Thesis

Miskolci Egyetem

Parser and Interpreter Development for the Fuzzy Behavior Description Language in MATLAB/Octave

Author:

Balázs Nagy

Computer Science Bsc.

Advisor:

Dr. Tamás Tompa

Miskolc, 2024

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Szakdolgozó Neve (N3P7UN) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: kulcsszavak, hasonlók

A szakdolgozat címe: A dolgozat címe

A feladat részletezése:

Ide kell a feladatkiírásban szereplő szöveget betenni.

(Kisebb tagolás lehet benne, hogy jól nézzen ki.)

Témavezető: Témavezető neve (beosztása)

A feladat kiadásának ideje:

. . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Szakdolgozó Neve**; Neptun-kód: `N3P7UN` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Szakdolgozat Címe* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év .... .hó .... .nap

....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat .... szövegoldalt

.... program protokollt (listát, felhasználói leírást)

.... elektronikus adathordozót (részletezve)

....

.... egyéb mellékletet (részletezve)

....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: ....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: ....

a bíráló javaslata: ....

a szakdolgozat végleges eredménye: ....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Contents

Chapter 1

Introduction

Presented in this work is an interpreter library/package for the Fuzzy Behavior Description Language (FBDL) implemented in the Octave programming language.

This initial description of the problem at hand can either be informative or utterly confusing for anyone reading it for the first time, simply because it entails many concepts perhaps still unknown to the reader. But it is quintessential to state the main topic being tackled as to not lose sight of it amidst the following discussions that will eventually lead up to the task itself. Anyone with a moderate to advanced knowledge in the field of programming can easily comprehend the workings of an interpreter if described properly, however simply showcasing that without any prior introduction to the underlying principles would still leave a fairly large gap in the reader's mind concerning the motivation behind such a language and also its use of a myriad of fuzzy logic related ideas. Therefore it is necessary to treat the subject as a whole and describe not only the technical implementations and results, but the theory as well, on which all of it is built.

With this in mind the work is split into two main sections along with this preceding foreword to allow for some clarifications and provide a greater description of the whole subject matter. The first half exposes the reader to core concepts related to fuzzy logic and slowly builds them up into its more complex and intricate applications. Also in this part the idea of behavior control and fuzzy state machines are presented along with various mathematical models to help with formalization; it concludes with the specifications of the aforementioned Fuzzy Behavior Description Language. The second half of the work delves into the implementation and inner workings of the interpreter. In the beginning, decisions regarding language specific implementation and other architectural considerations are discussed; possible alternatives are slightly touched upon. Following a general overview of the process of interpreting a language each stage and their operations are shown separately in detail along with possible corner cases that may require special attention and samples of unit tests to check the integrity and correct operation of the program. Finally the reader is provided with working examples of source code written in FBDL and also a demonstration of said code where the output of the interpreter can be verified.

The main difficulty lies in connecting the different parts of the overarching subject, so as to allow the reader to indulge in this work without getting lost consider the following short explanation as a guide to the various topics about to be presented.

The motivation behind creating a programming language, be it any kind, is always attributed to the existence of a specific problem it is trying to solve. This could range from low-level hardware management, such as those found in embedded systems, all the way to server-side applications and numerical analysis.

The language (FBDL) appearing in this work has been constructed to serve a particular application of fuzzy logic, namely that of behavior control. For example when a system, due to some event, reacts or behaves in a certain way based on predefined rules dictating its appropriate response. A fundamental property of fuzzy logic, essentially the fact that it is continuous, make it a prime candidate for such a use, since natural systems are hard, sometimes nigh impossible to accurately model with Boolean-logic.

Describing how such a system would operate, the rules it would follow and the actions it would take can be tricky to model with ordinary programming languages. Therefore an easier alternative was designed with the primary aim of facilitating the ease of use, particularly even if the user happens to lack any kind of previous formal experience in the field of programming. To further simplify the task of using this language it takes another useful property of fuzzy logic that arises from its continuity: we are able to describe the state of a fuzzy variable or in other words the degree to which it satisfies a certain statement with the help of natural language in contrast to using concrete numerical values.

There exits many applications for fuzzy logic and its extensions, some requiring complex calculations and methods that are quite conveniently present in scientific programming languages such as MATLAB and Octave. For this reason an interpreter library in these languages would provide great utility for programs already using fuzzy logic and open new opportunities for those seeking to venture into such areas.

These ideas constitute the majority of the work, so they shall be further examined at length in the following sections.

Chapter 2

Core Concepts and Previous Research

2.1 Fuzzy Logic

In order to gain a sound understanding of the idea of *fuzziness* we must first familiarize ourselves with the notion of fuzzy sets. The concept was first introduced and described by mathematician Lotfi A. Zadeh in 1965 as an extension to classical sets. The key difference between ordinary sets and fuzzy ones is simple: In the case of the former all elements are either a part of a set or not, where as in the world of fuzzy sets an element may belong to multiple sets. The measure of how much an element is part of a given set is referred to as its *degree of membership* and is calculated with the aid of the *membership function.*

2.1.1 Fuzzy Set

As opposed to classical sets, every element in a fuyz set has an additional property beside its value that being the degree to which that given element is the set. These aspects are more formally defined in the following section.

2.1. definíció. Let U, referred to as the *universe of discourse*, be a set containing all the elements we wish to describe and define $m : U \to [0, 1]$ as a membership function. The pair (U, m) forms a fuzzy set A in which $\forall x \in U$ the value given by $m(x)$ is called the degree of membership of x. The function $m(x)$ is equivalent to $\mu_a(x)$.

Taking the example from Claudio Moraga's *Introduction to fuzzy logic* (2005) [5]: given the interval $[0, 10]$ of the real line as our universe of discourse and the statement "x is between 3 and 5", we may represent it with the function $\mu_{3-5} : [0, 10] \to [0, 1]$. Where $\mu_{3-5}(x) = 1$ if $3 \leq x \geq 5$, and $\mu_{3-5}(x) = 0$ otherwise as seen on fig1.a. This function describes the classical set $[3, 5]$. Consider now the statement "x is near 4". The proximity, or nearness to the number 4 can be represented as $4 - \epsilon$, given the assumption that ϵ is a sufficiently small positive real number. Values obtained by the continued subtraction of ϵ will have a decreasing *"degree of nearness"* to 4 until the value, and subsequently those smaller then itself, is no longer considered to be "near" the number 4. Repeating this experiment with $4 + \epsilon$ and the continued addition of ϵ will yield symmetric results. If we take the function $\mu_{near4} : [0, 10] \to [0, 1]$ to represent this statement just as previously, it becomes apparent that it cannot be of the same kind as μ_{3-5} (that lead to a classical set). If we assume that 3 and 5 are acceptable limit points for "near 4" and marking these as , $\alpha_{min} = 3, \alpha_{max} = 5, beta = 4$, then

$$\mu_{near4}(x) = \begin{cases} 0, & x \leq \alpha_{min} \text{ or } x \geq \alpha_{max} \\ 1, & x = \beta \\ \frac{x-\alpha_{min}}{\beta-\alpha_{min}}, & \alpha_{min} < x < \beta \\ \frac{\alpha_{max}-x}{\alpha_{max}-\beta}, & \beta < x < \alpha_{max}. \end{cases}$$

The function will be continuous and increasing for $3 < x < 4 and$ will be continuous and decreasing for$4 < x < 5$. Without further information, linear transitions will be chosen as shown in fig1.b. μ_{near4} represents a **fuzzy set**.

[figures of the previous example]

(a) Classical set

(b) Fuzzy set

Figure 2.1: Difference in steepness during the transition from 0 to 1.

Other than assigning values linearly to elements not fully contained inside a fuzzy set any kind of membership function can be utilized, but by far the most common is the previously mentioned linear way, which produces a trapeziod shape during visualization (Triangles arise, when the upper side of the trapezoid is a point). As mentioned by [2], in terms of terminology the following expressions are defined regarding any given fuzzy set:

Core: Elements where the membership function is 1:

$$\text{core}(A) = \{x \in U | \mu_\alpha(x) \geq \alpha\}$$

Support: Elements where the membership function is greater than 0:

$$\text{support}(A) = \{x \in U | \mu_\alpha(x) > 0\}$$

Boundary: Elements where the membership function is between 0 and 1:

$$\text{boundary}(A) = \{x \in U | 0 < \mu_\alpha(x) < 1\}$$

Height: The height of the fuzzy set A is the maximum value taken on by the membership function:

$$\text{height}(A) = \{x \in U | \max(\mu_\alpha(x))\}$$

2.1.2 Various Membership Functions

An example of the different shapes that a membership function may take include the following cases and their respective definitions appearing in [2]:

Triangular (Same as in the above example, a special case of the trapezoid):

$$\mu_A(x) = \begin{cases} 0, & x \leq \alpha_{min} \text{ or } x \geq \alpha_{max} \\ 1, & x = \beta \\ \frac{x-\alpha_{min}}{\beta-\alpha_{min}}, & \alpha_{min} < x < \beta \\ \frac{\alpha_{max}-x}{\alpha_{max}-\beta}, & \beta < x < \alpha_{max}. \end{cases}$$

Trapezoidal (With the upper side (core) taking values from $[\beta_1, \beta_2]$):

$$\mu_A(x) = \begin{cases} 0, & x \leq \alpha_{min} \text{ or } x \geq \alpha_{max} \\ \frac{x-\alpha_{min}}{\beta_1-\alpha_{min}}, & \alpha_{min} < x < \beta_1 \\ \frac{\alpha_{max}-x}{\alpha_{max}-\beta_2}, & \beta_2 < x < \alpha_{max}. \end{cases}$$

Γ-**membership** function:

$$\mu_A(x) = \begin{cases} 0, & x < \alpha \\ 1 - e^{\gamma(x-a)^2}, & \alpha_{min} < x < \beta_1 \end{cases}$$

S-membership function:

$$\mu_A(x) = \begin{cases} 0, & x \leq \alpha_{min} \text{ or } x \geq \alpha_{max} \\ 2\left(\frac{x-\alpha_{min}}{\alpha_{max}-\alpha_{min}}\right) pow2, & \alpha_{min} < x < \beta \\ 1 - 2\left(\frac{x-\alpha_{min}}{\alpha_{max}-\alpha_{min}}\right) pow2, & \beta < x < \alpha_{max}. \end{cases}$$

Logistic function:

$$\mu_A(x) = \frac{1}{1 + e^{-\gamma x}}$$

Exponential-like function:

$$\mu_A(x) = \frac{1}{1 + \gamma(x - \beta)^2}$$

where $\gamma > 1$.

Gaussian function:

$$\mu_A(x) = e^{-\alpha(x-\beta)^2}$$

Besides these any function that fits the intended purpose of characterizing a certain fuzzy set is acceptable and it is left up to the discretion of experts in the given field to decide which one is most appropriate.

[figures some previous functions]

2.1.3 Dealing with Multiple Fuzzy Sets

There are some cases, where precise numerical measurements might not be required or even be detrimental, for example stating someone's age as being 17 years 32 days and 8 hours old, does not necessarily demand such accuracy. It is much more sensible to describe that person simply as young. This notion of the use of words in our statements instead of numerical values was introduced by Zadeh in 1975 and is called a **linguistic variable**. The varying values taken by such a variable can be described by **linguistic**

terms, such as *low, middle, high, very small, average, large*, meaning we are able to take advantage of natural language, thus making it easier to work with.

For a simple example consider one's age as a variable and the two sets: young and old. A person who is 5 years of age is considered very young and not at all old, similarly someone in their twenties may be called young, but slightly old as well, however a middle aged individual of 43 years is neither very young nor very old, but rather an even mix of both. Representing the two fuzzy sets, young and old, on fig3 we can see they overlap. Any value at this given interval of overlay is a linear combination of, in this particular case, both of these fuzzy sets. The number of sets can, of course, be extended and then the values at these intersection would be a linear combination of all the defined sets, given that they overlap. From giving proper definitions of how to operate on these linguistic variables arises the notion of **fuzzy logic** and subsequently it serves as the basis for many advanced concepts such as: inference, fuzzy decision making and fuzzy control.

[Overlapping fuzzy sets with point x demonstrating linear combination of values]

2.2 Applications of Fuzzy Logic

Many fields make use of fuzzy logic and all the differing unique characteristics from ordinary sets that it has to offer. Fuzzy logic is best suited for problems that may be hard to define or model precisely with Boolean-logic. A collection of some notable examples of already tried and implemented solutions explored by [6] are discussed in this section.

2.2.1 Healthcare

Due to the intrinsic non-linearity of biomedical systems it is difficult to accurately model various processes. Regulation of blood pressure in the case of medical patients has been tested with the help of a real time drug delivery system that used an integrated fuzzy controller. Separately, it has also been shown that test reports yield estimates of likelihood rather than confirmation of presence or absence of a disease, hence these empirical estimates can be treated as the output of a membership function and used as such in fuzzy inference modelling.

2.2.2 Chemical Science

A fuzzy control system was used to both apply current to a series of anodes protecting an underground pipeline and to minimize the system's power consumption. For comparison the system used 126 *fuzzy rules* (further discussed in the following chapter) and an empirically adjusted membership function to optimize the model. Another study conducted in relation to pH measurement in waste water adopted fuzzy logic to calculate errors and acceptable levels of pH in the data.

2.2.3 Optimization Problems (Operations Research)

Pappis and Mamdani (1977) applied fuzzy logic to control the flow of traffic at an intersection of two one-way streets and minimize traffic obstruction. Teodorovic and Kalie (1996) experimented with fuzzy logic based decisions for choosing the mode of transportation in order to minimize both the cost of travelling and the travel time. .Jarkko and Esko (2003) had applied fuzzy logic to minimize the waiting time and risk of collisions during the operation of traffic signals .

2.2.4 Behaviour Control

Perhaps the most intriguing application found in the field of fuzzy logic is the modelling of certain behaviors of systems. Of course the topic being the bulk of this work, its details and methods of operation will we furhter elaborated, but here we examine the interesting possibilites fuzzy logic and fuzzy control can offer us in the form of behavior control. The main difficulty of simulation or prediction of evolution regarding such systems comes from its dependence on a large number of variables and combinations of possible outcomes making it extremely difficult to model with great accuracy. Fuzzy logic allows, in a sense, to approximate these processes and provide a reasonably close solution to the problem. It closely resembles the natural ways of decision making as well, given the fact that there are no sharp boundaries needing to be crossed while considering a decision as it follows, by definition, a continous range of values.

2.3 Fuzzy Rules

Following [2], the behavior of a system can be represented by a simple model if we consider only its relevant aspects. Such a construct makes use of a set of rules in the "if - then" form. Fuzzy rules are categorized by two major types, Mamdani fuzzy rules and Takagi-Sueno fuzzy rules. In the general form of a fuzzy rule a list of antecednets is followed by a number of consquents such that:

> if $< antecedent_1 >$ and ... and $< antecedent_n >$, then $< consequent_1 >$ and ... and $< consequent_n >$

where the *antecedent* is of the form v_1 is S_1 and the consequent z_1 is W_1 respectively. v_i is an input variable belonging to the input fuzzy set S_i and z_j is an output variable of the output fuzzy set W_j. In the case of Takagi-Sueno fuzzy rules the consequents are replaced with functions of the input variables so that $z_j = f_p(v_1, \ldots, v_i)$, where f_p is any real function.

2.3.1 Fuzzy Inference

In order to use fuzzy logic for any sort of application we must first consider how to integrate it with existing Boolean-logic. More precisely, we are interested in a solution that operates on linguistic variables and an outcome that relies solely on fuzzy rules along with linguistic terms. Since the input variables to any given system are usually not fuzzy ones, they must be converted to satisfy this requirement in order to then later be used within the fuzzy application. This first step is called **fuzzification** and as the name implies we make sure to supply our further calculations with variables of the correct form. By taking the desired element $x \in U$ from our universe of discourse and some fuzzy set A we convert x to a membership function value, given by $\mu_A(x)$. Repeating this procedure for every element we wish to utilize yields a degree of membership for each one, therefore translating all discrete inputs to fuzzy ones.

Now that we have a number of fuzzy variables to work with the next step is to apply predefined rules of the form described in the previous section. This step is referred to as **inference** and mathematically it is a mapping of the antecedents (input variables) to the consequents (output variables), resulting in an output fuzzy set. A degree of membership of any variable in this resultant set is depended upon the degree of membership of values in the input set or sets that have been defined by the given rule. That is to say, let A and B be fuzzy set of the antecedent and C of the consequent respectively and X, Y, Z linguistic terms. Then, according to the rule

$$\text{if } A \text{ is } X \text{ and } B \text{ is } Y \text{ then } C \text{ is } Z$$

the inference process will calculate the output fuzzy set C based on the known values of $\mu_A(X)$ and $\mu_B(Y)$.

In a similar manner to the calculation of the membership functions there are a multitude of methods which are applicable in determining the output, these are functions that do the actual mapping between the two sets of antecedent and consequent. This process of **fuzzy rule interpolation** (FRI) entails a great number of ways in which different aspects and characteristics of the membership functions are considered and thereafter the calculations made, each having their advantages and difficulties.

[Figures and further description of the FRI process] [Greater detail about the calculation of the area under the combined membership function shape, determining the core's position (eg.: cog)] [Different methods of preforming FRI [1]]

[Defuzzification and graphical representation of the aggregated value given by the area of the interpolated membership functions]

2.4 Fuzzy Automaton and Behavior Control

Behavior is, in the most simple sense, a series of states, where a transition between two states occurs in response to some event. The closest and most accurate mathematical model to this notion are state machines, which follow an almost identical definition.

Integrating fuzzy logic into a state machine will result in a Fuzzy Finite-state Automaton (FFA); this work uses the same model as in [3]; where the definition is given in the following manner:

$$F = (S, X\delta, P, O, Y, \sigma, \omega),$$

where

S is a finite set of fuzzy states, $S = \{\mu_{s1}, \mu_{s2}, \ldots, \mu_{sn}\}$.
X is a finite dimensional input vector, $X = \{x_1, x_2, \ldots, x_m\}$.
$P \in S$ is the fuzzy start state of F.
O is a finite dimensional observation vector, $O = \{o_1, o_2, \ldots, o_p\}$.
Y is a finite dimensional output vector, $Y = \{y_1, y_2, \ldots, y_l\}$.
$\delta : S \times X \to S$ is the fuzzy state-transition function which is used to map the current fuzzy state to the next fuzzy state based on the input value.
$\sigma : O \to X$ is the input function which is used to map the observation to the input value.
$\omega : S \times X \times Y$ is the output function which is used to map the fuzzy state and input to the output value.

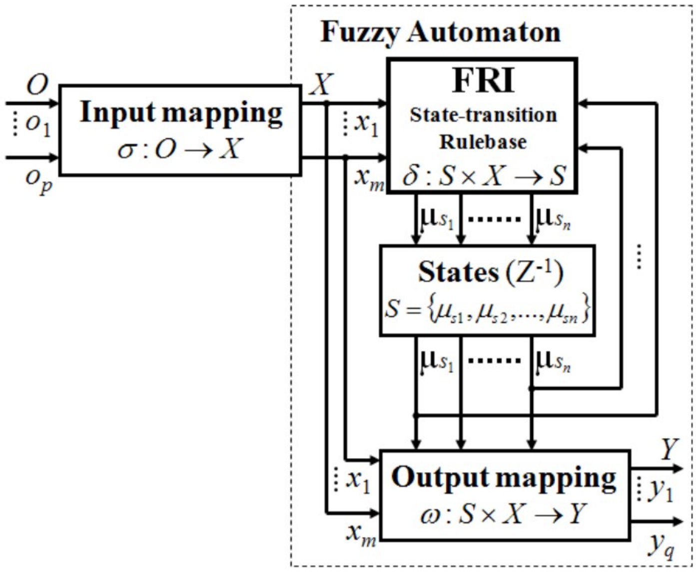


Figure 2.2: Fuzzy Automaton

2.5 Fuzzy Behavior Description Language

After having discussed in detail the needed prerequisites for greater understanding we can finally turn our attention to the main topic of this work, namely the language with which the aforementioned fuzzy behavior can be described.

2.5.1 Motivation

The language aims to provide an environment that enables the creation of programs utilizing fuzzy logic while requiring minimal to almost no prior knowledge in the field of programming. The target applications mainly entail those that delve into behavior control, as its name implies, and in order to facilitate the efficient development of such programs a higher level of abstraction is required leaving the specific implementations with regards to hardware constraints and fuzzy calculations to their respective layers of operation.

2.5.2 Language Specifications

An easy to use language encompasses not just logical abstraction, but is also simple in terms of syntax. An SQL-like syntax for verbosity and the lack of special characters or any complicated notation schemes makes it appeal to a wider audience than it would otherwise. The specifications for the grammar is provided below using the extended Backus-Naur form.

⟨*behavior*⟩ ::= universe+ rulebase+ [init]

⟨*universe*⟩ ::= '`universe`' string ['`description`' string] symbol+ '`end`'

⟨*symbol*⟩ ::= string number number

⟨*rulebase*⟩ ::= '`rulebase`' string ['`description`' string] rules '`end`'

⟨*rules*⟩ ::= rule+

⟨*rule*⟩ ::= '`rule`' ['`description`' string] ['`use`'] string ['`when`' predicates] '`end`'

⟨*predicates*⟩ ::= predicate ('`and`' predicate)*

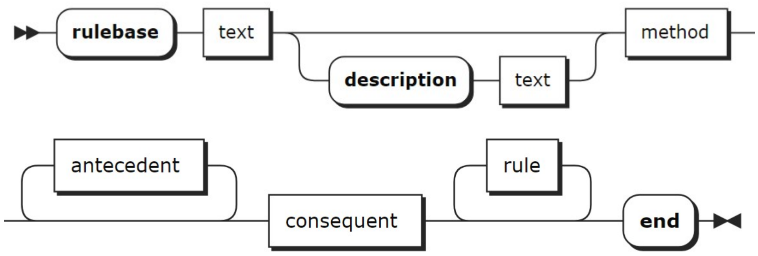
⟨*predicate*⟩ ::= string '`is`' string

⟨*init*⟩ ::= '`init`' ['`description`' string] (string (string | number))+ '`end`'

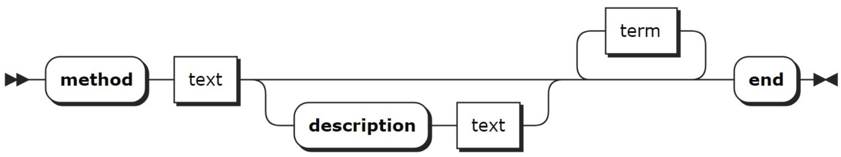
For a more concise representation a railroad diagram of each element is also included [3].

[Fix rule and term block]

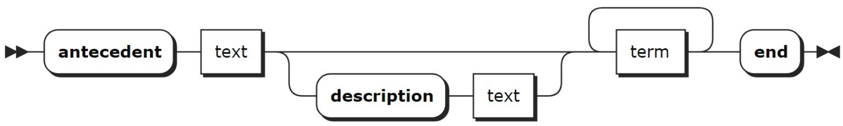
The interpreter discussed in this work is based on a JavaScript implementation from the same paper [3].



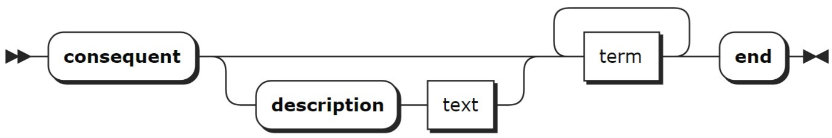
(a) Rulebase definition



(b) Method definition



(c) Antecedent definition



(d) Consequent definition

Figure 2.3: Syntax diagram of language elements

Chapter 3

Design

3.1 Requirements and Technical Considerations

Due to past research around fuzzy logic and its application in behavior control the need for having an adequate language along with a library or interpreter to make calculations easier also arose. For avoiding ambiguity, in the following discussions the words library, program and interpreter are used interchangeably and should be treated as referring to the same concept, namely the FBDL interpreter. The entire operation of the program is based on the preceding works and research in fuzzy behavior control and behavior-based systems [4].

Many of the applied examples were produced in the MATLAB language, therefore facilitating the need for developing a framework capable of operation in that environment. The similarity between the MATLAB and Octave programming languages presented a great opportunity; implementing the interpreter in such a way as to conform to both languages would make it more accessible and allow for wider usage. Therefore the decision was made to produce a program that uses in its operation only minimal parts that are found in both languages, thus making it compatible and interoperable.

To further elaborate, the usage of language specific components and features should be kept at a minimum or avoided altogether if possible, for example the definition of classes or application of several built in functions, function definitions, unit tests etc. are such areas where the two target languages tend to differ quite majorly. It also does not help that at each iteration and new versions these differences grow ever larger since new features are added and perhaps old ones modified or removed, hence implementation with only basic and fundamental features of each language should be prioritized to the utmost extent.

3.2 Structure of The Interpreter

The difference in function definitions, the most noticeable one being that in MATLAB scripts cannot contain any local function definitions before version R2016b, and also other incompatibilities related to the syntax of this operation are the main reason that the program is organized in a way so that every function definition occupies its own *.m* file.

[Add figure of program hierarchy, tree diagram]

Since the interpreter is not a standalone program, rather a library, its entry point is a function that gets called from the user of this library. From there it goes through each stage of interpreting the input; it is comprised of 3 major parts and a lot of smaller, auxiliary functions that help in completing the task, along with making the program code more readable and modular.

[Add figure of abstract operation, eg.: interpreter parts and data flow]

The program's entry point is a callable function that takes either a string or a file path as input and returns the solution vectors resulting from the fuzzy calculations. In both cases the content supplied must be a valid piece code written in FBDL.

[Extend with return value and engine/state machine]

```
function retval = main(input, type)
  retval = 0;
  addpath("lexer");
  addpath("lexer/utils");
  addpath("parser");
  addpath("engine");

  content = "";
  if strcmp(type, "f")
    content = fileread(input);
  elseif strcmp(type, "s")
    content = input;
  else
    % TODO: Print usage
  end

  behavior = parser(content);
  % Fuzzy state machine
  %stateMachine = createStateMachine(behavior);
  %step(stateMachine);
end
```

Functions that facilitate the usage of the program must first be "included" with path definitions treating the above entry point as root, since they are stored in their separate files. Supplying the function with either type of valid input will initiate its operation, anything other than that will result in an error and the user will be provided a message on basic usage.

The function *parser*() being called first might be a bit counter intuitive, but the lexer and parser operate simultaneously, not in a procedural manner, where the output of the lexer is the input to the parser. Of course doing it that way is also possible, returning a vector of tokens and then parsing that, however it is not just less efficient, but at the same time takes away the ability to report errors with correct positional messages, since that information is carried by the lexer and not the tokens.

The sequence of operations is perhaps best demonstrated via a simple model:

[Add pseudo code to describe functioning]

The parser continuously requests tokens from the lexer and at the same time it builds the syntax tree for the entire program, after which this completed structure is passed to the engine for calculations.

3.3 Data Structures

Regarding the data structures used in the program, the most compatible with both languages were found to be the *struct* and *cell*, hence all complex objects are stored in such a manner. For clarity, the *cell* data type is not used as frequently, but in some parts it is more appropriate than other solutions. The most important of these objects is the *lexer*, but others include *token* and the *syntax tree* along with other minor internal data structures that store and manage the information read during or after lexical analysis.

3.3.1 Lexer

This structure holds the input and several key information about the position of the cursor currently analyzing the text. Due to organizing every function into its own file the lexer can only be either a global structure that gets modified by any given function or it is created locally in the top most function in terms of calling hierarchy and is passed around by every other function that needs either access to the input stream or the metadata stored in it. A separate function for creating a lexer is provided in foresight to unit testing, so as to avoid having to create it in every single test case.

```
function lexer = createLexer(content)
  keywords = {
    "universe", "rulebase", "end", "description", "rule",...
    "when", "and", "is", "init",  "use"
  };

  lexer = struct(
    "content", content,
    "content_len", length(content),
    "cursor", 1,
    "line", 1,
    "beginning_of_line", 1,
    "token_begins", 1
  );

  lexer.keywords = keywords;
end
```

The input stream, or as *content* in the lexer structure, is processed by one character at a time and the *cursor* represents how many characters we have read so far, in other words our current position withing the supplied text. Lines are incremented with each encounter of a
n (newline) character. The staring position of the given token is also stored to allow for copying the value and report error messages, indicating the position of the incorrect token.

3.3.2 Token

Defined as having the fields: type and value, where the former can be any of the valid token types described by the grammar rules of the FBDL and the latter takes on the actual value read from the input stream. Below are the types accepted and the possible values a token may take:

- keyword: *universe, rulebase, init, end, description, rule, when, and, is, use*
- string: *Any character sequence between two " (double quote) symbols*
- number: *Any number, be it decimal or whole with . (dot) for separator in case of the former*
- identifier: **RESERVED** *Any ASCII alphanumeric sequence starting with either a letter or _ (underscore)*
- terminal: **RESERVED** *Special symbols such as (), [], {} , ::*

*Please note that tokens designated as **reserved** are either already in the program code, but not actively used or there are room for them should the need arise for future extension.*

Tokens are "produced" or emitted by the *emitToken* function that constructs a token structure with the correct type and value.

```
function token = emitToken (type, value)
  token = struct(
  "type", type,
  "value", value);
end
```

3.3.3 Syntax Tree

During the process of parsing every language element must be stored for later processing.

Chapter 4

Implementation

4.1 Lexical Analyzer

The first stage in the program is lexical analysis, also called tokenization, a process which dissects and "categorizes" the input based on some predefined rule and extracts a stream of tokens, or in this case the *getNextToken* function returns them one by one at each call. Everything else that is not needed or does not carry meaningful information is discarded such as white spaces, tabs, newline character, or any type of control character and only the allowed tokens are processed.

Upon receiving a call the function must first *trim* the input, simply skip the characters which are of no use to us such as all the control characters and those outside the alphanumeric range. Note that whenever we iterate over the input extensive checks are necessary in order to mitigate any indexing related issues.

```
function lexer = trim (lexer)
  while lexer.cursor <= lexer.content_len &&
                ((lexer.content(lexer.cursor) < 33) ||
                (lexer.content(lexer.cursor) == 127))
        if lexer.content(lexer.cursor) == "\n"
        lexer.line++;
        lexer.beginning_of_line = lexer.cursor;
      end
      lexer.cursor++;
   end
end
```

Then we must determine if we have run into a comment line, marked by a # *(hashtag)* symbol or reached the end of the file. In the case of the former we treat it in a similar manner to trimming, but only going until a
n (newline character is reached, whilst regarding the latter, a token is emitted with type *EOF* and no value .

Now we can finally start examining whether the stream of characters read from the input are part of an accepted token. Since we read individual characters from the input it is not possible to tell if it is going to be a valid token until we have read the whole word, however just from the first character we can categorize it as a possible token and then hand the procedure for checking to the appropriate function. In a case where an invalid character sequence is encountered the program raises a syntax error with the appropriate message and also displays the line and column numbers where the fault occurred, then exits.

4.1.1 Identifier

RESERVED: An identifier may start with an underscore or any letter from the English alphabet found in the ASCII character set; case sensitivity is not considered. Characters apart from the first one can include numbers as well. Since no functionality for handling identifiers are currently implemented in the current version of the interpreter, and neither in the grammar of the FBDL, only keywords are permitted in the supplied FBDL source code. However given the similarity of lexical analysis in both cases some room has been left for possible accommodation of this feature at a later time.

After a token is read as an identifier its value is compared against the list of available keywords, if a match is found the token type is changed to keyword and returned.

```
...
while lexer.cursor <= lexer.content_len &&
                isIdent(lexer.content(lexer.cursor))
  lexer.cursor++;
end
token.value = substr(lexer.content, lexer.token_begins,
                lexer.cursor - lexer.token_begins);
if any(strcmp(lexer.keywords, token.value))
  token.type = "keyword";
end
...
```

The list of keywords may be extended or have entries removed, granted the modification is permitted by the grammar of the language. This is the case with the *dominates* keyword as hierarchical rule dominance is not implemented in the program, but is found in the grammar as an optional language element.

4.1.2 Terminal

RESERVED: Similarly to identifiers it is not yet available in the FBDL grammar, but some consideration has been taken to allow for future extensions that include these elements.

4.1.3 String

Encountering a " (double quote) symbol implies that a string will follow and accordingly every character is skipped until the closing pair is not reached. The lexer holds every token's starting position, in this particular case that happens to coincide with the position of the double quote at the beginning, which will later be used as an index to copy the contents of the string and store it in a token. This procedure is used in case of every token that has a value.

The length of a given string is unknown when iterating over it, the only way to see if it is invalid, or in other words the closing double quotes are missing, is to see if we have reached the end-of-file beyond which there cannot be any more characters. Empty strings are permitted and no value copying happens in such a case.

Escape characters are not yet allowed in strings and there are no implementations to process them, they are simple copied as literal characters just as the rest of the string.

With EBNF :

$\langle string \rangle$ ::= " $\langle character \rangle$* "

4.1.4 Number

The first thing to consider when checking for numerical constants is the presence of the optional negative sign, since it is not itself a numerical value, most often being denoted with a - *(dash)*. The absence of such a sign implicitly implies a positive number, therefore the need for a + *(plus)* sign is eliminated and is not processed. Then a series of numbers, digits, must follow until the end of the token; the very first digit cannot be zero. Every number may contain a single negative sign before the first digit and a single decimal point between two adjacent digits marked by a . *(dot)* character. Failing to meet any of these conditions will result in a syntactical error being raised and the program exiting.

After passing these checks the series of digits is copied from the input text and is converted to a double precision floating point type, which is then returned in the token. This step allows direct referencing of the token's value during calculations in the engine without needing to do the conversion there.

With EBNF :

$\langle number \rangle$ ::= -? ($\langle integer \rangle$ | $\langle fraction \rangle$)

$\langle integer \rangle$ ::= $\langle digit\text{-}z \rangle$+ $\langle digit \rangle$*

$\langle fraction \rangle$ ::= ($\langle integer \rangle$ | 0) . $\langle digit \rangle$*

$\langle digit \rangle$ ::= 0 | $\langle digit\text{-}z \rangle$

$\langle digit\text{-}z \rangle$ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4.2 Parser

Every language conforms to some form of grammatical rule base from which it cannot deviate, otherwise it would not be valid. [Overview of each grammar rule] [Specific implementation to parse the given grammar with recursive descent parsing technique] [Building the syntax tree and considerations for the data structures being used] [Additional semantical checks after parsing] [Room for possible future extensions: warnings for missing elements, parsing optional arguments, rule domination hierarchy, extension of the existing grammar?, other parsing techniques required in case of certain grammar modification, eg.: precedence climbing for evaluation of mathematical expressions instead of using constants]

4.3 Engine

[Calculation of fuzzy rule interpolation and inference for resultant values] [Operations based on the mathematical fuzzy automaton] [Behavior control and further reusing output values]

4.4 Error Handling

[Built in Octave error handling, but a generic one should be used] [To avoid code entanglement and logically difficult to understand fault checking] [Reporting and error messages for helping find the apparent problem within the input code]

4.5 Future Extensions

With time programming languages usually evolve, and the FBDL is no exception, therefore it is quite sensible to employ an architecture that is capable of adaptation to changes in code and also leaves room for extensions. Various elements in the language such as strings, numbers, terminals and keywords are susceptible to change. Regarding the first in the list, strings might contain escape characters and as such the *lexer* must store a list of characters that are accepted as valid escape sequences.

Chapter 5

Tesztelés

A fejezetben be kell mutatni, hogy az elkészült alkalmazás hogyan használható. (Az, hogy hogyan kell, hogy működjön, és hogy hogy lett elkészítve, az előző fejezetekben már megtörtént.)

Jellemzően az alábbi dolgok kerülhetnek ide.

- Tesztfuttatások. Le lehet írni a futási időket, memória és tárigényt.
- Felhasználói kézikönyv jellegű leírás. Kifejezetten a végfelhasználó szempontjából lehet azt bemutatni, hogy mit hogy lehet majd használni.
- Kutatás kapcsán ide főként táblázatok, görbék és egyéb részletes összesítések kerülhetnek.

Chapter 6

Összefoglalás

Hasonló szerepe van, mint a bevezetésnek. Itt már múltidőben lehet beszélni. A szerző saját meglátása szerint kell összegezni és értékelni a dolgozat fontosabb eredményeit. Meg lehet benne említeni, hogy mi az ami jobban, mi az ami kevésbé jobban sikerült a tervezettnél. El lehet benne mondani, hogy milyen további tervek, fejlesztési lehetőségek vannak még a témával kapcsolatban.

Bibliography

[1] Szilveszter Kovács Alzubi M., Johanyák Zsolt Csaba. Fuzzy rule interpolation methods and fri toolbox. *Journal of Theoretical and Applied Information Technology*, 96(21):–, 2018.

[2] Sabri N. et al. Fuzzy inference system: Short review and design. *International Review of Automatic Control*, 6(4):441–449, 2013.

[3] Dávid Vincze Imre Piller, Szilveszter Kovács. Declarative language for behaviour description. *Acta Polytechnica Hungarica*, 316:103–112, 2015.

[4] Szilveszter Kovacs Imre Piller. Fuzzy behavior description language: A declarative language for interpolative behavior modeling. *Acta Polytechnica Hungarica*, 16:47–72, 2019.

[5] Claudio Moraga. Introduction to fuzzy logic. *Facta Universitatis. Series: Electronics and Energetics*, (18):319–328, 2005.

[6] Makkar R. Application of fuzzy logic: A literature review. *International Journal of Statistics and Applied Mathematics*, 3(1):357–359, 2018.

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).