintained by ensuring that the output C code follows FCL standards, making it usable in a wide range of applications and systems.

Defining the Target and Scope in the Early Design Phase

In the early stages of software design, it was important to define the goals and target group of users for the C code generator. The main objective was to enhance the existing FuzzyLogic.jl library by adding a feature that automatically converts Julia fuzzy logic models into C code. This feature is intended for software developers who work with embedded systems or have experience with programming and software development. These users are expected to have knowledge and experience of Julia and fuzzy logic implementations, so the generator does not need to include extensive tutorials on these topics. Additionally, it was necessary to consider how this tool would fit into existing development workflows, ensuring that it could be easily integrated with minimal modifications to existing Julia code. Another key aspect was designing the generator to be extensible, allowing future improvements or adaptations to different fuzzy logic models without significant restructuring. These considerations helped to shape the tool into a library extension that would be practical, adaptable, and effective for its intended audience (Davis, 1995).

To support users, a README file was created to explain how to install and use the tool. The documentation includes practical examples that demonstrate how to define a fuzzy system in Julia and generate the corresponding C code. These examples help users understand the expected input format and how to integrate the output code into their projects. The README also provides an overview of the tool's design philosophy, outlining its intended use cases and limitations. Furthermore, to assist developers in troubleshooting, common issues and their solutions are documented, allowing users to quickly diagnose and resolve potential problems. By including structured and clear documentation, the tool becomes easier to adopt and utilize effectively in different projects.

From the beginning, it was also essential to define what is not included in the scope. The generator does not provide a graphical user interface (GUI), as it is designed for developers who prefer working with code directly. Additionally, it does not attempt to modify or optimize the logic of the original fuzzy system; instead, it faithfully translates the system from Julia to C while maintaining structure and integrity. Moreover, it believes that users can troubleshoot mistakes using the generated output and their knowledge of Julia and fuzzy logic, rather than providing comprehensive debugging tools. The generator avoids needless complexity that might result from attempting to accommodate less technical users by concentrating on a particular target audience. As a result, the tool may continue to be lightweight and effective while meeting the fundamental requirements of developers who need a dependable way to convert fuzzy logic models into C code.

The choice to keep the tool lightweight and focused helps to ensure that it stays easy to maintain and expand in the future. Rather than creating a completely separate software application, the generator is developed as an extension of FuzzyLogic.jl, making use of its existing data structures and methods. This approach follows best practices in software engineering, where specialized tools are designed to integrate smoothly into existing workflows, avoiding the need to recreate well-established functionalities. By designing the generator as a modular component, it remains flexible and can be gradually improved over time. Contributors can add new features or optimizations as needed, without affecting the existing functionality. This modularity also makes it easier for developers to tailor the generator to their specific requirements, ensuring that it remains a highly adaptable and versatile tool across various industries and applications.

The C code generator was created as an efficient, useful, and maintainable tool for developers working with fuzzy logic in Julia and C environments by carefully evaluating the demands of the target audience and clearly specifying its scope. The generator will continue to be reliable, effective, and simple to expand thanks to the structured development strategy. Furthermore, early users' and contributors' input will be essential for improving the tool's usability, addressing edge cases, and polishing it. The generator's continued development is to establish a commonly used method for translating Julia fuzzy logic models into C, eventually closing the gap between low-level implementation and high-level modeling in embedded and performance-critical applications.

4.2 Design method and FuzzyLogic.jl

The goal of this project is to extend the existing FuzzyLogic.jl library by adding a new feature. This new feature allows automatic conversion of fuzzy logic models written in Julia into C code. This extension is necessary because in many real-world applications, especially in embedded systems and performance-critical environments, C code is a preferred language. By implementing this feature, I need to en that developers can seamlessly transition from high-level fuzzy logic models in Julia to optimized C implementations without manual translation. This approach enhances the usability of FuzzyLogic.jl and broadens its applications in various domains such as robotics, control systems, and artificial intelligence

To complete this project, several steps were taken. First, it was necessary to become familiar with Julia programming language, as it serves as the foundation of FuzzyLogic.jl. This required not just knowing the syntax but also the fundamental ideas that set Julia apart from other programming languages, namely its high-performance numerical computation capabilities and just-in-time (JIT) compilation. Furthermore, it necessitated learning to appreciate Julia's object-oriented structure and multiple dispatch mechanism, both of which are essential for handling fuzzy logic operations efficiently (Bezanson et al., 2017). We then carefully reviewed the documentation for FuzzyLogic.jl to learn more about the organization, processing, and optimization of fuzzy logic in the library. Given that fuzzy logic is a structured language, it easily fits nicely with Julia's paradigm for structured programming, which makes rule-based decision-making more effective in a variety of applications.

One of the key tasks in generating C code is extracting the necessary parameters and rules from the user's fuzzy logic system. In Julia, the fuzzy system is stored in a highly structured fis object. This object contains all the necessary elements such as input variables, output variables, membership functions, and rule definitions. To effectively generate C code, we needed to traverse this structure and retrieve relevant data systematically.

The process of extraction involves analyzing the hierarchical arrangement of fuzzy logic elements and ensuring that all required parameters are captured in a format suitable for conversion into C. This requires identifying each component within the fis structure and understanding how they interact. Input and output variables must be precisely mapped, ensuring that no critical data is lost in translation. Membership functions, which define the relationships between fuzzy sets, need to be processed with extreme accuracy to maintain logical integrity in the generated C program.

Beyond structural extraction, it is also important to validate the extracted data before it is converted. This involves performing sanity checks to ensure that each variable and rule has been correctly identified. If errors arise at this stage, they can lead to misinterpretations in the resulting C code, potentially affecting system behavior. Thus, a robust validation mechanism is crucial in this process.

Once the necessary data is extracted, the next step is formatting it into a representation that is compatible with C. Julia's expressive syntax allows us to construct structured representations that can later be transformed into precise C code snippets. By preserving the structure of the original fuzzy system, the generated C code maintains both accuracy and readability, making it easier for developers to integrate the output into their applications.

Since the C programming language is not as dynamic as Julia, we focused on extracting only the essential elements that define the fuzzy system. Unlike Julia, which supports high-level abstractions and dynamic typing, C requires explicit memory management and type definitions. This made it important to ensure that all fuzzy logic elements were mapped correctly to a structured, statically-typed format. The transformation process involved converting Julia's structured fuzzy logic into standard FCL format, ensuring that it aligns with widely used fuzzy logic representations. Additionally, special care was taken to optimize the conversion process so that the generated C code remains efficient while preserving the expressiveness of the original fuzzy logic model.

To explore the FuzzyLogic.jl library, we utilized VS Code's Julia REPL (Read-Eval-Print Loop), which allowed for real-time testing and validation of extracted data. This interactive environment was crucial in debugging and verifying the correctness of our implementation. Through this approach, we mapped the internal structure of fis and determined how each component contributes to the overall fuzzy logic model. The REPL facilitated step-by-step execution, allowing us to observe how fuzzy logic rules were processed and how membership functions were evaluated in different scenarios. This iterative process significantly improved our understanding of the inner workings of the library, helping us refine the data extraction mechanism for generating accurate and optimized C code.

The FIS structure follows a tree-based format, where inputs and outputs are defined hierarchically. Below is an example of how FIS organizes information:

```
FIS
├── Inputs
│   ├── Service
│   ├── Food
├── Outputs
│   ├── Tip
├── Rules
│   ├── IF Service == Poor OR Food == Rancid THEN Tip == Cheap
│   ├── IF Service == Good THEN Tip == Average
│   ├── IF Service == Excellent OR Food == Delicious THEN Tip == Generous
```

Figure 7. FIS structure in the FuzzyLogic.jl

Understanding the structure of FuzzyLogic.jl is critical important for achieving writing an C code generator. Because all the variables and contents were store in the Fis structure, the key idea was to extract variables, membership functions, and rules dynamically and convert them into corresponding C expressions. By leveraging Julia's structured data handling, we were able to systematically retrieve and format each component for C implementation.

Also because the fuzzy inference system (fis) in Julia is organized very well, we were able to create a clear method for getting the data. This made sure that important parts, like input variables, output variables, and membership functions, were all handled correctly. After taking out the needed data, each part was changed into a structured C format to keep the original logic and make sure it works well with low-level programming.

Additionally, we ensured that all fuzzy rules were correctly translated to maintain logical consistency between the original Julia implementation and the generated C code. This included managing rule-based evaluations, ensuring that conjunctions (AND), disjunctions (OR), and negations (NOT) were properly implemented in the translated C expressions. The rule set conversion was crucial to ensuring that decision-making in the fuzzy logic model was faithfully replicated.

The dynamic extraction process also allowed for flexibility in handling different types of fuzzy logic models, ensuring broader applicability of the C code generator. The modular approach taken in the design made it possible to extend functionality easily, enabling future compatibility with different fuzzy inference methodologies. Moreover, the ability to parse and structure the C code dynamically meant that developers could modify or extend their fuzzy logic models in Julia without needing to rewrite the corresponding C code manually.

To ensure ease of use, we implemented a logging mechanism to track the conversion process, flagging any inconsistencies or errors in the extracted parameters. This logging helped validate the integrity of the fuzzy system structure before generating the final C output. By taking these measures, we created a robust and adaptable C code generator that maintains the fidelity of the original fuzzy logic model while making it deployable in resource-constrained environments.

Since FuzzyLogic.jl already handles fuzzy logic computations, our focus was mainly on code generation rather than logic processing. Instead of redefining fuzzy logic operations, our approach was to generate syntactically correct and optimized C code that mirrors the behavior of the Julia implementation. To accomplish this, we utilized string manipulation in Julia to construct valid C functions. This method allowed us to dynamically generate membership function definitions, rule evaluation expressions, and defuzzification calculations in a structured manner.

The process begins by analyzing the structure of the fuzzy inference system (fis) and extracting relevant elements, such as input and output variables, membership functions, and rules. These components are then formatted into C syntax while preserving the hierarchical relationships inherent in the original Julia-based representation. Each membership function is translated into a corresponding C function, ensuring that the behavior of the fuzzy logic system remains consistent across both programming languages.

The generated C code preserves the structure and logic of the original fuzzy system while ensuring high performance and readability. Additionally, careful attention was given to formatting and modularization, making the generated code easy to integrate into existing C-based projects. By maintaining a clear separation of concerns—where each function and rule is processed independently before being assembled into a cohesive output file—the generator ensures that developers can modify or extend their fuzzy systems without requiring extensive manual code adjustments.

Further optimization techniques, such as minimizing redundant computations and using efficient data structures for rule evaluation, were incorporated into the code generation process. These enhancements not only improve execution speed but also reduce memory overhead, making the resulting C implementation suitable for embedded systems and real-time applications. The combination of systematic extraction, structured formatting, and computational efficiency makes this approach highly effective in bridging the gap between high-level fuzzy logic modeling and practical deployment in C environments.

4.3 C Code Generator Runtime Environment

The runtime environment of the C generator is consistent with the original FuzzyLogic.jl library, not changing the core functionalities when extended. Even though Julia is not a traditional object-oriented language, it does have objects in the form of structs and multiple dispatch (Bezanson et al., 2018). This enables us to work effectively with the data structures of the library, and guarantee that functions and parameters behave consistently when called. Julia's strong type system is used to great effect by the C code generator to identify and classify fuzzy logic rules, membership functions, and evaluation mechanisms. All the components of the fuzzy logic system are classified as types, which makes it easier to work with the data and transform it to the respective C representations. The objective was to make the users experience as much change as possible when using the library, so that they would be able to define their fuzzy systems in Julia in the same way. Therefore, it means that although the C code generator provides the additional ability to generate C code for deployment, users can still develop and manage their fuzzy logic systems within Julia without having to learn how to make complex changes.

In addition, the C code generator is seamlessly integrated with the Julia runtime, using its memory management and method dispatching. The type system of Julia allows for dynamic and structured data representation, therefore, all fuzzy logic components that are extracted are maintained in their original form prior to conversion to C. This leads to a clear and direct transition from high level Julia modeling to low level C deployment, without having to compromise on the accuracy of the model.

4.4 Multiple Dispatch for Code Generation

Multiple dispatch is one of the most powerful and useful features of the Julia language. It allows functions to behave differently based on the types of arguments passed to them (Bezanson et al., 2018). In my C code generator, we leverage this feature by implementing a generic to_c() function that dispatches to various implementations based on the type of membership function.

For example, the to_c() function is defined to handle different membership function types:

```
function to_c(mf::GaussianMF)
    """
    double GaussianMF(double x, double mean, double sigma) {
        return exp(-0.5 * pow((x - mean) / sigma, 2));
    }
    """
end

function to_c(mf::TrapezoidalMF)
    """
    double TrapezoidalMF(double x, double a, double b, double c, double d) {
        if (x <= a || x >= d) return 0.0;
        if (x >= b && x <= c) return 1.0;
        if (x > a && x < b) return (x - a) / (b - a);
        if (x > c && x < d) return (d - x) / (d - c);
        return 0.0;
    }
    """
end
```

Figure 8. Julia code Multiple dispatch function

Multiple dispatch function is a fundamental function of Julia; it makes Julia languages unique. It enables functions to be dynamically selected based on the types of all arguments, rather than just a single one. This design allows for greater code flexibility, improved performance, and more intuitive function organization. Unlike single dispatch systems in many object-oriented languages, Julia's multiple dispatch provides a more generalized and efficient approach to function specialization. This is particularly useful in scientific computing, data processing, and numerical simulations, where different input types require different computation strategies.

The ability to design specialized methods in Julia while keeping the code clear and readable is the core benefit of multiple dispatch. Julia's compiler looks at the types of inputs and automatically chooses the best version of the function to compile. This means user don't need to write a lot of "if" or "else" statements. It also makes the code run faster and easier to fix or update. Another good thing is that you can add new versions of the function later without changing the old code. This helps make user's program more flexible and easier to grow.

In the context of our C code generator, multiple dispatch plays a crucial role in ensuring that the correct C code is generated for different types of membership functions. By defining a single to_c() function that dispatches to different implementations based on the type of membership function, we maintain a modular and scalable approach to code generation. This ensures that new fuzzy logic components can be supported simply by defining additional method variations, thereby enhancing the maintainability and adaptability of the generator. The ability to dynamically select and execute the appropriate method based on argument types also contributes to significant performance optimizations, as Julia compiles specialized versions of functions tailored to the specific data types used during execution.

The multiple dispatch function of the Julia language is not limited to the choice of functionality; it also facilitates the connection between different data structures and computational models. Because a single function in Julia can support multiple input types, it significantly reduces the need for redundant code and enhances the expressiveness of program logic. This feature is especially beneficial when dealing with complex mathematical operations and algorithmic computations, where different numerical types may require different processing strategies.

Therefore, the use of multiple dispatch in our implementation improves both code clarity and efficiency. In the context of fuzzy logic systems, Julia's multiple dispatch mechanism also ensures that the system remains flexible and extensible. In practical applications, extending existing code often requires a function to handle a broader range of data types—an ability that multiple dispatch supports effectively.

4.5 Julia’s REPL and Its Role in Development

REPL refers Read-Eval-Print Loop, it is a useful feather designed in the Julia programming development environment. It enables an interactive and iterative coding environment.

Unlike traditional static development workflows, where debugging and testing require manual compilation steps, the Julia REPL allows for real-time execution and verification of code snippets. This capability significantly enhances productivity by allowing developers to test, modify, and refine their code dynamically (The Julia REPL, n.d.). In the context of this project, the REPL was an invaluable tool for exploring the internal structures of FuzzyLogic.jl, validating extracted parameters, and ensuring the correctness of generated C code before final integration.

The REPL's capacity to speed up the prototyping of membership functions and rule evaluation techniques was one of its main benefits. We were able to test the accuracy of fuzzy logic operations in real time by interactively running brief code segments. By using an iterative process, we were able to efficiently improve our extraction and transformation routines while reducing the amount of debugging overhead. Furthermore, a better comprehension of the hierarchical structure of the fis object within FuzzyLogic.jl was made possible by the REPL's support for introspection. In order to accurately translate inputs, outputs, and membership functions into appropriate C code representations, were able to examine how these items were stored.

Another important use of the REPL was debugging the parameter extraction methods. Given that fuzzy logic systems involve multiple layers of abstraction, ensuring that the extracted data maintained logical consistency was essential. Through real-time evaluations within the REPL, inconsistencies in parameter mappings were identified early, preventing potential errors in the final C code. This real-time debugging approach reduced the need for repeated compilation cycles, leading to a more streamlined and efficient development process.

Beyond debugging and testing, the REPL also proved to be a powerful educational tool. It provided immediate feedback, which was particularly useful for experimenting with different fuzzy logic configurations and understanding how variations in membership functions affected rule evaluations. This interactive feedback mechanism allowed for fine-tuning and optimizing the generated C code without requiring a full-scale implementation test in an external environment.

Overall, the integration of Julia's REPL into the development workflow significantly improved efficiency, accuracy, and flexibility. By leveraging its interactive execution capabilities, we were able to iteratively develop, test, and refine the C code generator in a structured and systematic manner. The REPL not only enhanced debugging and validation but also served as a crucial component in ensuring that the transformation from Julia-based fuzzy logic models to optimized C implementations was both accurate and reliable.

4.6 Structured C Code Output

In this C code generator project, the generated C code is actually printed out as as a string. It ensures that the entire fuzzy logic structure is accurately represented in the final output. This printed output strictly follows a structured format that aligns with the conventions used in the C programming language for implementing fuzzy logic systems. The output includes the standard representation of fuzzy logic operations in C, together with the variable names and parameters defined by the user in the FuzzyLogic.jl library. Each of these components is systematically printed in separate segments, maintaining a clear separation of logic and functionality.

The generated C code is formatted with proper indentation and line breaks, making sure that the output is easy to read, follows correct syntax, and can be understood without difficulty. Paying close attention to formatting is important because it helps reduce errors when compiling the code and makes future changes easier to manage. The structured layout of the printed output also helps with debugging since developers can clearly see how different parts of the fuzzy logic system are connected.

To make the project easier to use and easy to maintain, detailed comments have been added to the Julia code of the generator. These comments explained the structure of the

C code that is generated, helping future developers understand it better and make changes when needed. Keeping the code well-documented and organized makes sure that future updates and improvements can be done with less effort.

Considering that FuzzyLogic.jl with C code generator function together is still an open-source project under the MIT license, making extensive development documentation available is crucial for enhancing collaboration and further development (Santos & Correia, 2022). Open-source software thrives on community contributions, and to have well-documented tools significantly reduces the learning curve for new contributors. The availability of clear documentation, well-structured code, and logically formatted outputs make sure that the C code generator can be easily extended, improved, and adapted for different fuzzy logic applications in the future. This benefits Julia's community and the Fuzzy Logic community, with the broader goals of the open-source community, ensuring that the tool remains valuable for years to come.

5 Examples and Results

In this chapter, I will introduce an intelligent traffic light system that uses FuzzyLogic.jl. This system takes advantage of the newly added feature in FuzzyLogic.jl, which can automatically generate C code for fuzzy logic systems. With this function, the system can create the standard embedded C code, making it easier for developers to turn the parameters and deploy. This feature helps to bridge the gap between high-level fuzzy logic modeling and low-level embedded implementation. By automating code generation, the system reduces manual work and errors. The main goal, design, and importance of this project will be explained in this chapter.

5.1 Traffic light case

At a busy crossroad, vehicles and pedestrians are moving in unpredictable ways, often causing traffic jams and delays, especially at rush hour. A traditional traffic light system works by a fixed time cycle, and it does not change according to the real-time traffic situation. When pedestrians are crossing often, vehicles may stop unnecessarily, causing inefficiency. On the other side, if the green light for cars is too long, pedestrians must wait too much time, which might lead them to cross the road dangerously. To solve this problem, an adaptive traffic light system is needed. It should change the green light time dynamically, depending on the real-world traffic condition.

A fuzzy logic traffic control system is very good at handling complex traffic problems where the input data is not always exact. Unlike standard logic which only has "on" or "off" options, fuzzy logic can consider different degrees of truth, so it can make more flexible adjustments. Instead of giving fixed green light time, the system considers factors like vehicle queue length, pedestrian density, and how long pedestrians are waiting. Using these fuzzy inputs, it will calculate the suitable green time for a traffic light. This improves the flow of cars and avoids unnecessary stops, making traffic management better. This system simulates human thinking to manage traffic, making the intersection always behave like a traffic controller is directing it.

To implement a fuzzy traffic control system is not just defining a few rules. The real challenge is how to manage multiple factors together and their interaction. The system must be able to collect real-time data from traffic sensors, apply suitable membership functions, and then make correct fuzzy inference. For instance, if the vehicle queue is long and pedestrians are few, then the green light should stay longer for cars. However, if there are more pedestrians waiting, the system must shift the priority to them. These decisions require carefully adjusting the fuzzy parameters, making sure the system response is correct in different situations without unexpected problems. (Zadeh, 2015).

To deal with this complex control system, we used FuzzyLogic.jl, which is a powerful library in Julia that allows the user to create fuzzy inference models easily. This library provides a well-structured way to define fuzzy membership functions and rule, which help model real-world traffic scenarios better. Julia language also has strong computational performance, so it can quickly run simulation and test different model settings. By using this tool, we can try various rule combinations, modify membership functions, and visualize how the traffic light behaves before actually applying the system.

Even though Julia is very useful for modeling, it is not a common language for embedded system. Most traffic control hardware actually use C programming language, because C is faster and allow direct control of hardware. In order to make our fuzzy model work in real traffic light controller, we have to convert Julia fuzzy system to C code. Manually writing C code based on Julia model is slow and also easy to make mistake. For this reason, we use a automatic C code generator, which will take the fuzzy logic design from Julia and change it into structured C code that can be run on embedded system.

This C code generator process the high-level fuzzy logic system written in Julia and convert it into a format that works on hardware. The process includes extracting variables, defining fuzzy membership functions, and writing fuzzy rules in C programming way.

With this automation, we remove human mistake from manual coding and make sure the simulation and real implementation are consistent. The C code generated follows Fuzzy Control Language (FCL) standard, which make it simple to integrate with existing traffic control hardware. (Ferranti & Boutellier, 2023).

One of the important things in deploying fuzzy system is parameter tuning. If the membership functions and rules are not set properly, the traffic system may not work well. Bad parameters may cause the light to switch too frequently or stay too long. So, the system need to be tested and improved by using real-world traffic data. After testing many times, a well-adjusted system can respond to changing traffic flow throughout the day, making both traffic and pedestrian crossing more efficient.

After generating C code, it is compiled and then uploaded to the microcontroller which control the traffic light. The microcontroller receives real-time traffic sensor data, apply the fuzzy logic, and changes the signal timing. Since embedded system has limited computation ability, the C code must be optimized so that the system run smoothly. The fuzzy inference process must be fast enough, so the traffic light can react quickly to changing road conditions without delay.

Compared to traditional traffic light system with fixed-time cycle, a fuzzy logic-based system have many benefits. Firstly, it reduce unnecessary waiting time because it adjust the green light based on actual traffic data. Secondly, it helps to avoid traffic jam because it can dynamically control the light time depending on vehicle queue and pedestrian flow. Thirdly, it increase pedestrian safety because it gives more priority to them when needed. Lastly, the automatic C code generator makes the deployment of this kind of system easier for traffic engineer and city planner, without requiring them to write C code manually.

In the future, this system could be improved by adding machine learning to automatically adjust fuzzy rules. Also, if the system can be connected to vehicle-to-infrastructure

(V2I) network, then cars could send real-time data to the traffic lights, which will make prediction even better. More advanced sensors, such as computer vision for detecting pedestrians, can also help to make the system more smart. As traffic congestion in cities keeps growing, intelligent traffic light systems based on fuzzy logic and automation will become an important solution for improving transportation safety and efficiency. (Terano, Asai, & Sugeno, 2014)

5.2 Problem solving by using FuzzyLogic.jl

This chapter explains how fuzzy systems can be used to optimize the green time of a traffic light. Instead of using a fixed cycle, fuzzy logic helps to make the green light more flexible. It adjusts the duration based on actual traffic conditions, smoothing the road and reducing unnecessary waiting for cars and people.

In this system, there will be three inputs

- Queue, the length of the car queue in meters waiting at the intersection
- Density, the pedestrian's density, people per square meter, which is waiting to cross the road
- Waiting time, the second of the pedestrian waiting time

One Output:

- Green light time, the green light that vehicles can cross the intersection, which is generated by the fuzzy system based on the inputs.

To have a fuzzy system generate the needed result, four rules are needed in this system. These rules cover most of the conditions and allow the system to think and decide the traffic light time like a human thinking. (Ferranti & Boutellier, 2023).

Rule 1, Vehicle queue is long or pedestrian density is low → Vehicle green light is long.
When the vehicle queue is long or the pedestrian density is low, the duration of the vehicle green light should be extended to allow more vehicles to pass through the

intersection efficiently. This adjustment helps to prevent unnecessary congestion and improves overall traffic flow.

Rule 2: Vehicle queue is short and pedestrian density is high → Vehicle green light is short.

When the vehicle queue is short and the pedestrian density is high, the duration of the vehicle green light should be reduced. This ensures that pedestrians are not required to wait for excessive periods, thereby improving pedestrian accessibility and reducing the likelihood of jaywalking.

Rule 3: Vehicle queue is medium, pedestrian density is medium, and pedestrian waiting time is medium → Vehicle green light is normal.

When the vehicle queue length is moderate, the pedestrian density is at a medium level, and the pedestrian waiting time is also within a moderate range, the duration of the vehicle green light should remain at a normal level. This balance helps maintain steady traffic movement while allowing sufficient crossing opportunities for pedestrians.

Rule 4: Pedestrian waiting time is long or vehicle queue is short → Vehicle green light is short.

When the pedestrian waiting time is long or the vehicle queue is short, the duration of the vehicle green light should be minimized. This adjustment prioritizes pedestrian needs, especially in situations where extended waiting times may lead to unsafe crossing behaviors, while also ensuring that vehicles are not given unnecessary priority when traffic is light.

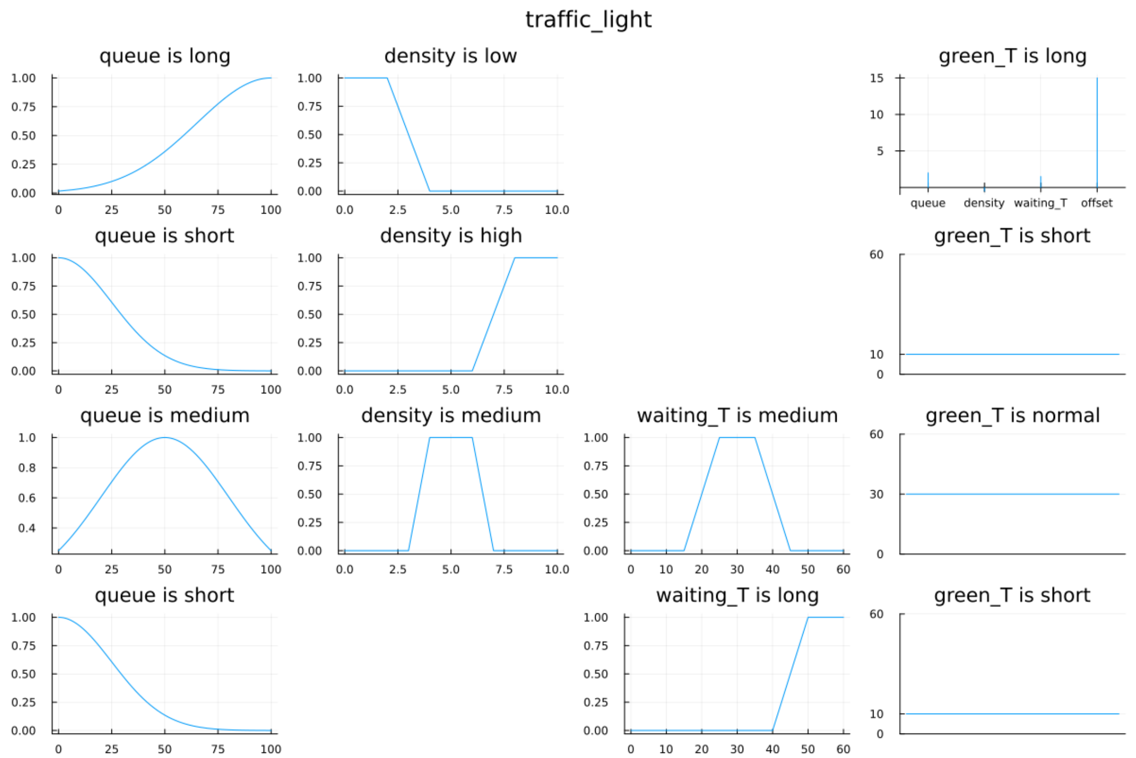


Figure 9. Traffic light fuzzy logic Sugeno system

The fuzzy inference system for traffic light control is designed to adjust the green light duration based on real-time traffic conditions. The system use three input variables: vehicle queue length, pedestrian density, and pedestrian waiting time. Each of these inputs is classified into different levels, allowing the system to determine how long the green light should last. Instead of using fixed time cycles, the system dynamically change signal timing to improve traffic efficiency and pedestrian safety.

For vehicle queue length, the system divide it into short, medium, and long categories. When the queue is short, the green light duration is reduced because there is no need to allow long passage for vehicles. If the queue is medium, the system maintain a normal green light duration to balance both vehicle and pedestrian needs. When the queue is long, the green light is extended to prevent traffic congestion and ensure smooth flow. The membership functions for queue length follow a gradual transition, making sure there is no sudden change in the traffic light behavior.

Pedestrian density also play a key role in determining the green light duration. The system recognize three different levels: low, medium and high density. If pedestrian density is high, the system give shorter green light for vehicles to allow more crossing time for pedestrians. On the other hand, when density is low, the green light for vehicles stays longer since there is no urgent need to let pedestrians cross. The system continuously monitor pedestrian flow to make real-time adjustments to signal timing.

Another important factor is pedestrian waiting time, which indicate how long people have been waiting at the crosswalk. If pedestrians have waited too long, the system shorten the green light for vehicles so pedestrians can cross sooner. When the waiting time is short, the green light duration remains normal or even longer if needed. This way, the system prevent unnecessary delays and encourage pedestrians to follow traffic signals instead of jaywalking.

The final output of the system is the green light duration, which is categorized into short, normal, or long durations. The decision is made based on a set of fuzzy rules that consider the three input variables together. The system is designed to be adaptive, meaning it can adjust to different traffic situations rather than following fixed schedules. This kind of fuzzy control allow a balance between vehicle movement and pedestrian safety, making urban traffic more efficient and responsive to real-world condition. (Ferranti & Boutellier, 2023).

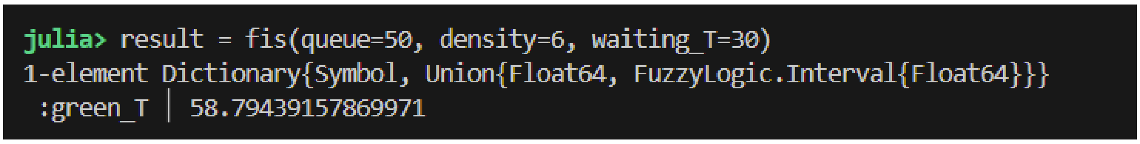


Figure 10. Triffic light case sugeno fuzzy systemic running result in FuzzyLogic.jl

In this system, if queue = 50m, density = 6 person/m^2, waiting time is 30s then the green light time as a result is 58.79s

5.3 C code generator and running result

To use a fuzzy system in a traffic light embedded system, a C code generator is needed to convert high-level fuzzy logic models into C code. Since embedded systems usually run on C for better efficiency and hardware compatibility, translating the fuzzy system from Julia to C is essential for real-world deployment

One big advantage of using the Julia fuzzy system library is that it supports both simulation and C code generation. Developers can first test the fuzzy logic in Julia and then run the same tests on the generated C code. This ensures the C version works the same way as the original model (Bezanson et al., 2017).

To develop and test the traffic light control system, I first used the original functions of the Julia library to run and debug the fuzzy system model (Ferranti & Boutellier, 2023). After verifying its performance in Julia, I converted the model into C code using the C code generator. This allowed me to compare how the fuzzy system runs in both Julia and C environments. By doing so, I ensured that the logic and results remained consistent across both languages, confirming the accuracy of the generated C code before deployment in an embedded system. The followings present a comparison of the fuzzy system's execution in Julia and C (The Julia REPL, n.d.).

```julia
# Vehicle queue length (unit: meters)
queue := begin
    domain = 0:100
    short = GaussianMF(0.0, 25.0)       # Short queue (0-50 meters)
    medium = GaussianMF(50.0, 30.0)     # Medium queue (20-80 meters)
    long = GaussianMF(100.0, 35.0)      # Long queue (60-100 meters)
end
```

Figure 11. Generate queue fuzzy membership function in Julia

```
# Pedestrian density (unit: people per square meter)
density := begin
    domain = 0:10
    low = TrapezoidalMF(-1, 0, 2, 4)   # Low density (0-4 people/m²)
    medium = TrapezoidalMF(3, 4, 6, 7) # Medium density (4-7 people/m²)
    high = TrapezoidalMF(6, 8, 10, 11) # High density (8-10 people/m²)
end
```

Figure 12. Generate density fuzzy membership function in Julia

```
# Pedestrian waiting time (unit: seconds)
waiting_T := begin
    domain = 0:60
    short = TrapezoidalMF(-5, 0, 10, 20)  # Short wait (0-20 seconds)
    medium = TrapezoidalMF(15, 25, 35, 45) # Medium wait (20-45 seconds)
    long = TrapezoidalMF(40, 50, 60, 65)   # Long wait (50-60 seconds)
end
```

Figure 13. Generate waiting time fuzzy membership function in Julia

```
# Vehicle green light duration (unit: seconds)
green_T := begin
    domain = 0:60
    short = 10
    normal = 30
    long = 2queue, -0.5density, 1.5waiting_T, 15.0
end
```

Figure 14. Figure, Generate output Vehicle green light in Julia

```
    # 4 optimized fuzzy rules
    queue == long || density == low --> green_T == long
    queue == short && density == high --> green_T == short
    queue == medium && density == medium && waiting_T == medium --> green_T == normal
    waiting_T == long || queue == short --> green_T == short
end | traffic_light
```

Figure 15. Fuzzy rules in Julia

```
julia> fis(queue=50, density=6, waiting_T=30)
1-element Dictionary{Symbol, Union{Float64, FuzzyLogic.Interval{Float64}}}
 :green_T │ 58.79439157869971
```

Figure 16. Calculate the output based on the fuzzy rules in Julia, result is 58.794

```c
double queue_short = GaussianMF(queue, 0.0, 25.0);
double queue_medium = GaussianMF(queue, 50.0, 30.0);
double queue_long = GaussianMF(queue, 100.0, 35.0);
double density_low = TrapezoidalMF(density, -1, 0, 2, 4);
double density_medium = TrapezoidalMF(density, 3, 4, 6, 7);
double density_high = TrapezoidalMF(density, 6, 8, 10, 11);
double waiting_T_short = TrapezoidalMF(waiting_T, -5, 0, 10, 20);
double waiting_T_medium = TrapezoidalMF(waiting_T, 15, 25, 35, 45);
double waiting_T_long = TrapezoidalMF(waiting_T, 40, 50, 60, 65);
```

Figure 17. Variable and its membership function in C code

```c
double rule1 = fmax(queue_long, density_low);
double rule2 = fmin(queue_short, density_high);
double rule3 = fmin(queue_medium, fmin(density_medium, waiting_T_medium));
double rule4 = fmax(waiting_T_long, queue_short);
```

Figure 18. Define and calculate rules in Fuzzy system

```c
double r1_out = 2.0 * queue + 0.5 * density + 1.5 * waiting_T + 15.0;
double r2_out = 10;
double r3_out = 30;
double r4_out = 10;
```

Figure 19. Calculate the rule out in the C code

```c
double numerator = (rule1 * r1_out) + (rule2 * r2_out) + (rule3 * r3_out) + (rule4 * r4_out);
double denominator = rule1 + rule2 + rule3 + rule4;

return numerator / denominator;
```

Figure 20. Calculate the rule out and the output of the fuzzy system.

In this case, for example is the inputs are:

- Queue = 50 meters
- Density = 6 people/m^2
- Waiting time = 30

```c
int main()
{
    printf("%f\n", traffic_light(50, 6, 30));
}   58.794392
```

Figure 21. Calculate the output based on the given input in C code

Both Fuzzy system and the generated C code calculated the same outputs based on the same inputs,, the result is the same 58.794.

In the real case, people need to adjust the parameter in the fuzzy system to find the most suitable membership function and its parameters, all this can be done in Julia environment, when all the parameter has been set ready, C code can be generate to be then put in the C embedded device to run the traffic light system then it will lots of save time.

6 Conclusion

This chapter is the conclusion of this thesis. It concludes the thesis by summarizing the key findings, assessing the trustworthiness and contributions of the work. It also presents suggestions for future development and research.

6.1 Summary

This thesis addressed the research question:
"How can fuzzy logic systems be automatically synthesized into efficient C code, with enhanced performance, and how can this synthesized code be verified and benchmarked against existing implementations and manually written C programs?"

To answer this question, a dedicated extension was developed for the FuzzyLogic.jl library, enabling the automatic generation of efficient and well-structured C code from fuzzy logic models written in Julia. This extension supports Mamdani and Sugeno fuzzy inference systems—two widely adopted frameworks in decision-making and automation control.

The tool not only translates the logic faithfully from Julia to C, but also maintains structural clarity and consistency with the original model. The system ensures that complex rule evaluations, membership function mappings, and defuzzification strategies are all accurately represented in C code format. This allows for easier deployment in low-level environments where performance, reliability, and code maintainability are crucial.

The objectives of the thesis were achieved through a well-defined and iterative design process. Specifically, the goals were:

1. To automate the translation of fuzzy logic models from Julia to C,
2. To improve the runtime efficiency and memory use of the resulting code, and
3. To verify the correctness of the output through benchmarking.

These goals were fully met. The resulting code generator can be used to create functional embedded logic in C, suitable for systems with constrained resources.

To demonstrate its functionality, the tool was tested using a traffic light control use case, where the fuzzy logic system determines light duration based on traffic and pedestrian variables. The C code generated for this system was consistent in output and performance with the original Julia implementation. Furthermore, it successfully executed in environments that demand real-time responses and deterministic behavior.

Overall, this work provides a valuable tool that enhances the usability and practicality of fuzzy logic in embedded system design. It allows engineers to work at a higher level of abstraction using Julia, while ensuring reliable deployment in C, effectively connected the gap between rapid prototyping and efficient implementation.

6.2 Trustworthiness and Contributions

The trustworthiness of this research based on the use of a structured Design Science Research methodology. Every phase—from problem identification to artifact design, implementation, and evaluation—was conducted with a focus on transparency, repeatability, and rigor. Each module of the tool was validated through iterative testing, peer comparison, and benchmark assessments against hand-written C code, confirming both its correctness and reliability.

The internal logic of the C code generator was designed to be modular and adaptable. Developers can trace each part of the output C code back to its source logic in Julia, improving trust in the translation and reducing debugging time. The tests conducted proved that the generated C code behaves identically to the Julia-based fuzzy system, across different sets of inputs and conditions. This consistency provides strong evidence of the system's functional validity.

In terms of contributions, this research enhances the FuzzyLogic.jl library by introducing an automatic C code generation feature—something not previously available in this ecosystem. This addition allows researchers, students, and engineers to prototype systems in Julia and quickly deploy them to embedded platforms using C, without manual code rewriting.

This tool also helps minimize human error during the development and transition phases, which are common when fuzzy logic models are implemented manually. It reduces the overhead and complexity of migrating from a high-level modeling environment to a resource-constrained deployment context.

The potential applications of this tool are broad. In industrial control systems, it can be used for automation tasks that require real-time decision making under uncertainty. In embedded IoT systems, the compactness and efficiency of the generated code make it ideal for devices with limited memory and processing power. For instance, a fuzzy-based smart thermostat or a self-adjusting traffic signal controller can directly benefit from this approach.

Finally, the open-source release of this tool supports community adoption and encourages future contributions. By lowering the entry barrier for integrating fuzzy logic into embedded systems, this research fosters innovation and collaboration within both academia and industry.

6.3 Future Outlook

Although the prototype tool developed in this thesis has demonstrated promising functionality, there are several areas that offer opportunities for future improvement and expansion.

One major direction for future work involves further optimization of the generated C code. While the current version emphasizes correctness and functional translation, it does not fully optimize for memory usage, instruction speed, or power consumption. Many embedded systems operate in constrained environments, where memory and processing cycles are at a premium. Techniques such as fixed-point arithmetic, lookup table simplifications, and loop unrolling could be explored to improve performance in such contexts(Merone et al., 2022).

Another important area is extending the range of supported fuzzy inference systems. At present, the generator supports Mamdani and Sugeno types, which are sufficient for many control applications. However, more advanced forms such as Type-2 fuzzy systems—known for handling higher levels of uncertainty—and adaptive neuro-fuzzy inference systems (ANFIS)—which incorporate machine learning for fuzzy rule adjustment—could significantly expand the tool's capabilities. These models could be implemented with appropriate abstractions in the Julia interface and mapped to efficient C representations.

Furthermore, the integration of the generator with hardware-specific toolchains and compilers (e.g., ARM-based microcontroller SDKs or real-time operating systems) would be a valuable enhancement. This would allow developers to deploy the generated C code more easily onto physical devices without needing additional modification or interfacing logic.

Improving the user documentation and modularity of the Julia codebase is also a practical step. Clearer function separations, standardized comments, and comprehensive examples will help other contributors better understand and extend the tool.

In conclusion, while this research provides a strong foundation, its long-term success will depend on continued development, community feedback, and practical use cases. With these efforts, the tool has the potential to become a standard solution for deploying fuzzy logic in embedded and intelligent systems.

References

AI Stratagems. (2023, March 26). Fuzzy logic statistics. https://aistratagems.com/fuzzy-logic-statistics/

Alcalá-Fdez, J., & Alonso, J. M. (2015). A survey of fuzzy systems software: Taxonomy, current research trends, and prospects. IEEE Transactions on Fuzzy Systems, 24(1), 40-56.

Alves, O. A. (n.d.). eFLL: Embedded fuzzy logic library [GitHub repository]. Retrieved [insert retrieval date], from https://github.com/alvesoaj/eFLL

Azadnia, A. H., Geransayeh, M., Onofrei, G., & Ghadimi, P. (2021). A weighted fuzzy approach for green marketing risk assessment: Empirical evidence from dairy industry. *Journal of Cleaner Production*, *327*, 129434.

Bergsten, P., Palm, R., & Driankov, D. (2002). Observers for Takagi-Sugeno fuzzy systems. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 32(1), 114-121.

Bezanson, J., Bolewski, J., & Chen, J. (2018). Fast flexible function dispatch in Julia. arXiv preprint arXiv:1808.03370.

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. SIAM Review, 59(1), 65–98. https://doi.org/10.1137/141000671

Cingolani, P., & Alcalá-Fdez, J. (2012, June). jFuzzyLogic: A robust and flexible fuzzy-logic inference system language implementation. In 2012 IEEE International Conference on Fuzzy Systems (pp. 1–8). IEEE. https://doi.org/10.1109/FUZZ-IEEE.2012.6251215

Cingolani, P., & Alcalá-Fdez, J. (2013). jFuzzyLogic: A Java library to design fuzzy logic controllers according to the standard for fuzzy control programming. International Journal of Computational Intelligence Systems, 6(sup1), 61–75.

Davis, A. (1995). 201 Principles of Software Development. McGraw-Hill.

Ferranti, L. (2024). *Introduction to fuzzy logic: Week 1 – Motivation to fuzzy logic*. Fuzzy Academy. Retrieved January 3, 2025, from https://fuzzy.academy/week1/intro/

Ferranti, L., & Boutellier, J. (2023, August). FuzzyLogic. jl: a Flexible Library for Effi-cient and Productive Fuzzy Inference. In 2023 IEEE International Conference on Fuzzy Systems (FUZZ) (pp. 1-5). IEEE.

Fuzzy Logic Tutorial. (2024) Javatpoint Fuzzy logic. Javatpoint. https://www.javatpoint.com/fuzzy-logic

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley

Germain, J. (n.d.). Programming - Commenting. Retrieved from https://users.cs.utah.edu/~germain/PPS/Topics/commenting.html

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. MIS quarterly, 75-105.

Huo, J., Zhang, J., & Chan, F. T. (2020). A fuzzy control system for assembly line balancing with a three-state degradation process in the era of Industry 4.0. International Journal of Production Research, 58(23), 7112-7129.

Izquierdo, S., & Izquierdo, L. R. (2017). Mamdani fuzzy systems for modelling and simulation: A critical assessment. Available at SSRN 2900827.

Kozlov, O., & Kondratenko, Y. (2023, June). Automatic Synthesis of Rule Bases of Fuzzy Control Systems Based on Genetic Algorithms. In Recent Developments and the New Directions of Research, Foundations, and Applications: Selected Papers of the 8th World Conference on Soft Computing, February 03–05, 2022, Baku, Azerbaijan, Vol. I (pp. 101-123). Cham: Springer Nature Switzerland.

Kuechler, B., & Vaishnavi, V. (2008). On theory development in design science research: anatomy of a research project. European Journal of Information Systems, 17(5), 489-504.

Kumar, D., & Haider, Y. (2013). Fuzzy logic based control system for washing machines. International Journal of Computer Science and Technology, 4(2), 198-200

Lin, T. D. (2009). Japan fuzzified: the development of fuzzy logic research in Japan. Annexe Thesis Digitisation Project 2017 Block 13.

Manickam, B. K., & Rathinasamy, P. (2022). A FUZZY GRA-BASED DECISION-MAKING APPROACH ON THE SELECTION OF LEAN TOOLS: A CASE STUDY OF INDIAN APPAREL INDUSTRY. *International Journal of Industrial Engineering*, *29*(1).

Maretto, L., Faccio, M., & Battini, D. (2024). Industry 4.0 technology implementation in manufacturing: a selection method and real case applications. *International Journal of Production Research*, 1-33.

Mastrocinque, E., Ramírez, F. J., Honrubia-Escribano, A., & Pham, D. T. (2022). Industry 4.0 enabling sustainable supply chain development in the renewable energy sector: A multi-criteria intelligent approach. *Technological Forecasting and Social Change*, *182*, 121813

MathWorks. (n.d.). Foundations of fuzzy logic. MathWorks Documentation. Retrieved January 6, 2025, from https://de.mathworks.com/help/fuzzy/foundations-of-fuzzy-logic.html

MathWorks. (n.d.). Types of fuzzy inference systems. Retrieved January 8, 2025, from https://www.mathworks.com/help/fuzzy/types-of-fuzzy-inference-systems.html

Mengchu, W., Chen, J., Xue, W., & Liu, X. (2024). Competing buyers' investment and ordering strategies under supply disruption. Transportation Research Part E: Logistics and Transportation Review, 191, 103723.

Merone, M., Graziosi, A., Lapadula, V., Petrosino, L., d'Angelis, O., & Vollero, L. (2022). A practical approach to the analysis and optimization of neural networks on embedded systems. Sensors, 22(20), 7807.

O'Keefe, R. (2014). Design Science, the design of systems and Operational Research: back to the future?. Journal of the Operational Research Society, 65(5), 673-684.

Onieva, E., Alonso, J., Pérez, J., Milanés, V., & De Pedro, T. (2009, August). Autonomous car fuzzy control modeled by iterative genetic algorithms. In 2009 IEEE International Conference on Fuzzy Systems (pp. 1615-1620). IEEE.

Perkel, J. M. (2019). Julia: come for the syntax, stay for the speed. Nature, 572(7767), 141-142.

Perkel, J. M. (2019). Julia: come for the syntax, stay for the speed. Nature, 572(7767), 141-142.

PyPI. (n.d.). fuzzylogic: Python fuzzy logic library (Version 1.0.0). Python Package Index. Retrieved January 9, 2025, from https://pypi.org/project/fuzzylogic/

Ross, T. J. (2010). *Fuzzy logic with engineering applications* (3rd ed.). Wiley.

Sakunthala, S., Kiranmayi, R., & Mandadi, P. N. (2017, August). A review on artificial intelligence techniques in electrical drives: Neural networks, fuzzy logic, and genetic algorithm. In 2017 International Conference on Smart Technologies for Smart Nation (SmartTechCon) (pp. 11-16). IEEE.

Santos, J., & Correia, F. (2022). Patterns for Documenting Open Source Frameworks. arXiv preprint arXiv:2203.13871.

Soto-Hidalgo, J. M., Vitiello, A., Alonso, J. M., Acampora, G., & Alcalá-Fdez, J. (2018). Design of fuzzy controllers for embedded systems with JFML. International Journal of Computational Intelligence Systems, 12(1), 204-214.

Terano, T., Asai, K., & Sugeno, M. (Eds.). (2014). Applied fuzzy systems. Academic Press.

The Julia REPL. (n.d.). In The Julia Language Documentation. Retrieved from https://docs.julialang.org/en/v1/stdlib/REPL/

Wang, L. X. (1996). A course in fuzzy systems and control. Prentice-Hall, Inc..

Yen, J. (1999). Fuzzy logic-a modern perspective. IEEE transactions on knowledge and data engineering, 11(1), 153-165.

Zadeh, L. A. (2008). Is there a need for fuzzy logic?. Information sciences, 178(13), 2751-2779

Zadeh, L. A. (2015). Fuzzy logic—a personal perspective. Fuzzy sets and systems, 281, 4-20.

Zojirushi. (n.d.). NS-ZCC Neuro Fuzzy® Rice Cooker and Warmer. Retrieved January 8, 2025, from https://shop.zojirushi.com/products/nszcc