Spis treści

Wstęp

Pomysł na logikę rozmytą zrodził się w 1960 r na Uniwersytecie Kalifornijskim w Berkley. Została ona zaproponowana przez Dr Lotfi Zadecha, który pracował nad problemem rozumienia naturalnego języka przez komputer. Naturalny język, podobnie jak praktycznie wszystko w otaczającym nas świecie ciężko jest przetłumaczyć na komputerowe 0/1. Samo pytanie czy wszystko da się wytłumaczyć binarnie jest czysto filozoficzne, choć zazwyczaj dane które mamy do przetworzenia dostajemy już w tej postaci z innego, elektronicznego źródła.

Logika rozmyta do pojęć prawdy i fałszu dodaje również całą gamę wartości pośrednich. Takie podejście znane było już wcześniej w logice trójwarościowiej Łukasiewicza (L3) [1], podobne próby podejmował również Clarence Lewis. Trzecim stanem w logice Łukasiewicza był stan nieznany (0.5). Lotfi ten pomysł rozwinął na całą przestrzeń wartości <0,1>.

W logice rozmytej wartości 0 i 1 stanowią ekstremalne przypadki prawdy (albo stanu ważności, faktu), może ona jednak przybierać dowolne wartości ze zbioru <0,1> co daje możliwość stwierdzenia, że coś jest jednocześnie np. w 0.7 wysokie i w 0.3 ciepłe. Jest to bardzo bliskie naszemu naturalnemu procesowi myślowemu, gdzie nic nigdy nie jest do końca czarne ani białe. Logika rozmyta nie wyklucza swoim istnieniem logiki binarnej, jest jedynie jej uogólnieniem. Prawa, które nią rządzą dają się zaaplikować do logiki Boole'a dlatego, że w zadbano o zachowanie praw Augusta De Morgana [2].

Logika rozmyta rozpoczyna się od wykreowania reguł spisanych w naturalnym języku. System rozmyty konwertuje te reguły do ich matematycznego odpowiednika na podstawie odpowiedniego modelu rozmytego. Upraszcza to w dużej mierze pracę projektanta takiego systemu, ponieważ reprezentacja rozwiązania problemu jest bardzo czytelna dla człowieka. Również rezultaty prac takiego systemu są celniejsze niż te z systemów opartych na logice dwuwartościowej, ponieważ system pracuje w rzeczywistym świecie, gdzie rozmycie wartości wejściowych i wyjściowych jest czymś naturalnym. Kolejnym plusem logiki rozmytej jest jej prostota i elastyczność. Może ona rozwiązywać problemy na podstawie niekompletnych, niedokładnych danych. Może też modelować nieliniowe funkcje o dużej złożoności. Przydaje się to szczególnie w przypadkach, gdy mamy do czynienia ze środowiskiem, które może się zmieniać, ponieważ przeprojektowanie modelu funkcji rozmytych nie wymaga wiedzy na temat złożoności algorytmicznej problemu,
tylko wiedzy eksperckiej - reguły wyrażamy w czytelnym naturalnym języku.

W praktyce można niewielkim nakładem prac napisać system, który będzie obsługiwał każdy przypadek danych wejściowych. Teoria zbiorów rozmytych cieszy się dość dużą popularnością w dziedzinach związanych ze sterowaniem oraz przetwarzaniem języka. Produkowane są mikroprocesory których rozkazy projektowane są pod przetwarzanie rozmyte. Powstało do tego czasu kilka sposobów reprezentacji wiedzy systemu rozmytego. Jednym z nich jest FCL.

Fuzzy Control Language (FCL) jest językiem który został ustandaryzowany przez Międzynarodową Komisję Elektrotechniczną (IEC) w dokumencie IEC 61131-7. Język ten nie zawiera żadnych funkcjonalności nie związanych z logiką rozmytą, co sprawia, że nie da się w nim napisać najprostszego programu który da jakiekolwiek rezultaty (np "Hello World"). Może on być jednak częścią większego programu. Biblioteka napisana przeze mnie w języku Java ma na celu interpretację pliku skryptowego FCL i jego reprezentację w postaci struktury powiązanych obiektów. Do biblioteki dołączyłem aplikację webową, która jest środowiskiem IDE do języka FCL. Można ją wykorzystać do sterowania inną aplikacją, która np. modeluje problem. Aplikacja pozwala na ukazanie wewnętrznych mechanizmów zaprojektowanego systemu rozmytego

I. Podstawy teorii modelowania systemów rozmytych

1. Reguły sterownika rozmytego

Reguły którymi opisuje się sterownik rozmyty tworzone są następująco :

```
if <przesłanka> then <konkluzja>
```

Przesłankę tworzy się za pomocą serii predykatów połączonych spójnikami logicznymi, Konkluzję tworzą kolejne predykaty. Struktura ta przypomina swą budową bazy wiedzy znane z systemów ekspertowych, jednak wewnątrz każdej reguły "dzieje się" tu dużo więcej.

W zależności od zastosowanego modelu rozmytego możemy spotkać się z różnymi formami predykatów. Aplikacja, która jest przedmiotem tej pracy, bazuje na modelu Mamdaniego [3]. W tym modelu predykaty są postaci :

```
A is B
```

gdzie :
A - nazwa danej zmiennej lingwistycznej
B - jeden z termów opisujących tą zmienną.

Teoria zbiorów rozmytych zakłada, że termy określające zmienną lingwistyczną mają postać zbiorów rozmytych. Zmienna jest więc zbiorem nazwanych zbiorów rozmytych, które mają nazwy wzięte z języka naturalnego. Predykaty w tej postaci może wyglądać tak :

```
temperatura is niska
```

w miejsce niska możemy wstawić też np. wysoka, średnia. Definiujemy w ten sposób kolejne termy ze zbioru termów zmiennej temperatura. Termom tym przypisane są pewne zbiory rozmyte które opisują małą, średnią i wysoką temperaturę.

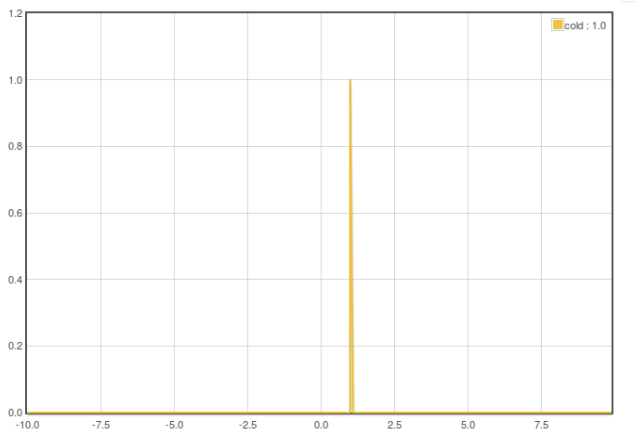
Zmienne lingwistyczne które występują po lewej stronie reguły nazywamy zmiennymi wejściowymi sterownika. Natomiast zmienne znajdujące się w konkluzji to zmienne w-wyjściowe. Do wnioskowania rozmytego jako podstawę użyto uogólnioną na teorię zbiorów rozmytych regułę *modus ponens*. Zachowano element, który określał, że prawdziwość przesłanki implikacji pozwala wnioskować o prawdziwości konkluzji. Dodano pojęcie stopnia prawdziwości albo stopnia spełnienia zarówno przesłanki jak i konkluzji.

2. Bloki i proces wnioskowania rozmytego

2.1. Blok rozmywania

Bazując cały czas na modelu Mamdaniego wyróżniamy kolejne etapy(bloki) wnioskowania rozmytego. Pierwszym z nich jest blok rozmywania. Blok ten przekształca wartość zmiennej wejściowej (zmiennej lingwistycznej) na stopień spełnienia predykatów - przesłanek reguł, które angażują tę zmienną.

Najczęściej spotykaną i najprostszą obliczeniowo metodą, która realizuje te cele jest metoda typu singleton. Polega ona na utworzeniu funkcji - zbioru rozmytego X następującej postaci :



rys I.2.1/1 Singleton

Który zdefiniowany jest funkcją :

$$f(x) = \begin{cases} 1 : x = C \\ 0 : x \neq C \end{cases}$$

Następnie jako wartość spełenia predykatu przesłanki A is B uznaje się zbiór powstały w wyniku przecięcia zbioru X ze ze zbiorem skojarzonym z termem B. Jest to prosta metoda w wyniku której zbiorem wynikowym predykatu przesłanki jest liczba mówiąca o stopniu aktywacji tego predykatu. Istnieje wiele innych, bardziej skomplikowanych metod rozmywania [4].

2.2. Blok i metody inferencji

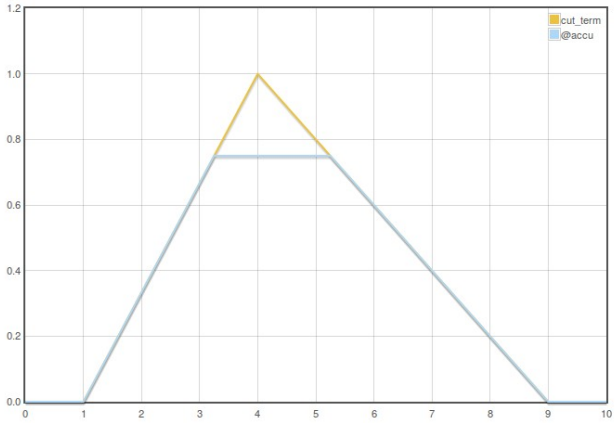
Inferencja jest kolejnym etapem działania sterownika rozmytego. Jest ona jednocześnie najbardziej złożona obliczeniowo. Możemy wyróżnić trzy etapy prac tego bloku :

- agregacja (aggregation)
- wnioskowanie
- kumulacja (accumulation)

Wszystkie te etapy wykonywane są w oparciu o przygotowaną przez eksperta baze reguł. W poprzednim bloku dane zmiennych wejściowych zamienione zostały na stopnie spełnienia odpowiednich predykatów w przesłankach reguł. W tym bloku następuje uruchomienie wszystkich reguł, których przesłanki są spełnione, wyliczenie zbioru rozmytego który jest wynikiem tych reguł oraz kumulacja wyników w każdym bloku reguł.
Na etapie agregacji stopień spełnienia każdej z reguł obliczany jest na podstawie stopnia spełnienia ich przesłanek. W tym celu używane są logiczne operatory rozmyte znane z logiki Boole'a : AND, OR oraz NOT. W zastosowanej przeze mnie implementacji języka Fuzzy Control Language dostępne są następujące funkcje (t-normy) dla tych spójników :

	AND	OR
MIN	$\mu_{A,B}(x,y) = min(\mu_A(x),\mu_B(y))$	$\mu_{A,B}(x,y) = max(\mu_A(x),\mu_B(y))$
PROD	$\mu_{A,B}(x,y) = \mu_A(x)\cdot\mu_B(y)$	$\mu_{A,B}(x,y) = \mu_A(x)+\mu_B(y)-\mu_A(x)\cdot\mu_B(y)$

Te reguły dla których stopień spełnienia zagregowanych przesłanek jest niezerowy zostają aktywowane. W wyniku wnioskowania obliczany jest zbiór rozmyty stanowiący konkluzję danej reguły. Operatorem implikacji jest operator MIN, a polega to obrazowo na "obcięciu" zbioru termu wyjściowego na wysokości stopnia spełnienia zagregowanych przesłanek, na przykład :



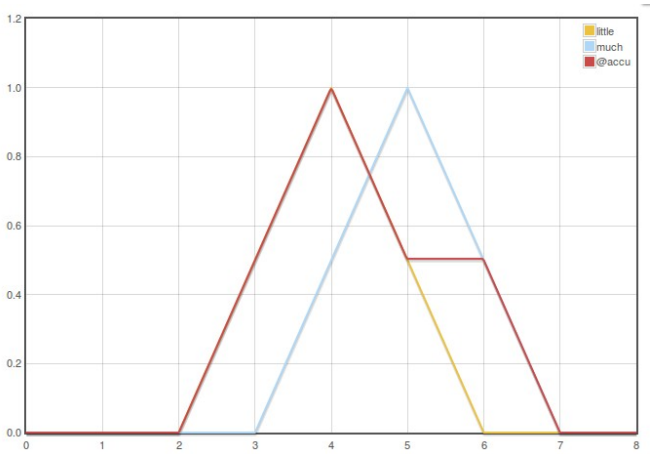
rys I.2.2/1 Przykład obciętego termu

Dodatkowo w konkluzji każdej reguły możemy podać kilka predykatów dotyczących różnych lub nawet tej samej zmiennej, rozdzielając je operatorem AND. Użycie operatora o tej nazwie podyktowane było zachowaniem czytelności reguł, nie ma on żadnego znaczenia pod względem matematycznym.

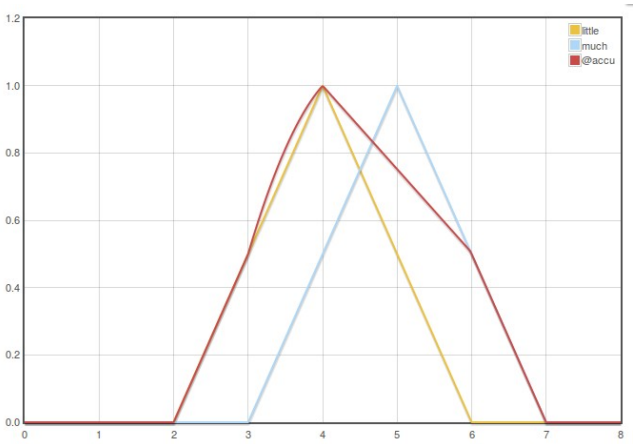
Wynikowe zbiory rozmyte kumulują się w ramach danego bloku funkcji sterownika w zbiór rozmyty, który poddawany jest następnie procesowi wyostrzania. Kumulacja polega na zebraniu wszystkich wynikowych zbiorów rozmytych danej zmiennej w jeden zbiór za pomocą funkcji kumulacji. Funkcje kumulacji dostępne w implementowanym języku FCL :

	ACCU
MAX	$\mu_{A,B}(x,y) = max(\mu_A(x), \mu_B(y))$
PROD	$\mu_{A,B}(x,y) = \mu_A(x) + \mu_B(y) - \mu_A(x) \cdot \mu_B(y)$

Obie metody nieco się różnią kształtami zbiorów wynikowych np.:



rys I.2.2/1 Akumulacja metodą MAX



rys I.2.2/2 Akumulacja metodą PROD

Różnice te nieznacznie wpływają na rezultat działania całego systemu.

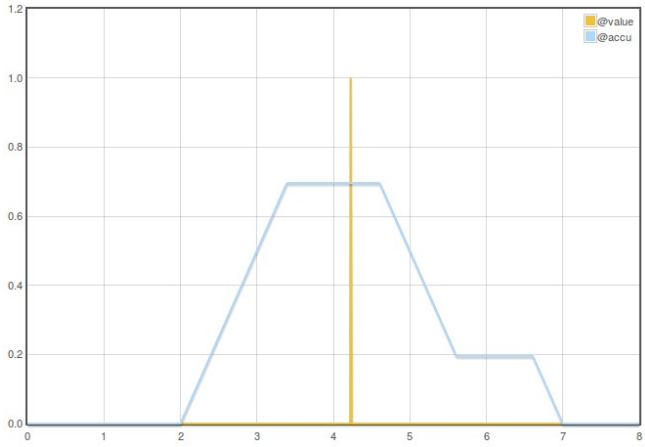
2.3. Blok wyostrzania.

Wyostrzanie ma na celu przekształcenie wynikowego zbioru rozmytego na określoną wartość rzeczywistą stanowiącą wartość wyjścia modelu.

W tworzonym systemie możliwymi operatorami są :

- Środek ciężkości (CoG -center of gravity).
- Środek ciężkości dla singletonów (CoS - center of singleton)

	WZÓR
COG	$v = \frac{\int_{v_{min}}^{v_{max}} v\mu_v(v)\,dv}{\int_{v_{min}}^{v_{max}} \mu_v(v)\,dv}$
COS	$v = \frac{\sum_{t \in S} \mu_t(t)\,t}{\sum_{t \in S} \mu_t(t)}$, *gdzie* S *– zbiór zagregowanych singletonów*



rys I.2.3/1 Wyostrzanie metodą COG

3. Przykład wnioskowania rozmytego – zraszanie trawnika

3.1 Zbieranie danych

Załóżmy, że mamy trawnik o sporej powierzchni, który codziennie w miarę potrzeb podlewa sztab ogrodników. W celu zmniejszenia kosztów zatrudnienia chcemy utworzyć system, który będzie automatycznie zraszał nasz trawnik i będzie to robił w zależności od aktualnych warunków pogodowych – wilgotności gruntu i temperatury. W tym celu rozmieszczamy czujniki temperatury, wilgotności i nasłonecznienia w strategicznych punktach trawnika, tworzymy prosty system filtrujący błędne dane (powstałe w przypadku awarii jednego z czujników) i stajemy przed problemem implementacji właściwej aplikacji sterującej zraszaniem. Przy projektowaniu takiej aplikacji, w której zastosujemy logikę rozmytą, możemy skorzystać z pomocy jednego z ogrodników, który do tej pory pracował w naszym gospodarstwie. W tym celu prosimy go o spisanie zasad, którymi kierował się podlewając trawnik. Otrzymujemy od niego mniej więcej taki dokument:

Jeżeli temperatura jest bardzo niska to nie podlewamy trawnika
Jeżeli wilgotność jest duża i temperatura jest niska to nie podlewamy trawnika
Jeżeli wilgotność jest duża i temperatura jest średnia to nie podlewamy trawnika
Jeżeli wilgotność jest duża i temperatura jest wysoka to ustawiamy słabe zraszanie
Jeżeli wilgotność jest średnia i temperatura jest niska to nie podlewamy trawnika
Jeżeli wilgotność jest średnia i temperatura jest średnia to ustawiamy słabe zraszanie
Jeżeli wilgotność jest średnia i temperatura jest wysoka to ustawiamy średnie zraszanie
Jeżeli wilgotność jest niska i temperatura jest niska to ustawiamy słabe zraszanie
Jeżeli wilgotność jest niska i temperatura jest średnia to ustawiamy średnie zraszanie
Jeżeli wilgotność jest niska i temperatura jest wysoka to ustawiamy silne zraszanie

Z racji, że pewne określenia zawarte w tym dokumencie nie do końca są jasne zbieramy kolejne informacje:
temperatura bardzo niska jest poniżej 5°C
temperatura niska to około 10°C
temperatura średnia to około 18°C
temperatura wysoka to 25 °C i więcej
dowiadujemy się też, że około to +- 8°C

wilgotność niska to poniżej 10%
wilgotność średnia to około 15%
wilgotność duża to to powyżej 20%
około to +- 4%

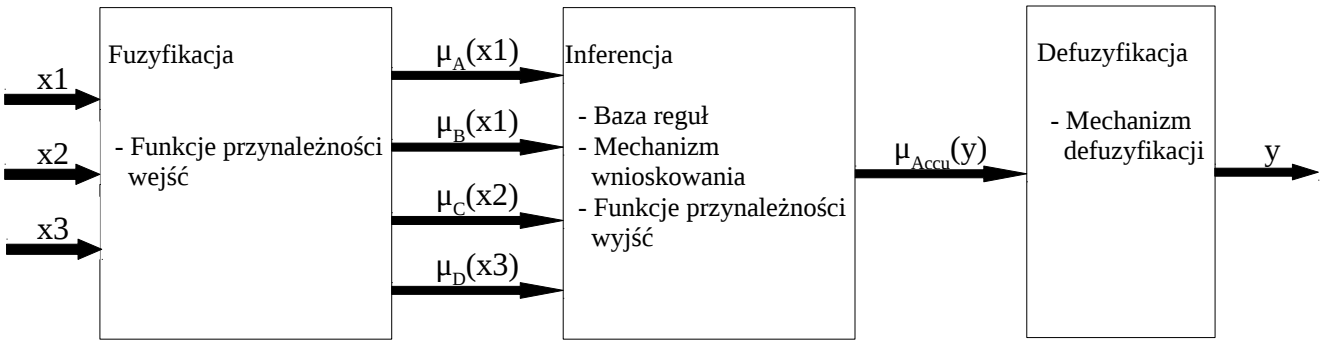
zraszanie słabe to 20%
zraszanie średnie to 40%
zraszanie silne to 60%

Większego zraszania nigdy nie ustawiali, ponieważ podmywało glebę. Z rozmowy wynika również, że sprawdzali warunki co pół godziny i dostrajali cały system, można jednak dostrajać go w trybie ciągłym, cały czas próbkując dane w małych odstępach czasu – wpłynie to pozytywnie na zużycie wody.

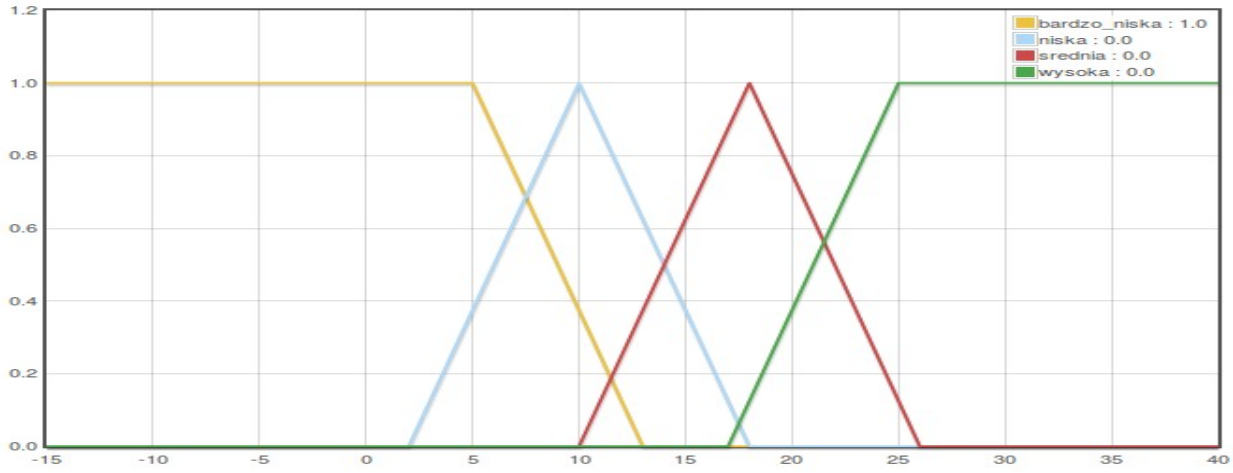
Z tak zebranymi danymi możemy przystąpić do zaprojektowania naszego systemu rozmytego.

3.2 Modelowanie rozmyte

Do implementacji naszej aplikacji zraszającej trawnik użyjemy modelu Mamdaniego. W ogólnym zarysie model ten prezentuje się następująco:



Z naszych danych tworzymy funkcje przynależności zmiennych wejściowych:

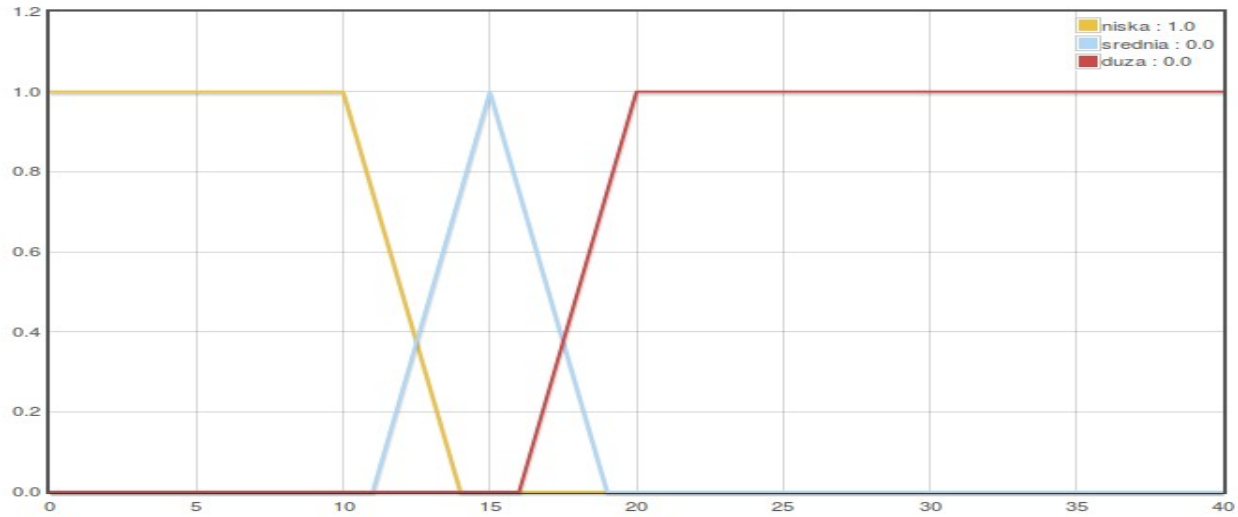


rys I.3.2/1 Zraszanie trawnika, funkcje przynależności termów zmiennej temperatura

Przy tworzeniu zbiorów korzystamy z punktów które przedstawił nam ogrodnik opisując własne podejście do temperatury tworząc z nich kolejne termy:

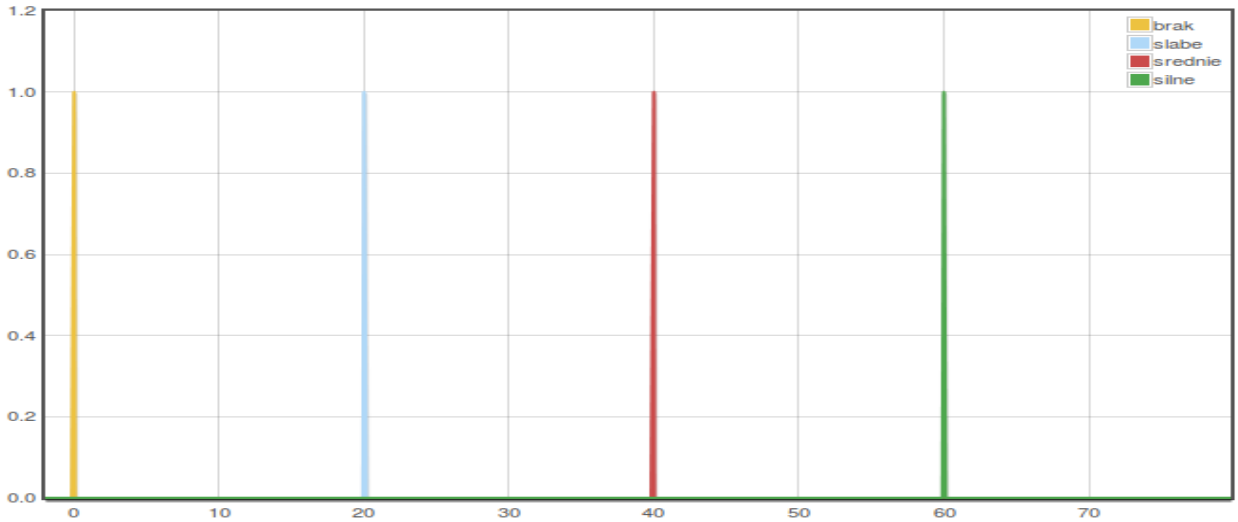
- TERM bardzo_niska := (-15,1) (5,1) (13,0);
- TERM niska := (2, 0) (10, 1) (18, 0);
- TERM srednia := (10, 0) (18, 1) (26,0);
- TERM wysoka := (17,0) (25, 1) (40, 1);

Kolejną zmienną wejściową jest wilgotność gruntu dla uproszczenia nazwana dalej wilgotnością:



rys I.3.2/2 Zraszanie trawnika, funkcje przynależności termów zmiennej wilgotność

Pozostaje jeszcze zmienna wyjściowa zraszanie, którą opiszemy specjalnym rodzajem termów – singletonami :



rys I.3.2/3 Zraszanie trawnika, funkcje przynależności termów zmiennej wyjściowej zraszanie

Do tego potrzebujemy jeszcze zestawu reguł, które opisał nam ogrodnik. Dla ułatwienia zapiszę je w notacji używanej w FCL:

- `RULE 1: if temperatura is bardzo_niska then zraszanie is brak;`
- `RULE 2: if wilgotnosc is duza and temperatura is niska then zraszanie is brak;`
- `RULE 3: if wilgotnosc is duza and temperatura is srednia then zraszanie is brak;`
- `RULE 4: if wilgotnosc is duza and temperatura is wysoka then zraszanie is slabe;`
- `RULE 5: if wilgotnosc is srednia and temperatura is niska then zraszanie is brak;`
- `RULE 6: if wilgotnosc is srednia and temperatura is srednia then zraszanie is slabe;`
- `RULE 7: if wilgotnosc is srednia and temperatura is wysoka then zraszanie is srednie;`
- `RULE 8: if wilgotnosc is niska and temperatura is niska then zraszanie is slabe;`
- `RULE 9: if wilgotnosc is niska and temperatura is srednia then zraszanie is srednie;`
- `RULE 10: if wilgotnosc is niska and temperatura is wysoka then zraszanie is silne;`

Tak zdefiniowane reguły i funkcje przynależności termów zmiennych wejściowych i wyjściowych opisują model Mamdaniego. Przeprowadzimy teraz przykładowy proces wnioskowania od początku do końca.

3.3 Rozmywanie (fuzyfikacja)

Załóżmy stan zmiennych wejściowych :

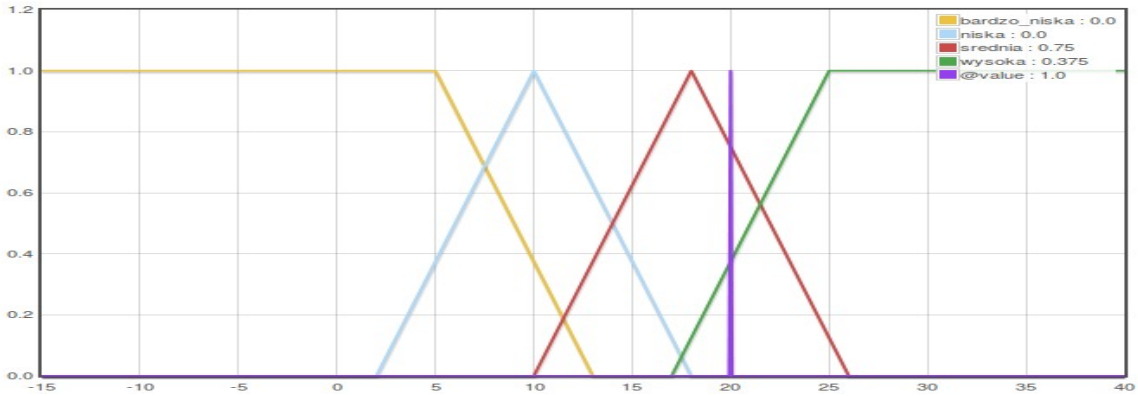
- temperatura : 20°C
- wilgotność : 16%

Obliczamy wartości funkcji przynależności szczególnych predykatów reguł. Otrzymujemy:

$$\mu_{TEMPERATURA\ IS\ SREDNIA}(20) = 0{,}75$$

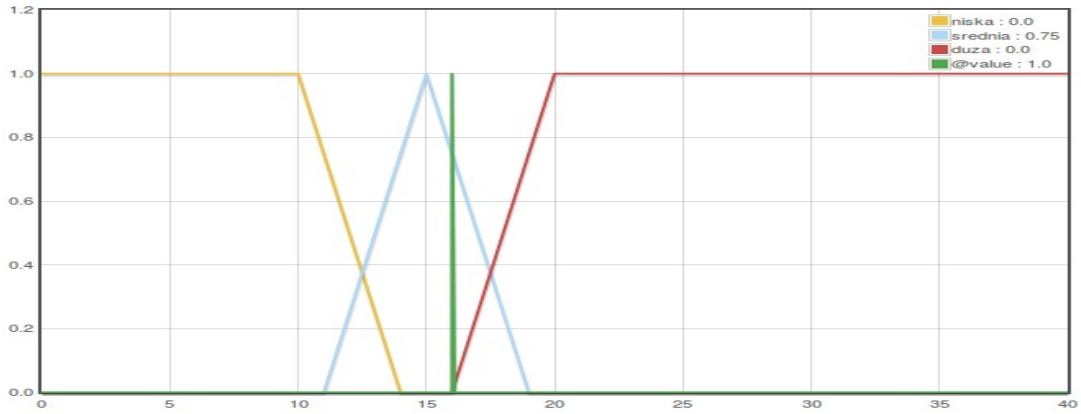
$$\mu_{TEMPERATURA\ IS\ WYSOKA}(20) = 0{,}375$$

Dla pozostałych predykatów występujących w regułach wartość funkcji przynależności wynosi 0. Widać to na poniższym wykresie – przecięcie wykresów termów z fioletowym singletonem @value:



rys I.3.3/1 Funkcje aktywacji termów zmiennej wejściowej temperatura

Fuzyfikujemy również zmienną wilgotność :



rys I.3.3/2 Funkcje aktywacji termów zmiennej wejściowej wilgotność

$$\mu_{WILGOTNOSC\ IS\ SREDNIA}(16) = 0{,}75$$

Dla pozostałych termów wartość funkcji przynależności wynosi 0.

3.4 Wnioskowanie (inferencja)

Wnioskowanie rozpoczyna się od wyszukania tych reguł, których predykaty przesłanek mają niezerowe wartości funkcji aktywacji:

- RULE 1: if temperatura is bardzo_niska then zraszanie is brak;
- RULE 2: if wilgotnosc is duza and temperatura is niska then zraszanie is brak;
- RULE 3: if wilgotnosc is duza and **temperatura is srednia** then zraszanie is brak;
- RULE 4: if wilgotnosc is duza and **temperatura is wysoka** then zraszanie is slabe;
- RULE 5: if **wilgotnosc is srednia** and temperatura is niska then zraszanie is brak;
- RULE 6: if **wilgotnosc is srednia** and **temperatura is srednia** then zraszanie is slabe;
- RULE 7: if **wilgotnosc is srednia** and **temperatura is wysoka** then zraszanie is srednie;
- RULE 8: if wilgotnosc is niska and temperatura is niska then zraszanie is slabe;
- RULE 9: if wilgotnosc is niska and **temperatura is srednia** then zraszanie is srednie;
- RULE 10: if wilgotnosc is niska and **temperatura is wysoka** then zraszanie is silne;

Następnie obliczamy wartość funkcji aktywacji całych przesłanek reguł, w których występują niezerowe funkcje aktywacji predykatów. Zakładamy, że spójnik AND wyraża się t-normą MIN

- RULE 3: if wilgotnosc is duza and **temperatura is srednia** then zraszanie is brak;

$$\mu_{RULE\,3}(16{,}20) = min(0.0\ ,\ 0.75)$$

$$\mu_{RULE\,3}(16{,}20) = 0$$

Jak widzimy wartość funkcji aktywacji tej reguły wynosi 0. Możemy to odnieść do pozostałych reguł, w których przesłankach występują predykaty o zerowej wartości funkcji przynależności. Pozostają nam następujące reguły :

- RULE 6: if **wilgotnosc is srednia** and **temperatura is srednia** then zraszanie is slabe;
- RULE 7: if **wilgotnosc is srednia** and **temperatura is wysoka** then zraszanie is srednie;

Obliczamy wartości funkcji aktywacji tych reguł :

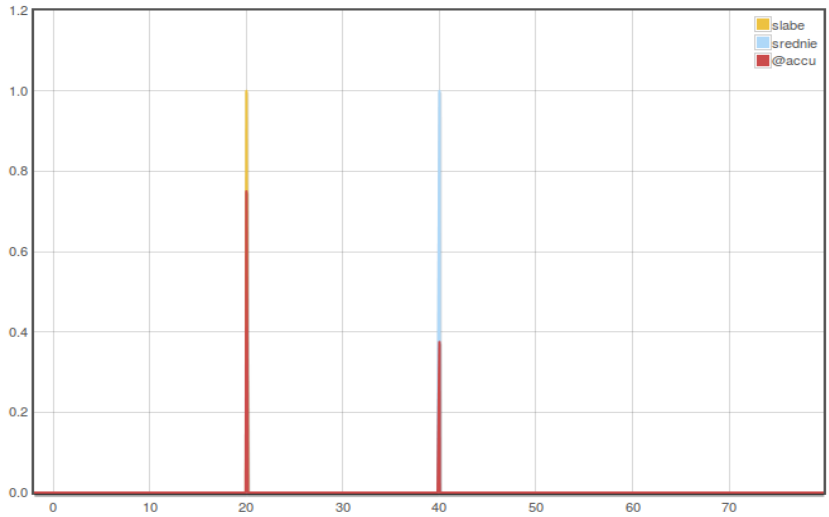
$$\mu_{RULE\,6}(16{,}20) = min(0.75\ ,\ 0.75)$$

$$\mu_{RULE\,6}(16{,}20) = 0.75$$

$$\mu_{RULE\,7}(16{,}20) = min(0.75\ ,\ 0.375)$$

$$\mu_{RULE\,7}(16{,}20) = 0.375$$

Zgodnie z konkluzją naszych reguł otrzymujemy zraszanie słabe na poziomie 0,75 i średnie na poziomie 0,375. Przeprowadzamy implikację na tych regułach (operator MIN) co prowadzi do powstania następujących termów :



rys I.3.4/1 Aktywacja predykatów konkluzji wraz z akumulacją termów

Na tym wykresie czerwoną linią zaznaczono poziom obcięcia singletonów termów SLABE I SREDNIE. Jednocześnie czerwony wykres jest akumulacją zbiorów tych dwóch singletonów. Użyty operator akumulacji to MAX.

3.5 Defuzyfikacja

Do defuzyfikacji użyjemy metody CoS (Centre of Singleton) zwaną również CoGS (Centre of Gravity for Singletons). Nie musimy obliczać całek oznaczonych jak w przypadku metody COG, ponieważ znamy z góry kształt funkcji przynależności zakumulowanych termów.

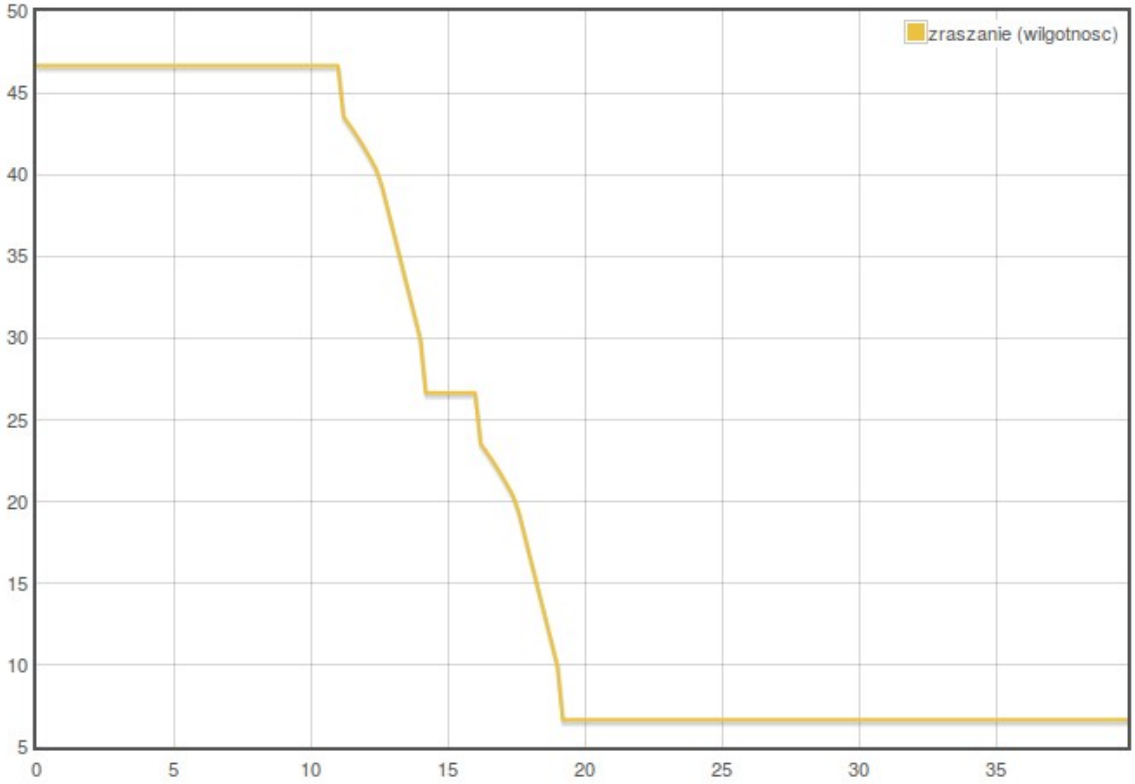
Wzór służący do obliczania tą metodą to :

$$v = \frac{\sum_{t \in S} \mu_t(t)\, t}{\sum_{t \in S} \mu_t(t)}, \textit{gdzie } S - \textit{zbiór zagregowanych singletonów}$$

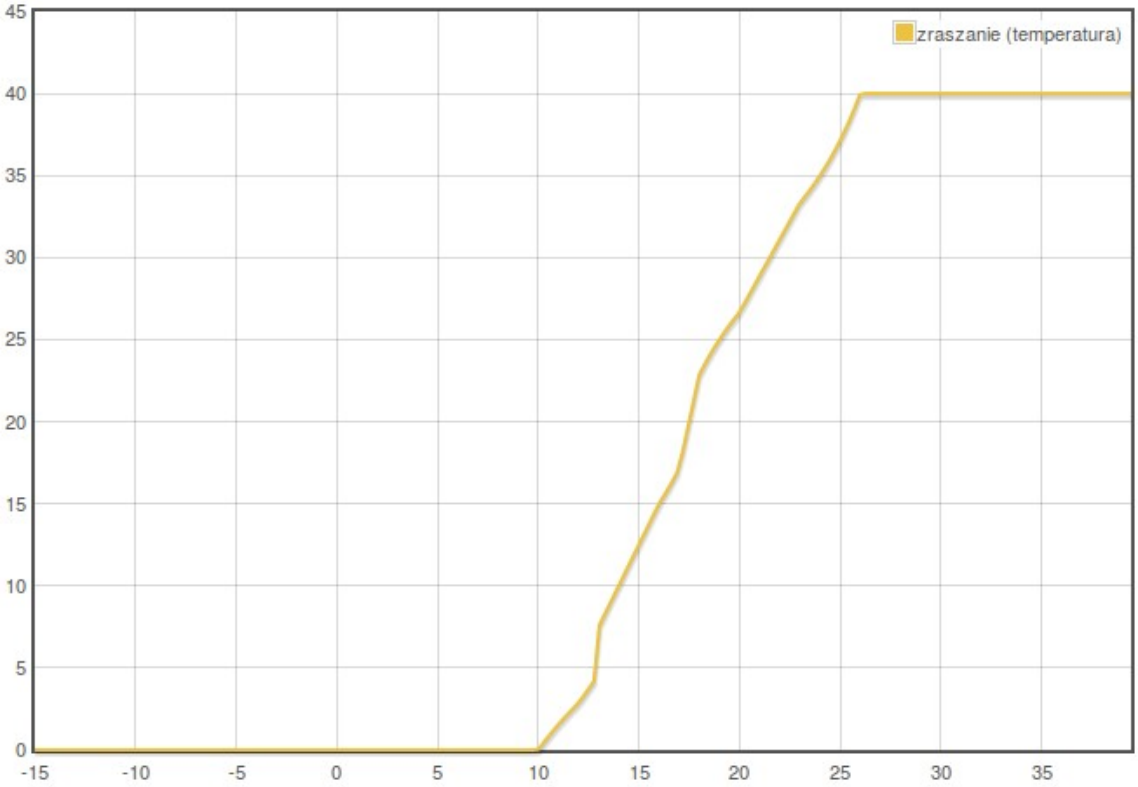
Obliczamy dla naszych danych:

$$v = \frac{0{,}75 \cdot 20 + 0{,}375 \cdot 40}{0{,}75 + 0{,}375} = \frac{30}{1{,}125} = 26{,}(6)$$

Warto zauważyć, że model Mamdaniego daje nam możliwość tworzenia złożonych funkcji na podstawie prostych reguł opisanych przez osobę, która może być laikiem w kwestii projektowania aplikacji. Z racji, że mamy do czynienia z płaszczyzną w przestrzeni następne wykresy będą dotyczyły stanu wyjściowego z poprzedniego wnioskowania :

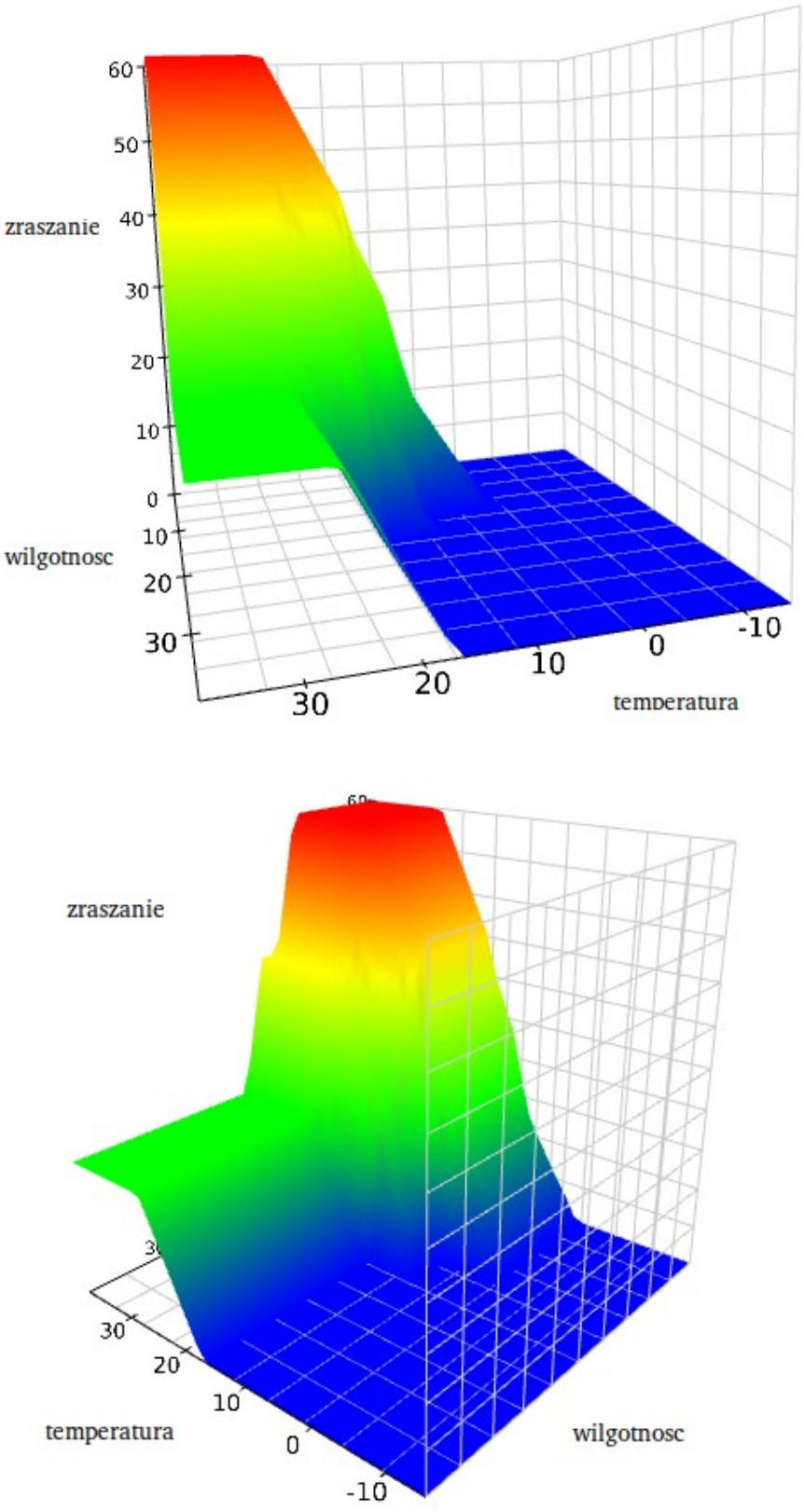


rys I.3.5/1 Zraszanie w funkcji wilgotności dla temperatury 20°C



rys I.3.5/2 Zraszanie w funkcji temperatury dla wilgotności 16%

Całą złożoność opisanego modelu widać jednak dopiero na wykresie 3D przedstawiającym $zraszanie(temperatura, wilgotnosc)$:



rys I.3.5/3 Zraszanie w funkcji temperatury i wilgotności

II. Opis składni języka Fuzzy Control Language

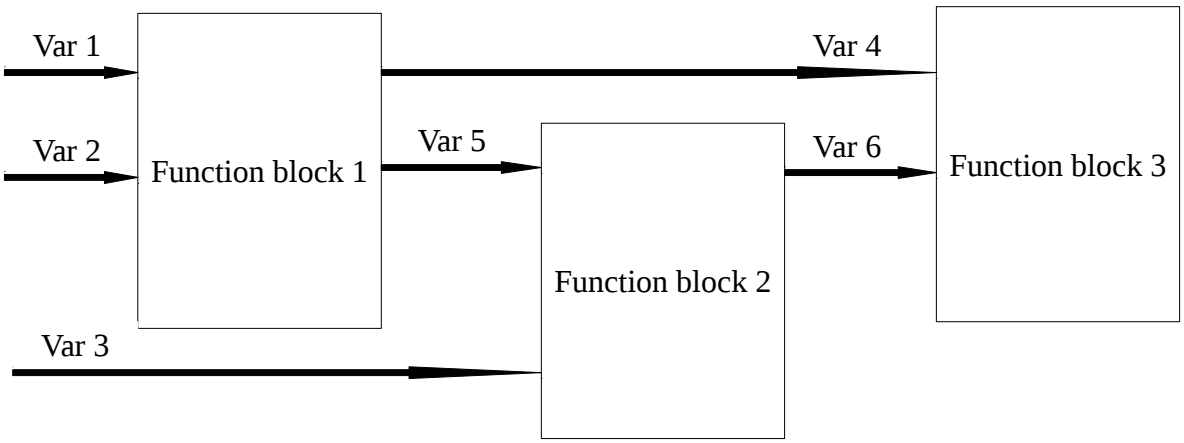
1. IEC 61131-7

Fuzzy Control Language zwany dalej FCL jest to standard opisany przez Międzynarodową Komisję Elektrotechniczną (International Electrotechnical Commision - http://www.iec.ch/) w dokumencie IEC 61131-7 w styczniu 1997 roku. Zawiera on 53 strony, 6 rozdziałów i 6 dodatków opisujących każdy detal standardu wraz z przykładami. W ramach tej pracy opis standardu zostanie ograniczony do niezbędnego minimum. [5]

Język FCL jest językiem domenowym. Oznacza to, że nie da się w nim napisać standardowego „Hello World!”, posiada instrukcje związane stricte z logiką rozmytą. Może jednak stanowić część większej aplikacji [6]. Istnieją płatne produkty implementujące ten standard, jak również aplikacje open source, np. http://jfuzzylogic.sourceforge.net/html/index.html. Celem tej pracy było zaimplementowanie tego systemu od podstaw z mocnym ukierunkowaniem na rozszerzalność aplikacji, jak również przedstawienie możliwości tego języka. Standard FCL przewiduje w większości przypadków więcej rozwiązań i metod niż zaimplementowano w tej pracy.

2. Bloki funkcji

Aplikacja w języku FCL składa się z bloków funkcji. Ich celem jest rozdzielenie logiki poszczególnych elementów rozwiązywanego problemu. Umożliwia łączenie poszczególnych bloków za pomocą zmiennych wejściowych i wyjściowych.



Rys II.2/1 Przykładowy schemat połączeń bloków funkcji

System nie zakłada żadnych zabezpieczeń przed rekursją połączeń, powinien o to zadbać deweloper tworzący system. Definicja bloku funkcji wygląda następująco

```
FUNCTION_BLOCK <nazwa bloku>
…
END_FUNCTION_BLOCK
```

Uwaga: w języku FCL wielkość liter nie ma znaczenia

3. Blok zmiennych

Blok zmiennych służy do deklaracji zmiennych użytych w danym bloku funkcji. Tylko zmienne zdefiniowane w tym bloku mogą być użyte w pozostałych elementach bloku funkcji. Definiuje się go następująco:

```
VAR_INPUT
     <nazwa zmiennej wejściowej 1> : <typ zmiennej – REAL|INT>;
     <nazwa zmiennej wejściowej 2> : <typ zmiennej>
END_VAR
VAR_INLINE
     <nazwa zmiennej wewnętrznej 3> : <typ zmiennej>;
END_VAR
VAR_OUTPUT
     <nazwa zmiennej wyjściowej 4> : <typ zmiennej>;
END_VAR
```

Zmienne wejściowe i wyjściowe są niezbędne do prawidłowej komunikacji z aplikacją zewnętrzną lub innymi blokami funkcji. Zmienne wewnętrzne służą do obliczeń pośrednich. W przypadku gdy w aplikacji znajduje się kilka bloków funkcji należy uważać na nazewnictwo zmiennych wewnętrznych ponieważ zmienne o tej samej nazwie w różnych blokach wskazują na tę samą zmienną w całym systemie. W aplikacji będącej przedmiotem tej pracy zaimplementowano jedynie typ zmiennej REAL.

4. Blok fuzyfikacji

```
FUZZIFY temperatura
     TERM <nazwa termu 1> := (x₁,μ₁) (x₂,μ₂) (...);
     TERM <nazwa termu 2> := (x₁,μ₁) (x₂,μ₂) (...);
     TERM <nazwa termu 3> := x;
END_FUZZIFY
```

Blok fuzyfikacji służy do zdefiniowania funkcji przynależności termów zmiennych wejściowych. Można je definiować na dwa sposoby:

- Podając kolejne punkty, których połączenie utworzy funkcję przynależności termu (x,μ) przy czym μ musi się zawierać w przedziale <0,1> i punkty nie mogą się powtarzać
- Podając jedną wartość rzeczywistą która utworzy funkcję singletonu

Zakres zmiennej (min,max) zostanie obliczony na podstawie skrajnych punków wszystkich termów. Bloki te powtarzają się dla każdej zmiennej wejściowej

5. Blok defuzyfikacji

Tym blokiem opisuje się zmienne wewnętrzne oraz wyjściowe. Definicja tego bloku wygląda następująco:

```
DEFUZZIFY <nazwa zmiennej wyjściowej lub wewnętrznej>
     TERM <nazwa termu 1> := (x₁,μ₁) (x₂,μ₂) (...);
     TERM <nazwa termu 2> := (x₁,μ₁) (x₂,μ₂) (...);
     TERM <nazwa termu 3> := x;
     ACCU : <nazwa metody akumulacji MAX|PROD>;
     METHOD : <nazwa metody defuzyfikacji COS|COG>;
     DEFAULT := <opcjonalne – rzeczywista wartość domyślna>;
END_DEFUZZIFY
```

Podobnie jak w przypadku bloku fuzyfikacji znajdują się tu definicje termów, które mogą występować w dwóch postaciach punktów albo singletonu. Dodatkowo dochodzą opcje niedostępne bloku fuzyfikacji:

- ACCU – metoda akumulacji termów wyjściowych poszczególnych reguł których predykaty dotyczą opisywanej zmiennej (MAX albo PROD)
- METHOD – metoda wyostrzania (defuzyfikacji). W zależności od użytych funkcji przynależności termów do dyspozycji są dwie metody COG (Centre of Gravity) – dla punktów, COS (Centre of Singleton) – dla singletonów
- DEFAULT – domyślna wartość tej zmiennej, jeśli żadna reguła nie wpływa na jej wartość

6. Bloki reguł

W jednym bloku funkcji może istnieć kilka bloków funkcji. Taka możliwość została dodana w celu rozbicia reguł na zbiory spójne strukturalnie. Nie ma to jednak wpływu na sam proces agregacji i defuzyfikacji, ponieważ jest on przypisany bezpośrednio do zmiennej wyjściowej lub wewnętrznej. Pozwala jedynie na zdefiniowanie indywidualnej t-normy dla operatora AND (OR). Defnicja bloku wygląda następująco:

```
RULEBLOCK No1
     AND : MIN;
     RULE 1: if <zmienna wej> is <term> then <zmienna wyj> is <term>;
     RULE 2: if <zmienna wej> is <term> <and|or> <zmienna wej> is <term> then
     <zmienna wyj> is <term> and <zmienna wyj> is <term>;
END_RULEBLOCK
```

Jednym z elementów bloku reguł jest zadeklarowanie t-normy dla spójnika AND (co pociąga za sobą odpowiadającą wg praw De Morgana t-normę spójnika OR). Szczegóły opisu reguł znajdują się w rozdziale I.3.2

III. Szczegóły implementacji sterownika FCL

1. API

1.1 Wstęp

Sterownik został opracowany z myślą wykorzystania go w aplikacjach napisanych w języku JAVA. Stanowi on część całości aplikacji odpowiadającą za prawidłowe modelowanie struktur na podstawie modlu Mamdaniego oraz parsowanie dokumentów napisanych w języku FCL.

1.2 Wymagania

- Środowisko uruchomieniowe Java SE Runtime Enviroment w wersji minimum 8
- Plik zawierający bibliotekę fcl.jar

1.3 Uruchamianie i wczytywanie skryptów fcl

Tworzenie instancji parsera :

```
Parser parser = new Parser (String document);
```

Parsowanie danych :

```
parser.parse();
```

Wyciąganie aplikacji :

```
Application application = parser.getApplication();
```

Aplikacja jest już gotowa do użycia – wszystkie reguły ustalone w dokumencie FCL są dostępne i odpalane automatycznie przy zmianie wartości zmiennych które znajdują się w ich części przesłanki.

Ustawianie zmiennych

```
application.setValue(<nazwa zmienniej - String>,<wartość double>);
```

Po takim ustawieniu odpalą się właściwe dla tej zmiennej reguły i można używać obliczonych wartości zmiennych wyjściowych

```
double value = application.getValue(<nazwa zmiennej - String>);
```

2. Biblioteka FCL

2.1 Wstęp

Zadaniem biblioteki jest udostępnienie programistycznego API umożliwiającego ładowanie skryptów FCL oraz korzystanie z wyników ich działania w innej aplikacji.

2.2 Moduły

Biblioteka dzieli się na następujące moduły :

- Moduł główny research.fcl.library
- Moduł parsera research.fcl.library.parser
- Moduł środowiska research.fcl.library.enviroment
- Moduł bloku funkcji research.fcl.library.functionblock
- Moduł zmiennych blokowych research.fcl.library.variables
- Moduł reguł research.fcl.library.rules
- Moduł termów research.fcl.library.terms
- Moduł akumulacji research.fcl.library.accumulation
- Moduł spójników research.fcl.library.andmethods
- Moduł defuzyfikacji research.fcl.library.defuzzification

Ze względu na dużą złożoność biblioteki (ponad 90 klas) Moduły zostaną przedstawione rozdzielnie.

2.3. Moduł główny biblioteki

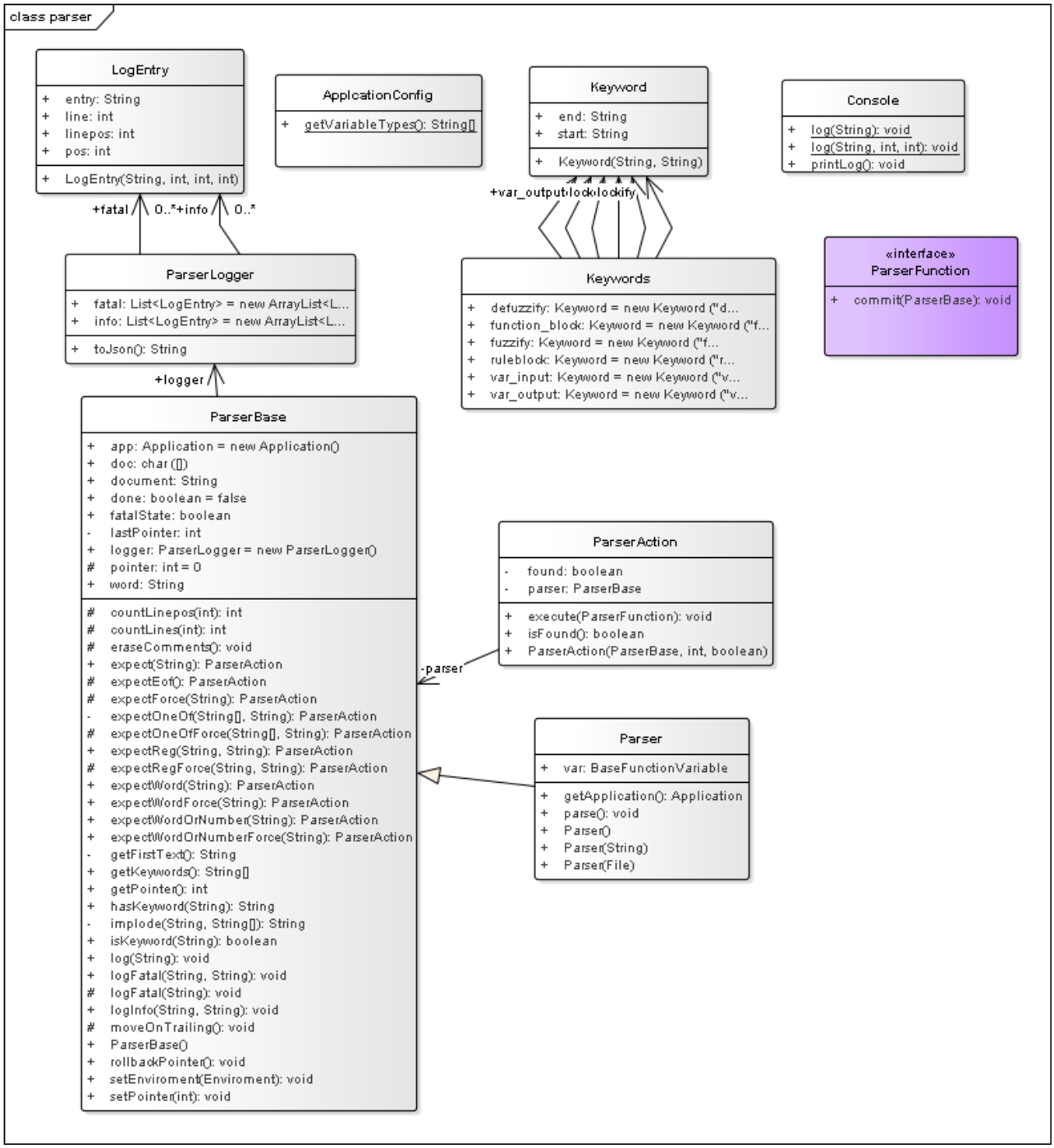
W module głównym biblioteki znajduje się jedna klasa, której zadaniem jest reprezentacja modelu opisanego w załadowanym skrypcie FCL. Sygnatura klasy wygląda następująco :

rys. II.2.3/1 Sygnatura klasy Application

Wszystkie gettery zostały udostępnione z myślą o rozszerzaniu aplikacji o nowe funkcjonalnści. Klasa posiada również delegaty ułatwiające operacje na środowisku zmiennych :

- `getValue(String)`
- `setValue(String,double)`

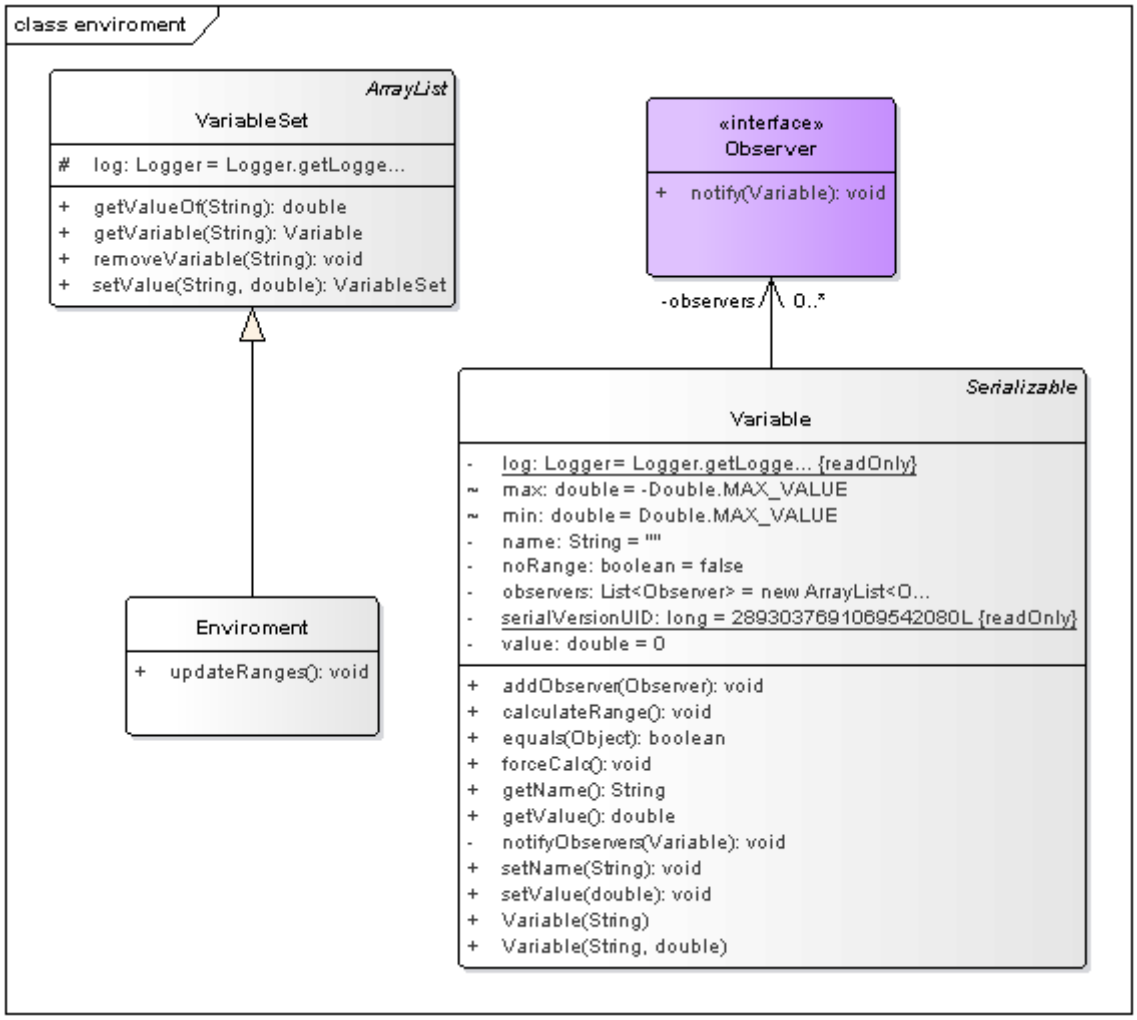
2.4 Moduł parsera



Rys III.2.4/1 Parser wraz zależnościami

Parser jest jednym z bardziej złożonych elementów aplikacji. Korzysta on z mechanizmu Regex do wyszukiwania słów odpowiadających wzorcom przygotowanym do konstrukcji skrytpu FCL. Posiada również mechanizm logowania błędów, które mogą wystąpić w skryptach.

2.5 Moduł środowiska

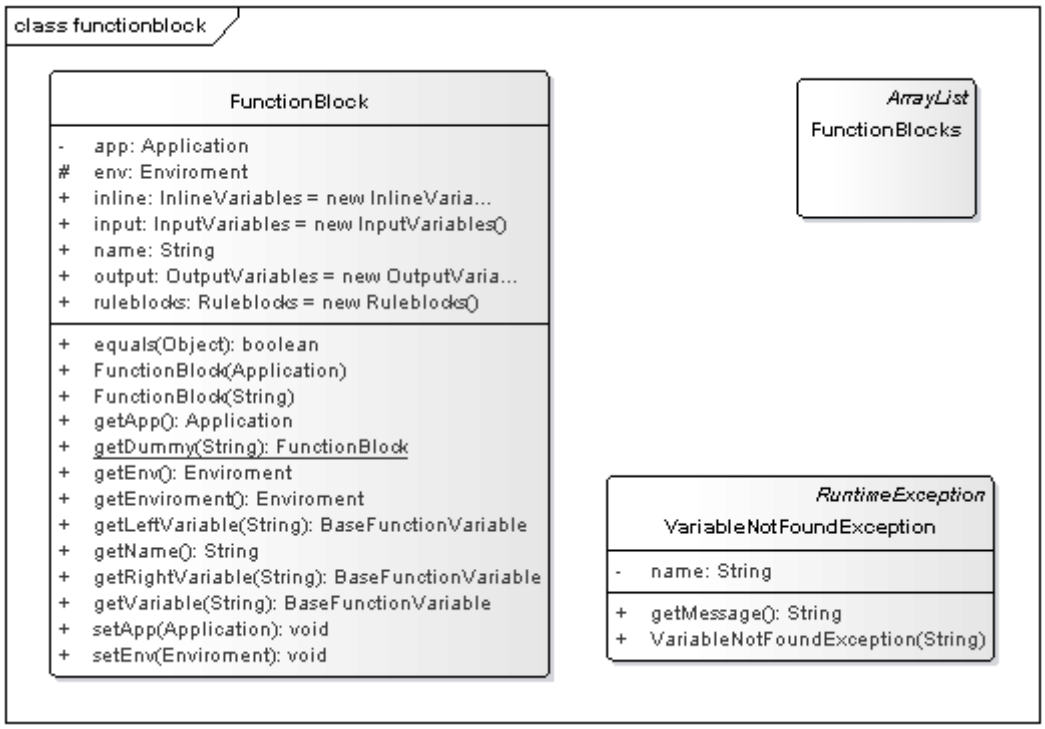


Rys III.2.5/1 Środowisko wraz z zależnościami

Środowisko jest mocno uniezależnione od pozostałej części aplikacji. Zmienne blokowe odwołują się do zmiennych środowiska implementując interfejs `Observer` i dodając się do listy obserwatorów odpowiedniej zmiennej. Środowisko jest mapą `String→double`. Istnieje możliwość Utworzenia nowej aplikacji na podstawie innych skryptów przy zachowaniu wartości zmiennych środowiska. Warunkiem jest pokrywanie się nazw zmiennych w nowych skryptach.

2.6 Moduł bloku funkcji

Aplikacja składa się z mapy bloków funkcji, z których każdy posiada swoją nazwę. Blok odpowiada za przechowywanie zmiennych wejściowych, wewnętrznych i wyjściowych. Daje również możliwość wyciągnięcia zmiennych przeznaczonych do użycia w przesłankach – zwanych dalej lewymi oraz tych, które można użyć w konkluzjach – zwane dalej prawymi. Wiąże się to z faktem, że zmienne wewnętrzne mogą należeć do obu zbiorów zmiennych lewych i prawych. Oprócz zmiennych zawiera również listę bloków reguł.

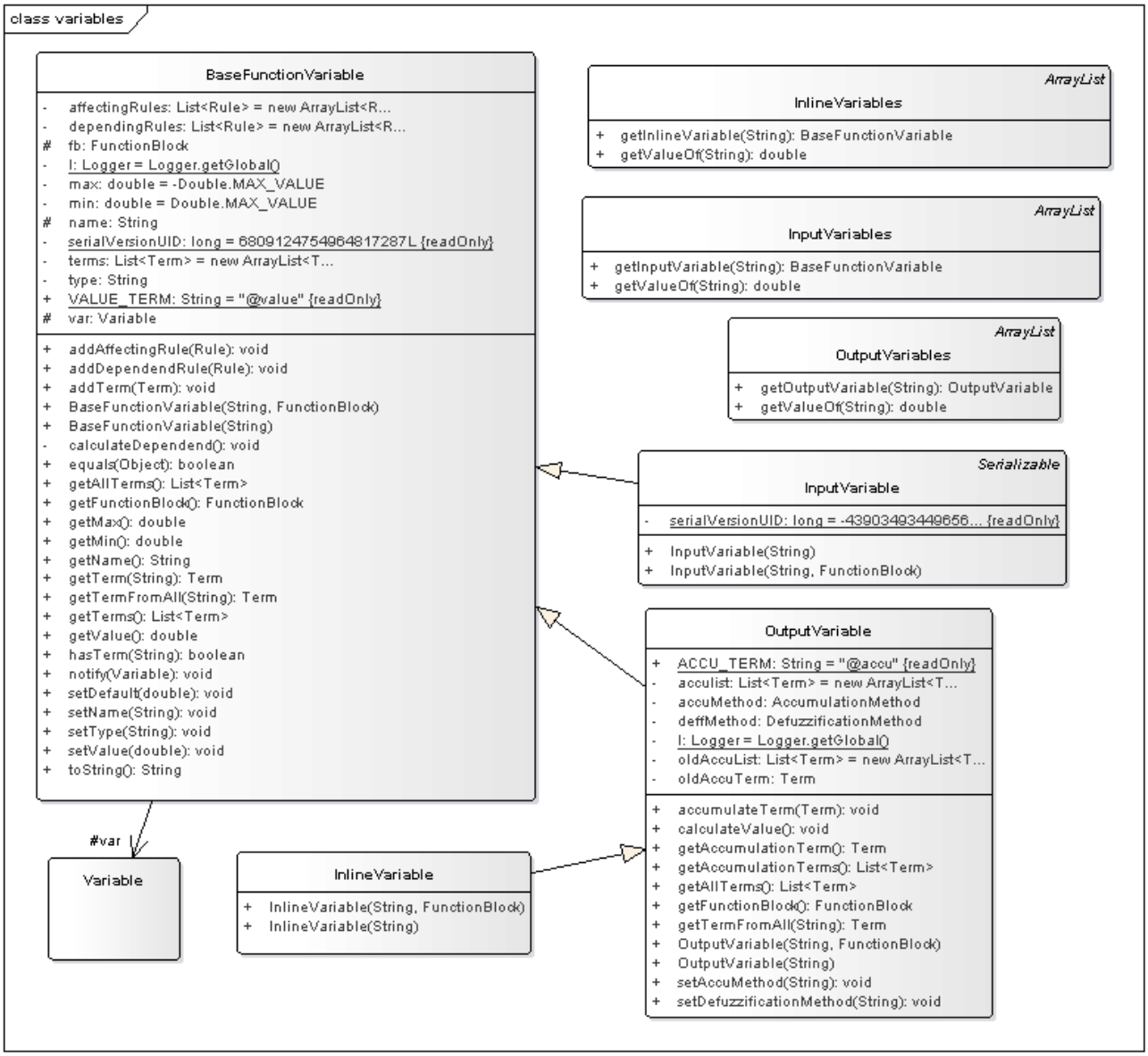


Rys III.2.6/1 Blok funkcji wraz z zależnościami

2.7 Moduł zmiennych blokowych

Zmienne blokowe wskazują na zmienne środowiskowe i reagują na ich zmiany implementując interfejs `Observer`. Zmienne środowiskowe informują o każdej zmianie uruchamiając metodę `notify(Variable)`. Klasa BaseFunctionVariable po której dziedziczą wszystkie pozostałe typy zmiennych blokowych implementuje ten interfejs. Godnymi uwagi elementami tej klasy są :

- lista reguł od których zależy ta zmienna (zmienna występuje po stronie konkluzji w tych regułach)
- lista reguł które zależą od tej zmiennej (zmienna występuje po stronie przesłanki w tych regułach)
- lista termów które są przypisane do tej zmiennej
- lista pojedynczych termów zebranych z wszystkich predykatów konkluzji
- funkcję przynależności dla zakumulowanych termów dla ostatniego przeliczenia
- funkcję ustawiającą domyślną wartość tej zmiennej
- funkcję służącą do prezentacji wszystkich termów tej zmiennej – włącznie z zakumulowanymi

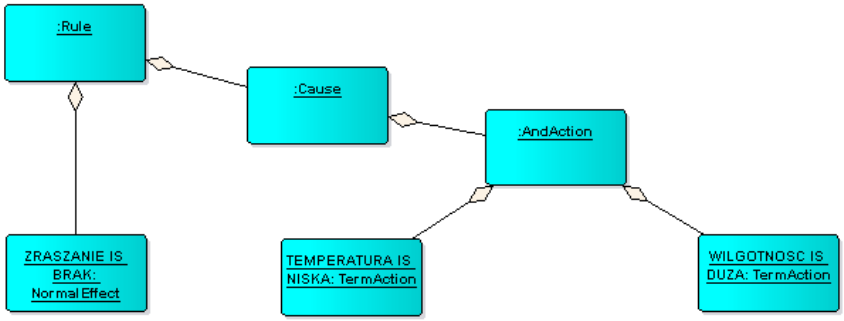


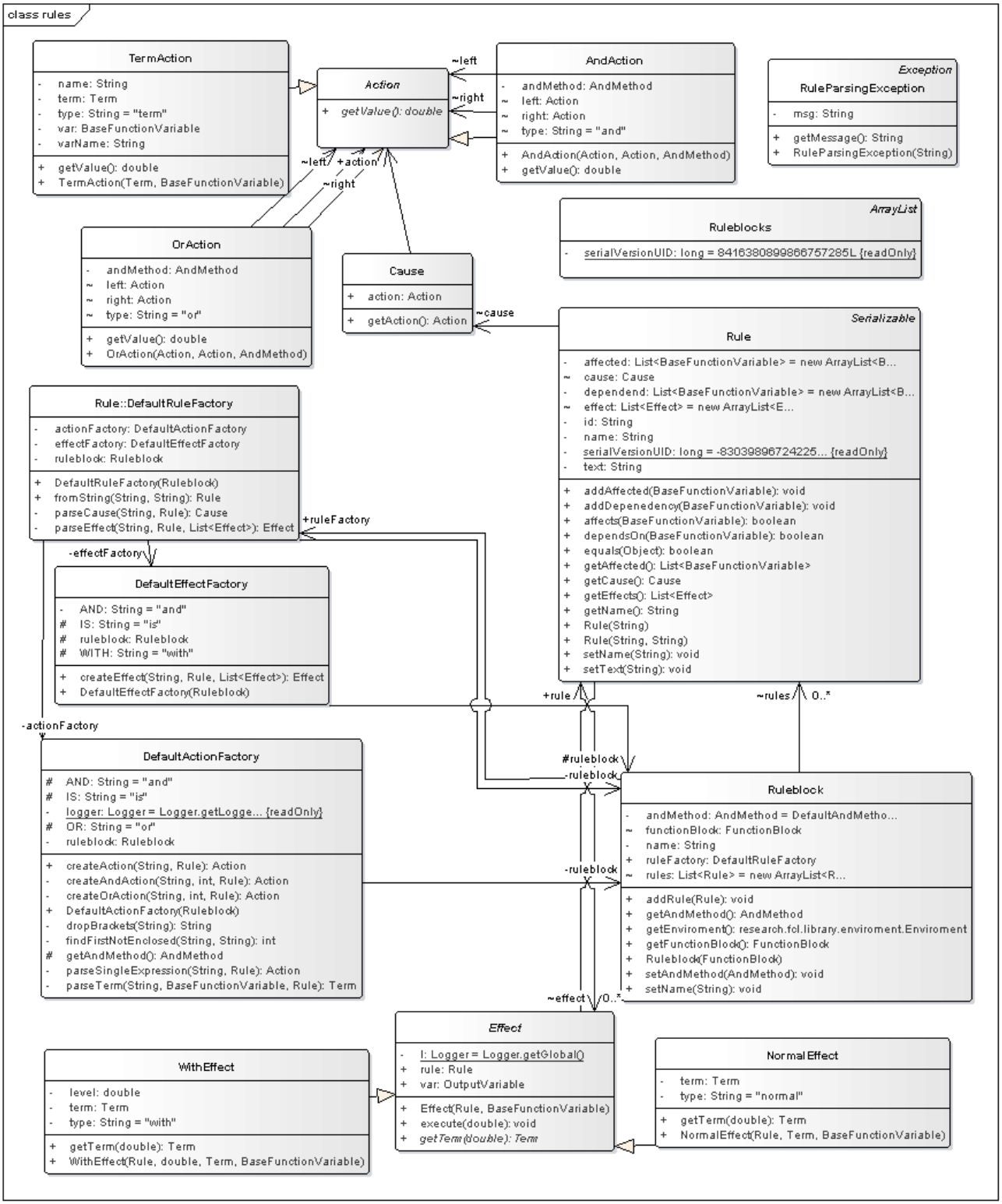
Rys III.2.7/1 Diagram klas modułu zmiennych blokowych

2.8 Moduł reguł

Moduł ten odpowiada za prawidłowe parsowanie treści reguł i przekonwertowanie ich do drzewa obiektów wykonujących właściwe operacje. Przykładowe drzewo reguły :

```
if wilgotnosc is duza and temperatura is niska then zraszanie is brak;
```




class rules
TermAction
- name: String
- term: Term
- type: String = "term"
- var: BaseFunctionVariable
- varName: String
+ getValue(): double
+ TermAction(Term, BaseFunctionVariable)
Action
+ getValue(): double
~left
~right
AndAction
- andMethod: AndMethod
~ left: Action
~ right: Action
~ type: String = "and"
+ AndAction(Action, Action, AndMethod)
+ getValue(): double
Exception
RuleParsingException
- msg: String
+ getMessage(): String
+ RuleParsingException(String)
~left
+action
~right
OrAction
- andMethod: AndMethod
~ left: Action
~ right: Action
~ type: String = "or"
+ getValue(): double
+ OrAction(Action, Action, AndMethod)
Cause
+ action: Action
+ getAction(): Action
ArrayList
Ruleblocks
- serialVersionUID: long = 8416380899866757285L {readOnly}
~cause
Serializable
Rule
- affected: List<BaseFunctionVariable> = new ArrayList<B...
~ cause: Cause
- dependend: List<BaseFunctionVariable> = new ArrayList<B...
~ effect: List<Effect> = new ArrayList<E...
- id: String
- name: String
- serialVersionUID: long = -83039896724225... {readOnly}
- text: String
+ addAffected(BaseFunctionVariable): void
+ addDepenedency(BaseFunctionVariable): void
+ affects(BaseFunctionVariable): boolean
+ dependsOn(BaseFunctionVariable): boolean
+ equals(Object): boolean
+ getAffected(): List<BaseFunctionVariable>
+ getCause(): Cause
+ getEffects(): List<Effect>
+ getName(): String
+ Rule(String)
+ Rule(String, String)
+ setName(String): void
+ setText(String): void
Rule::DefaultRuleFactory
- actionFactory: DefaultActionFactory
- effectFactory: DefaultEffectFactory
- ruleblock: Ruleblock
+ DefaultRuleFactory(Ruleblock)
+ fromString(String, String): Rule
- parseCause(String, Rule): Cause
- parseEffect(String, Rule, List<Effect>): Effect
+ruleFactory
-effectFactory
DefaultEffectFactory
- AND: String = "and"
IS: String = "is"
ruleblock: Ruleblock
WITH: String = "with"
+ createEffect(String, Rule, List<Effect>): Effect
+ DefaultEffectFactory(Ruleblock)
+rule
~rules
0..*
-actionFactory
DefaultActionFactory
AND: String = "and"
IS: String = "is"
- logger: Logger = Logger.getLogge... {readOnly}
OR: String = "or"
- ruleblock: Ruleblock
+ createAction(String, Rule): Action
- createAndAction(String, int, Rule): Action
- createOrAction(String, int, Rule): Action
+ DefaultActionFactory(Ruleblock)
- dropBrackets(String): String
- findFirstNotEnclosed(String, String): int
getAndMethod(): AndMethod
- parseSingleExpression(String, Rule): Action
- parseTerm(String, BaseFunctionVariable, Rule): Term
#ruleblock
-ruleblock
Ruleblock
- andMethod: AndMethod = DefaultAndMetho...
~ functionBlock: FunctionBlock
- name: String
+ ruleFactory: DefaultRuleFactory
~ rules: List<Rule> = new ArrayList<R...
+ addRule(Rule): void
+ getAndMethod(): AndMethod
+ getEnviroment(): research.fcl.library.enviroment.Enviroment
+ getFunctionBlock(): FunctionBlock
+ Ruleblock(FunctionBlock)
+ setAndMethod(AndMethod): void
+ setName(String): void
-ruleblock
~effect
0..*
Effect
- l: Logger = Logger.getGlobal()
+ rule: Rule
+ var: OutputVariable
+ Effect(Rule, BaseFunctionVariable)
+ execute(double): void
+ getTerm(double): Term
WithEffect
- level: double
- term: Term
- type: String = "with"
+ getTerm(double): Term
+ WithEffect(Rule, double, Term, BaseFunctionVariable)
NormalEffect
- term: Term
- type: String = "normal"
+ getTerm(double): Term
+ NormalEffect(Rule, Term, BaseFunctionVariable)

Bibliografia

1) https://pl.wikipedia.org/wiki/Logika_tr%C3%B3jwarto%C5%9Bciowa
2) https://en.wikipedia.org/wiki/De_Morgan%27s_laws
3) Mamdani, E. H., Application of fuzzy algorithms for the control of a simple dynamic plant. In *Proc IEEE* (1974)
4) Andrzej Piegat; „Modelowanie i sterowanie rozmyte”; Akademicka Oficyna Wydawnicza EXIT, Warszawa 1999
5) http://www.fuzzytech.com/binaries/ieccd1.pdf
6) https://en.wikipedia.org/wiki/Fuzzy_Control_Language