POLITECHNIKA KOSZALIŃSKA

WYDZIAŁ ELEKTRONIKI I INFORMATYKI

INFORMATYKA

PROGRAMOWANIE KOMPUTERÓW

I SIECI INFORMATYCZNE

Łukasz Zieliński

62093

PROGRAMOWE MODELOWANIE STRUKTUR BLOKOWYCH SYSTEMÓW ROZMYTYCH

MODELING BLOCK STRUCTURES OF FUZZY SYSTEMS

Praca inżynierska wykonana pod kierunkiem

dr inż. Marek Popławski

Streszczenie pracy:

Tematem niniejszej pracy jest modelowanie struktur blokowych systemów rozmytych z wykorzystaniem języka Fuzzy Control Language. Główne elementy pracy to krótkie omówienie teorii logiki rozmytej i dokumentacja techniczna aplikacji. Część teoretyczna zawiera historyczny zarys powstawania logiki rozmytej oraz szczegółowy opis modelu Mamdaniego. Dokumentacja składa się z opisu elementów inżynierii oprogramowania oraz instrukcji obsługi aplikacji.

Słowa kluczowe:

logika rozmyta, modelowanie, systemy rozmyte, Mamdani, Java, aplikacje webowe

Title:

Modeling block structures of fuzzy systems

Abstract:

Subject of this document is modeling block structures of fuzzy systems with Fuzzy Control Language. Key elements are brief dive into theory of fuzzy logic and technical documentation of application. Theoretical part contains outline of history behind fuzzy logic and details of Mamdani's model. Documentation consists of elements of software engineering and application's user manual.

Keywords:

fuzzy logic, modeling, fuzzy systems, Mamdani, Java, web applications

Oświadczenie

Łukasz Zieliński
(Imię i Nazwisko studenta)

Numer 62093
(Nr albumu)

Oświadczam, że moja praca pt.: *Programowe modelowanie struktur blokowych systemów rozmytych*. została przygotowana przeze mnie samodzielnie*:

a) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. Nr 24., poz. 83 z późn. zm.) oraz dóbr osobistych chronionych prawem,

b) nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,

c) nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie ani innej osobie.

Ponadto oświadczam, że treść pracy przedstawionej przez mnie do obrony, zawarta na przekazywanym nośniku elektronicznym, jest identyczna z jej wersją drukowaną.

……. (data)

…… (podpis studenta)

*Uwzględniając merytoryczny wkład promotora (w ramach prowadzonego seminarium dyplomowego)

Spis treści

I. 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 trójwartościowiej 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 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.1. Cel i zakres pracy

Celem tej pracy jest utworzenie oprogramowania umożliwiającego naukę podstaw logiki rozmytej, modelowania struktur blokowych systemów rozmytych oraz ułatwiającego pracę z nimi. Zadanie to ma być zrealizowane za pomocą kompilatora języka Fuzzy Control Language (FCL). Oprogramowanie ma wspierać projektowanie struktur rozmytych wizualizując elementy istotne przy modelowaniu takie jak:

- funkcje przynależności poszczególnych termów
- funkcje przynależności wyników akumulacji
- funkcje przynależności predykatów poszczególnych reguł
- strukturę projektu modelowanego systemu
- funkcje zależności poszczególnych zmiennych
- informacje o błędach w kodzie FCL w czasie rzeczywistym
- zmianę wartości zmiennych w czasie rzeczywistym

Dodatkowo powinno umożliwić zapis aktualnego stanu prac użytkownika oraz jego autoryzację.

Cele dodatkowe :

- łatwa rozszerzalność aplikacji o kolejne elementy języka FCL, które zostały pominięte podczas realizacji tej pracy
- wykorzystanie metod inżynierii oprogramowania
- zamodelowanie przykładów w języku FCL
- umożliwienie podłączenia aplikacji symulacyjnej bezpośrednio do symulowanego systemu

II. Podstawy teorii modelowania systemów rozmytych

II.1. Historia postępu i stan aktualny

Ponad trzydzieści lat minęło od sformułowania podstaw teorii zbiorów rozmytych przez Lotofi Zadeha z Uniwersytetu Kalifornijskiego[3]. W trakcie trwania ostatniego ćwierćwiecza teoria ta się bardzo rozwinęła. Wstępując zarówno do wielu działów matematyki, techniki i, co jest najbardziej zadziwiające, również do ekonomii i wielu innych dziedzin. To, że teoria zbiorów rozmytych jest przydatna jest widoczne patrząc na dowody w postaci zastosowania ich w technice sterowania i w systemach ekspertowych o rozmaitym przeznaczeniu, np. w technologii maszyn. Najbardziej spektakularnym osiągnięciem ostatnich lat w dziedzinie sterowania rozmytego jest zaprojektowanie oraz zrobienie modelu helikoptera, który był bezzałogowy. Reagował on na rozmyte rozkazy, które były przekazywane drogą radiową. W przemyśle samochodowym również zastosowano sterowanie rozmyte, dotyczące automatycznych przekładani biegów.

II.2. Podstawowe pojęcia i teoria zbiorów rozmytych

Obserwując cały rozwój logiki, od czasów starożytnych do pierwszych dziesięcioleci obecnego wieku, można tę podstawową gałąź matematyki, jaką jest logika, utożsamić z logiką dwuwartościową, mimo tego że zagadnienia logiki wielowartościowej były wówczas znane[4]. Miało to związek z tendencją do ścisłego i precyzyjnie sformalizowanego opisu wszelkich obiektów, jak i pewnej niechęci dla niejednoznaczności.

Aby poprawnie opisać zbiór rozmyty można stosować różne aspekty. Przyjmuje się, że w zbiorze rozmytym nie ma ostrej granicy pomiędzy elementami, które należą do zbioru, a tymi, które należą do dopełnienia. W takim razie określa się stopień przynależności elementów do zbioru po to, aby móc scharakteryzować zbiór rozmyty poprzez funkcje przynależności, która przyjmuje wartości [0,1].

Za pomocą zbiorów rozmytych możemy formalnie określić pojęcia nieprecyzyjne i wieloznaczne, takie jak „wysoka temperatura”, „młody człowiek”, „średni wzrost” lub „duże miasto”. [4]Przed podaniem definicji zbioru rozmytego musimy ustalić tzw. obszar rozważań,w przypadku pojęcia wieloznacznego „dużo pieniędzy” inna suma będzie uważana za dużą, jeżeli ograniczymy się do obszaru rozważań [0.1000 zł], a inna – jeżeli przyjmiemy przedział [0;1000 000 zł]. [4] Obszar rozważań nazywany jest w dalszym ciągu zbiorem lub przestrzenią, oznaczymy go jako X, pamiętając że X jest zbiorem nierozmytym. [4]

Definicja [4]

Zbiór rozmyty A w pewnej (niepustej) przestrzeni X definiowany jest przez pary

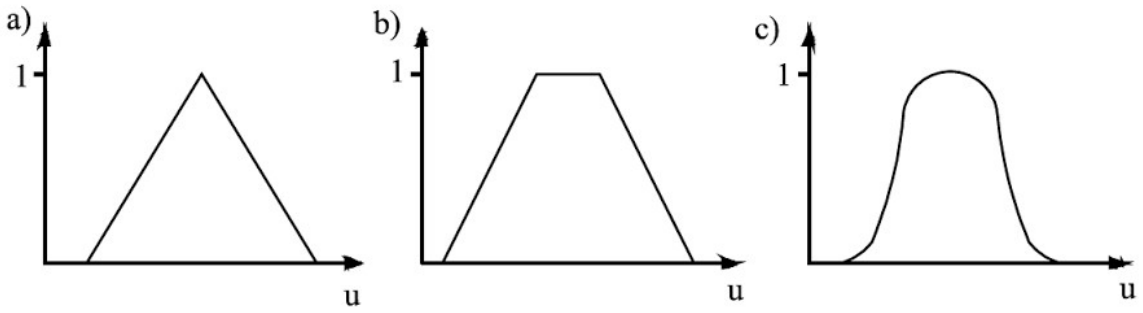
$F=\{u,\ \mu_F(u) \mid u \in U\}$ [4]

w którym μa:X->[0.1] jest funkcją przynależności zbioru rozmytego A[5]. W odróżnieniu od klasycznego podejścia teorii zbiorów, która mówi o funkcji przypisującej dwie wartości {0,1}, w zbiorach rozmytych wyróżniamy trzy przypadki:

μa(x)=1 – pełna przynależność do zbioru rozmytego A [6]

μa(x)=0 – brak przynależności elementu x do zbioru rozmytego A [6]

0<μa(x)<1 – częściowa przynależność elementu x do zbioru rozmytego A [6]



rys. II.2./1Różne formy funkcji przynależności: a) trójkątna, b) trapezoidalna, c) kształt dzwonu[7]

II.3. Operacje na zbiorach rozmytych

Poniżej zostaną opisane najważniejsze operacje na zbiorach rozmytych. Wśród wielu liczb operacji można wyróżnić operacje mnogościowe oraz algebraiczne.

Z operacji mnogościowych wymienić trzeba następujące operacje:

- **jednoargumentowa operacja dopełnienia zbioru rozmytego**

- **dwuargumentowa operacja sumy mnogościowej dwóch zbiorów**

$A, B, C \in X$

$C = A \vee B$

$\forall x \in X$:

μc(x) = max[$\mu_A(x), \mu_B(x)$]

- **przecięcie**

$A, B, D \in X$

D= $a \wedge b$

$\forall x \in X$:

$\mu_D(x) = min$ [$\mu_A(x), \mu_B(x)$]

Definicje, które są zawarte wyżej nadają zbiorowi podzbiorów rozmytych zbioru X strukturę, podobną do algebry Boolea, jednak jest ona inna poprzez brak obowiązywania fundamentalnych zasad teorii klasycznych:

- **zasada sprzeczności**, bo: $A \cap \bar{A} \neq \emptyset$ [8]
- **zasada łączonego ośrodka,** ponieważ: $A \cup \bar{A} \neq \emptyset$ [8]

Oprócz operacji mnogościowych, trzeba wymienić jeszcze szereg operacji algebraicznych w zbiorach rozmytych. Należą do nich operacje mnożenia i potęgowania funkcji przynależności zbiorów rozmytych. Do operacji potęgowania należą: operacja koncentracji zaostrzająca zbiór rozmyty, operacja rozcieńczania, spłaszczająca zbiór rozmyty oraz intensyfikacja kontrastu zbioru rozmytego. [7]

Istnieje jednak jeszcze bardzo duża ilość operacji stosowanych – własności, które zajmują ważne miejsce w zbiorach rozmytych i są szczególnie interesujące [8]:

-**przemienność:**

$A \cup B = B \cup A$

$A \cap B = B \cap A$

-idempotentność

$A \cup B = A$

$A \cap B = A$

-łączność

$A \cup (B \cup C) = (A \cup B) \cup C = A \cup B \cup C$

$A \cap (B \cap C) = (A \cap B) \cap C = A \cap B \cap C$

- rodzielność

$A \cap (B \cap C)(A \cap B) \cup (A \cap C)$

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

- prawa de Morgana

$\overline{A \cup B} = \bar{A} \cap \bar{B}$

$\overline{A \cap B} = \bar{A} \cup \bar{B}$

Najważniejsze pojęcia teorii zbiorów rozmytych, które decydują o przeznaczeniu w praktyce, zostaną przedstawione poniżej.

Zmienna lingwistyczna jest to wielkość, która może przyjmować wartości lingwistyczne (wartości to zdania wyrażone w języku naturalnym). Na przykład szybkość, zmienna ta może przyjmować różne wartości (inaczej zwane termami). Termy mogą być opisane numerycznie.

Przetwarzanie danych przez człowieka oparte jest na zmiennych lingwistycznych. Zdolność przetwarzania zmiennych oraz rozumowania na ich podstawie, to niezwykłe zdolności ludzkiego umysłu. Żeby wykorzystać to, trzeba jeszcze opisać zależności między zmiennymi. Można posłużyć się do tego zdaniami warunkowymi, które opisują zależności tzw. przyczyna-skutek. Nazywane są one regułami rozmytymi.

Reguły rozmyte przedstawiane są w postaci „Jeżeli A i B to C, A i B i C” - to twierdzenia rozmyte np. (a jest małe). Można tworzyć je na podstawie prób numerycznych lub na podstawie doświadczenia specjalisty. Jeżeli dana sytuacja jest opisana zestawem reguł, to tworzą one bazę reguł. Taka baza stanowi punkt wyjściowy do problemu wnioskowania, kiedy informacje nie są precyzyjnie podane lub wnioskowania przybliżonego. Wnioskowanie takie jest zautomatyzowane w systemach wnioskowania rozmytego.

II.4. 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" 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 [9]. W tym modelu predykaty są w 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 mogą 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.

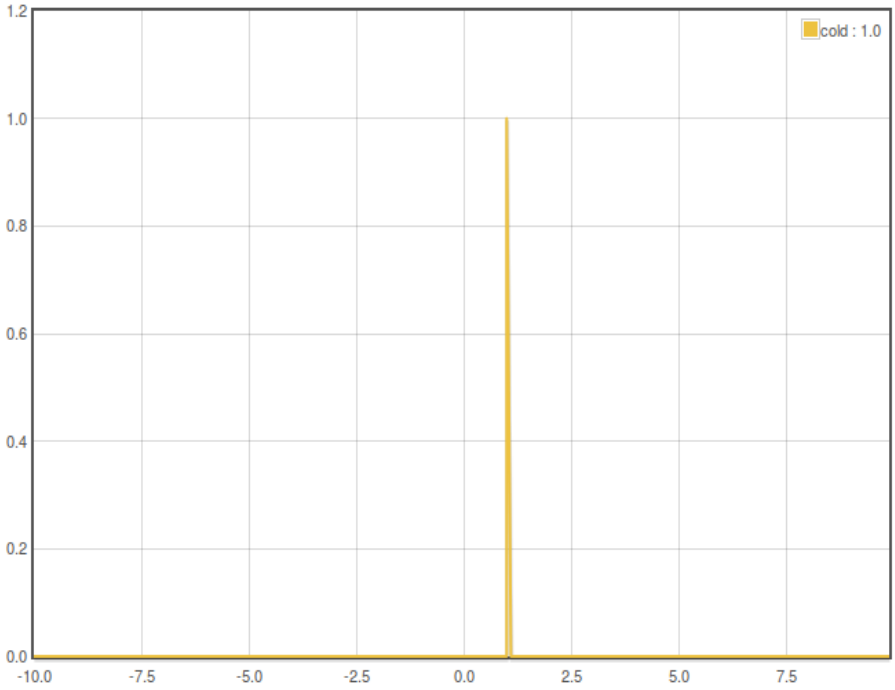
Zmienne lingwistyczne które występują po lewej stronie reguły, nazywamy zmiennymi wejściowymi sterownika. Natomiast zmienne znajdujące się w konkluzji to zmienne wyjściowe. Do wnioskowania rozmytego jako podstawę użyto uogólnioną na teorię zbiorów rozmytych, regułę *modus ponens*. Zachowano element,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

II.5. Bloki i proces wnioskowania rozmytego

II.5.1. 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. II.5.1./1. Singleton

Który zdefiniowany jest funkcją :

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

Następnie, jako wartość spełnienia predykatu przesłanki A is B, uznaje się zbiór powstały w wyniku przecięcia zbioru X 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 [10].

II.5.2. Blok i metody inferencji

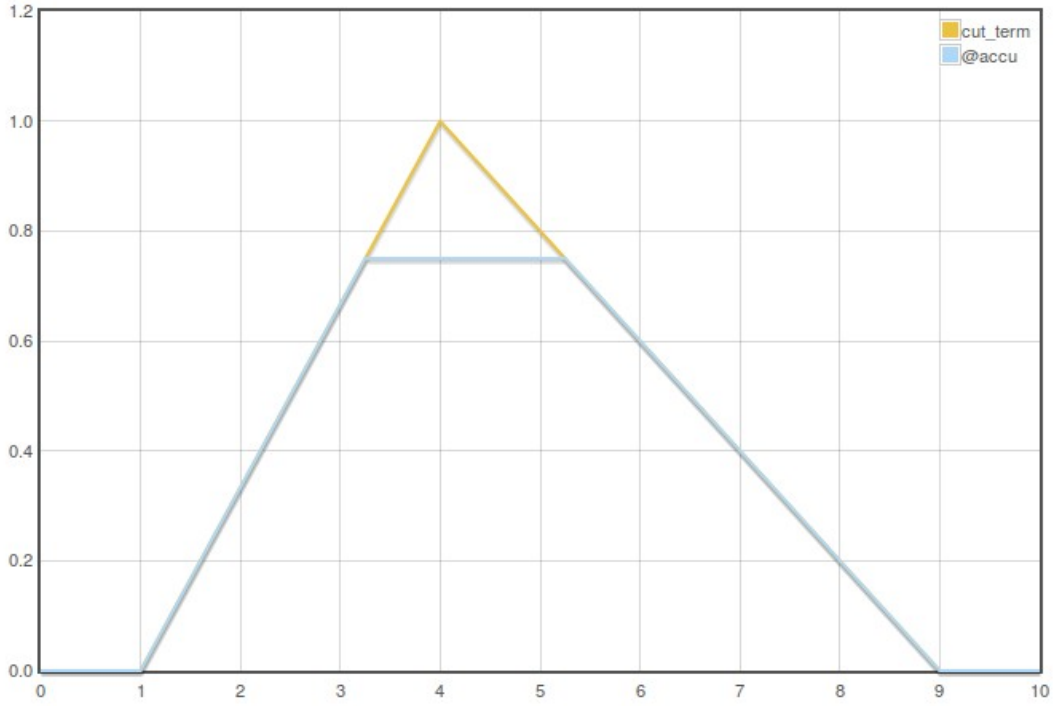
Inferencja jest kolejnym etapem działania sterownika rozmytego. Jest ona jednocześnie najbardziej złożona obliczeniowo. Możemy wyróżnić trzy etapy prac tego bloku :

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

Wszystkie te etapy wykonywane są w oparciu o przygotowaną przez eksperta bazę 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 :



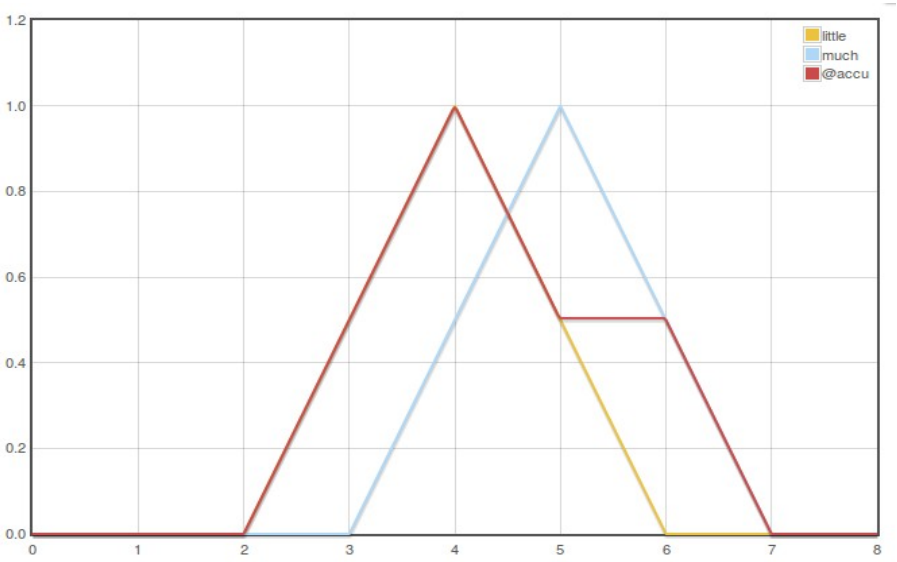
rys II.5.2./1. Przykład obciętego termu

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.

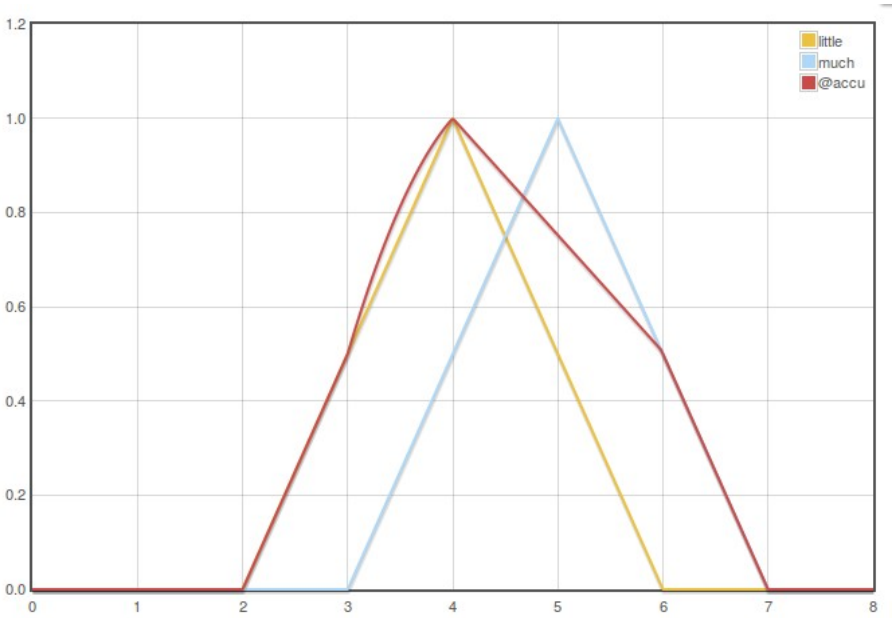
Wynikowe zbiory rozmyte kumulują się w ramach danego bloku funkcji sterownika w zbiór rozmyty, który poddawany jest następnie procesowi wyostrzania. Kumulacja polega na zebraniu wszystkich wynikowych zbiorów rozmytych danej zmiennej w jeden zbiór za pomocą funkcji kumulacji. Funkcje kumulacji dostępne 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. II.5.2./1 akumulacja metodą MAX



rys II.5.2./2 Akumulacja metodą PROD

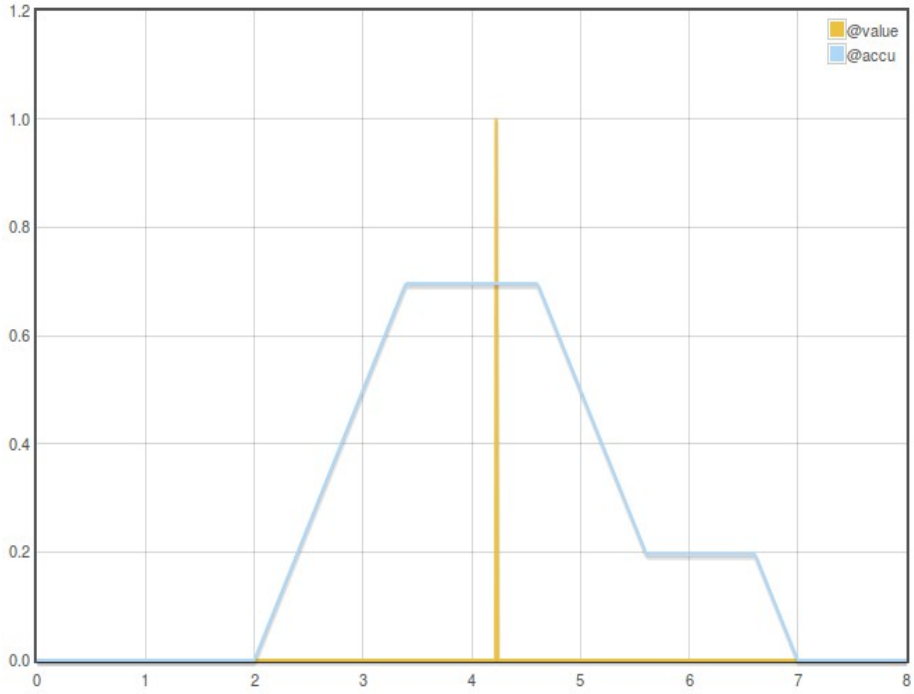
Różnice te nieznacznie wpływają na rezultat działania całego systemu.

II.5.3. 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) [11]

	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. II.5.3./1 Wyostrzanie metodą COG

II.6. Przykład wnioskowania rozmytego – zraszanie trawnika

II.6.1. 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

Ze względu na to, 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 powyżej 20%
około to +- 4%

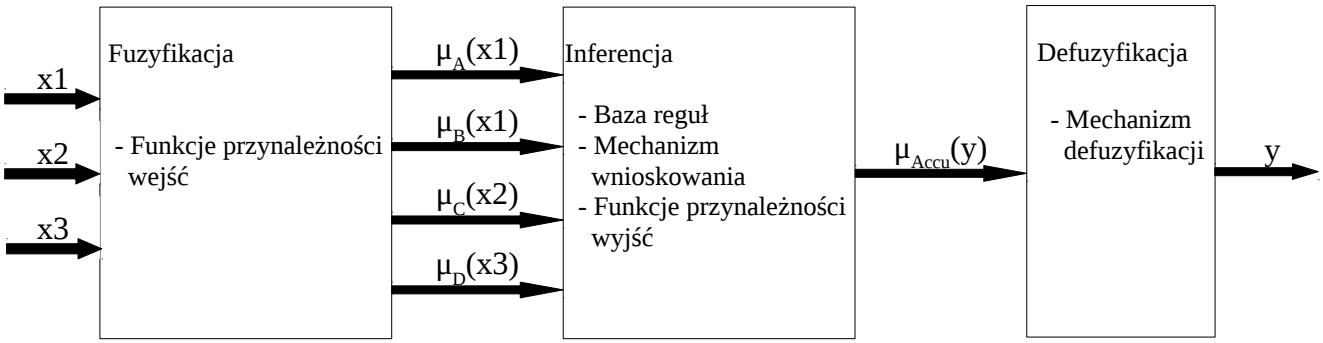
zraszanie słabe to 20%
zraszanie średnie to 40%
zraszanie silne to 60%

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.

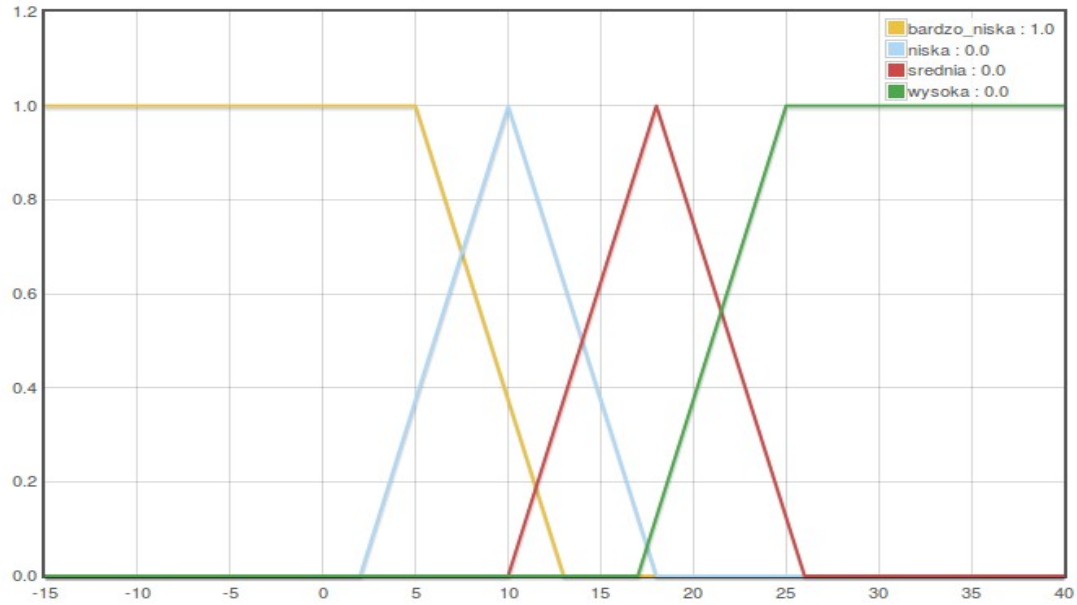
Z tak zebranymi danymi możemy przystąpić do zaprojektowania naszego systemu rozmytego.

II.6.2. 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:

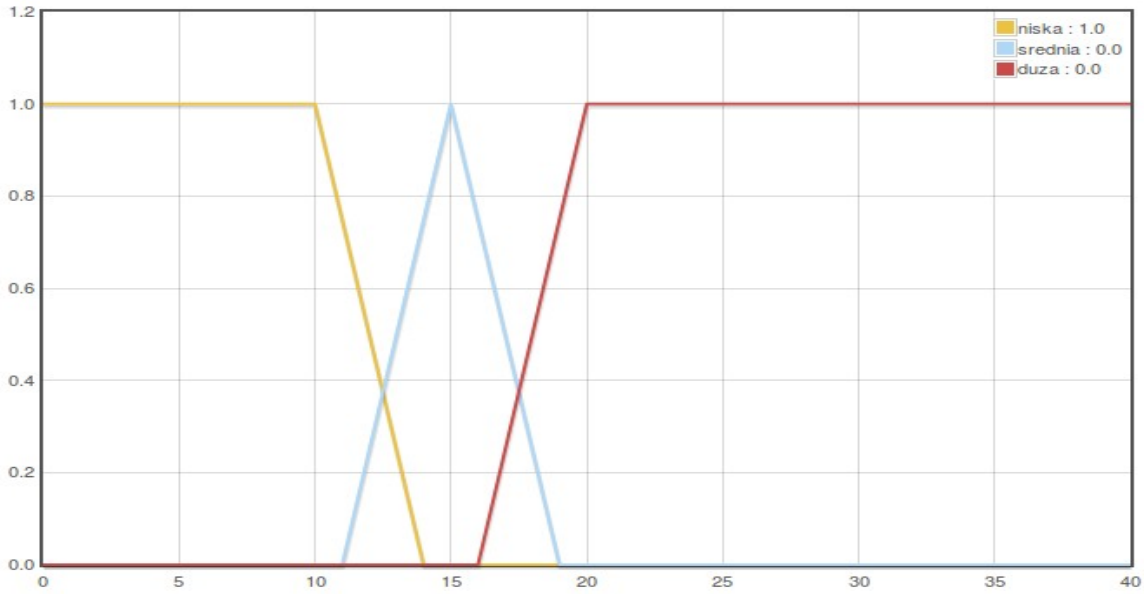


rys. II.6.2./1 Zraszanie trawnika, funkcje przynależności termów zmiennej temperatura

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

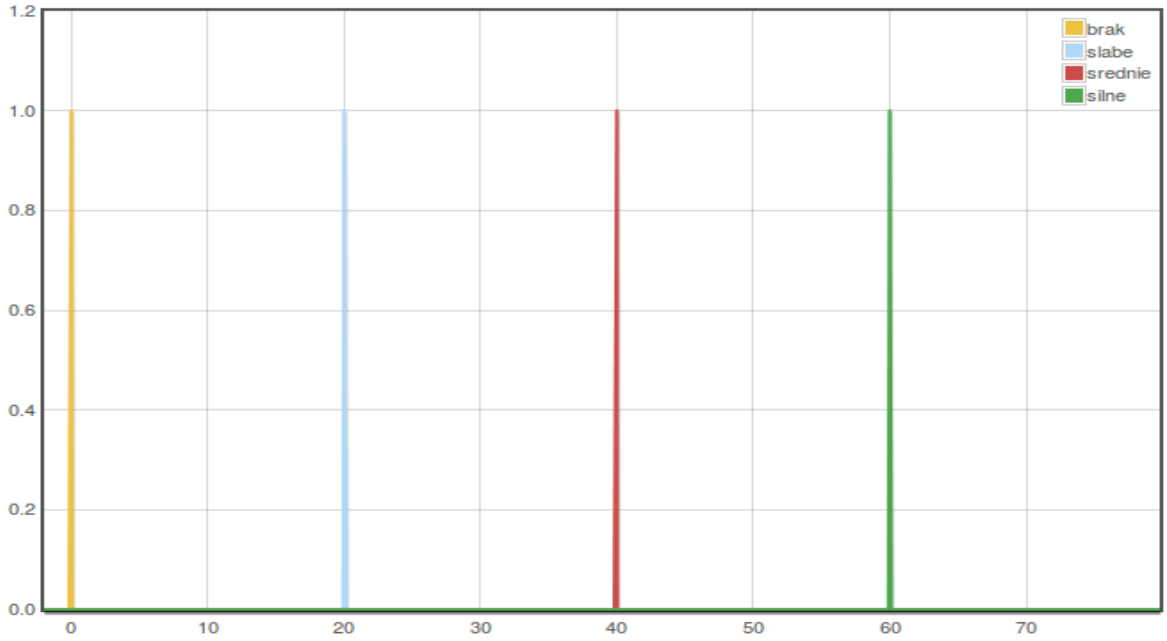
- TERM bardzo_niska := (-15,1) (5,1) (13,0);
- TERM niska := (2, 0) (10, 1) (18, 0);
- TERM srednia := (10, 0) (18, 1) (26,0);
- TERM wysoka := (17,0) (25, 1) (40, 1);

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



rys. II.6.2./2 Zraszanie trawnika, funkcje przynależności termów zmiennej wilgotność

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



rys. II.6.2./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.

II.6.3. Rozmywanie (fuzyfikacja)

Załóżmy stan zmiennych wejściowych :

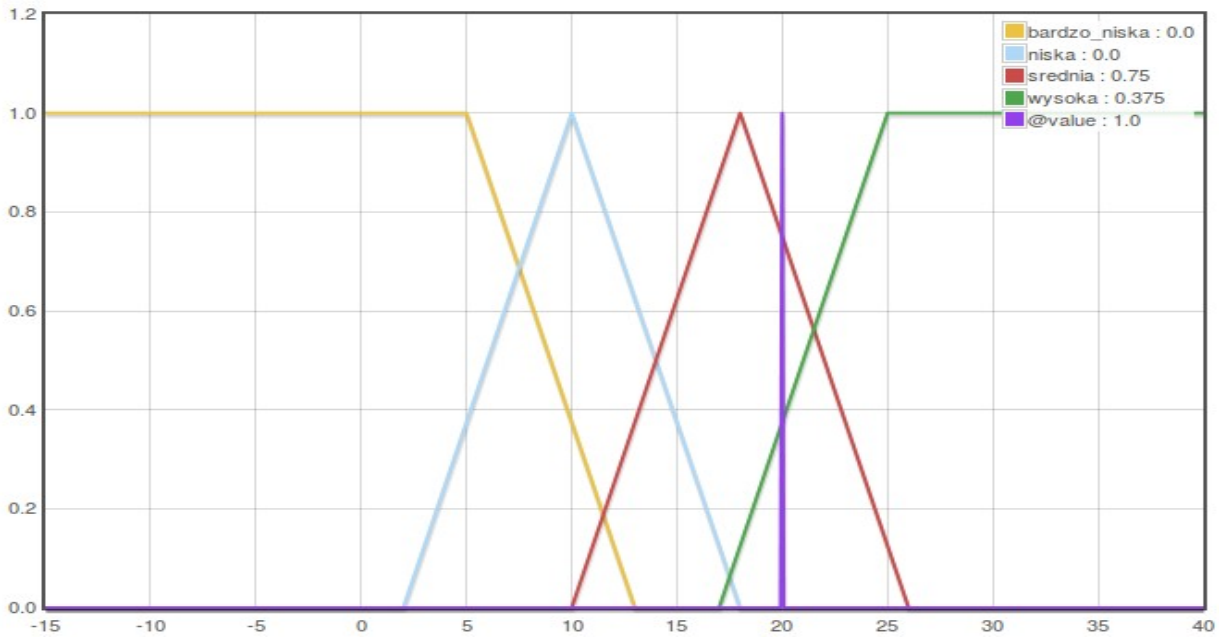
- temperatura: 20°C
- wilgotność: 16%

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 II.6.3./1 Funkcje aktywacji termów zmiennej wejściowej temperatura

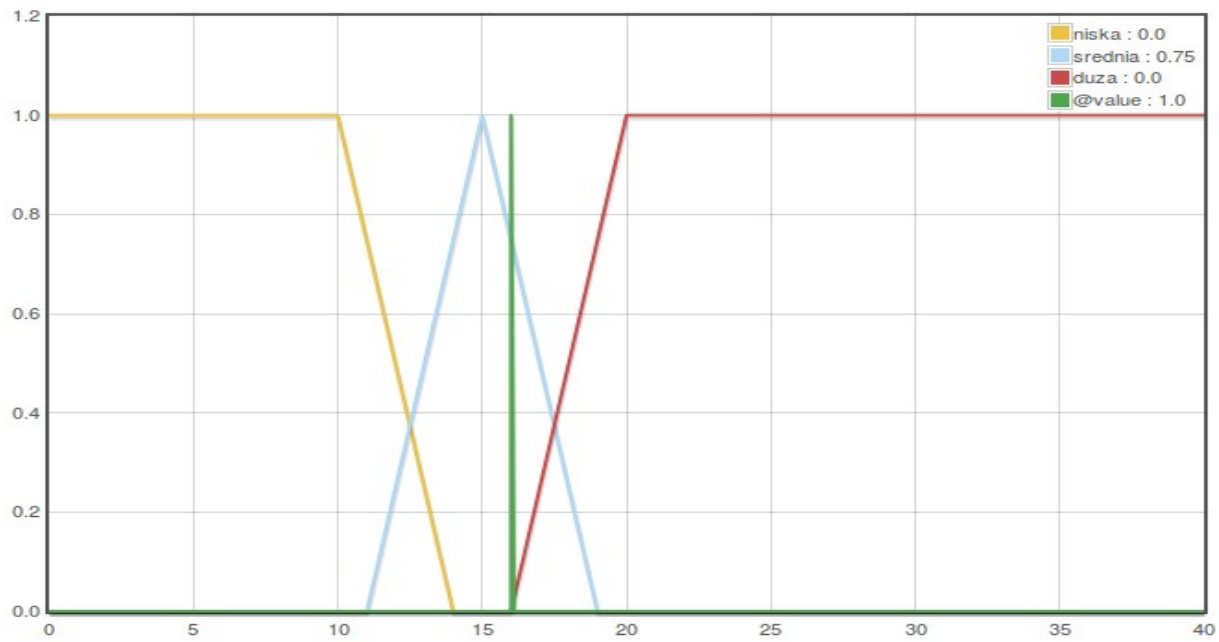
Fuzyfikujemy również zmienną wilgotność:

rys. II.6.3./2 Funkcje aktywacji termów zmiennej wejściowej wilgotność

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

Dla pozostałych termów wartość funkcji przynależności wynosi 0.

II.6.4. 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:

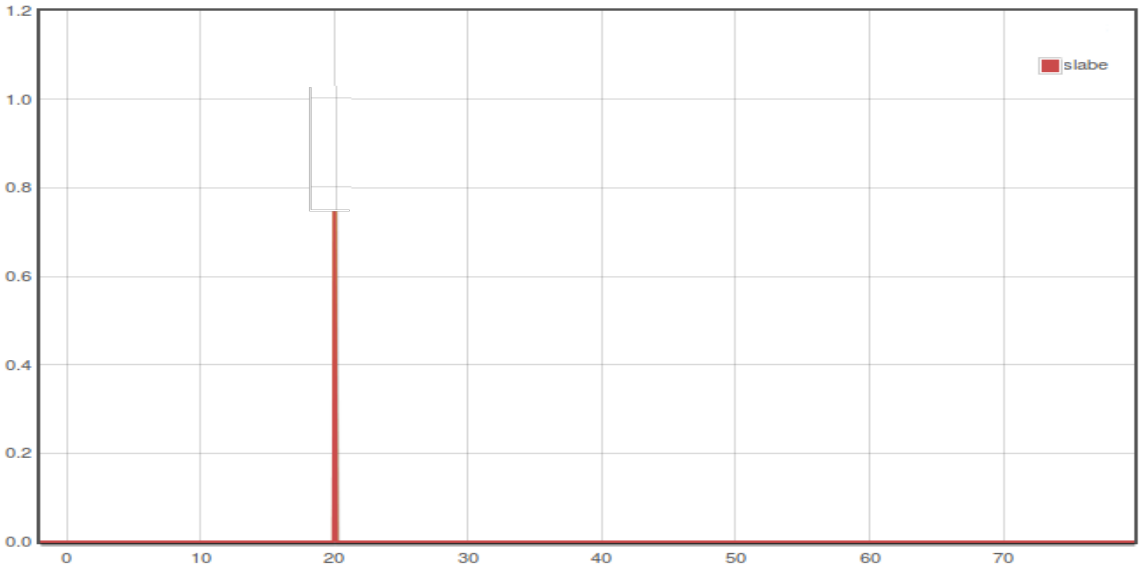
$$\mu_{RULE\,6}(16{,}20) = min(0.75\ ,\ 0.75)$$

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

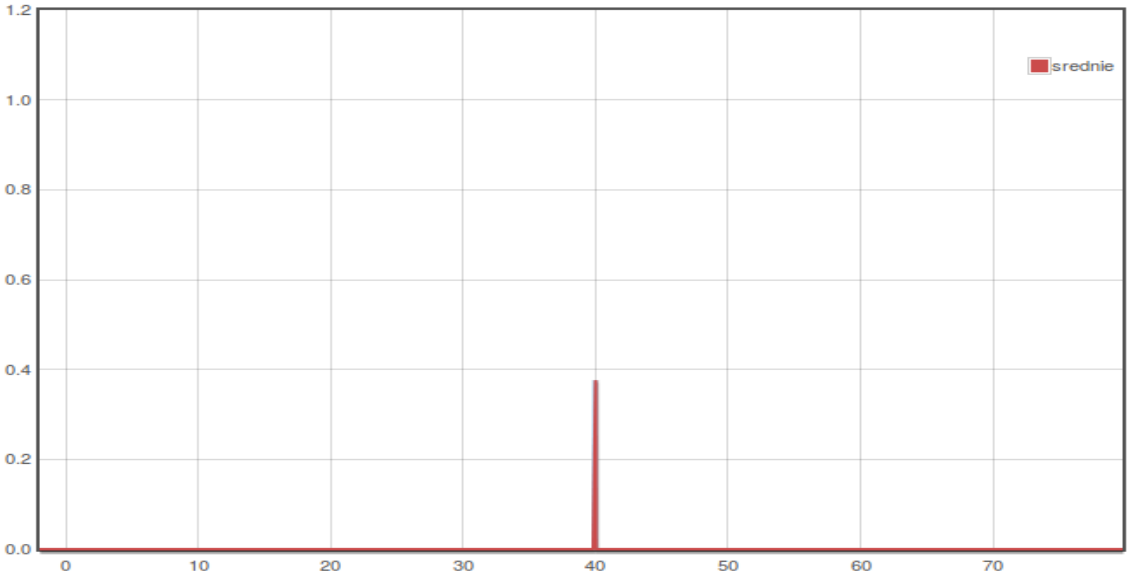
$$\mu_{RULE\,7}(16{,}20) = min(0.75\ ,\ 0.375)$$

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

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:

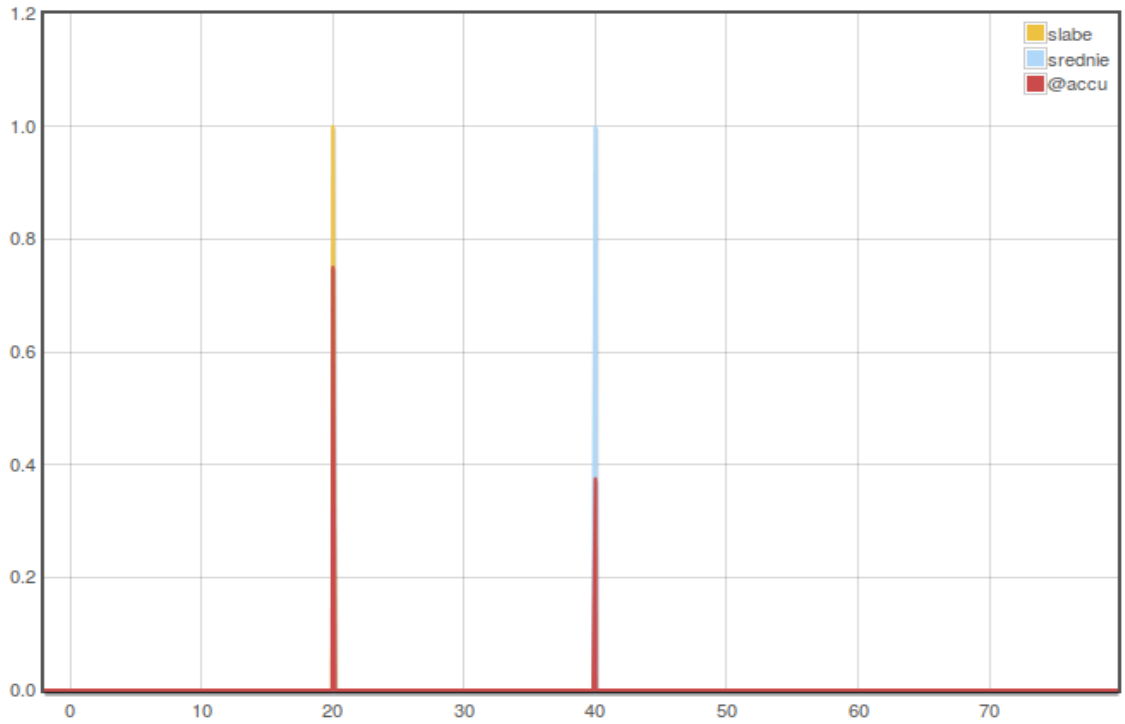


rys. II.6.4./1 Aktywacja dla predykatu zraszanie jest słabe



rys. II.6.4./2 Aktywacja dla predykatu zraszanie jest średnie

Na tak zebranych funkcjach przynależności przeprowadza się proces akumulacji zgodnie ze zdefiniowaną metodą (tutaj MAX). Prowadzi on do powstania następującego zbioru rozmytego:



rys. II.6.4./3 Akumulacja zebranych zbiorów rozmytych (kolor czerwony)

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.

II.6.5. 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.

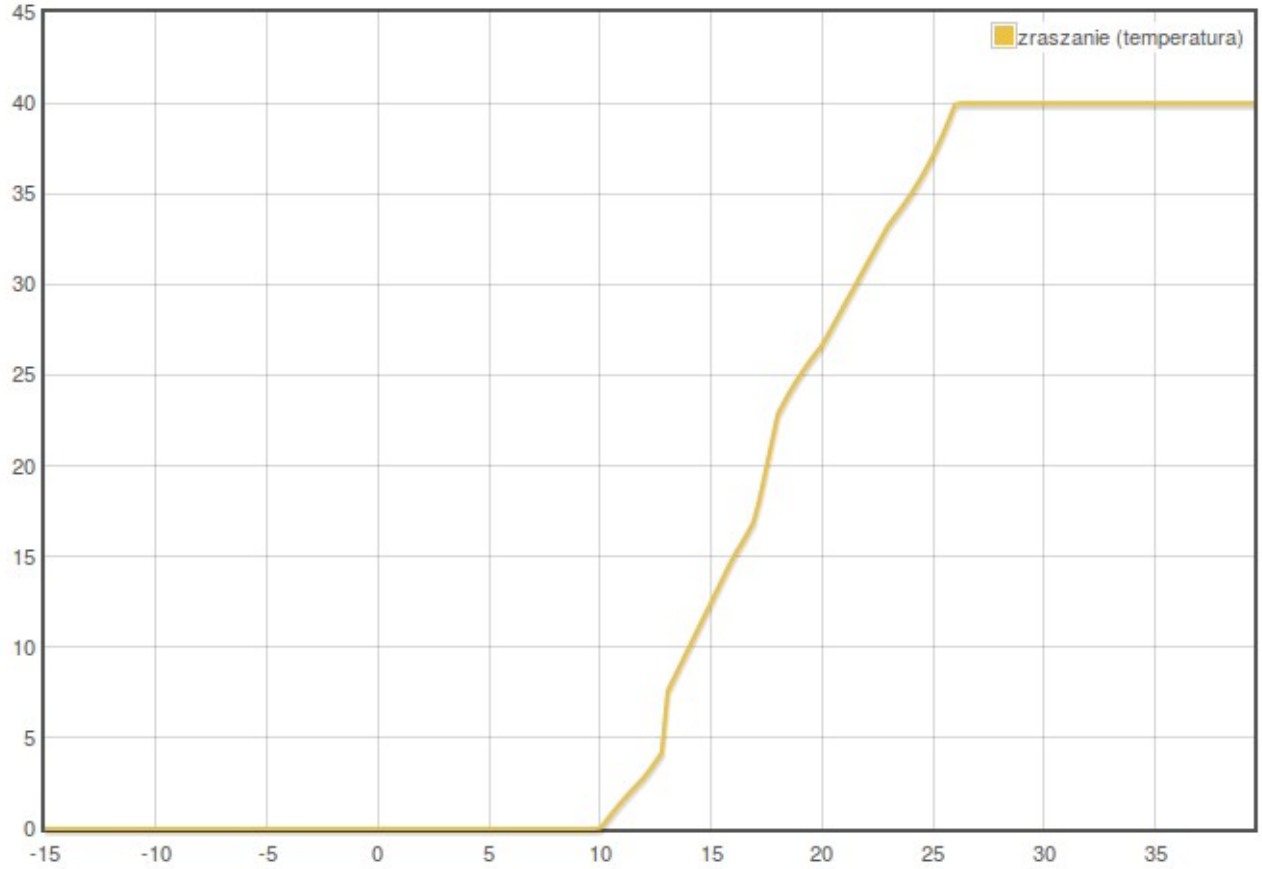
Wzór służący do obliczania tą metodą to: [11]

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

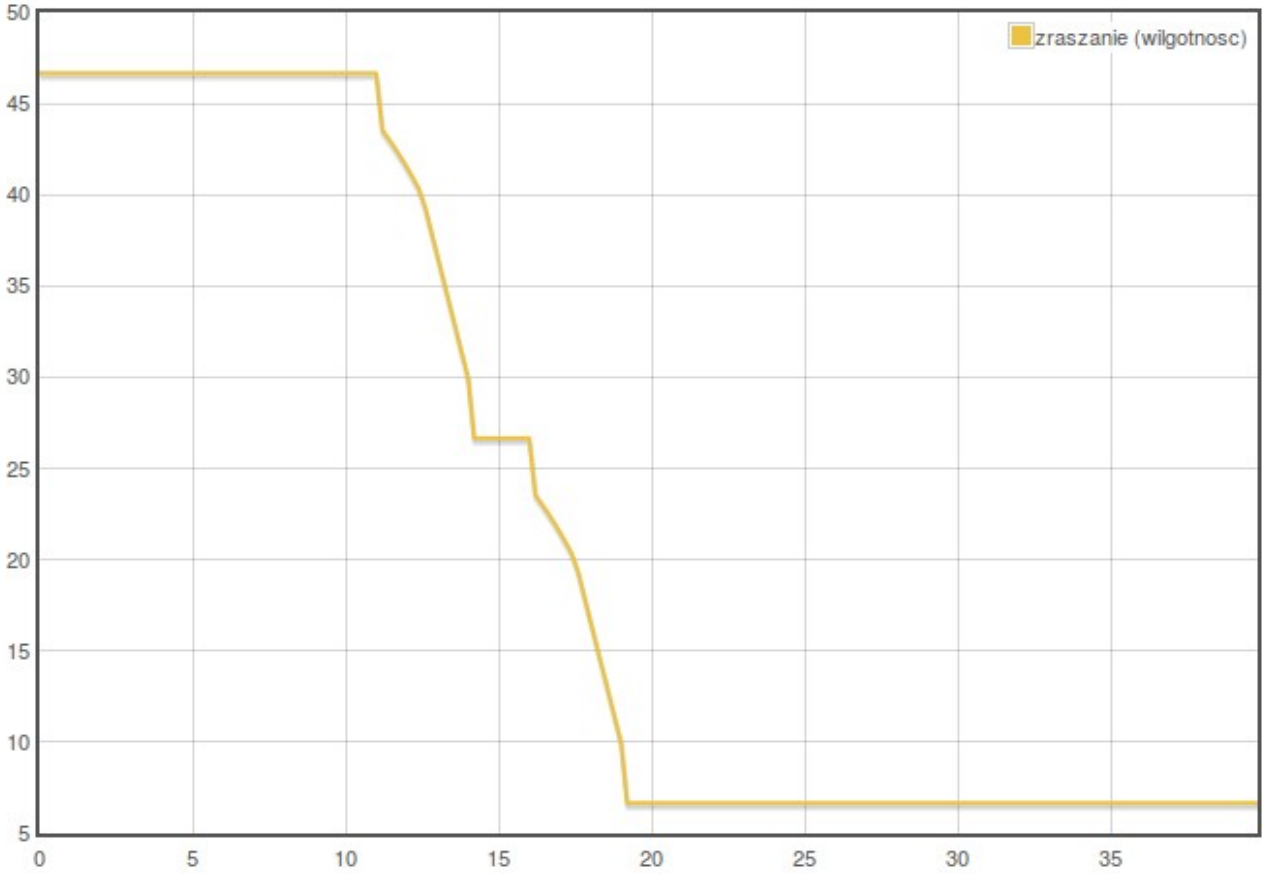
Obliczamy dla naszych danych:

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

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 tego, że mamy do czynienia z płaszczyzną w przestrzeni, następne wykresy będą dotyczyły stanu wyjściowego z poprzedniego wnioskowania:



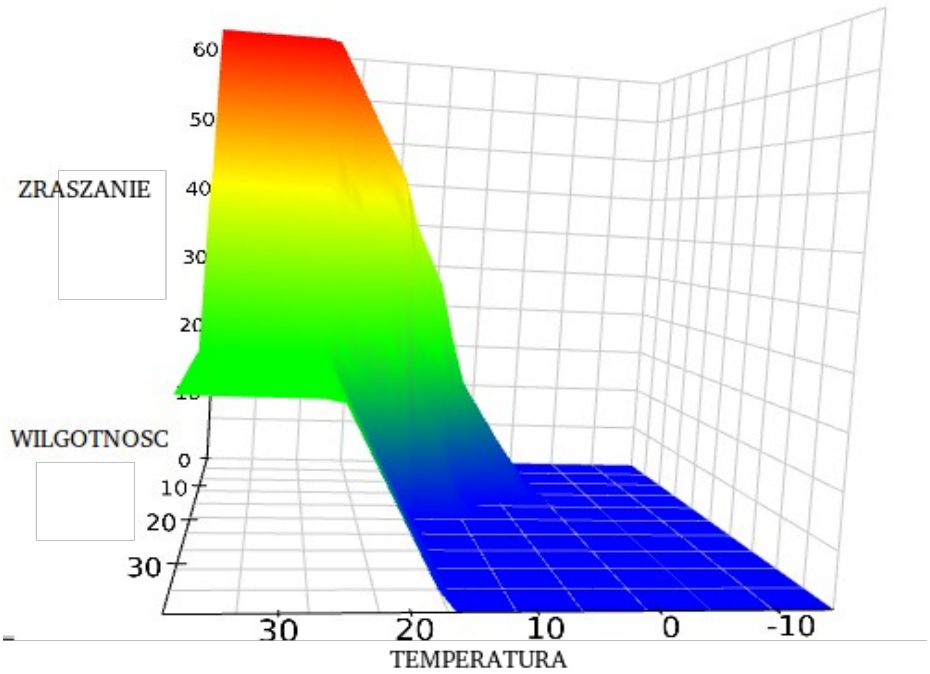
rys. II.6.5./1 Zraszanie w funkcji wilgotności dla temperatury 20°C



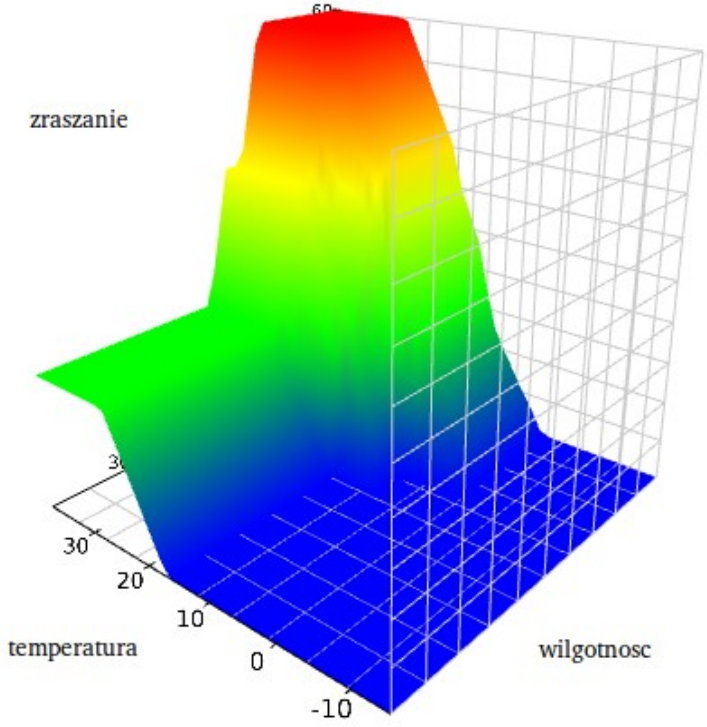
rys. II.6.5./2 Zraszanie w funkcji temperatury dla wilgotności 16%

Całą złożoność opisanego modelu widać jednak dopiero na wykresie 3D przedstawiającym przestrzeń rozwiązań (model) opracowanego systemu rozmytego:

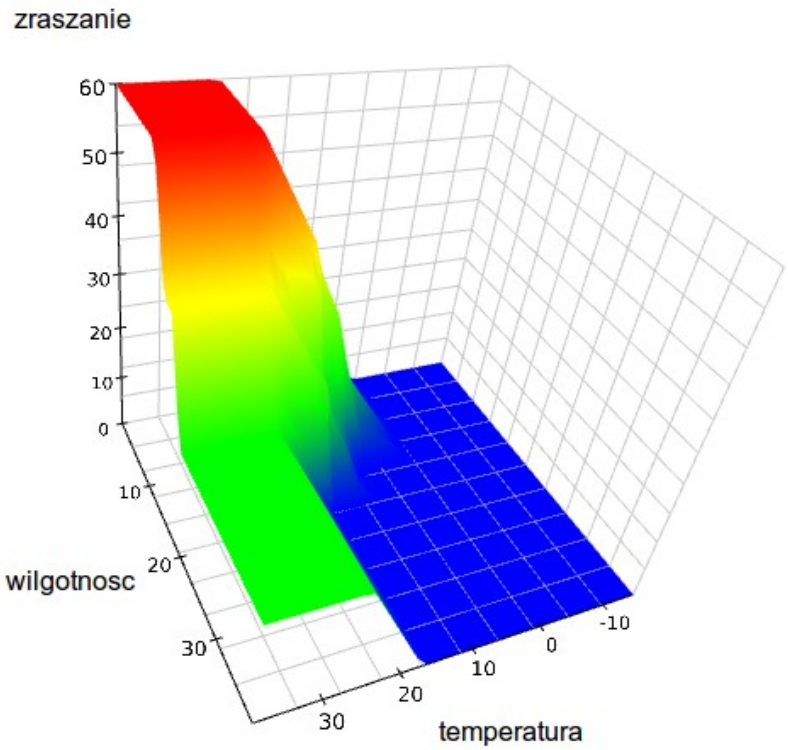
$$zraszanie(temperatura, wilgotnosc)$$



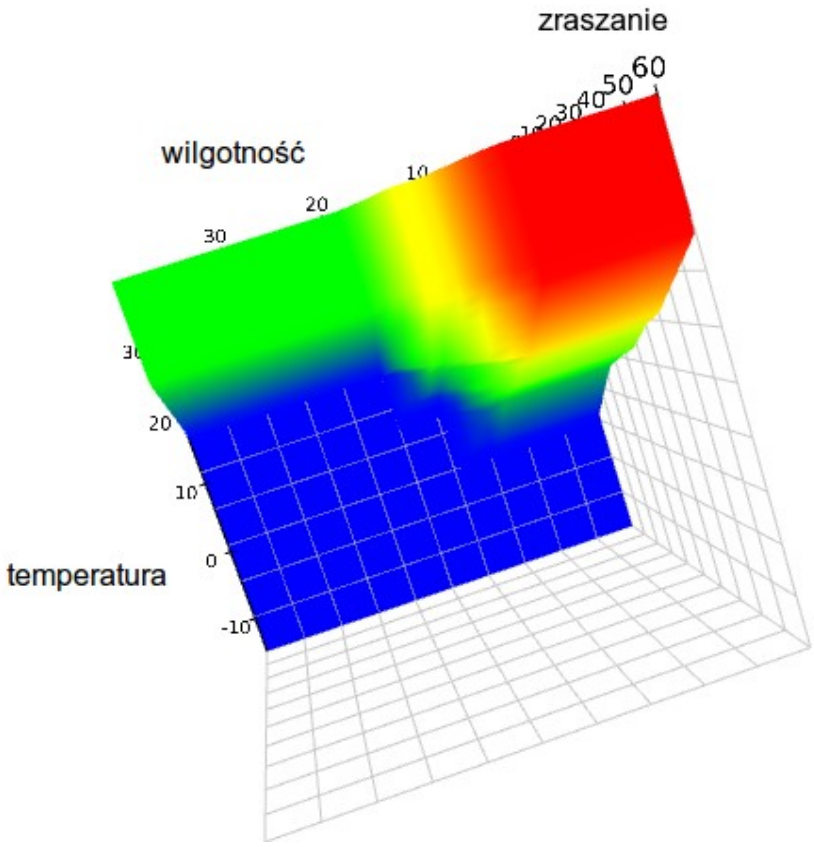
rys. II.6.5./3a Zraszanie w funkcji temperatury i wilgotności



rys. II.6.5./3b Zraszanie w funkcji temperatury i wilgotności



rys. II.6.5./3c Zraszanie w funkcji temperatury i wilgotności



rys. II.6.5./3d Zraszanie w funkcji temperatury i wilgotności

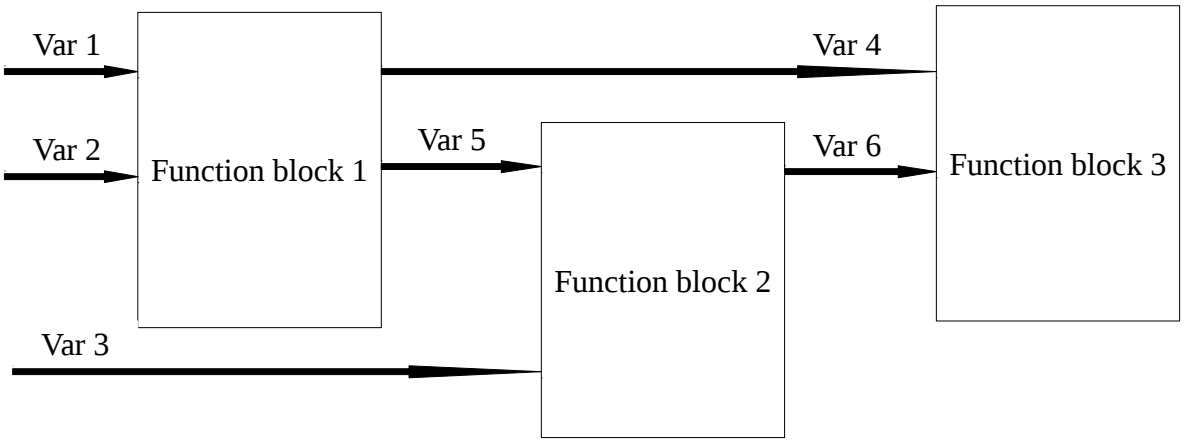
III. Opis składni języka Fuzzy Control Language

III.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. [11]
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 [12]. 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.

III.1.1. 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 III.1.1./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
I.1.2. 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.

III.1.2. 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 punktów wszystkich termów. Bloki te powtarzają się dla każdej zmiennej wejściowej.

III.1.3. 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 w 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

III.1.4. 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). Definicja 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

I.1.6. Przykład

Przedstawienie rozwiązania problemu ogrodnika z rozdziału pierwszego w postaci skryptu FCL.
Kod skryptu :

```
FUNCTION_BLOCK Fuzzy_FB
      VAR_INPUT
            wilgotnosc: REAL;
            temperatura : REAL;
      END_VAR
      VAR_OUTPUT
            zraszanie : REAL;
      END_VAR
    FUZZIFY temperatura
            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);
      END_FUZZIFY
      FUZZIFY wilgotnosc
            TERM niska := (0, 1) (10, 1) (14, 0);
            TERM srednia := (11, 0) (15, 1) (19,0);
            TERM duza :=  (16,0) (20, 1) (40, 1);
      END_FUZZIFY

      DEFUZZIFY zraszanie
          TERM min := (-2,0);
          TERM brak := 0;
            TERM slabe := 20;
            TERM srednie := 40;
            TERM silne := 60;
            TERM max := (80,0);
            ACCU : MAX;
            METHOD : COS;
            DEFAULT := -.2;
      END_DEFUZZIFY

      RULEBLOCK No1
          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;
            AND : MIN;
      END_RULEBLOCK
END_FUNCTION_BLOCK
```

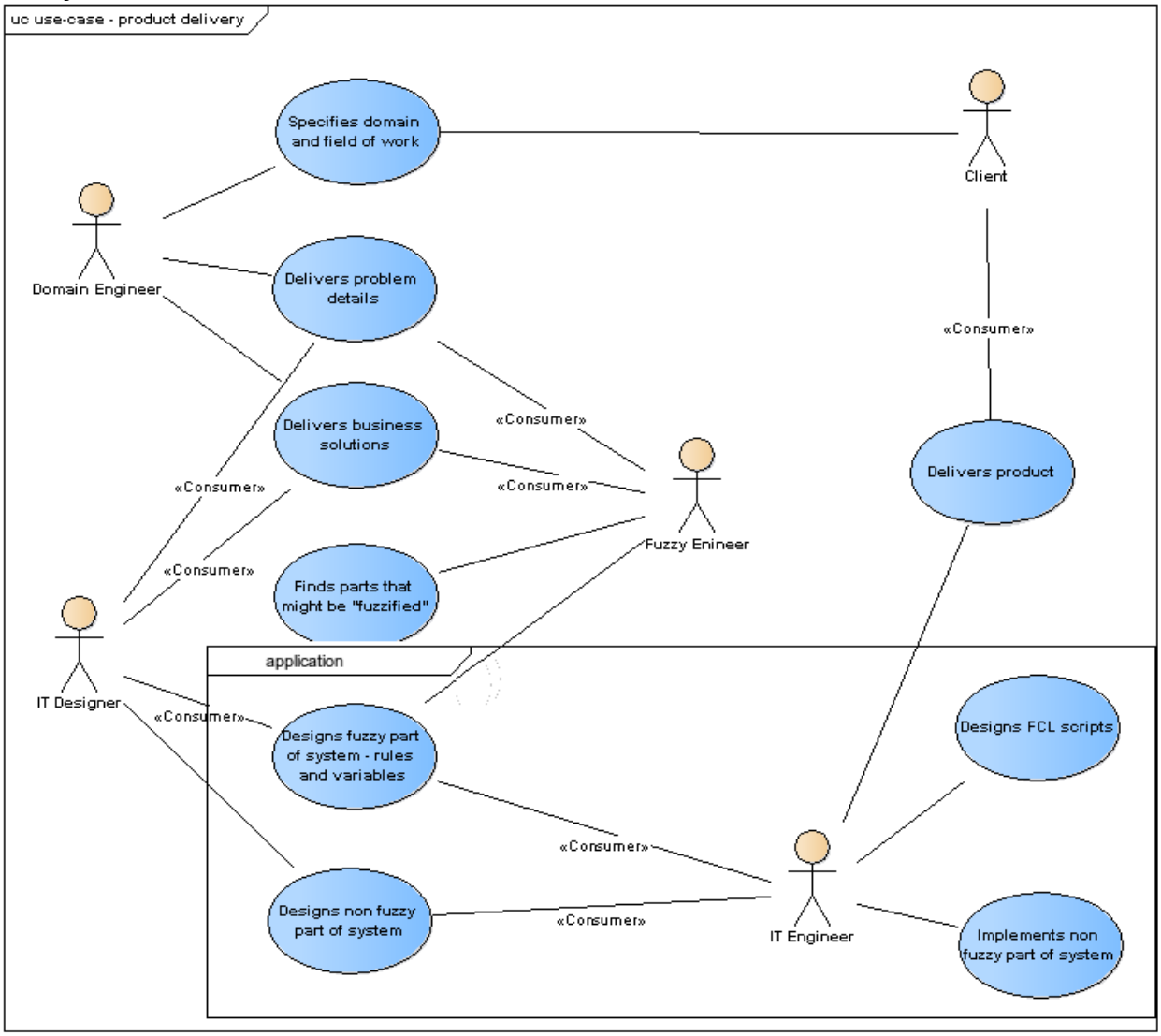
Wcięcia w kodzie nie są obowiązkowe w tym języku, poprawiają jednak czytelność kodu. Skrypt ten jest bardzo krótki w porównaniu do aplikacji, w dowolnym innym języku programowania, która miałaby realizować rozwiązanie tego problemu za pomocą logiki rozmytej. Reguły w tym przypadku są dość proste i krótkie. Można jednak tworzyć dowolnie długie reguły i skomplikowane wzory funkcji przynależności termów kreując przy tym skomplikowane systemy rozmyte do sterowania dowolną przestrzenią zmiennych. Taki bardziej złożony problem oraz jego rozwiązanie za pomocą FCL zostanie przedstawiony w ostatnim rozdziale tej pracy.

IV. Implementacja

Opis implementacji aplikacji został podzielony na dwie części. Część pierwsza to opis biblioteki FCL, która stanowi element bazowy aplikacji będącej przedmiotem tej pracy, może jednak być dołączona do dowolnej innej aplikacji. Część druga, to środowisko programistyczne, które ułatwia programowanie w FCL, uwidaczniając struktury wewnętrzne projektowanego sterownika rozmytego.

IV.1. Implementacja biblioteki

Biblioteka została opracowana z myślą wykorzystania jej w aplikacjach napisanych w języku JAVA. Stanowi ona część całości aplikacji będącej przedmiotem tej pracy. Odpowiada za prawidłowe modelowanie struktur na podstawie modelu Mamdaniego oraz parsowanie dokumentów napisanych w języku FCL. Wprowadzenie języka FCL w przypadku projektowania aplikacji, w której zastosowanie logiki rozmytej ma uzasadnienie, daje możliwość wydzielenia tej części ze standardowego przebiegu projektowania aplikacji. Przykładowy use-case modelowania takiej aplikacji:



rys. IV.1./1 Modelowanie aplikacji z wykorzystaniem FCL

Elementy, których realizację wspiera aplikacja, zostały ujęte w obszar application. Projektowanie i oprogramowanie elementów nie związanych z logiką FCL zostało wsparte możliwością podłączenia aplikacji do interfejsu restowego w przypadku, gdy FCL jest jeszcze w fazie projektowania. Umożliwia to bezpośrednią współpracę i wprowadzanie poprawek, których rezultat ma natychmiastowy efekt w projektowanej aplikacji, element istotny w popularnych zwinnych metodykach programowania.

Jak widać, w modelowanie takiej aplikacji można włączyć inżyniera logiki rozmytej, który może zaprojektować osobno część systemu, która daje się „zfuzyfikować”. Co ważniejsze utrzymanie takiego produktu – drobne dostrajanie aplikacji pod potrzeby klienta nie wymaga przebudowy całego modułu, jedynie skryptów FCL. Implementacja takiej aplikacji również zyskuje na czasie, ponieważ zamodelowanie od podstaw algorytmów odpowiadających za obliczenia jest bardzo czasochłonne.

IV.1.1. 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>);
```

IV.1.2. Wymagania systemowe

- Środowisko uruchomieniowe Java SE Runtime Enviroment w wersji minimum 8

IV.1.3. Wymagania funkcjonalne

Lista wymagań funkcjonalnych, które nałożyły się na powstanie biblioteki w obecnej formie:

ID	Opis	Priorytet	Harmonogram
FR1	System umożliwia podanie kodu w formie pliku	Wysoki	Release 1
FR2	System umożliwia podanie kodu w formie zmiennej tekstowej	Wysoki	Release 1
FR3	System umożliwia tworzenie zmiennych wejściowych	Wysoki	Release 1

FR4	System umożliwia tworzenie zmiennych wewnętrznych	Wysoki	Release 1
FR5	System umożliwia tworzenie zmiennych wyjściowych	Wysoki	Release 1
FR6	System pozwala na zdefiniowanie funkcji przynależności termów dla l-zmiennych	Wysoki	Release 1
FR7	System pozwala na zdefiniowanie funkcji przynależności termów dla r-zmiennych	Wysoki	Release 1
FR8	System pozwala na tworzenie termów singletonów	Wysoki	Release 1
FR9	System pozwala na zdefiniowanie wartości domyślnej zmiennej	Wysoki	Release 1
FR10	System pozwala na dodawanie nowych typów termów za pomocą dziedziczenia po klasie bazowej generatora termu i dodanie instancji do fabryki termów	Wysoki	Release 1
FR11	System pozwala na zdefiniowanie metody akumulacji r-zmiennej	Wysoki	Release 1
FR12	System pozwala na zdefiniowanie nowych metod akumulacji za pomocą dziedziczenia po klasie bazowej akumulacji i dodanie jej instancji do fabryki metod akumulacji w aplikacji	Wysoki	Release 1
FR13	System pozwala na wybranie metody wyostrzania dla każdej r-zmiennej	Wysoki	Release 1
FR14	System pozwala na zdefiniowanie nowych metod wyostrzania za pomocą dziedziczenia po klasie bazowej defuzyfikacji oraz dodanie nowej instancji do fabryki metod defuzyfikacji	Wysoki	Release 1
FR15	System umożliwia zdefiniowanie reguł	Wysoki	Release 1
FR16	System umożliwia ustawienie na stałe poziomu „obcięcia” funkcji przynależności termów r-zmiennych za pomocą operatora with	Wysoki	Release 1
FR17	System umożliwia wybór metod and/or dla każdego bloku reguł	Wysoki	Release 1
FR18	System umożliwia tworzenie nowych metod and/or za pomocą dziedziczenia po klasie bazowej metod oraz dodanie instancji do fabryki metod	Wysoki	Release 1
FR19	System umożliwia dodanie modyfikatorów rozmytych (bardzo, nieco, nie) do termów w regułach.	Wysoki	Release 1
FR20	System umożliwia tworzenie nowych modyfikatorów za pomocą dziedziczenia po klasie bazowej modyfikatora	Wysoki	Release 1

	oraz dodanie wpisu do fabryki modyfikatorów.		

IV.1.4. Moduły

Biblioteka dzieli się na następujące moduły, które będą omówione szczegółowo w podanych rozdziałach:

- Moduł główny research.fcl.library – rozdział IV.1.5
- Moduł środowiska research.fcl.library.enviroment – rozdział IV.1.6
- Moduł zmiennych blokowych research.fcl.library.variables – rozdział IV.1.7
- Moduł reguł research.fcl.library.rules – rozdział IV.1.8
- Moduł parsera research.fcl.library.parser – rozdział IV.1.9
- Moduł bloku funkcji research.fcl.library.functionblock – rozdział IV.1.10
- Moduł termów research.fcl.library.terms – rozdział IV.1.11
- Moduł akumulacji research.fcl.library.accumulation – rozdział IV.1.12
- Moduł spójników research.fcl.library.andmethods – rozdział IV.1.13
- Moduł defuzyfikacji research.fcl.library.defuzzification – rozdział IV.1.14

IV.1.5. Moduł główny biblioteki

IV.1.5.1) Struktura

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

rys. IV.1.5.1)/1 Sygnatura klasy Application

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

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

IV.1.5.2) Fabryki

Klasa posiada również settery do fabryk wszystkich „ruchomych” komponentów aplikacji, co pozwala na łatwą modyfikację zachowania aplikacji na poszczególne fragmenty skryptu FCL. Należą do nich:

- Fabryka metod defuzyfikacji – metoda *get/set DefuzzificationMethodsFactory*
- Fabryka metod akumulacji – metoda *get/set AccumulationMethodsFactory*
- Fabryka t-norm/s-nrom dla operatorów and i or – *metoda get/set AndMethodsFactory*
- Fabryka termów – *metoda get/set TermFactory*
- Fabryka modyfikatorów rozmytych *(not/very/fairly) get/set ModifierFactory*
- Fabryka akcji dla części przyczynowej reguł – *metoda getActionFactory*

- Fabryka efektów dla części skutkowej reguł – *metoda getEffectFactory*
- Fabryka tłumacząca tekst na reguły *getRuleFactory*

IV.1.5.3) Rozszerzalność

Ostatnie trzy fabryki, z uwagi na wysoką złożoność, nie udostępniają setterów. Aby je zmodyfikować należy odziedziczyć po klasie aplikacji oraz parsera np.:

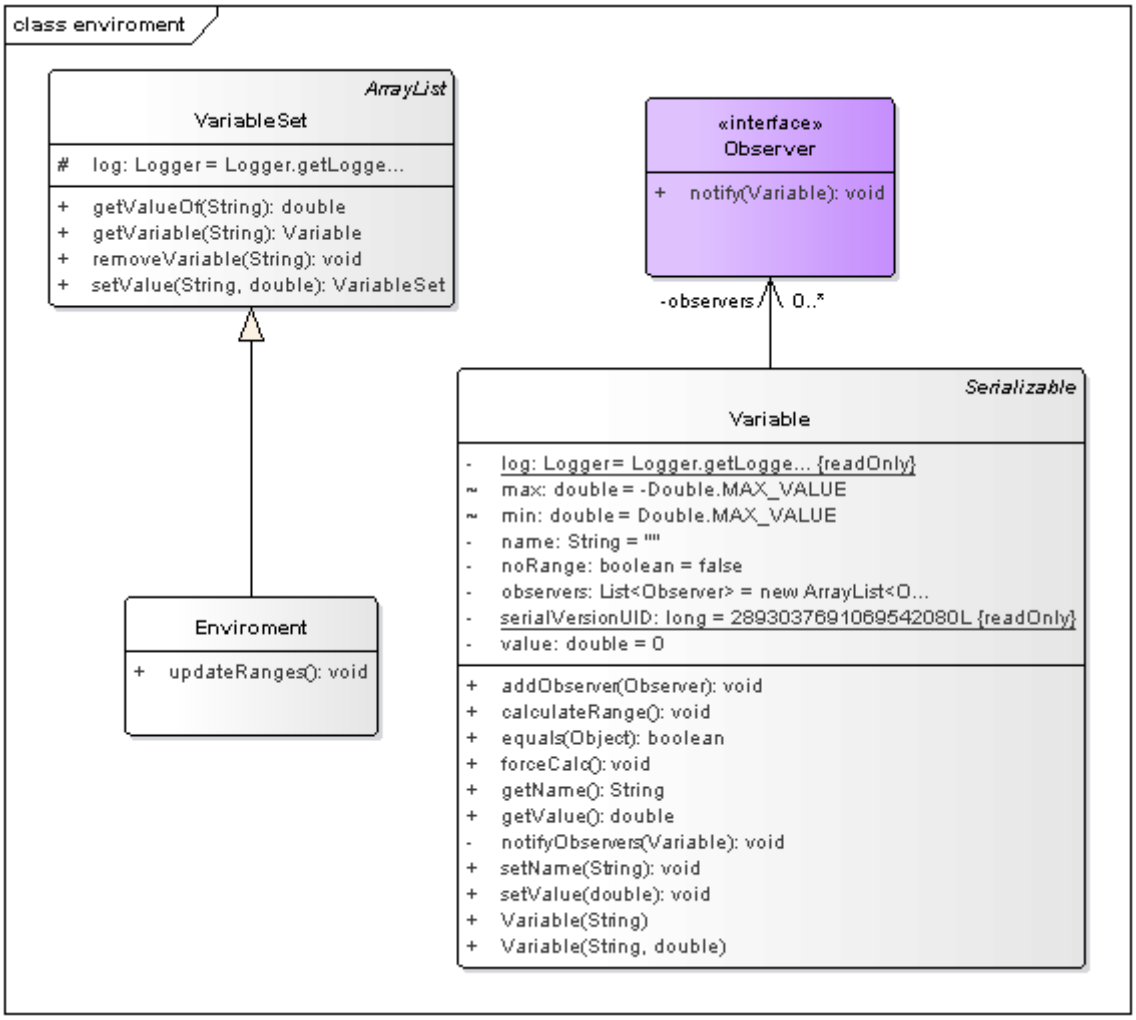
```
//tworzymy nową fabrykę reguł modyfikując potrzebne elementy
public class MyRuleFactory extends DefaultRuleFactory {
      //add new term generators here
}
//tworzymy nową klasę aplikacji dodają  getter dla nowej fabryki reguł
public class MyApplication extends Application {
      public RuleFactory getRuleFactory(Ruleblock ruleblock) {
            return new MyRuleFactory(ruleblock);
      }
}
//nadpisujemy kreatora instancji aplikacji w parserze
public class MyParser extends Parser {
      public Application createApplication () {
            return new MyApplication ();
      }
}
```

Tak utworzony parser będzie używał nowej fabryki reguł do mapowania skryptów FCL na instancję aplikacji.

IV.1.6. Moduł środowiska

Środowisko udostępnia dostęp RW do zmiennych w nim zawartych. Jest mapą <String>/<Variable>. Jego standardowym zachowaniem przy braku oczekiwanego elementu jest utworzenie nowego, dodanie go do mapy i zwrócenie użytkownikowi. Środowisko może istnieć niezależnie od samej aplikacji i taka sytuacja jest wskazana, gdy istnieje potrzeba zmodyfikowania i przeparsowania pliku FCL z jednoczesnym zachowaniem aktualnego stanu zmiennych.

IV.1.6.1) Struktura.



rys. IV.1.6.1)/1 Budowa modułu środowiska

Każda zmienna środowiskowa posiada zakres <min;max>, wykorzystywany przez kolejny opisywany moduł. Posiada również listę obserwatorów, którzy są informowani o każdej zmianie wartości zmiennej. Zabieg ten został zastosowany w celu rozdzielenia reprezentacji skryptu fcl od samych wartości zmiennych oraz umożliwienia różnym zmiennym skryptowym, np. wejściowej w jednym bloku i wyjściowej w innym o tej samej nazwie, na wskazywanie tej samej zmiennej środowiskowej.

IV.1.6.2) Wstrzykiwanie środowiska

Biblioteka umożliwia podmianę aktualnego stanu zmiennych w aplikacji. Fragment kodu, który wykonuje taką podmianę po przeparsowaniu nowej treści skryptu fcl:

```
Parser p = new Parser(body); //w body treść skryptu fcl
//pobieramy środowisko z dowolnego miejsca (w tym przypadku sesja www)
Enviroment env = (Enviroment) session.getAttribute("env");
p.parse();
if (env == null)
env = p.getApplication().getEnv();
p.setEnviroment(env);
```

IV.1.6.3) Fragmenty kodu

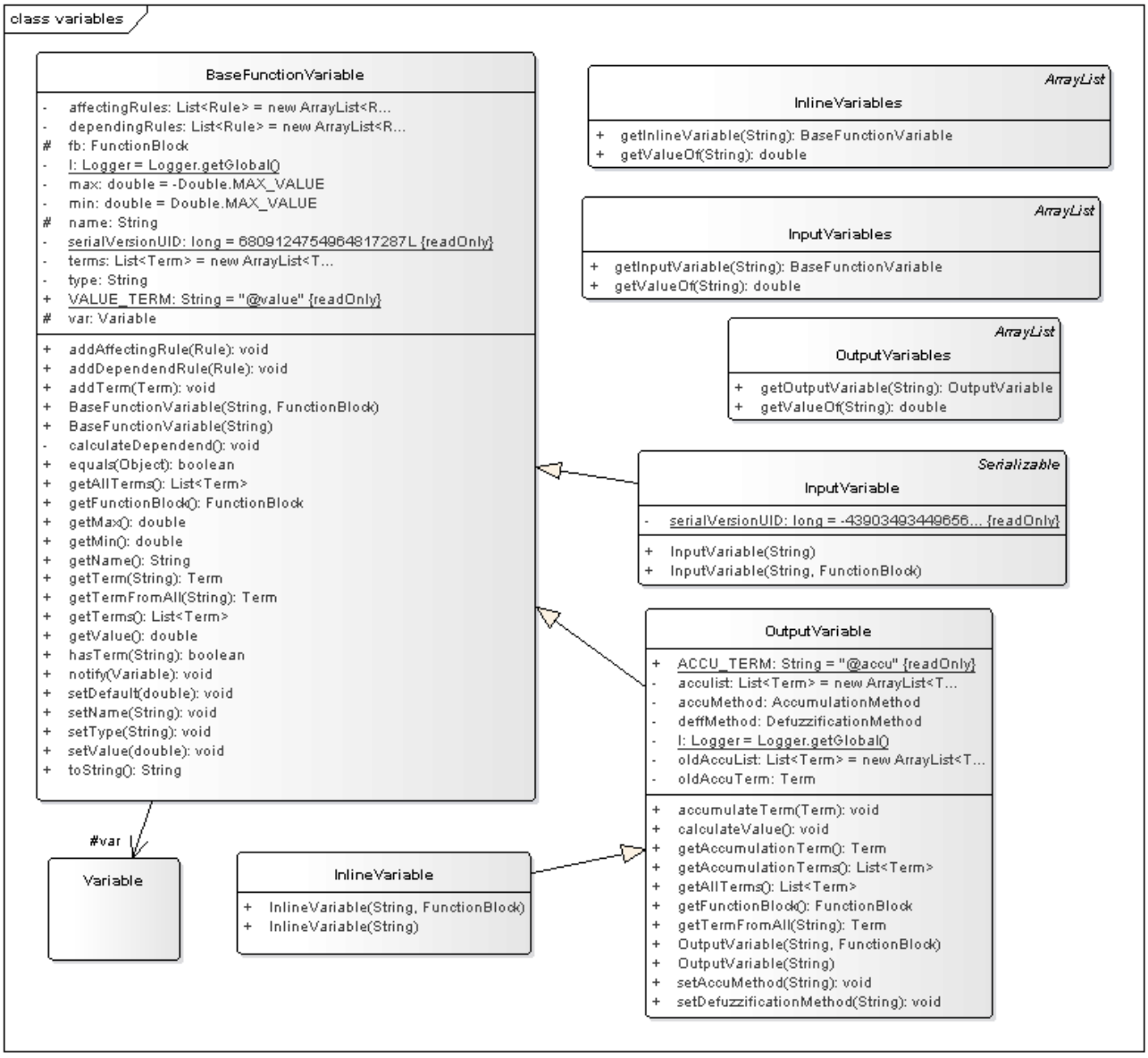
Przedstawiona poniżej implementacja zbioru zmiennych, po których dziedziczy środowisko, to przeróbka listy na mapę obiektów. Pozwala na wyciąganie zmiennych wg ich nazwy. Jeśli dana zmienna nie zostanie znaleziona w zbiorze, będzie ona utworzona i dodana do zbioru:

```
public class VariableSet extends ArrayList<Variable>{

	protected Logger log =
Logger.getLogger(this.getClass().toGenericString());

	public Variable getVariable (String name) {
		int index = -1;
		if ((index = this.indexOf(new Variable(name))) >= 0 ) {
			return this.get(index);
		}
		log.info ("Variable " + name + " does not exist, creating new.");
		Variable var = new Variable(name);
		this.add(var);
		return var;

	}
	public void removeVariable (String name) {
		int index = this.indexOf (new Variable (name));
		if (index < 0) return;
		this.remove (index);
	}
	public VariableSet setValue (String name, double value) {
		this.getVariable(name).setValue(value);
		return this;
	}
	public double getValueOf (String name) {
		int index = -1;
		if ((index = this.indexOf(new Variable(name))) >= 0 ) {
			return this.get(index).getValue();
		}
		log.info ("Variable " + name + " does not exist, creating new.");
		Variable var = new Variable(name);
		this.add(var);
		return var.getValue();
	}
}
```

IV.1.7. Moduł zmiennych blokowych

IV.1.7.1) Struktura

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.



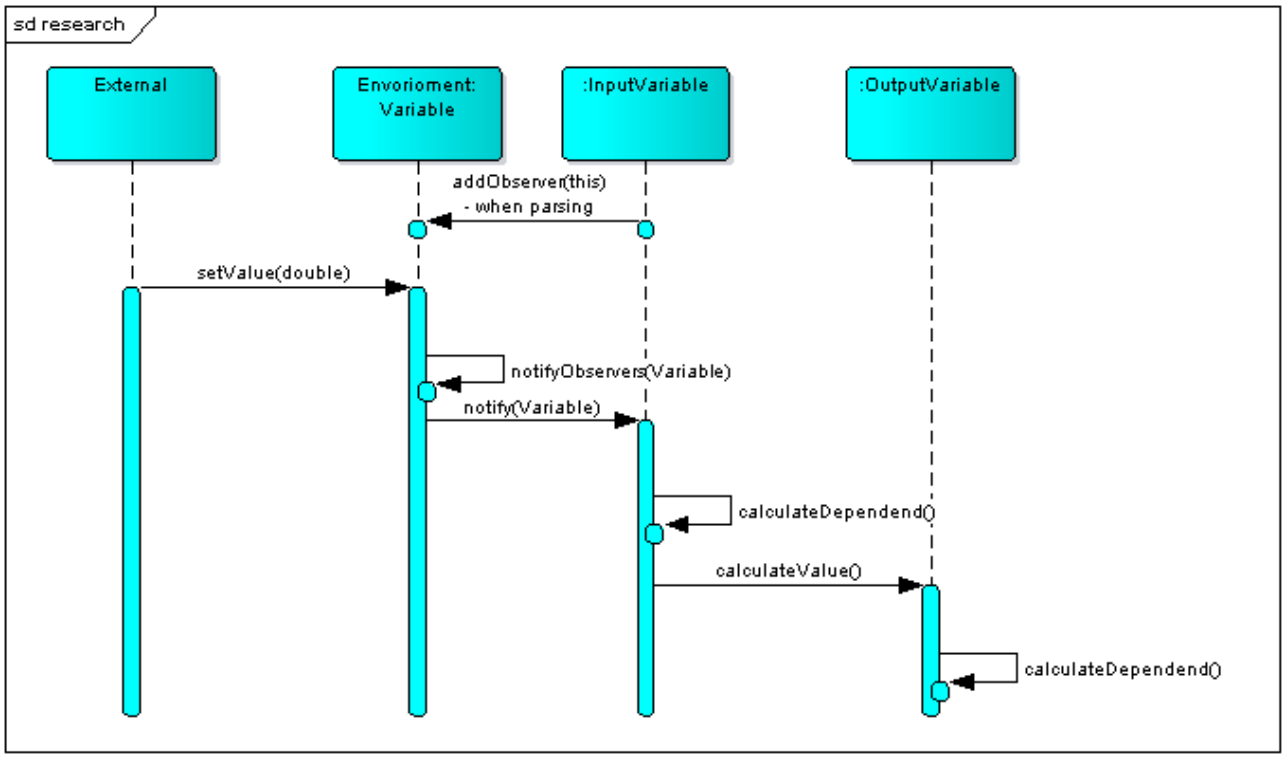
rys. IV.1.7.1)/1 Budowa modułu zmiennych blokowych

Istotne elementy klasy :

- 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 termami.

IV.1.7.2) Obliczanie wartości zmiennych

Obliczanie wartości zmiennych bazuje na wzorcu obserwatora. Wszystkie zmienne bloku funkcji nasłuchują na swoich odpowiednikach ze zmiennych środowiska. Sekwencja działania tego procesu wygląda następująco:



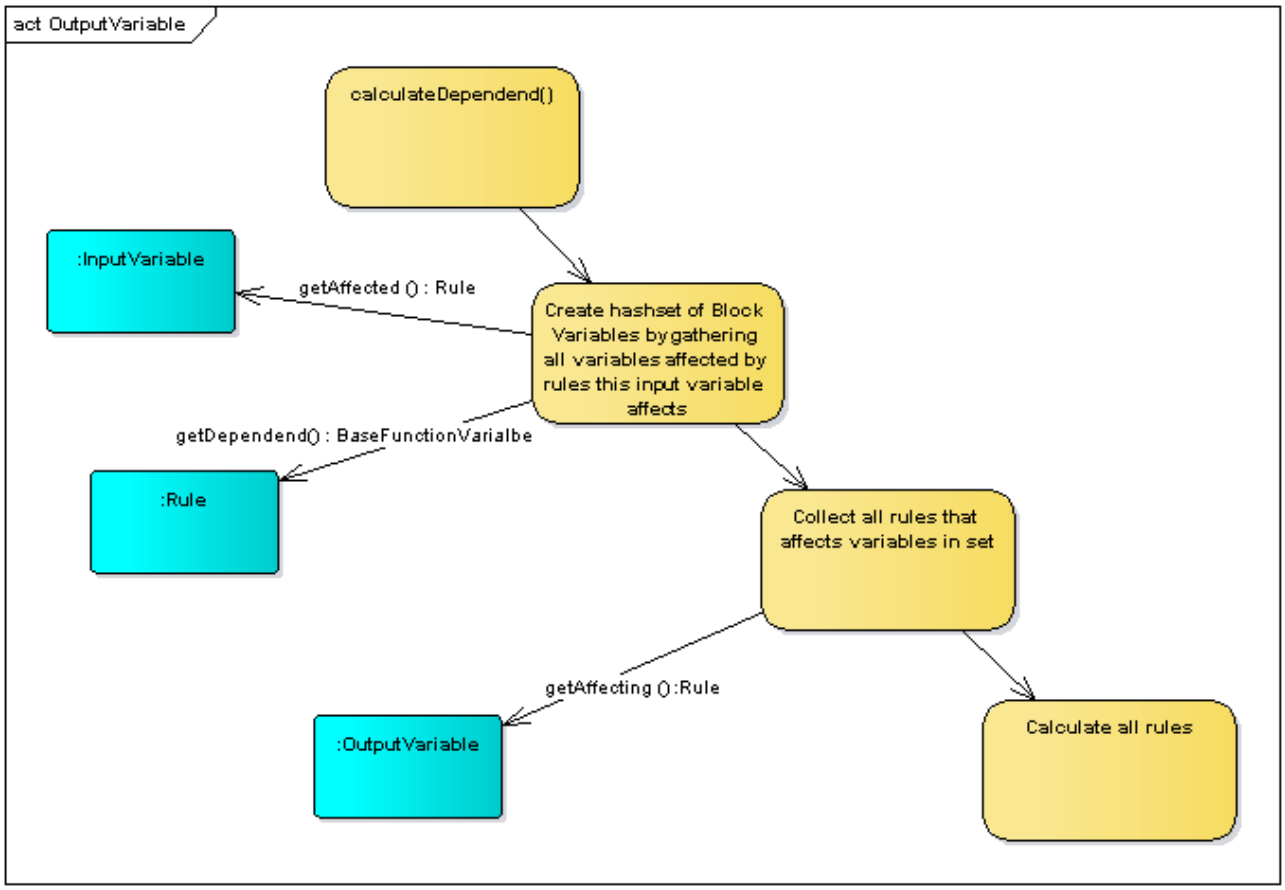
rys. IV.1.7.2)/1Diagram sekwencji notyfikacji zmiennych

Jak widać przeliczane są tylko wartości tych zmiennych wyjściowych, na które wpływ ma dana zmienna wejściowa. Takie podejście zoptymalizowało proces obliczania wartości zmiennych, ponieważ unika „pustych przebiegów”.

Proces obliczania wartości zmiennej wyjściowej podlega określonemu algorytmowi, który zapewnia przeliczenie wszystkich reguł, które wpływają na wszystkie zmienne wyjściowe, na które wpływa modyfikowana zmienna wejściowa. Wygląda to następująco:

1) Znajdź wszystkie reguły, na które wpływa modyfikowana zmienna wejściowa (referencje są dodawane automatycznie w czasie parsowania kodu FCL)
2) Zbierz wszystkie zmienne wyjściowe, które zmieniane są przez te reguły
3) Zbierz wszystkie reguły, w których uczestniczą te zmienne wyjściowe
4) Oblicz wszystkie reguły

Przedstawiony jest na poniższym diagramie:



rys. IV.1.7.2)/2 Diagram wyszukiwania reguł do przeliczenia.

IV.1.7.3) Fragmenty kodu

Przeliczanie wartości zmiennej:

```
public class BaseFunctionVariable implements Observer,Serializable {

	//(…)

	private void calculateDependend() {
		if (dependingRules.size() == 0) {
			l.info("No rules for variable " + this.name);
			return;
		}

		Set<BaseFunctionVariable> dependend = new
HashSet<BaseFunctionVariable>();
		l.info("Found rules for this variable, count: " +
dependingRules.size());


		//zbieranie wszystkich zmiennych wyjściowych zmienianych przez
		//reguły na które wpływ ma ta zmienna
		dependingRules.forEach(r -> dependend.addAll(r.getAffected()));
		l.info("All variables that depends on " + this.name + " count : "
				+ dependend.size());
		Set<Rule> rules = new HashSet<Rule>();


		//zbieranie wszystkich reguł zmieniających te zmienne
```

```
            //poniewaz do przeliczenia ich potrzebujemy wszystkich
            //składowych akumulacji
            dependend.forEach(v ->   rules.addAll(v.affectingRules));
            l.info("All rules to calculate : " + rules.size());

            //odpalanie wszystkich wymaganych reguł
            //zmienne wyjściowe kolekcjonują kolejne składowe akumulacji
            rules.forEach(r -> r.calculate());

            //przeliczanie wartości wszystkich zmiennych wyjściowych
            //na które wpływ mają znalezione reguły.
            dependend.forEach(v -> {
                  try {
                        ((OutputVariable) v).calculateValue();
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            });
      }

      //(…)

}
```

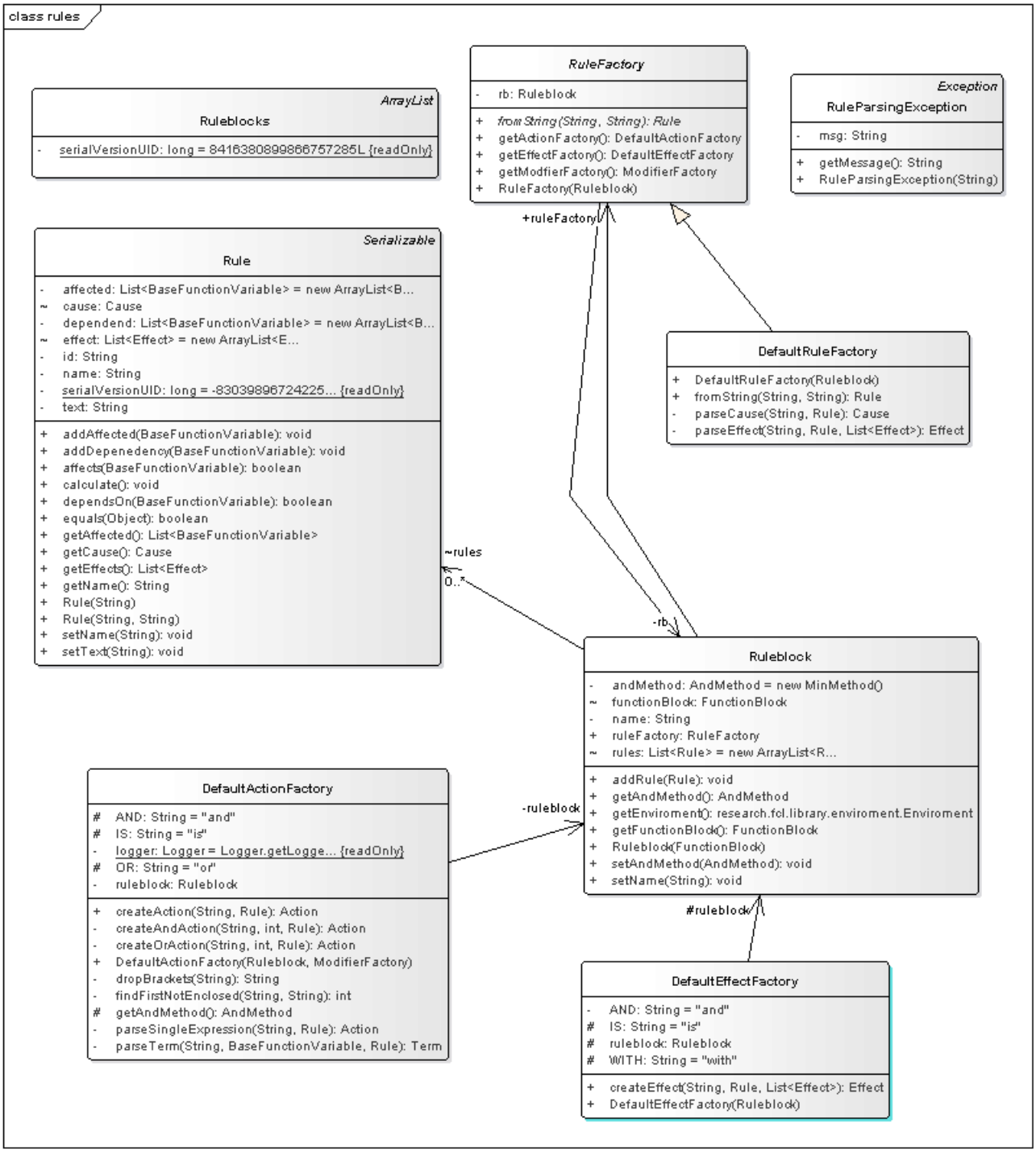
Ten fragment implementuje diagram wyszukiwania i przeliczania reguł z poprzedniego punktu.

IV.1.8. Moduł reguł

IV.1.8.1) Struktura

Moduł ten odpowiada za prawidłowe parsowanie treści reguł i przekonwertowanie ich do drzewa obiektów wykonujących właściwe operacje. W celu uproszczenia implementacji wydzielone zostały dodatkowo trzy pakiety wewnątrz modułu odpowiadające za:

- Przyczynę (`Cause`) – reprezentują elementy znajdujące się w części przesłanki reguł. Tworzy drzewo akcji (Action)
- Efekty (`Effect`) – reprezentują elementy znajdujące się w części konkluzji reguł.
- Modyfikatory (`Modifier`) – mogą być użyte w części przesłanki w celu modyfikacji funkcji przynależności termów zmiennych wejściowych. Na zewnątrz tych pakietów zostały wyprowadzone jedynie domyślne implementacje fabryk akcji i efektów.

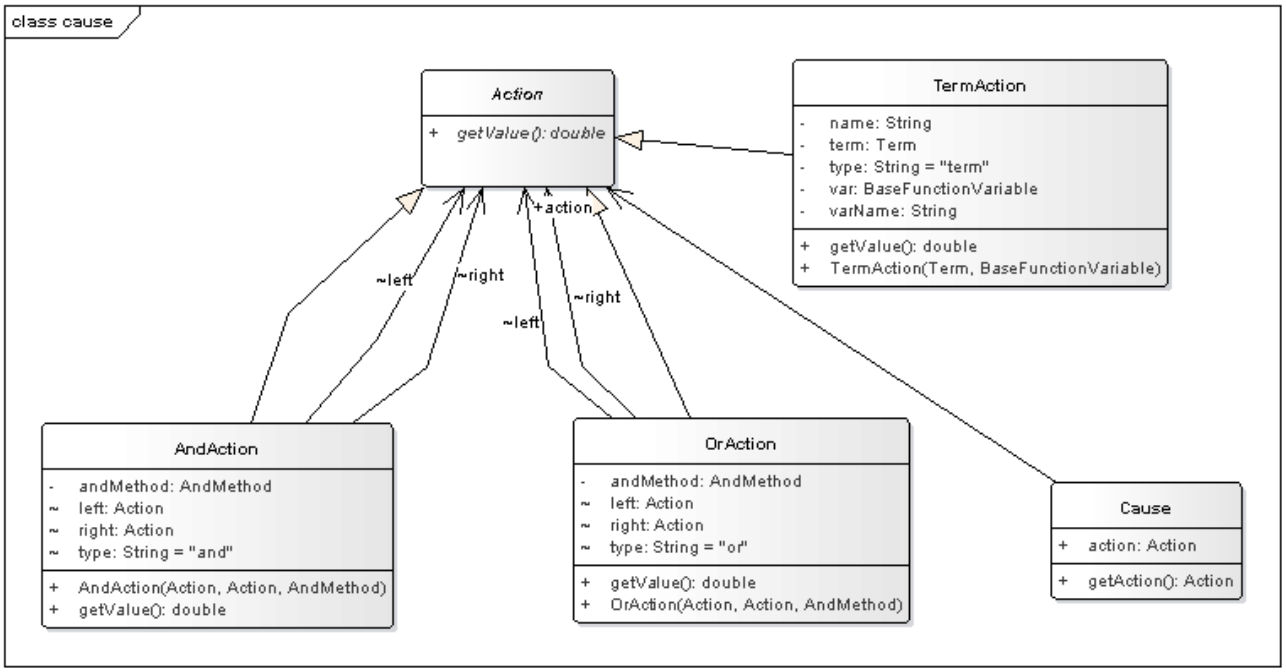


rys. IV.1.8.1)/1 Diagram klas modułu reguł.

Pakiet akcji zawiera trzy implementacje bazowej klasy Action:

- AndAction – wykonuje operację rozmytej koniunkcji (za pomocą odpowiedniej t-normy)
- OrAction – wykonuje operację alternatywy (za pomocą odpowiedniej s-normy)
- TermAction – wylicza wartość na podstawie zadanej funkcji przynależności, która może być uprzednio potraktowana modyfikatorem rozmytym.

Zasadniczo zadaniem akcji jest obliczenie wartości typu double. W tym celu każda akcja implementuje abstrakcyjną metodę klasy bazowej – `double getValue ()`. Struktura pakietu wygląda następująco:

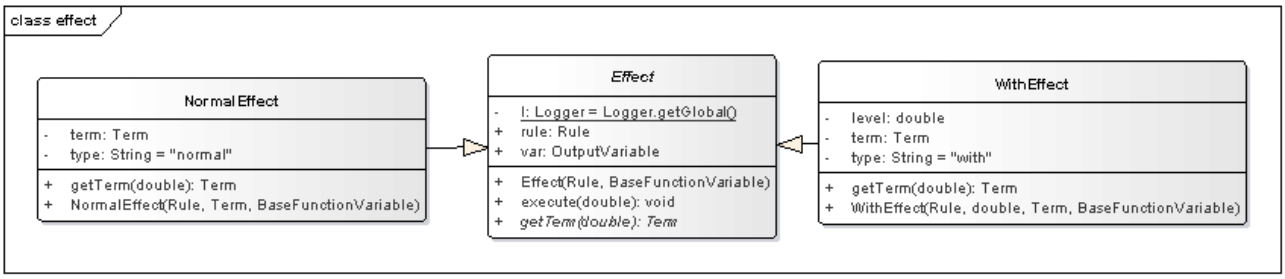


rys. IV.1.8.1)/2 Diagram klas pakietu przyczyny w module reguł

Klasa Cause odpowiada jedynie za dostęp do korzenia drzewa akcji.

Pakiet skutku zawiera dwie implementacje klasy abstrakcyjnej Effect:

- `WithEffect` – funkcja przynależności, w której określono poziom obcięcia.
- `NormalEffect` – funkcja przynależności, w której poziom obcięcia ustalany jest na podstawie wartości aktywacji.



rys. IV.1.8.1)/3 Diagram klas pakietu przyczyny w module reguł

Abstrakcyjna metoda getTerm odpowiada za zwrot prawidłowej funkcji przynależności na podstawie wartości aktywacji/operatora with utworzonej z funkcji przynależności predykatu konkluzji.

IV.1.8.2) Budowa reguły

W systemie rozmytym reguła składa się z dwóch elementów:

- przesłanka
- konkluzja

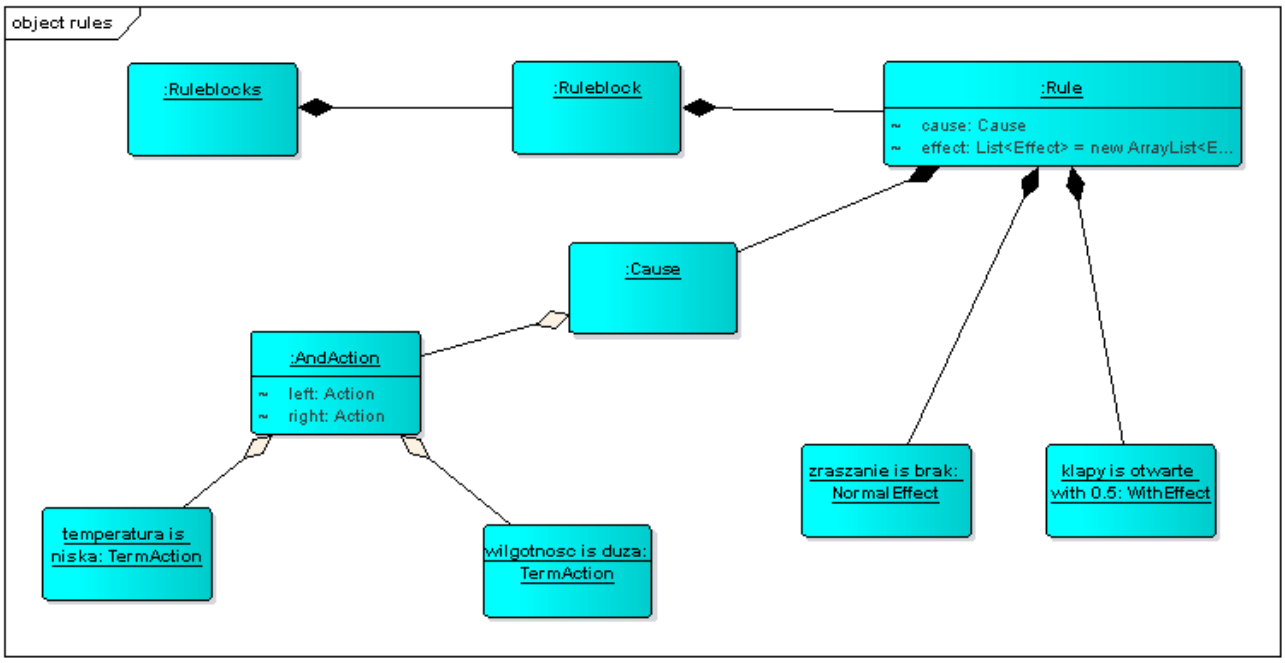
Przykładowa reguła :

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

Dzieli się na :

Przesłanka	Konkluzja
`Wilgotność is duza and temperatura is niska`	`Zraszanie is brak and klapy is otwarte with 0.5`

Przesłanka składa się z dwóch predykatów oraz operatora rozmytego and (t-normy). Konkluzja składa się również z dwóch predykatów, przy czym drugi z nich deklaruje poziom obcięcia funkcji przynależności termu, który mu odpowiada – 0.5. W pamięci aplikacji taka reguła będzie miała następującą reprezentację:



rys. IV.1.8.2)/1 Części składowe przykładowej reguły

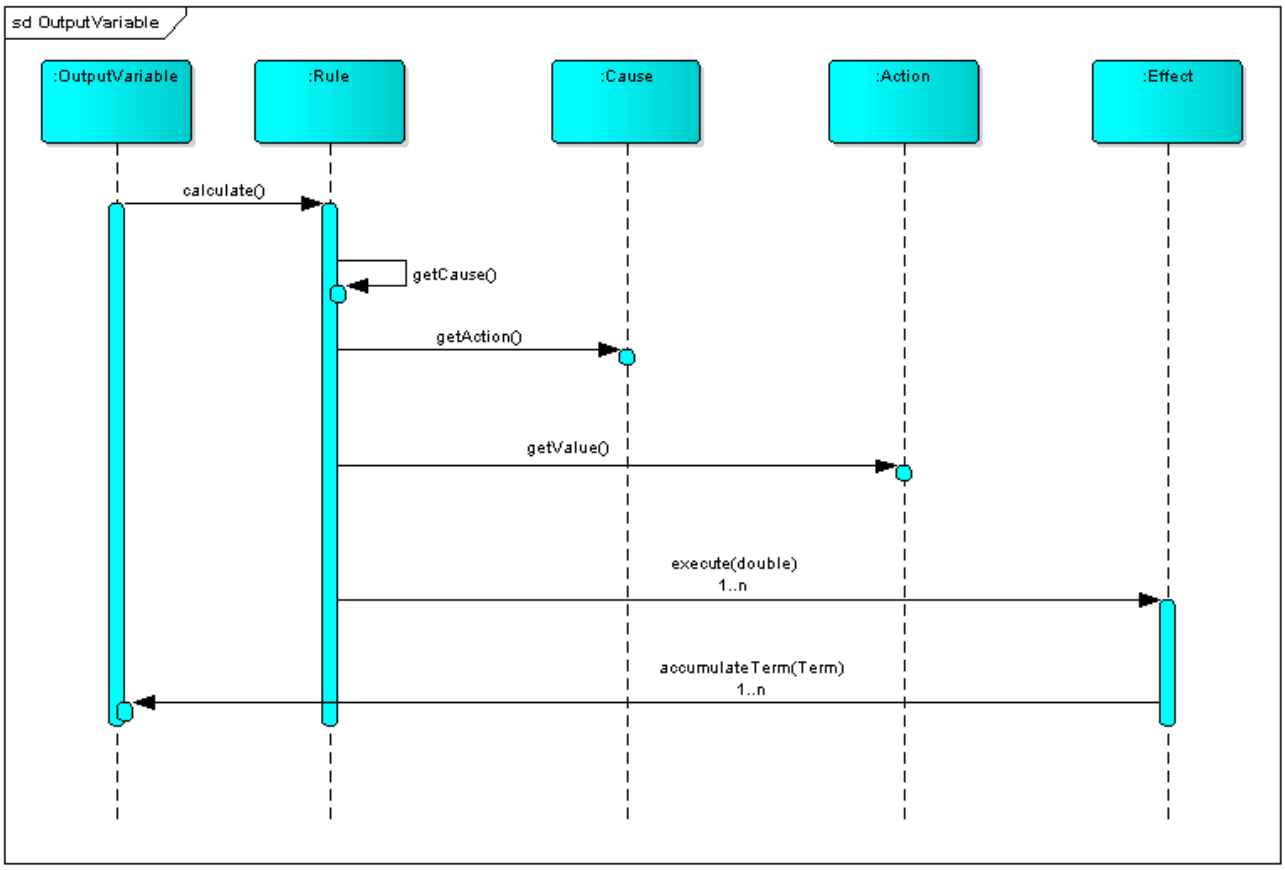
Kolejność odpalania akcji z sekcji przyczynowej reguły ma znaczenie (operator AND przed OR, nawiasy), dlatego z tej części reguły tworzone jest drzewo akcji. Operatory AND i OR wykorzystują do prawidłowego działania wybraną t/s-normę, która będzie omówiona w jednym z kolejnych rozdziałów. Sama akcja `TermAction` wylicza wartość funkcji aktywacji przypisanego do niej termu.

Kolejność odpalania efektów reguł, czyli tworzenie kolejnych funkcji przynależności termów konkluzji (obcinanie), nie ma znaczenia, dlatego zostały one zagregowane w jedną listę. Obowiązuje tzw. *lazy evaluation*, czyli najpierw tworzymy funkcje, a dopiero później na etapie akumulowania funkcji przynależności wyliczamy konkretne wartości.

Przeliczanie funkcji przynależności, która ma zostać zakumulowana w zmiennej wyjściowej odbywa się następująco:

1. Pobierana jest wartość (double) funkcji aktywacji głównej akcji – przeliczane są wszystkie akcje z drzewa akcji
2. Uruchamiane są elementy efektu reguły z obliczonym poziomem aktywacji z poprzedniego punktu
3. Następuje akumulacja funkcji przynależności (termu) dla każdego efektu.

Ten proces lepiej przedstawia poniższy diagram sekwencji :



rys. IV.1.8.2)/2 Diagram sekwencji odpalania reguły.

IV.1.8.3) Modyfikatory rozmyte

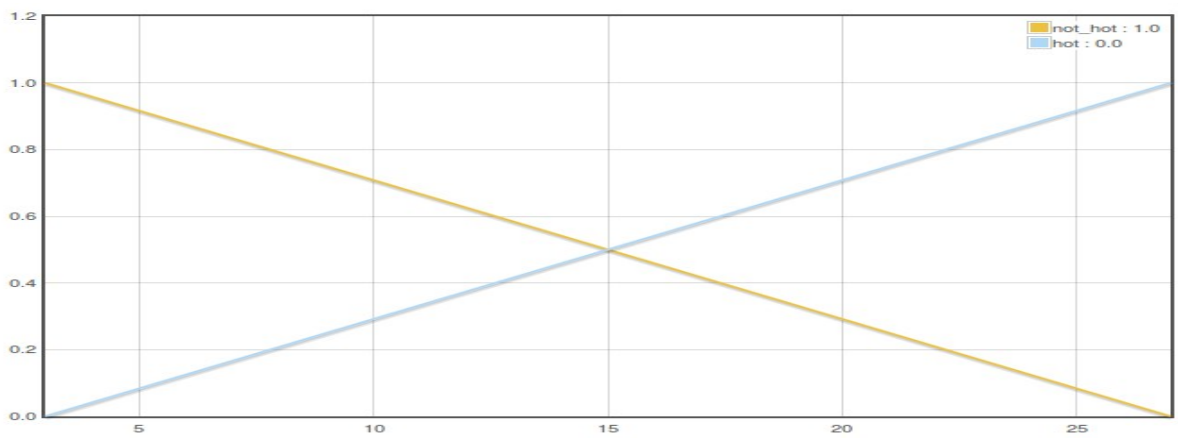
Modyfikatory rozmyte pozwalają na zmianę kształtu krzywej funkcji przynależności termu, do którego zostaną zastosowane. W aplikacji zdefiniowano trzy modyfikatory:

bardzo: $g(x)=f(x)^2$

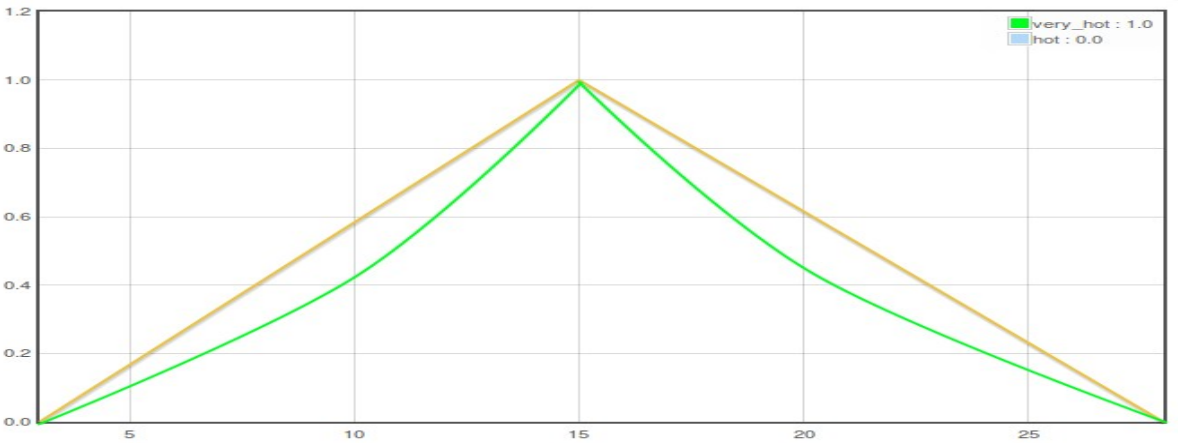
nieco: $g(x)=\sqrt{f(x)}$

nie: $g(x)=1-f(x)$

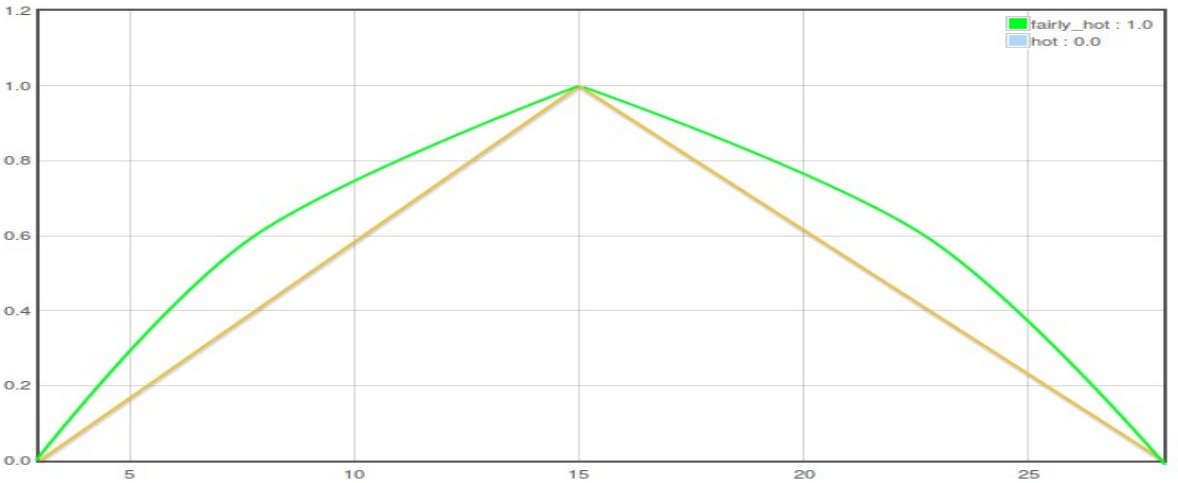
Znajdują one zastosowanie w części przesłanki reguł. Efekty działania tych modyfikacji przedstawiono na poniższych wykresach:



rys. IV.1.8.3)/1 Efekt działania modyfikatora not (żółty)

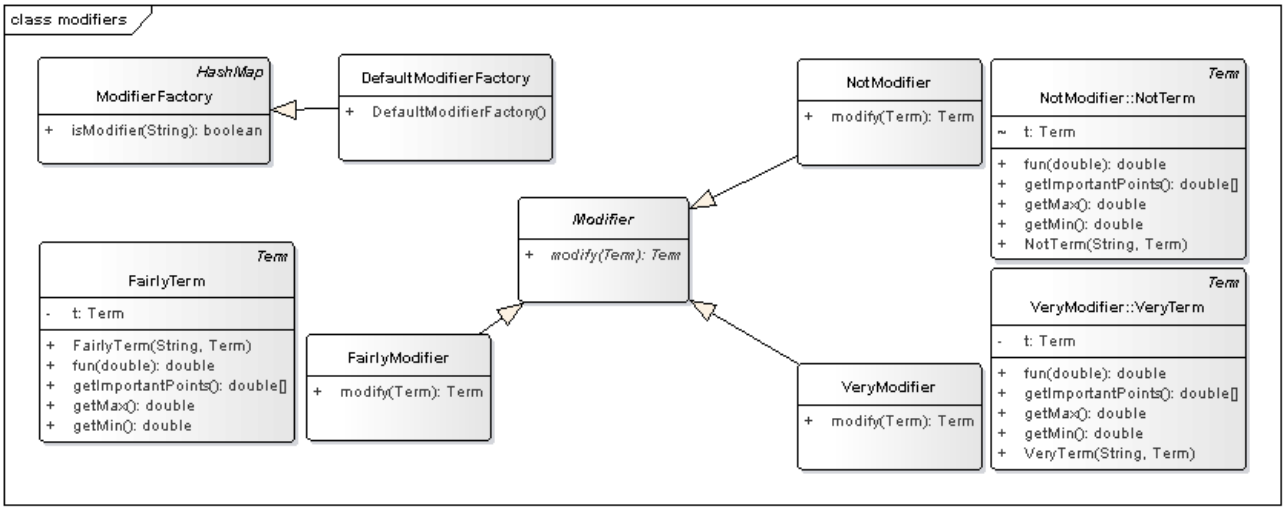


rys. IV.1.8.3)/2 Efekt działania modyfikatora very (bardzo – zielony)



rys. IV.1.8.3)/3 Efekt działania modyfikatora fairly (nieco – zielony)

Struktura pakietu modyfikatorów wygląda następująco:



rys. IV.1.8.3)/4 Diagram klas modyfikatorów.

Trzy definicje modyfikatorów implementują swoje własne funkcje przynależności (termy), które bazują na podstawie podanej do modyfikatora funkcji. Oprócz tego istnieje jeszcze fabryka, która ułatwia rozszerzanie tego pakietu o kolejne implementacje.

IV.1.8.4) Kod źródłowy

Reprezentacja reguły:

```
public class Rule implements Serializable{
	private static final long serialVersionUID = -8303989672422512125L;
	@Expose
	private String text;
	@Expose
	private String name;
	@Expose
	Cause cause;
	@Expose
	List<Effect> effect = new ArrayList<Effect> ();
	private List<BaseFunctionVariable> dependend = new
ArrayList<BaseFunctionVariable>();
	private List<BaseFunctionVariable> affected = new
ArrayList<BaseFunctionVariable>();
	private String id;
	@Override
	public boolean equals(Object obj) {
		return this.id.equals(((Rule)obj).id);
	}
	public void setText(String text) {
		this.text = text;
	}

	public void setName(String name) {
		this.name = name;
	}
	public String getName () {
		return this.name;
	}

	public Rule(String name) {
		this.name = name;
		Random r  = new Random();
		this.id = name + r.nextLong();
	}
```

```
	public Rule(String name, String text) {
		this(name);
		this.text = text;
	}

	public void addDepenedency(BaseFunctionVariable v) {
		if (!this.dependend.contains(new BaseFunctionVariable(v.getName())))
			this.dependend.add(v);
	}

	public void addAffected(BaseFunctionVariable v) {
		if (!this.affected.contains(new BaseFunctionVariable(v.getName())))
			this.affected.add(v);
	}

	public boolean affects(BaseFunctionVariable var) {
		return this.affected.contains(new BaseFunctionVariable(var.getName()));
	}
	public boolean dependsOn(BaseFunctionVariable var) {
		return this.dependend.contains(new BaseFunctionVariable(var.getName()));
	}

	public Cause getCause() {
		return cause;
	}

	public List<Effect> getEffects() {
		return effect;
	}

	public List<BaseFunctionVariable> getAffected() {
		return this.affected;
	}
	public void calculate() {
		double activation = this.getCause().getAction().getValue();
		if (activation > 0)
			this.getEffects().forEach(e -> {
				e.execute(activation);
			});
	}
}
```

Klasa stanowi jedynie kontener zawierający listy zmiennych wejściowych i wyjściowych reguły, listę efektów oraz obiekt Cause stanowiący korzeń drzewa akcji.

Przykładowa akcja (AndAction):

```
public class AndAction extends Action {

	@Expose
	String type = "and";
	@Expose
	Action left;
	@Expose
	Action right;
	private AndMethod andMethod;

	public AndAction(Action l, Action r, AndMethod andMethod) {
		this.left = l;
		this.right = r;
		this.andMethod = andMethod;
	}

	@Override
```

```
	public double getValue() {
		return andMethod.and(left.getValue(), right.getValue());
	}

}
```

Akcja agreguje dwie kolejne akcje występujące po lewej stronie operatora and. Mogą to być końcowe akcje (TermAction) albo kolejne akcje operatorów. Pozwala to na zastosowanie rekurencji przy tworzeniu akcji (klasa DefaultActionFactory, rekurencja oznaczona na czerwono w komentarzu):

```
public class DefaultActionFactory {
	(...)

	public Action createAction(String text, Rule r) throws //rekurencja
RuleParsingException, InlineVariableNotFoundException,
InputVariableNotFoundException, TermNotFoundException {
		text = text.trim();
		text = this.dropBrackets (text);
		int pos = findFirstNotEnclosed ("or",text);
		if (pos<0) { //no or, search for and
			pos = findFirstNotEnclosed ("and",text);
				if (pos<0) {
					return this.parseSingleExpression(text,r);
				}
			return this.createAndAction (text,pos,r);
		}
		return this.createOrAction (text,pos,r);
	}
	private Action createOrAction(String text, int pos, Rule ru) throws
RuleParsingException, InlineVariableNotFoundException,
InputVariableNotFoundException, TermNotFoundException {
		String left = text.substring(0, pos);
		String right = text.substring(pos + OR.length());
		Action l = this.createAction(left,ru); //rekurencja
		Action r = this.createAction(right,ru); //rekurencja
		if (l==null || r ==null) throw new RuleParsingException("Unknown error
at parsing rule expression");
		return new OrAction (l,r,this.getAndMethod());
	}
	private Action createAndAction(String text, int pos, Rule ru) throws
RuleParsingException, InlineVariableNotFoundException,
InputVariableNotFoundException, TermNotFoundException {
		String left = text.substring(0, pos);
		String right = text.substring(pos+AND.length());
		Action l = this.createAction(left,ru); //rekurencja
		Action r = this.createAction(right,ru); //rekurencja
		if (l==null || r ==null) throw new RuleParsingException("Unknown error
at parsing rule expression");
		return new AndAction (l,r,this.getAndMethod());
	}
	private Action parseSingleExpression(String text, Rule r) throws
RuleParsingException, InlineVariableNotFoundException,
InputVariableNotFoundException, TermNotFoundException {
	(...)
}
```

Przykładowy efekt (WithEffect) :

```
public class WithEffect extends Effect {
	private Term term;
```

```
	@Expose
	private double level;
	@Expose
	private String type = "with";
	public WithEffect(Rule r, double level, Term t, BaseFunctionVariable v) throws
RuleParsingException {
		super (r,v);
		this.level = level;
		this.term = t;
	}
	public Term getTerm(double level) {
		return new CutTerm (this.rule.getName(), this.level,this.term);
	}
}
```

Klasa tworzy nowy term obcięty na podstawie podanego w konkluzji na ustalonym w skrypcie poziomie.

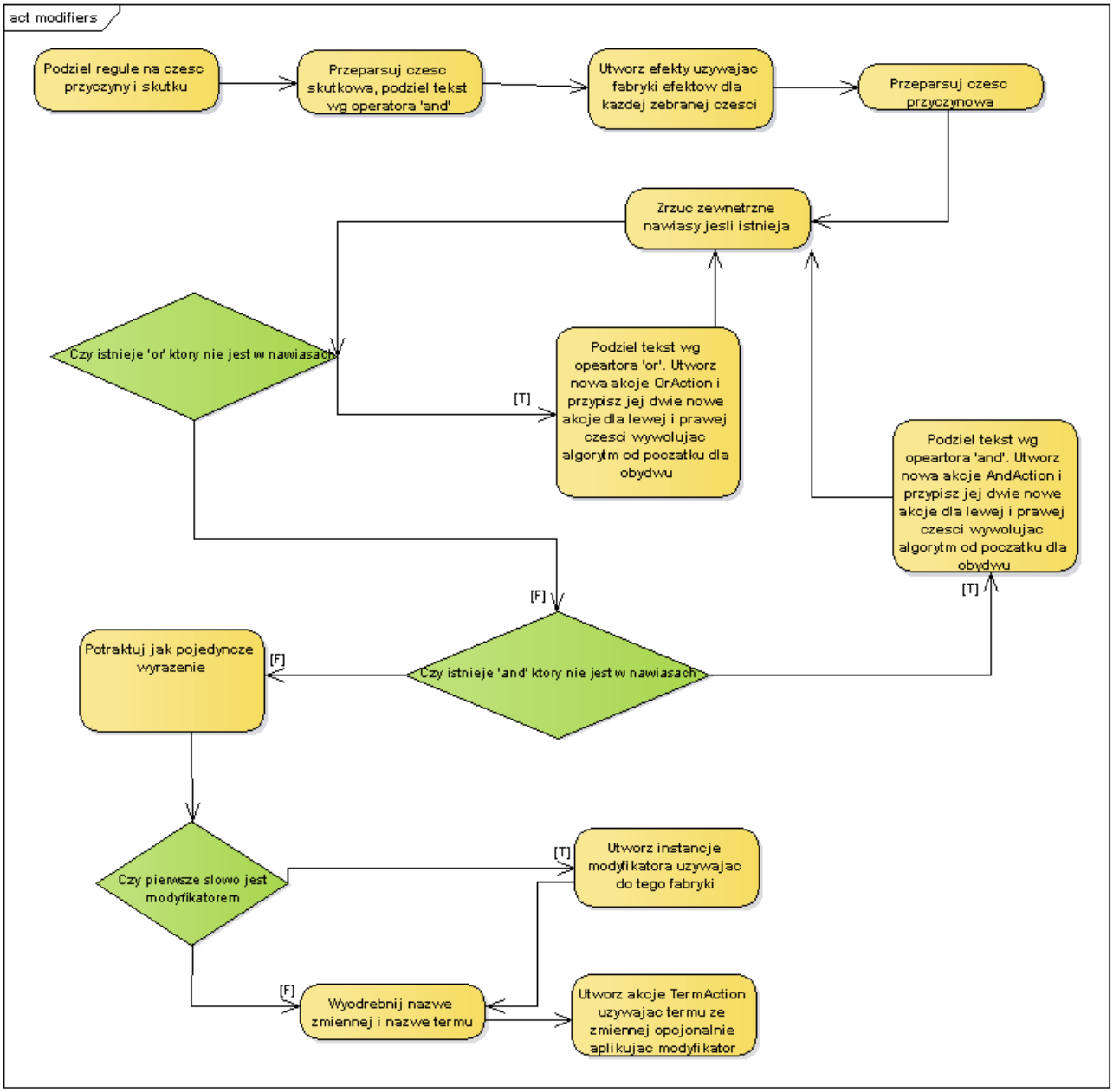
Przykładowy modyfikator (VeryModifier) :

```
public class VeryModifier extends Modifier {
	public class VeryTerm extends Term {
		private Term t;
		public VeryTerm(String name, Term t) {
			super(name);
			this.t = t;
		}
		@Override
		public double getMax() {
			return t.getMax();
		}
		@Override
		public double getMin() {
			return t.getMin();
		}
		@Override
		public double[] getImportantPoints() {
			return t.getImportantPoints();
		}
		@Override
		public double fun(double val) {
			double z=t.fun(val);
			return z*z;
		}
	}
	@Override
	public Term modify(Term t) {
		return new VeryTerm(t.getName(), t);
	}
}
```

Klasa definiuje nową klasę wewnętrzną na bazie klasy termu – VeryTerm, delegując wszystkie metody oprócz *fun(double)* – ta metoda zwraca kwadrat rezultatu poprzedniego termu.

IV.1.8.5) Tworzenie reguły

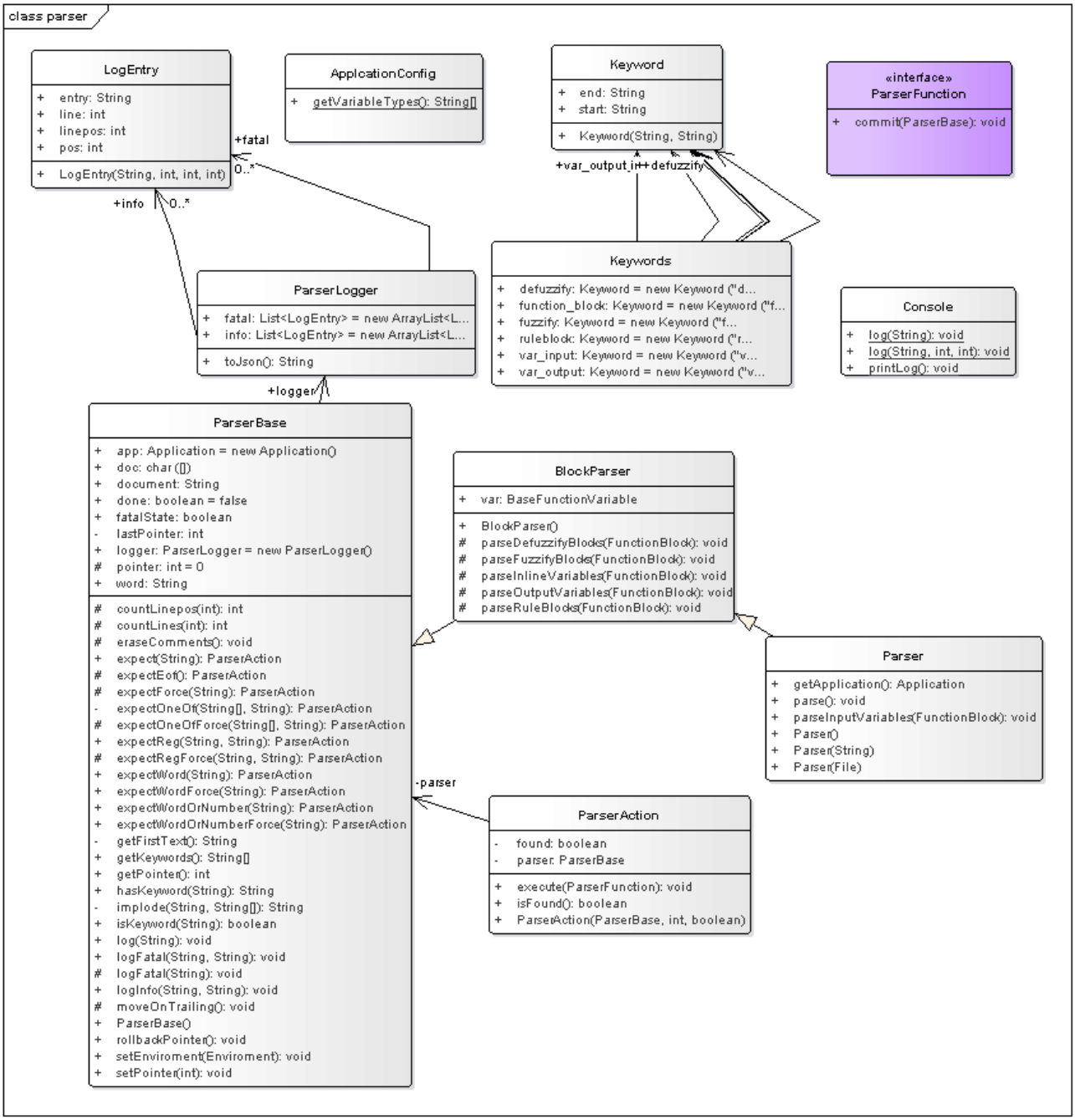
Algorytm tworzenia reguły zawiera rekurencje. Poniższy diagram aktywności przedstawia proces parsowania treści reguły w uproszczeniu (happy path). W rzeczywistości każdy fragment jest podatny na błędy, ponieważ stanowi treść wprowadzaną przez użytkownika. Z tego względu poszczególne funkcje rzucają wyjątki, które są wychwytywane w warstwie obsługi wyjątków w głównym parserze.



rys. IV.1.8.5)/1 Diagram sekwencji tworzenia

IV.1.9. Moduł parsera

IV.1.9.1) Struktura.

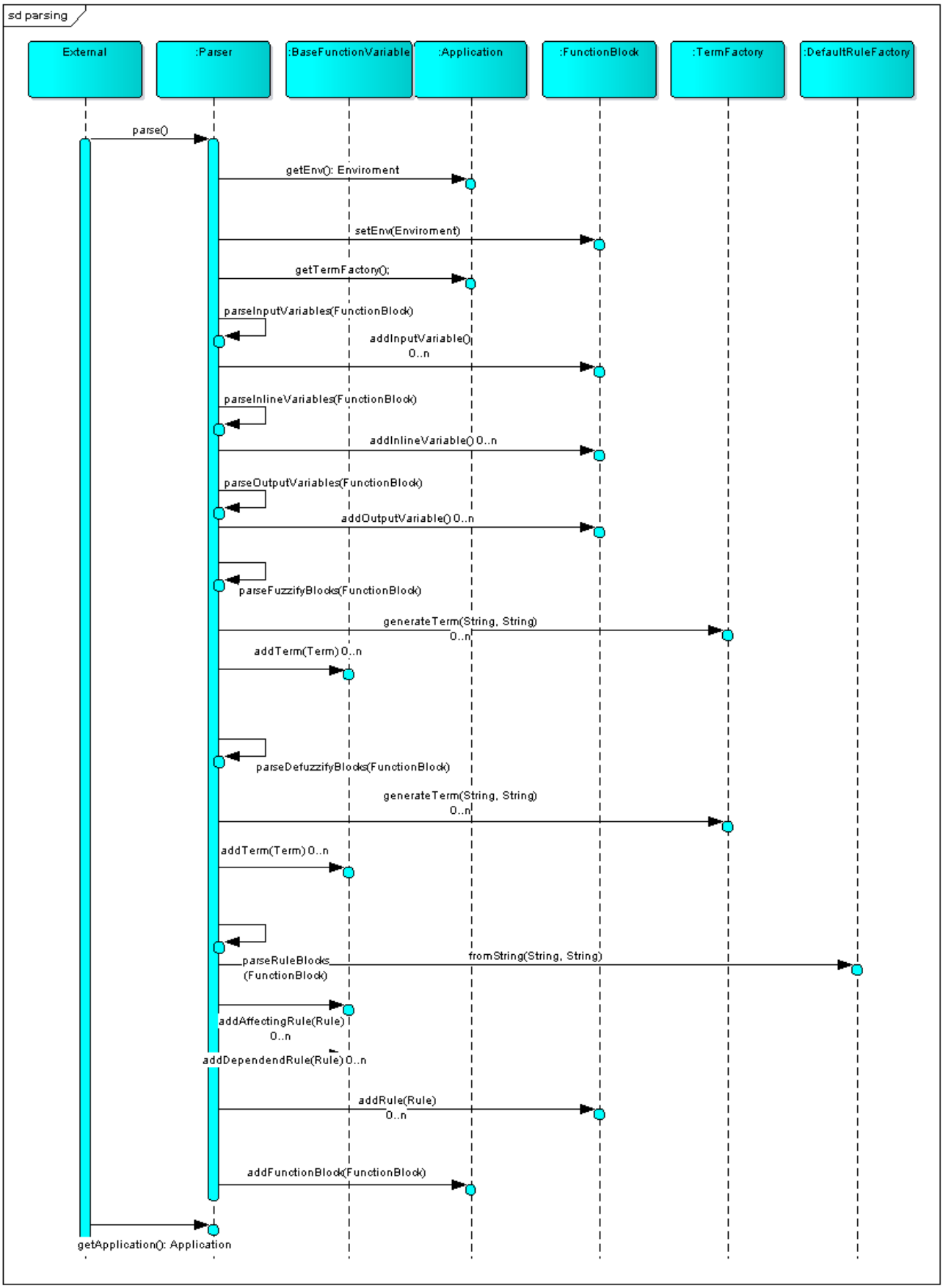


Rys IV.1.9.1)/1 Parser wraz zależnościami

Parser jest jednym z bardziej złożonych elementów aplikacji. Z tego względu odpowiedzialność za część prac przejęły klasy bazowe. Do rozpoznawania treści używany jest mechanizm Regex, który rozpoznaje wzorce przygotowane do konstrukcji skryptu FCL. Posiada również mechanizm logowania błędów, które mogą wystąpić w skryptach.

Sama procedura parsowania oparta jest na stosunkowo nowym w Javie (od wersji 1.8) mechanizmie funkcji anonimowych (lambd). Przykład zostanie przedstawiony przy fragmentach kodu.

IV.1.9.2) Mechanizm parsowania



Rys IV.1.9.2)/1 Mechanizm parsowania

Parser przeprowadza kompilację skryptu FCL. Mechanizm jest liniowy, wyszukiwane są wszystkie bloki we właściwej kolejności. Elementy stałe konstrukcji FCL parsowane są bezpośrednio przez metody tej klasy. Elementy, którym nadano elastyczność i możliwość wprowadzania zmian, kreowane są przez właściwe im fabryki. Funkcja parsowania uruchamia się wielokrotnie dla każdego spotkanego bloku funkcji.

IV.1.9.3) Fragmenty kodu

Metoda odpowiadająca za przetłumaczenie fragmentu opisującego zmienne wyjściowe na odpowiednią reprezentację obiektową w systemie. Obiekty p1, p2, p3 itd. to kolejne instancje parsera, który jest odpowiednio zmodyfikowany (przesunięty pointer, ustawiona flaga itp.). Doklejenie słowa Force do metody expect oczekuje od niej wyrzucenia wyjątku w przypadku, gdy słowo nie zostanie znalezione. Wyjątek ten jest wychwytywany w głównej metodzie parsującej.

```
protected void parseOutputVariables(FunctionBlock fb) {
	expectForce("var_output").execute(
	p2-> { while (!expect("end_var").isFound()) {
		expectWordForce("variable name or 'end_var'").execute(
		p3 -> {
			if (isKeyword(p3.word)) {
				logFatal("variable name", "keyword " + p3.word);
				}
			OutputVariable var = new OutputVariable(p3.word,fb);
			expectForce(":").execute(
			p4 -> {
			expectOneOfForce(
				ApplcationConfig.getVariableTypes(),
				"variable type").execute(

				p5 -> {
					var.setType(p5.word);
					expectForce(";").execute(
					p6 -> {
						fb.output.add(var);

					});
				});
			});
		});
	}});
}
```

Główna funkcja parsująca odpowiada za znajdowanie całych bloków funkcji, następnie deleguje prace do pozostałych funkcji:

```
	public void parse() {
		try {
		this.eraseComments();
		if (this.doc.length==0) return;
		while (!expectEof ().isFound())
		expectForce("function_block").execute(
				p1 -> {
					FunctionBlock fb = new FunctionBlock(app);
					fb.setEnv(app.getEnv());
					app.addFunctionBlock(fb);
					expectWordForce("function block name").execute(p2 ->
{

						if (isKeyword(p2.word)) {
							logFatal("function block name", "keyword
" + p2.word);
						}
```

```
                                    fb.name = p2.word;
                              });
                              this.parseInputVariables(fb);
                              this.parseInlineVariables(fb);
                              this.parseOutputVariables(fb);
                              this.parseFuzzifyBlocks (fb);
                              this.parseDefuzzifyBlocks (fb);
                              this.parseRuleBlocks(fb);
                              expectForce("end_function_block");
                        });
            }
            catch (Exception e) {
                  this.rollbackPointer();
                  this.logger.fatal.add(new LogEntry (e.getMessage(),
this.countLines(getPointer()), getPointer(), this.countLinepos(getPointer())));
                  e.printStackTrace();
            }
            this.app.logger=this.logger;
      }
```

Logowanie błędów, oprócz wiadomości z Exception, zapisuje również aktualną pozycję wskaźnika, numer linii i pozycję w linii. Klasa odpowiadająca za logowanie błędów posiada dwa poziomy błędów:

- info – ten poziom informuje o każdej czynności, która jest wykonywana przez parser. Metody typu expect automatycznie wrzucają na ten poziom informacje na temat wykonywanych przez nie czynności
- fatal – błąd, który przerwał parsowanie i nie pozwala na podjęcie dalszych czynności. Instancja aplikacji została jednak utworzona, choć na pewno brakuje jej funkcjonalności skryptu od miejca wystąpienia błędu

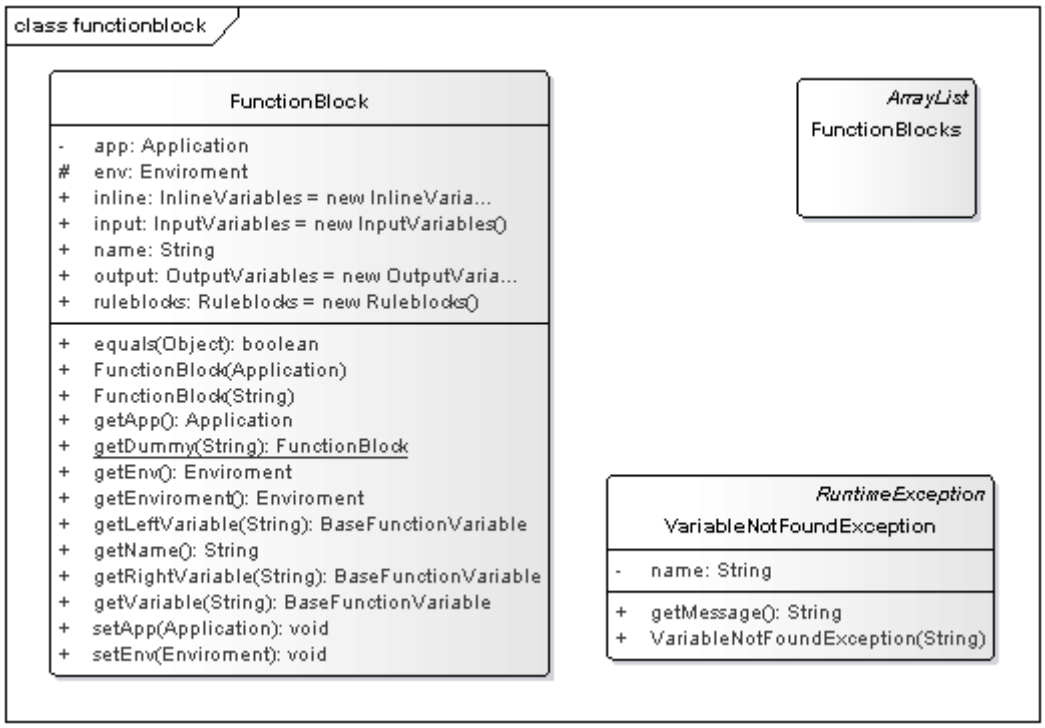
```
public class ParserLogger {
      //@Expose
      public List<LogEntry> info = new ArrayList<LogEntry>();
      @Expose
      public List<LogEntry> fatal = new ArrayList<LogEntry>();

      public String toJson() throws FileNotFoundException {

            Gson gson = new GsonBuilder().excludeFieldsWithoutExposeAnnotation()
                        .setPrettyPrinting().create();
            return gson.toJson(this);

      }
}
```

Parser jest skromną klasą, która daje dostęp z zewnątrz do swoich elementów. Posiada również metodę, która przerabia wszystkie logi na wygodny w odbiorze plik tekstowy w formacie JSON.

IV.1.10. Moduł bloku funkcji

IV.1.10.1) Struktura

Zadaniem tego modułu jest reprezentacja bloku funkcji skryptu fcl. Zbudowany jest następująco:



Rys IV.1.10.1)/1 Reprezentacja bloku funkcji

Jedyne funkcjonalności modułu to udostępnienie kolejnych elementów reprezentacji skryptu – zmiennych i reguł.

IV.1.11. Moduł termów

IV.1.11.1) Struktura

Głównym elementem modułu jest klasa reprezentująca funkcję przynależności wraz z kilkoma przydatnymi informacjami:

- największa wartość, która wg skryptu fcl ma znaczenie dla tej funkcji przynależności
- najmniejsza wartość jw.
- nazwa typu termu (tylko informacyjnie - nie ma wpływu na działanie aplikacji).
- nazwa termu
- term bazowy – jeśli mamy do czynienia z np. obciętą funkcją przynależności.
- istotne punkty na osi zmiennej (dotyczy głównie singletonów oraz wszelkich zmian kierunku krzywej funkcji – potrzebne do prawidłowego rysowania wykresów termów)

Najważniejszą metodą klasy jest abstrakcyjna funkcja *fun(double)*. Implementacja tej funkcji należy do wszystkich klas dziedzicznych, które reprezentują w ten sposób funkcję przynależności danego termu.

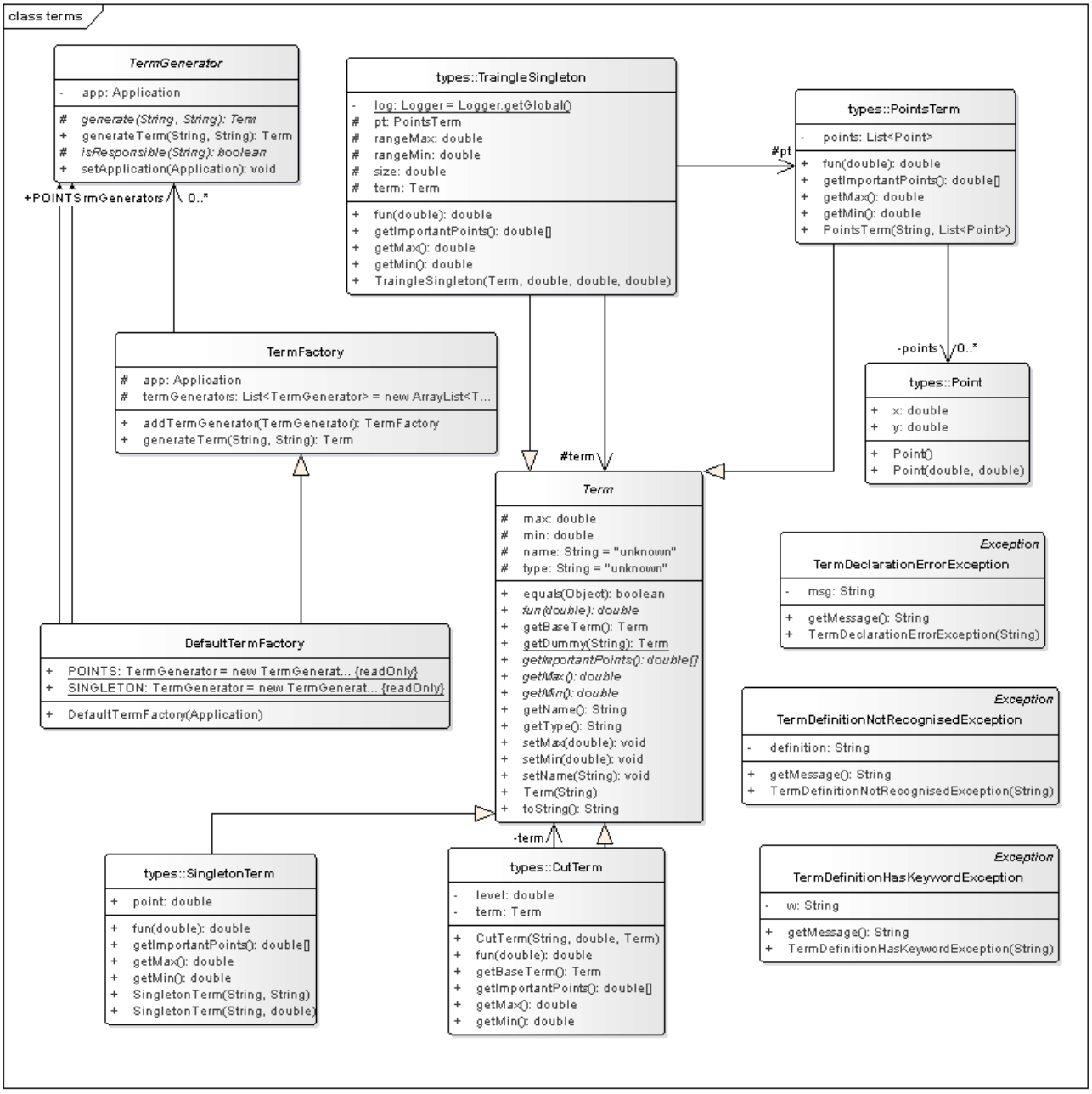
Osobnym elementem w module jest fabryka termów oraz ich generatory. Jeden generator wytwarza jeden rodzaj termu. Zadaniem generatora jest rozpoznanie czy tekst stanowiący definicję termu (FCL) jest tym, czego potrzebuje do zdefiniowania termu typu, który mu odpowiada. Fabryka

odpytuje kolejne generatory i jeśli któryś zwróci jej gotowy term, kończy pracę. W przeciwnym przypadku wyrzuca wyjątek.

Pakiet zawiera też definicje kilku termów:

- PointsTerm – tworzy krzywe z linii połączonych punktami.
- CutTerm – obcina istniejący term na określonej wysokości za pomocą funkcji min.
- SingletonTerm – posiada tylko jeden punkt, gdzie wartość funkcji przynależności wynosi 1, w pozostałym zakresie jest to 0.

Struktura klas pakietu:



Rys IV.1.11.1)/1 Diagram klas modułu termów

IV.1.11.2) Fragmenty kodu.

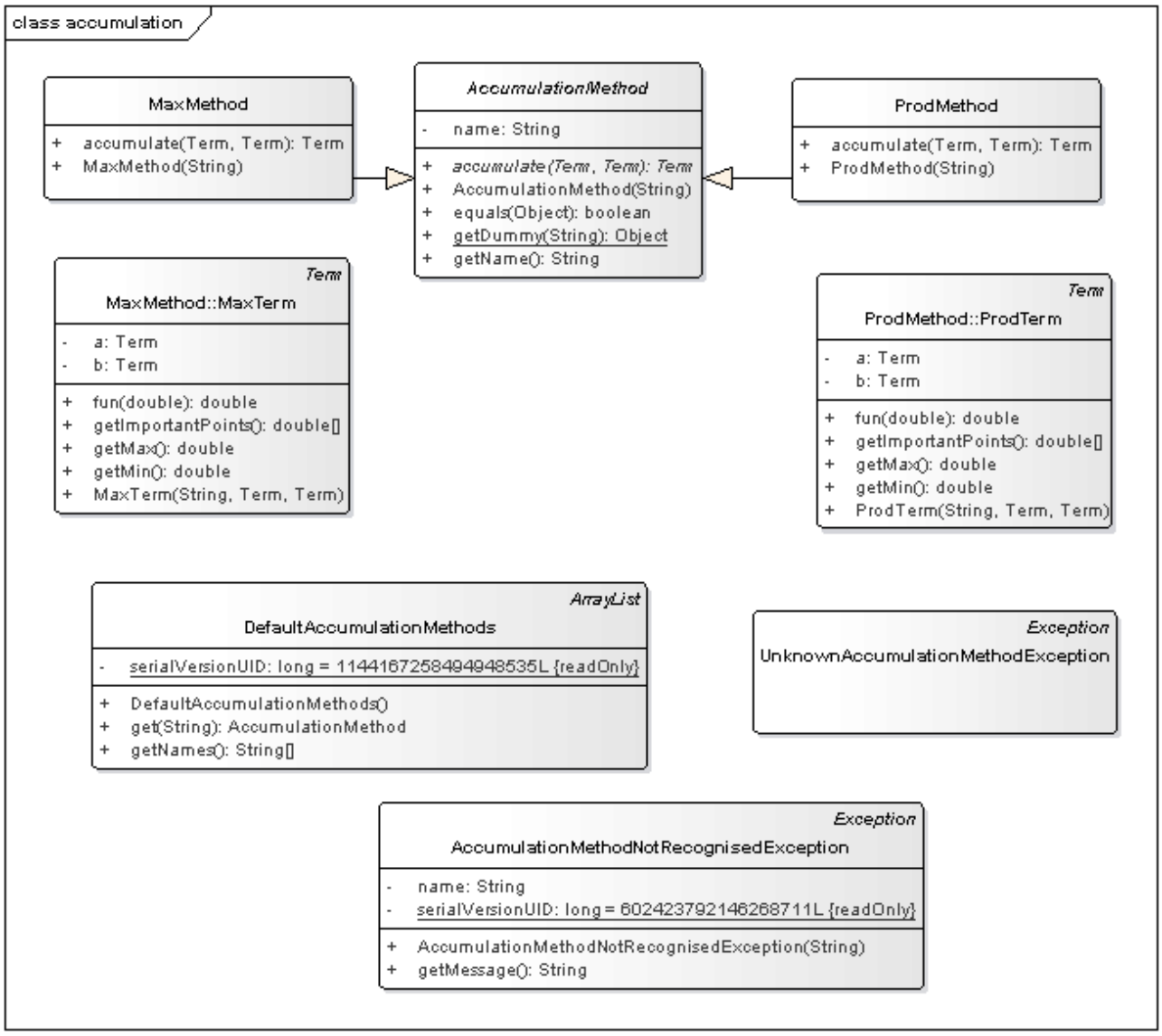
Przykład typu termu (PointsTerm):

```
public class PointsTerm extends Term {

	@Expose
	private List<Point> points;

	public PointsTerm(String termName, List<Point> p) {
		super(termName);
		this.max = p.get(p.size()-1).x;
		this.min = p.get(0).x;
		this.type = "points";
		this.points=p;
	}

	@Override
	public double fun(double val) {
		if (val <= points.get(0).x)
			return points.get(0).y;
		if (val >= points.get(points.size()-1).x)
			return points.get(points.size()-1).y;
		for (int i = 0; i < points.size() - 1; i++) {
			if (points.get(i).x <= val && points.get(i + 1).x >= val) {
				double x1 = points.get(i).x;
				double x2 = points.get(i + 1).x;
				double y1 = points.get(i).y;
				double y2 = points.get(i + 1).y;
				return (y2 - y1) * (val - x1) / (x2 - x1) + y1;
			}
		}
		return Double.NaN;
	}

	@Override
	public double getMax() {
		return this.max;
	}

	@Override
	public double getMin() {
		return this.min;
	}

	@Override
	public double[] getImportantPoints() {
		double [] list = new double [this.points.size()];
		for (int i=0; i<list.length; i++)
			list[i]=points.get(i).x;
		return list;
	}
}
```

Zadaniem klasy jest utworzenie funkcji, która składa się z punktów połączonych odcinkami. Do wyliczenia funkcji odcinków wykorzystano wzór na prostą przechodząca przez dwa punkty. Poszukiwany jest zakres podanej wartości w liście punktów, następnie obliczana jest wartość w danym punkcie z ww. wzoru.

IV.1.12. Moduł akumulacji

IV.1.12.1) Struktura



Rys IV.1.12.1)/1 Diagram klas modułu akumulacji

Moduł ten zawiera definicje metod akumulacji oraz fabryki dobierające odpowiednią metodę na podstawie nazwy metody użytej w skrypcie FCL. Metoda pobiera dwie funkcje przynależności i za pomocą odpowiedniej funkcji łączy je w jedną. Moduł zawiera definicje dwóch metod akumulacji oraz dwie definicje termów, których zadaniem jest połączenie dwóch termów w jeden.

Konstruktor metod akumulacji pobiera nową nazwę, która będzie identyfikować tę metodę w skrypcie FCL. Akumulacja polega na wielokrotnym utworzeniu instancji termu akumulacji z poprzedniej instancji i termu do zakumulowania. Powstałe w ten sposób warstwy tworzą na końcu jeden term przypisany do zmiennej wyjściowej. W tym celu metoda akumulacji implementuje metodę accumulate, która jako parametry pobiera dwa termy (funkcje przynależności).

IV.1.12.2) Fragmenty kodu

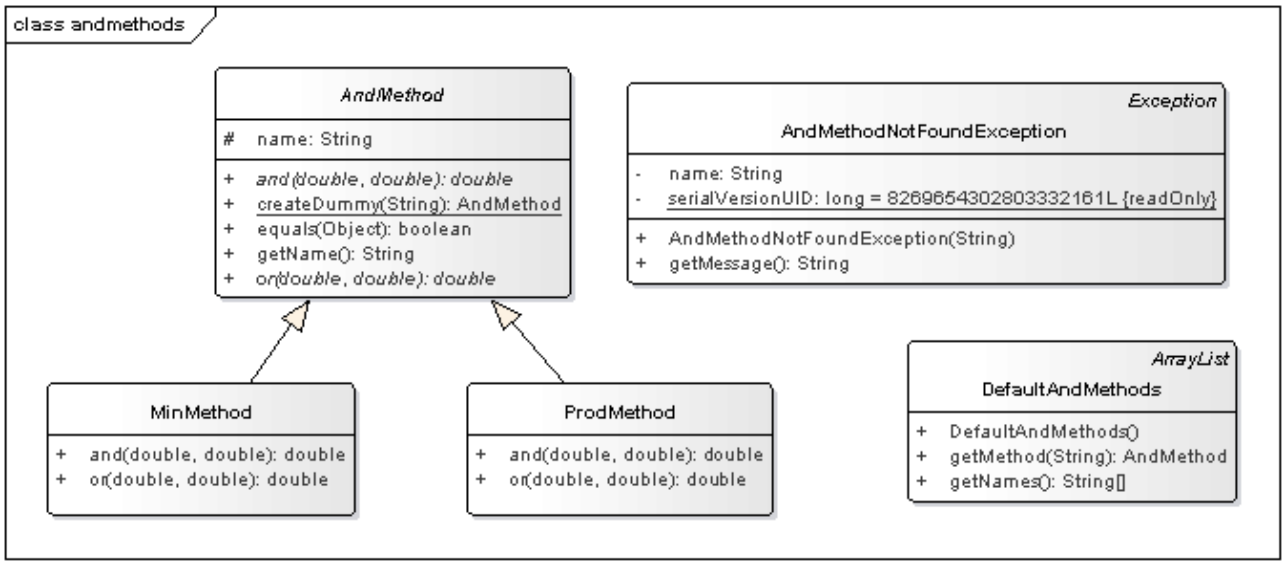
Przykład implementacji metody akumulacji (ProdMethod)

```
public class ProdMethod extends AccumulationMethod {

	public class ProdTerm extends Term {

		private Term a;
		private Term b;

		public ProdTerm(String name, Term a, Term b) {
			super(name);
			this.a = a;
			this.b = b;
			this.max=this.getMax();
			this.min = this.getMin();
			this.type = "accumulation";
		}
		@Override
		public double getMax() {
			return Math.max(a.getMax(), b.getMax());
		}

		@Override
		public double getMin() {
			return Math.min(a.getMin(), b.getMin());
		}

		@Override
		public double fun(double val) {
			return a.fun(val) + b.fun(val) - a.fun(val)*b.fun(val);
		}

		@Override
		public double[] getImportantPoints() {
			return ArrayUtils.addAll
(a.getImportantPoints(),b.getImportantPoints());
		}



	}
	public ProdMethod(String name) {
		super(name);
	}

	@Override
	public Term accumulate(Term aa, Term bb) {
		return new ProdTerm ("@accu", aa,bb);
		}
}
```

Powyższa klasa, oprócz implementacji metody accumulate, niezbędnej do przeprowadzenia obliczeń, zawiera również własną implementację klasy Term, która jako funkcję przynależności definiuje metodę prod na dwóch pobranych funkcjach przynależności. Każde wywołanie metody tworzy nową funkcję przynależności na podstawie dwóch podanych.

IV.1.13. Moduł spójników.

IV.1.13.1) Struktura.



Rys IV.1.13.1)/1 Diagram klas modułu spójników

Moduł zawiera fabrykę, która jest w stanie wyprodukować dwa typy spójników na podstawie ich nazw:

- min
- prod

Każda metoda musi zaimplementować klasę AndMethod która posiada dwie metody abstrakcyjne:

- *and(double,double)*
- *or(double,double)*

Na ich podstawie obliczana jest wartość wynikowa aktywacji przesłanek.

IV.1.13.2) Fragmenty kodu

Implementacja spójnika and i or metodą prod (ProdMethod):

```
public class ProdMethod extends AndMethod  {
	{
		this.name="prod";
	}
	@Override
	public double and(double a, double b) {
		return a*b;
	}
	@Override
	public double or(double a, double b) {
		return a+b-a*b;
	}

}
```

IV.1.14. Moduł defuzyfikacji

IV.1.14.1) Działanie

Moduł ten definiuje metody defuzyfikacji. Każda nowa metoda musi dziedziczyć po klasie abstrakcyjnej DefuzzificationMethod implemenując funkcję calculate. Funkcja otrzymuje :

- funkcję przynależności wszystkich zakumulowanych termów
- listę funkcji przynależności wynikowych z poszczególnych reguł
- referencję zmiennej wyjściowej dla której liczona jest wartość
- metodę akumulacji przypisaną do tej zmiennej

Zdefiniowane są dwie metody defuzyfikacji:

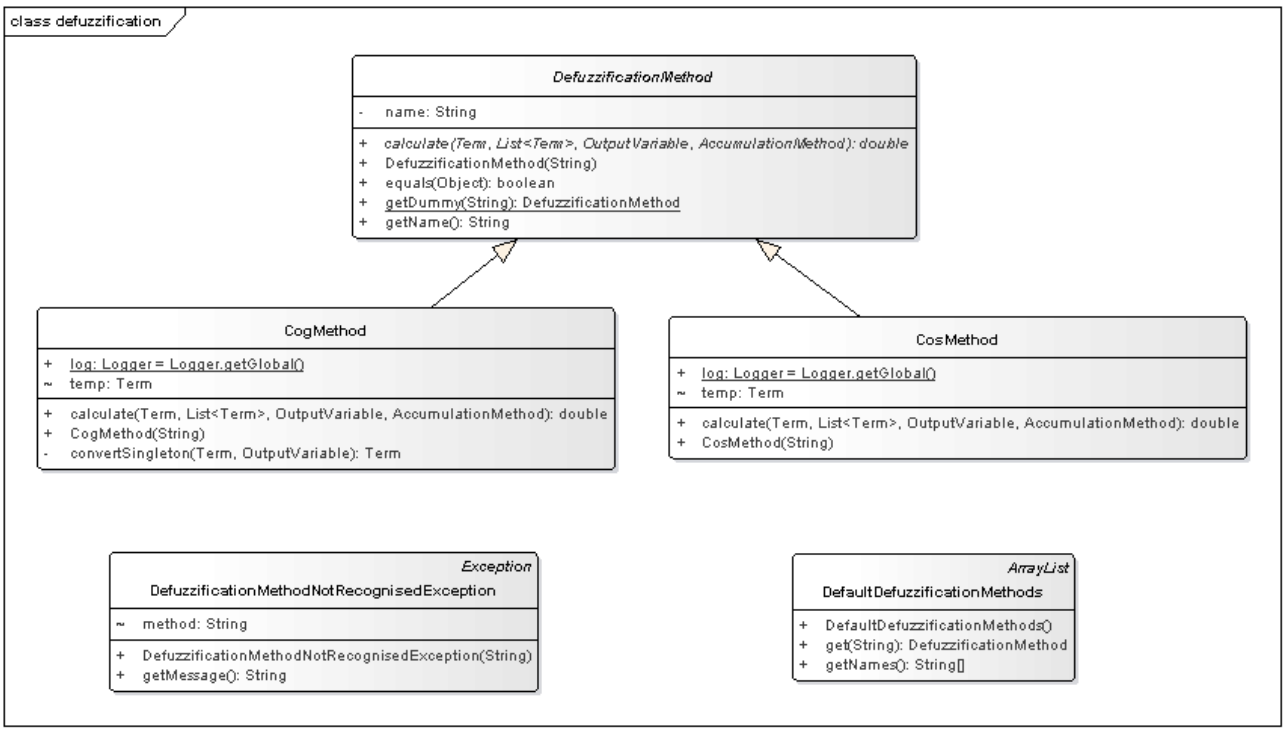
- COG - Center of Gravity
- COS – Center of (gravity for) Singletons

Metoda COG bazuje na całkowaniu numerycznym – metodą trapezów. Polega ona na dzieleniu zakresu całkowania na przedziały i obliczaniu pola trapezu, którego dwa punkty leżą na osi x, dwa pozostałe zaś znajdują się na wykresie funkcji.

$$\int_{x_p}^{x_k} f(x)\,dx \approx \frac{x_k - x_p}{n}\left(\sum_{i=1}^{n} f\left(x_k + i\frac{x_k - x_p}{n}\right) + \frac{f(x_k)+f(x_p)}{2}\right)$$

Metoda COS wylicza wartość za pomocą wzoru opisanego w II.2.3.

IV.1.14.2) Struktura

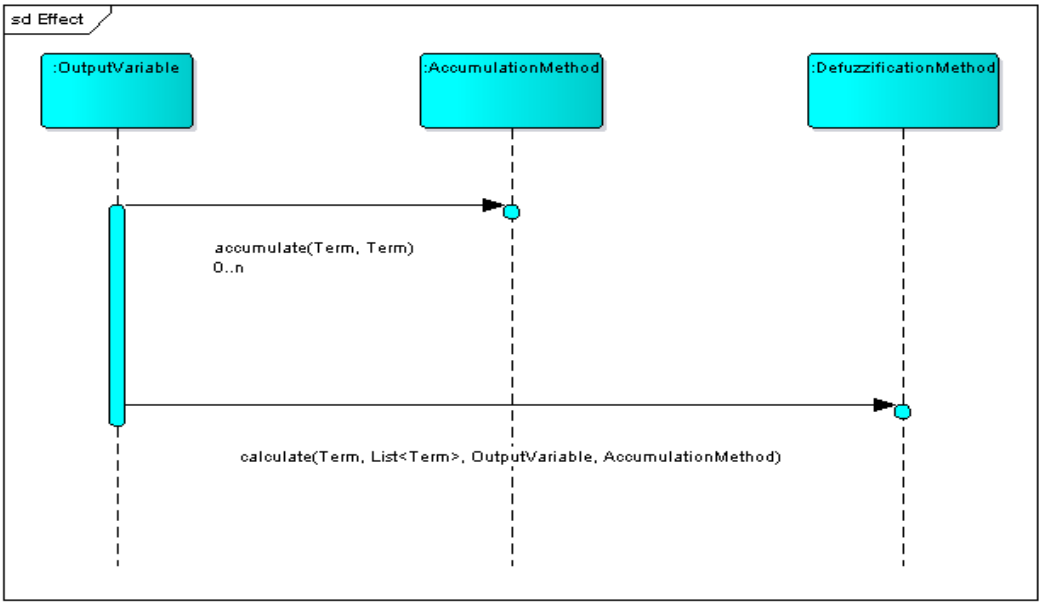


Rys IV.1.14.2)/1 Diagram klas metod defuzyfikacji

Moduł zawiera definicję klasy abstrakcyjnej DefuzzificationMethod, która jest rozszerzana przez szczegółowe metody opisane w poprzednim punkcie. Zawiera również fabrykę, która zwraca instancję metody na podstawie jej nazwy.

IV.1.14.3) Obliczanie wartości zmiennej za pomocą defuzyfikacji

Obliczanie wartości zmiennej prezentuje poniższy diagram sekwencji:



Rys IV.1.14.3)/1 Diagram sekwencji defuzyfikacji i akumulacji.

Za pomocą akumulacji gromadzone są wszystkie funkcje przynależności z odpalonych reguł (zapisane wcześniej), następnie uruchamiana jest metoda calculate z właściwej instancji obiektu, którego klasa dziedziczy po DefuzzificationMethod (pobranej z fabryki).

Metoda akumulacji musi zaimplementować jedną funkcję, której zadaniem będzie połączenie dwóch funkcji przynależności w jedną. Sama akumulacja uruchamiana jest dla każdego zakumulowanego termu w zmiennej wyjściowej z wynikiem poprzedniej akumulacji.

Metoda defuzyfikacji również musi zaimplementować jedną funkcję, której zadaniem jest wyliczenie zmiennoprzecinkowej wartości zmiennej, na podstawie zakumulowanej funkcji przynależności. Dodatkowo, z tego względu, że niektóre metody potrzebują większej ilości informacji, przekazywane są również:

- lista zakumulowanych termów
- zmienna wyjściowa
- metoda akumulacji

IV.1.14.4) Fragmenty kodu.

```
public class CogMethod extends DefuzzificationMethod {
	public static Logger log = Logger.getGlobal() ;
	public CogMethod(String method) {
		super(method);
	}
	Term temp;
	@Override
	public double calculate(Term first, List<Term> acculist,
			OutputVariable var, AccumulationMethod accuMethod) {
		//recalculating main term - converting singletons to triangular terms
		Term main = convertSingleton (acculist.get(0),var);
		for (int i=1; i< acculist.size(); i++) {
			main = accuMethod.accumulate(main, convertSingleton
(acculist.get(i),var));
		}
```

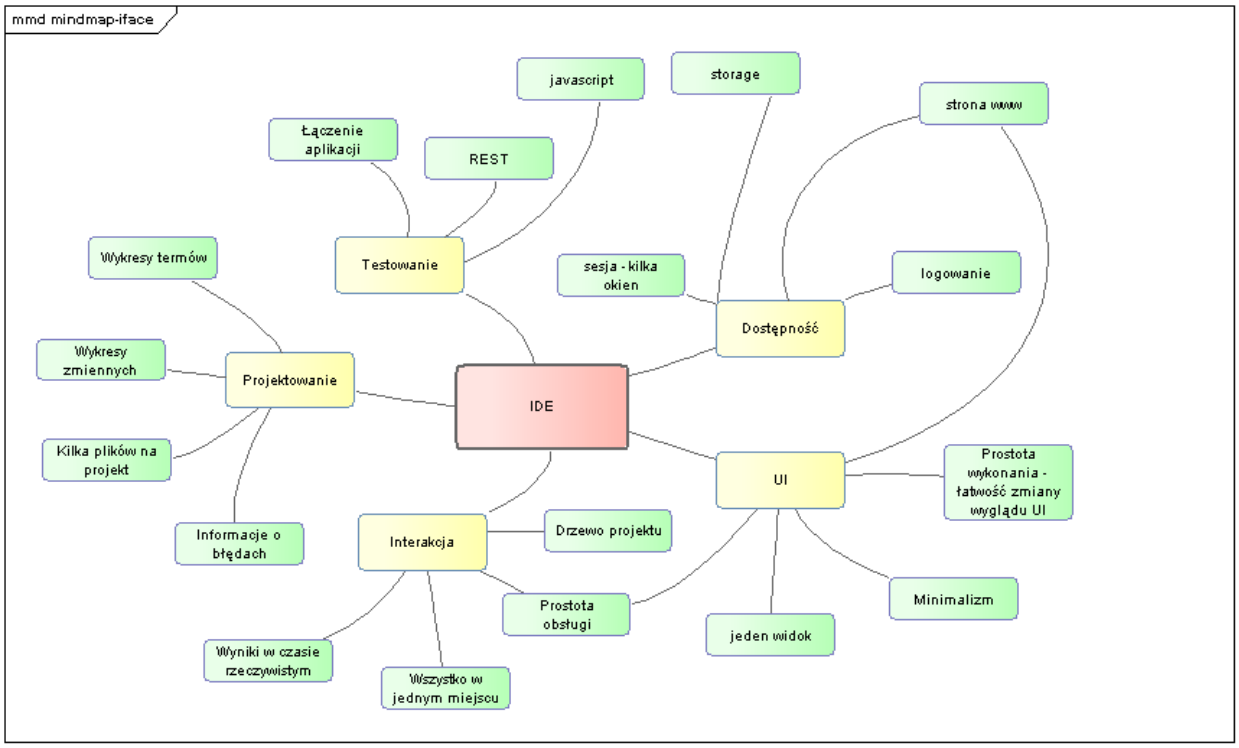
```
            temp = main;
            TrapezoidIntegrator u = new TrapezoidIntegrator();
            TrapezoidIntegrator l = new TrapezoidIntegrator();
            double min = /*temp.getMin()==temp.getMax()? var.getMin() :*/
temp.getMin();
            double max = /*temp.getMin()==temp.getMax()? var.getMax() :*/
temp.getMax();
            log.info ("Counting simpson for " + var.getName() + " min : " +
temp.getMin() + " max : " + temp.getMax() );
            double a = u.integrate(Integer.MAX_VALUE, v->{return
v*temp.fun(v)*1000;}, min , max);
            log.info ("a : " + a );
            double b = l.integrate(Integer.MAX_VALUE, v->{return temp.fun(v)*1000;},
min , max);
            log.info ("b : " + b );
            log.info ("Calculated value : " + a/b);
            return b!=0?a/b:var.getValue();
      }

      private Term convertSingleton(Term term, OutputVariable var) {

            Term base = term.getBaseTerm();
            if (base instanceof SingletonTerm) {
                  log.info("CONVERTING SINGLETON !!!");
                  return new TraingleSingleton (term, var.getMin(), var.getMax(),
20);
            }
            return term;
      }

}
```

Metoda korzysta z funkcji całkujących otwartej biblioteki Apache. Oprócz obliczania całek potrzebnych do wyliczenia wartości, metoda podmienia również singletony na trójkątne - całkowalne funkcje przynależności. Ta metoda nie zawsze, ale czasem (przy względnie małych zakresach całkowania), pozwala w przybliżeniu wyliczyć ich „wagę”.

IV.2. Implementacja aplikacji IDE

IV.2.1. Założenia

Jednym z celów tej pracy było utworzenie środowiska programistycznego, którego zadaniem jest ułatwienie projektowania aplikacji FCL. Przy planowaniu prac powstała mapa myśli, której zadaniem było wyszukanie najlepszych rozwiązań:



Rys IV.2.1./1Mapa myśli IDE

IV.2.2. Technologie

Technologie wybrane do napisania środowiska IDE:

- Baza danych MySQL
- Serwer aplikacji JAVA EE 7 Wildfly 8.2
- Hibernate ORM
- Commons Math: The Apache Commons Mathematics Library
- JQuery
- Flot (wykresy 2D)
- D3 Data Driven Documents + Elegans (wykresy 3D)
- Google GSON
- JAX RS
- ACE Editor
- Bootstrap CSS/JS
- JSTree

IV.2.3. Wymagania funkcjonalne

ID	Opis	Priorytet	Harmonogram
FR1	System umożliwia edycję kodu	Wysoki	Release 1
FR2	System umożliwia przedstawienie relacji dwóch zmiennych w formie wykresu	Wysoki	Release 2
FR3	System raportuje o błędach w kodzie	Wysoki	Release 1
FR4	System umożliwia przedstawienie relacji trzech zmiennych	Wysoki	Release 2
FR5	System przedstawia projekt w formie drzewa	Niski	Release 2
FR6	System umożliwia przedstawienie zmiennej wejściowej za pomocą wykresu z funkcjami przynależności termów tej zmiennej	Wysoki	Release 2
FR7	System umożliwia przedstawienie zmiennej wyjściowej za pomocą wykresu z funkcjami przynależności termów tej zmiennej oraz aktualną funkcję przynależności akumulacji rezultatów odpalonych reguł i aktualną wartość zmiennej	Wysoki	Release 2
FR8	System umożliwia zmianę wartości zmiennych	Wysoki	Release 2
FR9	System umożliwia komunikację z zewnętrzną aplikacją za pomocą protokołu http	Niski	Release 2
FR10	System umożliwia edycję kilku dokumentów	Niski	Release 3
FR11	System umożliwia zapis aktualnego stanu projektu w bazie danych	Niski	Release 3
FR12	System umożliwia uwierzytelnianie użytkowników	Niski	Release 3
FR13	System umożliwia przedstawienie ostatnich odpalonych reguł	Niski	Release 3

IV.2.4. Opis funkcjonalny

System składa się z następujących komponentów:

- Serwer
- Aplikacja użytkownika

Serwer
Serwer JBoss to podstawowy element systemu FCL IDE. Odpowiada on za działanie systemu w sieci. Format przesyłanych danych do serwera to JSON. JSON jest formatem tekstowym bazującym na podzbiorze języka JavaScript.

Aplikacja użytkownika
Aplikacja ta może być użyta do projektowania systemów rozmytych w języku FCL lub dla celów edukacyjnych. Umożliwia ona:

Edytowanie projektów w języku FCL
Funkcjonalność ta odpowiada za edycję projektów wcześniej napisanych w języku FCL. Również w celu ich kontynuowania jak i sprawdzenia poprawności ich działania.

Tworzenie swojego projektu w języku FCL
Można stworzyć własny projekt w języku FCL. Temat projektu może być dowolny.

Raportowane błędów
Raportowanie znajduje się w edytorze, gdy jest popełniany błąd to komunikat błędu znajdzie się po lewej stronie.

System pozwala na zdefiniowanie domyślnej wartości zmiennej
Użytkownik po odpaleniu systemu dostaje zainicjowaną zmienną, czyli domyślną.

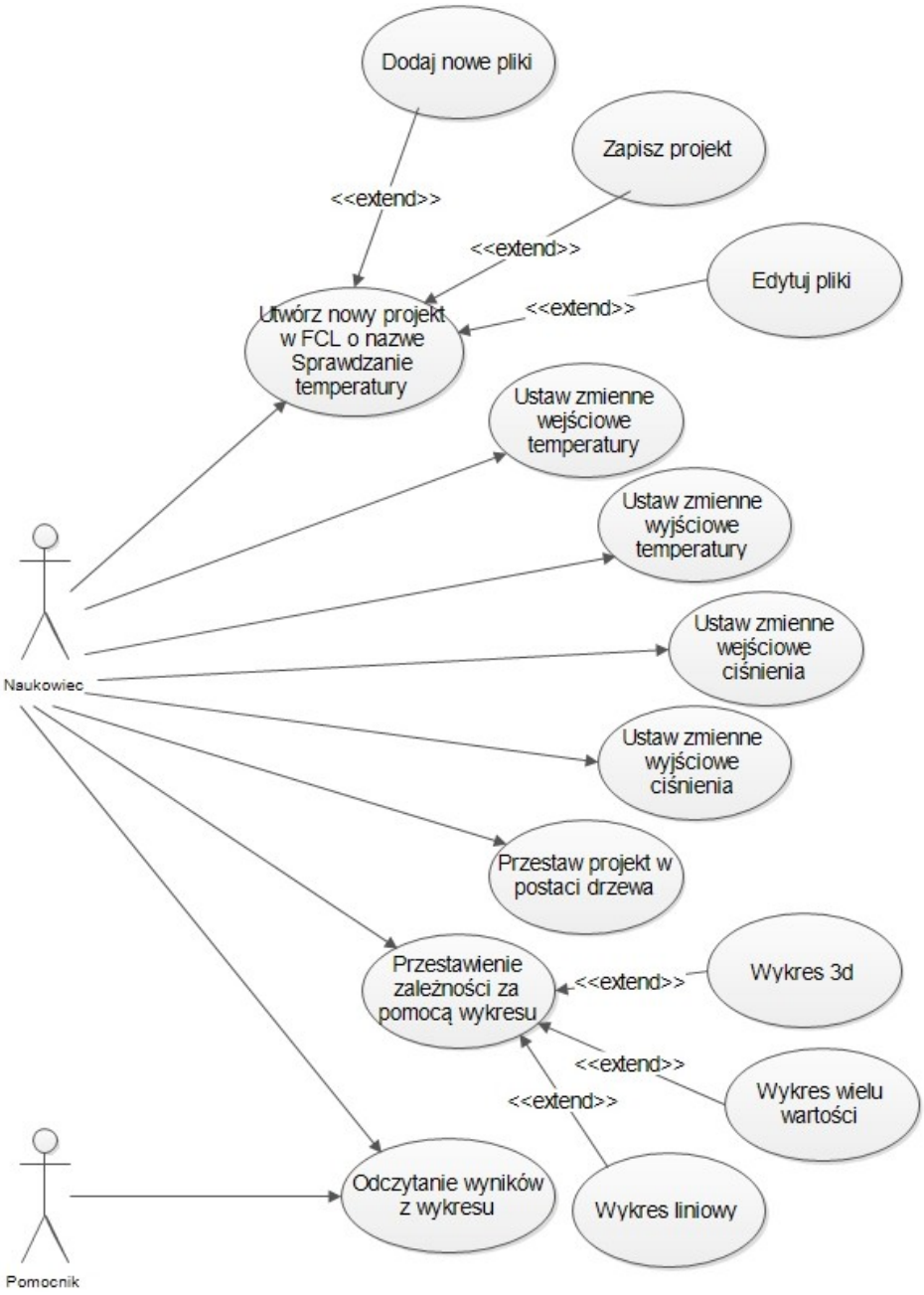
System umożliwia tworzenie zmiennych wewnętrznych
Zmienne wewnętrzne służą do obliczeń pośrednich w aplikacji, traktowane są jak zmienna wyjściowa i wejściowa jednocześnie – może znajdować się z prawej i lewej strony reguły. System pozwala na zdefiniowanie domyślnej wartości zmiennej.

System umożliwia dodanie modyfikatorów rozmytych (bardzo, nieco, nie) do termów w regułach
Kształt krzywej funkcji przynależności termów może być modyfikowany poprzez użycie słów kluczowych języka FCL very, little, not.

System umożliwia tworzenie zmiennych wejściowych i wyjściowych
System zamienia zmienne bloku funkcji języka FCL na swoją reprezentację wewnętrzną, daje możliwość zmiany wartości zmiennych wejściowych i prawidłowo oblicza wartości zmiennych wyjściowych.

IV.2.5. Diagram przypadków użycia

Opisuje system z punktu widzenia użytkownika. Pokazuje co robi system, a nie jak to robi. Diagram ten sam w sobie nie daje nam wielu informacji, dlatego potrzebna jest do tego dokumentacja w postaci dobrze napisanego przypadku użycia. Poniżej znajduje się diagram przypadku użycia związanego z systemem FCL IDE:



Rys IV.2.5./1 Przykładowy diagram przypadków użycia aplikacji

IV.2.6. `Opis przypadków użycia`

Poniżej znajdują się dwa przykładowe przypadki użycia aplikacji:

Pełna nazwa podmiotu sporządzającego dokumentację	„Test przypadku użycia” Łukasz Zieliński	T01
Pełna nazwa systemu informatycznego:	FCL IDE	
Nazwa przypadku testowego	Zmiana danych w projekcie	

Testowana funkcjonalność	Edytowanie
Sposób dostępu	Należy uruchomić aplikacje poprzez wpisanie adresu strony internetowej. Po uruchomieniu aplikacji należy edytować projekt, utworzyć dwie zmienne wejściowe i jedną wyjściową oraz zależności między nimi. Otworzyć drzewo projektu. Wybrać zmienną wyjściową, otworzyć wykres 3D, sprawdzić poprawność wykresu z wprowadzonymi wartościami.
Dane wejściowe	Wprowadzenie skryptu
Dane wyjściowe	Poprawny wykres 3D

Pełna nazwa podmiotu sporządzającego dokumentację	„Test przypadku użycia” Łukasz Zieliński	T02
Pełna nazwa systemu informatycznego:	FCL IDE	
Nazwa przypadku testowego	Edytowanie projektu w celu wpisania innych zmiennych	
Testowana funkcjonalność	Raportowanie błędów	
Sposób dostępu	Należy uruchomić aplikację poprzez wpisanie adresu strony internetowej. Po uruchomieniu aplikacji należy edytować projekt. Zaznaczyć miejsca z zmiennymi wejściowymi i wyjściowymi, wpisać nowe zmienne. Przy wpisywaniu zrobiona została literówka. Pojawiły się komunikaty o błędach.	
Dane wejściowe	Edycja projektu	
Dane wyjściowe	Komunikaty o błędzie	

IV.2.7. Endpointy

Interfejs aplikacji napisany jest w całości w języku JavaScript. Wymaga jednak dostarczenia/przetwarzania danych po stronie serwera. W tym celu powstał backend na serwerze JBoss obsługujący endpointy do których odwołuje się JS. Format przesyłanych danych to JSON. Aplikacja wykorzystuje bibliotekę GSON do mapowania obiektów Javy na ten format. Docelowo wykorzystany będzie do tego celu framework JAX RS.

IV.2.7.1) Punkty dostępowe

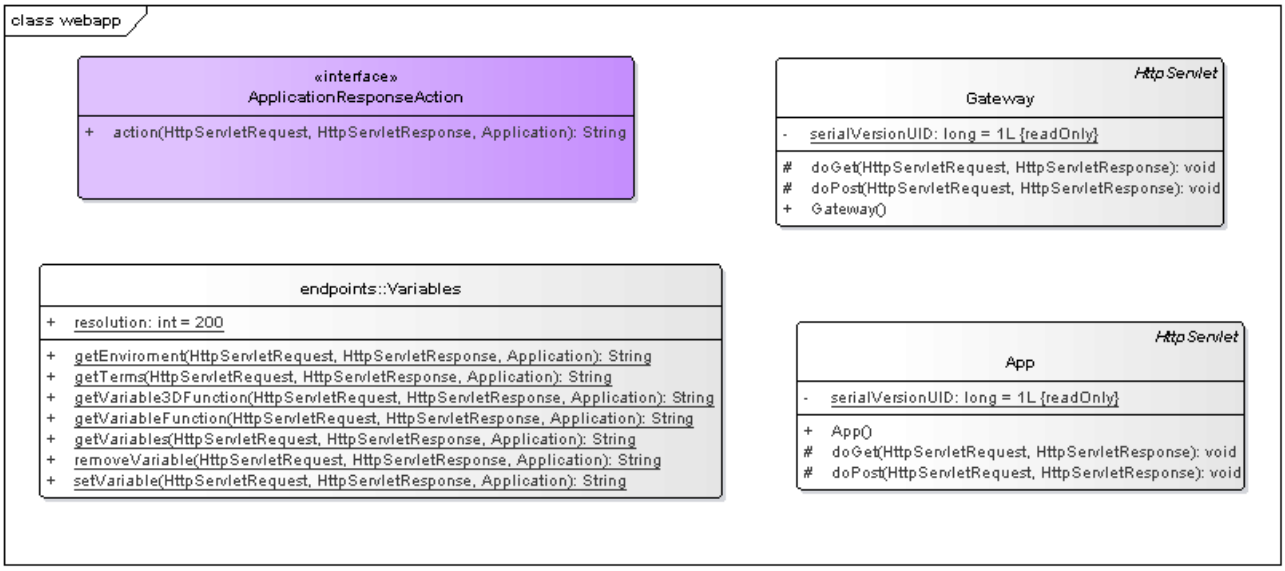
Utworzono dwa główne punkty dostępowe:

- /Gateway – służy do przesyłania kodu skryptu FCL, działa tylko w jedną stronę. Sukces zwraca html o treści 1
- /App – Przesyłanie pozostałych danych (błędy, wykresy, drzewo itp.), daje możliwość odwołania się do akcji poprzez parametr action.

IV.2.7.2) Struktura

Punkty dostępowe obsługiwane są przez servlety o tych samych nazwach. Servlet Gateway z racji wykonywania tylko jednej operacji zawiera się w całości w sobie. Servlet App zawiera request dispatcher, który przekierowuje dane requestu wraz z dodatkowymi danymi z sesji, do klas

implementujących funkcjonalny interfejs ApplicationResponseAction. W Javie 8 możemy zamiast implementacji klas podać metody, i tak też to zostało zrealizowane.



Rys IV.2.7.2)/1 Struktura aplikacji punktów dostępu.

IV.2.7.3) Przykład kodu

Realizacja implementacji ApplicationResponseAction z poprzedniego punktu, request dispatcher:

```
@WebServlet("/App")
public class App extends HttpServlet {
	private static final long serialVersionUID = 1L;
	protected void doGet(HttpServletRequest request,
		HttpServletResponse response) throws ServletException, IOException {
		doPost (request,response);
	}
	protected void doPost(HttpServletRequest request,
			HttpServletResponse response) throws ServletException,
IOException {
		HttpSession session = request.getSession();
		String actionString = request.getParameter("action");
		PrintWriter pw = response.getWriter();
		Application app = (Application) session.getAttribute("app");
		synchronized (app) {
		if (app==null) {
			pw.print("false");
			return;
		}
		Map<String,ApplicationResponseAction> dispatchMap = new
HashMap<String,ApplicationResponseAction> ();

		dispatchMap.put ("setVariable", Variables::setVariable);
		dispatchMap.put ("getErrorLog", LoggerEndpoint::getErrorLog);
		dispatchMap.put ("getEnviroment", Variables::getEnviroment);
		dispatchMap.put ("getTreeData", Tree::getTreeData);
		dispatchMap.put ("getTermsData", TermEndpoint::getTerms);
		dispatchMap.put ("getVariables", Variables::getVariables);
		dispatchMap.put ("remove-var", Variables::removeVariable);
		dispatchMap.put ("getTerms", Variables::getTerms);
		dispatchMap.put ("getVariableFunction", Variables::getVariableFunction);
		dispatchMap.put ("getVariable3DFunction",
Variables::getVariable3DFunction);
		ApplicationResponseAction action = dispatchMap.get(actionString);
```

```
                if (action != null) pw.write(action.action(request, response, app));
                else pw.write("404");
                }
        }
}
```

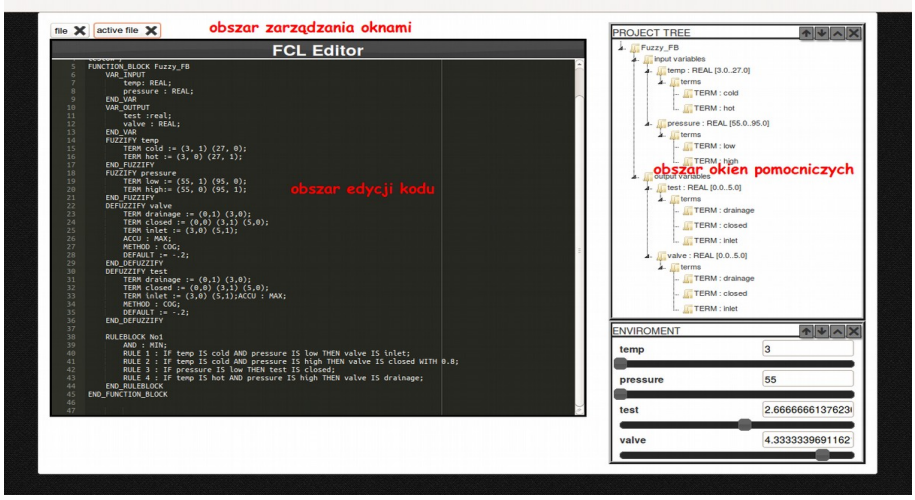
Implementacja :

```
public class Variables {
        //(...)
        public static String getEnviroment(HttpServletRequest request,
                        HttpServletResponse response, Application app) {
                try {
                        Enviroment env = app.getEnv();
                        env.updateRanges();
                        DefaultGsonMapper mapper = new DefaultGsonMapper(env);
                        return mapper.toJson();
                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                        return "false";
                } catch (IOException e) {
                        e.printStackTrace();
                        return "false";
                }
        }
}
```

IV.2.8. Interfejs użytkownika

Na aplikację składa się jeden widok użytkownika, na którym dostępne są wszystkie możliwe opcje dotyczące edycji dokumentu. Dodatkowo użytkownik ma możliwość otwarcia kilku okien przeglądarki, w których może edytować kilka dokumentów naraz. Główny widok dzieli się na następujące obszary:

- obszar zarządzania oknami
- obszar edycji kodu
- obszar okien pomocniczych



Rys IV.2.8./Główny widok aplikacji

IV.2.8.1) Obszar zarządzania oknami

Obszar ten składa się z przycisków. Przyciski po lewej stronie obszaru reprezentują otwarte okna dokumentów danego projektu. Przycisk po prawej stronie obszaru po kliknięciu rozwija listę wszystkich dokumentów otwartego projektu. Wybór (kliknięcie) jednego z nich otwiera ten

dokument. Przycisk aktywnego dokumentu (wyświetlony w obszarze edycji kodu) oznaczony jest ramką.

Rys IV.2.8.1)/1 Fragment służący do zarządzania oknami

IV.2.8.2) Obszar edycji kodu

Obszar składa się z okna edycji treści z numeracją wierszy. Obsługiwane są operacje:

kopiowania (CTRL + c)

wycinania (CTRL + x)

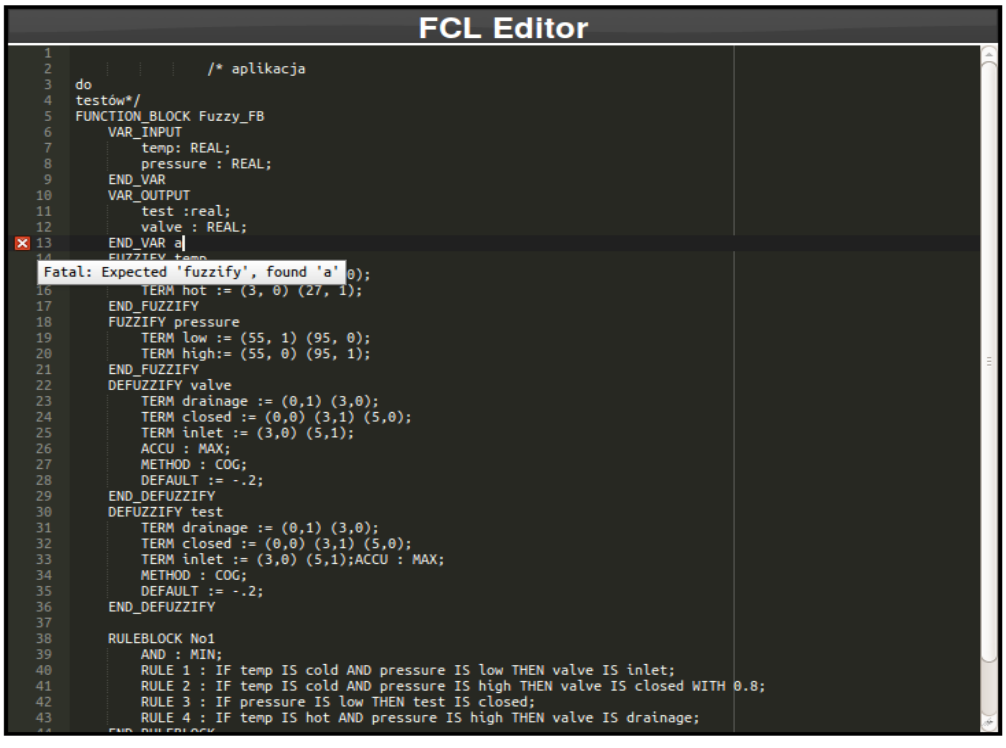
wklejania (CTRL + v)

cofania ostatniej operacji (CTRL + z)

przywracania ostatniej operacji (CTRL + y)

zaznaczania całej treści (CTRL + a)

Zaprzestanie edycji na dłużej niż pięć sekund powoduje przesłanie kodu do backendu aplikacji, kompilacje kodu, następnie odświeżenie wszystkich okien na ekranie. W przypadku popełnienia błędu w skrypcie FCL edytor po zaprzestaniu edycji dokumentu oznaczy linijkę z błędem krzyżykiem. Po najechaniu myszką na krzyżyk otrzymamy dymek z informacją o szczegółach błędu jak na rys. poniżej:

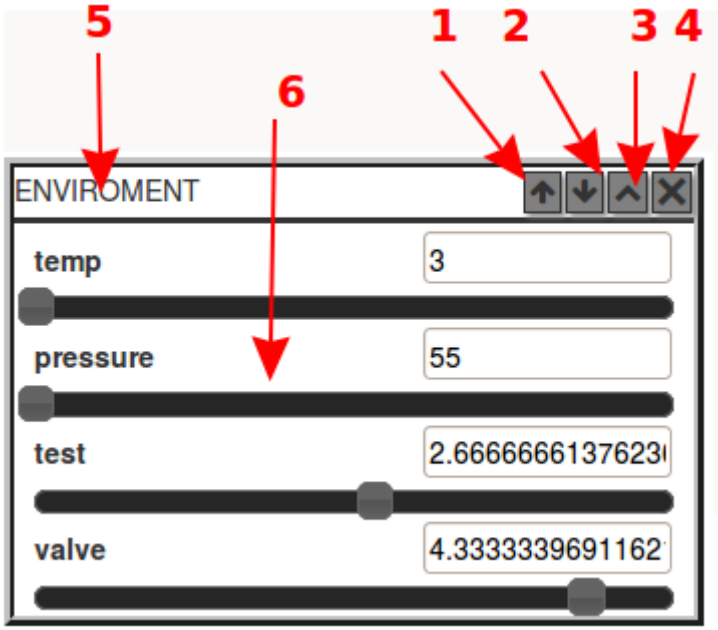


Rys IV.2.8.2)/1 Edytor kodu

IV.2.8.3) Obszar okien pomocniczych

Obszar pozwala na wyświetlenie i zarządzanie oknami pomocniczymi. Okna te posiadają tytuł, rząd

przycisków umożliwiających zarządzanie oknem oraz zawartość. Jeśli zawartość okna nie mieści się w oknie, dołączane są automatycznie paski przewijania. Dodatkowo każde okno ma ruchomą dolną krawędź, co pozwala na zmianę jego wielkości.



Rys IV.2.8.3)/1 Budowa okna pomocniczego

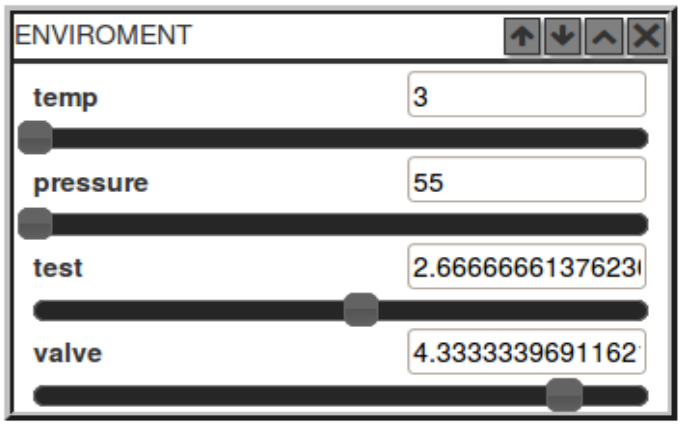
Elementy składowe z rysunku IV.2.8.3)/1 :

1. Zamień okno z oknem znajdującym się wyżej (nie działa dla pierwszego okna)
2. Zamień okno z oknem znajdującym się niżej (nie działa dla ostatniego okna)
3. Zminimalizuj okno (okno zminimalizowane na rys IV.2.8.3)/2)
4. Zamknij okno
5. Tytuł okna
6. Obszar zawartości okna

Rys IV.2.8.3)/2 Okno w stanie zminimalizowanym

Okna pomocnicze zostaną omówione w kolejnych punktach.

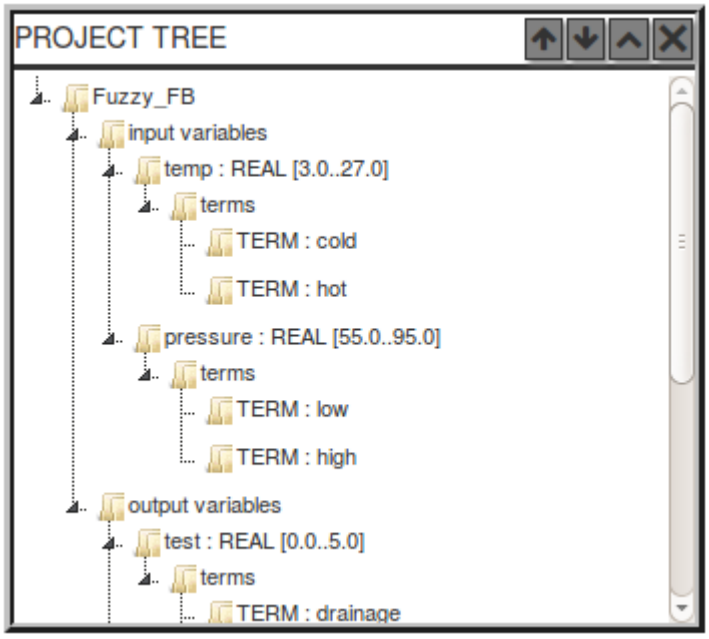
IV.2.8.4) Okno edycji zmiennych



Rys IV.2.8.4)/1 Okno edycji zmiennych (środowiska)

Okno pozwala na zmianę wartości wszystkich zmiennych utworzonych w całym projekcie. Zmiana następuje po przesunięciu suwaka lub wpisaniu wartości z klawiatury. Po zmianie wartości następuje przeładowanie (z małą częstotliwością) wartości zmiennych wyjściowych. Pozwala to na obserwację „na żywo”, jak opracowywany system ewaluuje zmienne. Dodatkowo zmiana wartości wpływa na wszystkie okna, których zawartość zależy od aktualnego stanu środowiska. Po zaprzestaniu zmian wartości następuje przeładowanie tych okien.

IV.2.8.5) Okno drzewa projektu



Rys IV.2.8.5)/1 Okno drzewa projektu

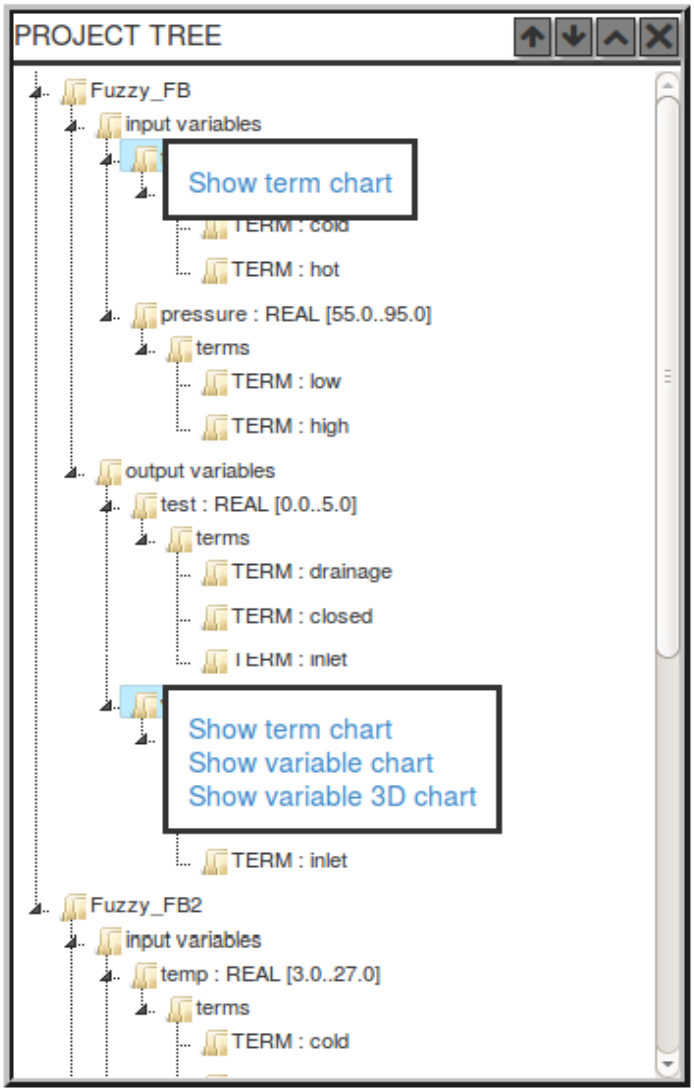
Okno przedstawia strukturę wygenerowaną przez skrypty FCL w projekcie. Wyświetlają się:

- nazwy bloków funkcji
- nazwy zmiennych wejściowych wraz z zakresami
- nazwy zmiennych wewnętrznych wraz z zakresami
- nazwy zmiennych wyjściowych wraz z zakresami

Przy każdej zmiennej widoczna jest lista termów jej przypisanych. Kliknięcie na zmienną

wejściową lub wyjściową otwiera menu kontekstowe (tutaj pokazane są obydwa typy, opis w kolejności):

1. dla zmiennej wejściowej:
    1. okno termów wej. - pokazuje funkcje przynależności termów tej zmiennej wraz z wartością aktywacji
2. dla zmiennej wyjściowej:
    1. okno termów wyj. - pokazuje funkcje przynależności termów tej zmiennej oraz funkcję przynależności zakumulowanych przesłanek odpalonych reguł i singleton wartości tej zmiennej
    2. okno funkcji zmiennej – pokazuje wykres zmian wartości tej zmiennej w funkcji jednej zmiennej wejściowej
    3. okno funkcji zmiennej 3D – pokazuje wykres zmian wartości tej zmiennej w funkcji dwóch zmiennych wejściowych.

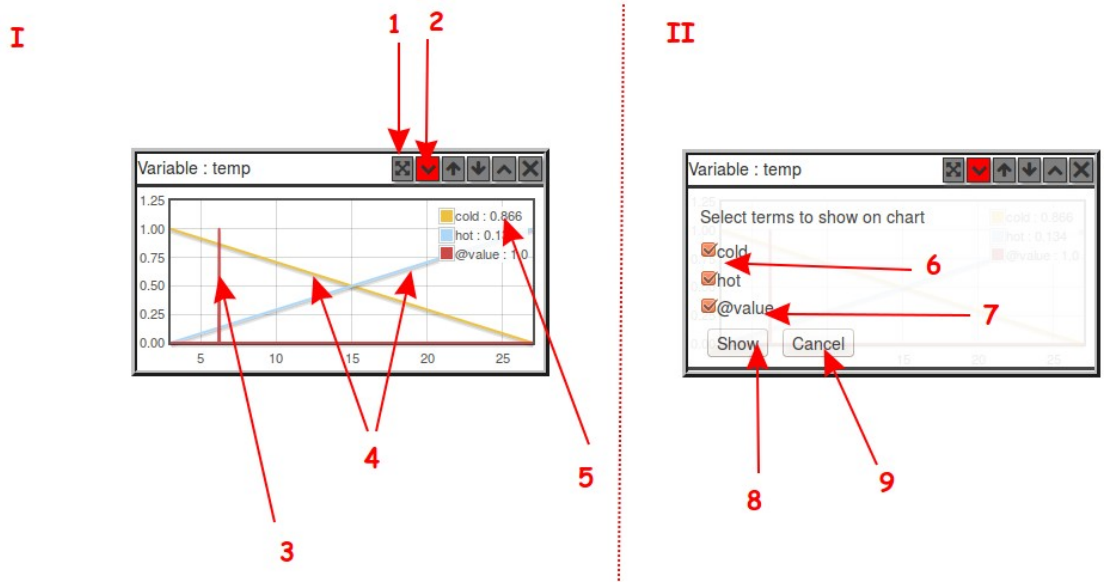


Rys IV.2.8.5)/1 Okno drzewa z menu kontekstowym

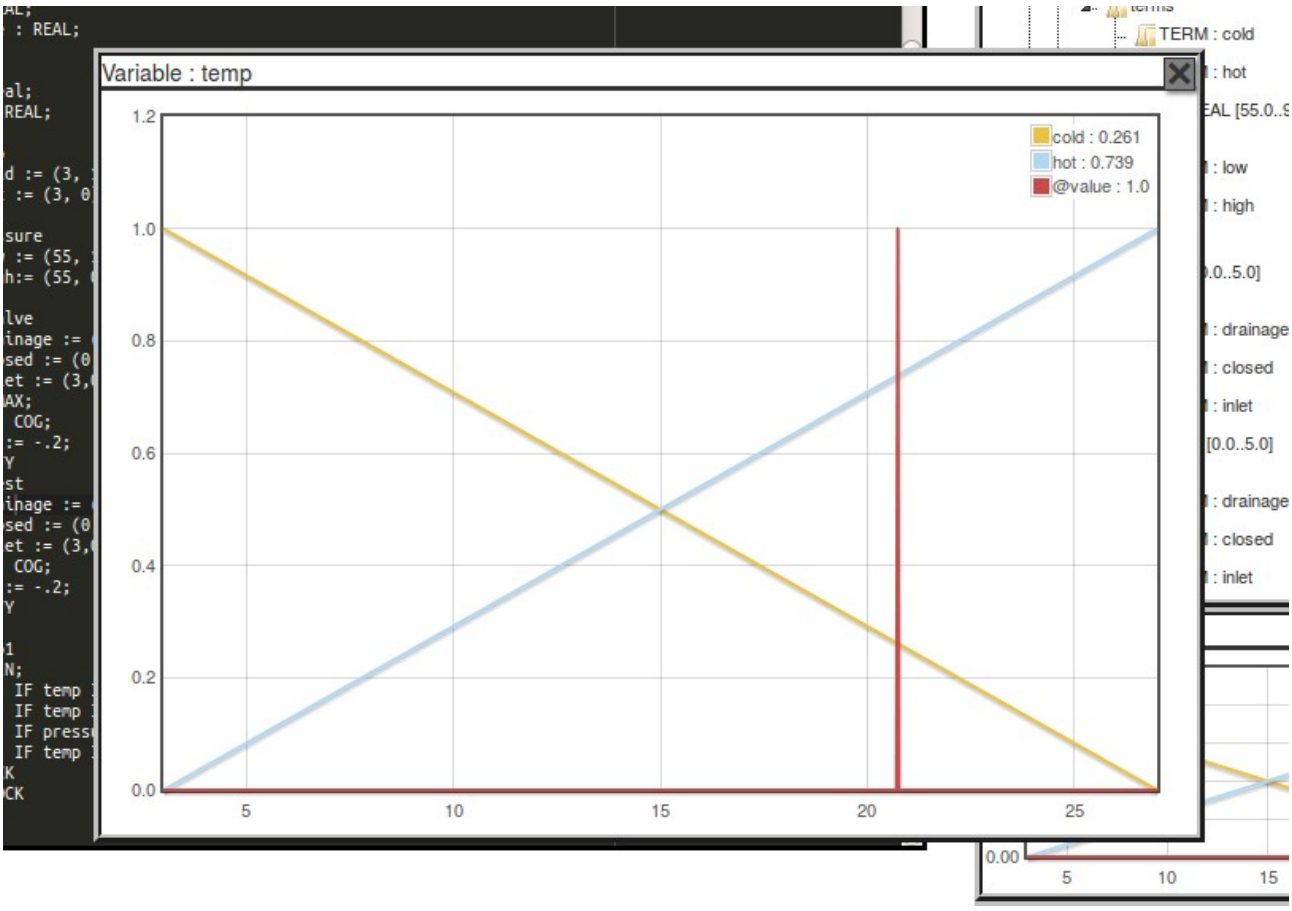
IV.2.8.6) Okno termów zmiennej wejściowej

Okno posiada dwa dodatkowe przyciski w pasku nawigacji. Odpowiadają one za powiększenie wykresu oraz rozwinięcie menu z dodatkowymi opcjami. Okno w podstawowej wersji przedstawia obszar I poniższego rysunku, okno z rozwiniętym menu znajduje się na obszarze II. Elementy okna:

1. przycisk powiększenia – powiększa wykres do szerokości 500px
2. przycisk rozwinięcia menu – rozwija menu przedstawione w obszarze II
3. singleton przedstawiający aktualną wartość zmiennej
4. wykresy funkcji przynależności termów tej zmiennej wejściowej
5. aktualne wartości funkcji aktywacji poszczególnych termów
6. lista termów zmiennej – pozwala na wybranie funkcji, które mają być przedstawione na wykresie
7. singleton przedstawiający wartość funkcji – pozwala na ukrycie bądź pokazanie tej wartości
8. przycisk akceptacji wyboru – zwija okno menu i zatwierdza wybór funkcji do pokazania
9. przycisk anulowania wyboru – zwija okno menu i neguje wybrane opcje



Rys IV.2.8.6)/1 Wygląd okna termów zmiennej wejściowej

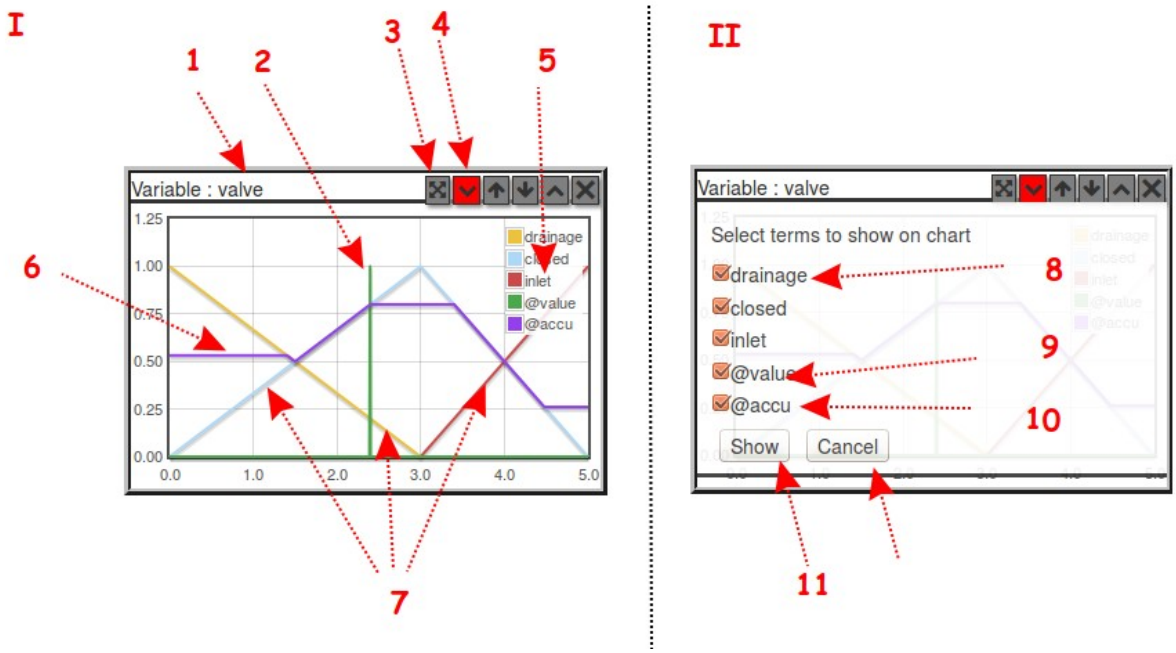


Rys IV.2.8.6)/2 Powiększony wykres

IV.2.8.7) Okno termów zmiennej wyjściowej

Okno wykresu jest podobne do okna termów zmiennej wejściowej, zawiera jednak kilka dodatkowych elementów. Przedstawia wykres zakumulowanych funkcji przynależności, powstałych w wyniku uruchomienia reguł, dzięki którym wyliczono ostatnią wartość zmiennej. Elementy okna:

1. nazwa zmiennej w nagłówku
2. singleton przedstawiający aktualną wartość zmiennej
3. przycisk powiększający wykres do szerokości 500px
4. przycisk rozwijający dodatkowe menu (obszar II)
5. lista przedstawionych funkcji przynależności (nazwy termów)
6. wykres przedstawiający zakumulowaną funkcję przynależności
7. wykresy przynależności poszczególnych termów tej zmiennej
8. lista umożliwiająca wybór przedstawionych termów
9. wyświetla singleton wartości zmiennej
10. wyświetla zakumulowaną funkcję przynależności
11. zatwierdza wybór i zwija menu
12. anuluje wybór i zwija menu

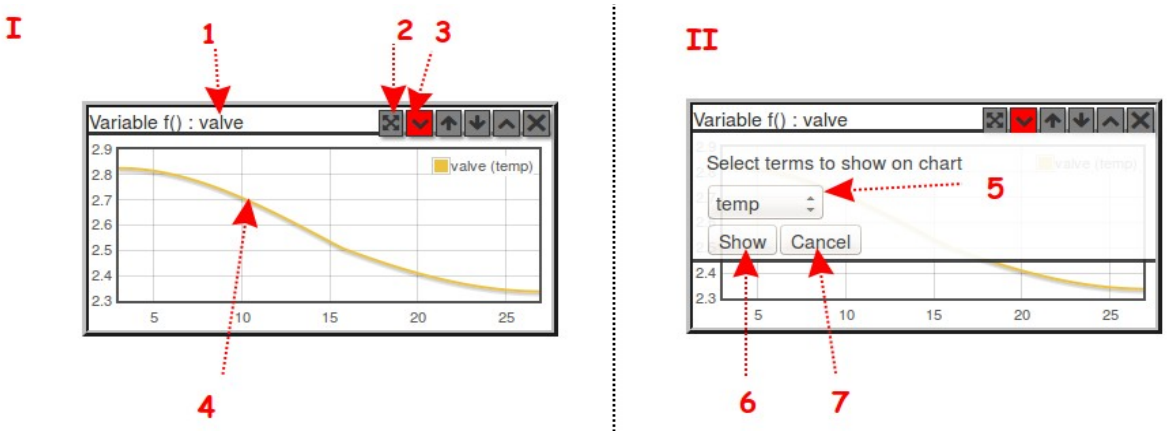


Rys IV.2.8.7)/1 Okno termów zmiennej wyjściowej

IV.2.8.8) Okno wartości zmiennej wyjściowej w funkcji wybranej zmiennej wejściowej

Okno przedstawia wykres funkcji zmiennej jak w tytule. Posiada dwa dodatkowe przyciski, podobnie jak poprzednie okna wykresów termów. Jeżeli zmienna wyjściowa zależy od więcej niż jednej zmiennej wejściowej, stan pozostałych zmiennych przy przeliczaniu wykresu funkcji jest taki, jak aktualny stan środowiska zmiennych. Okno składa się z następujących elementów:

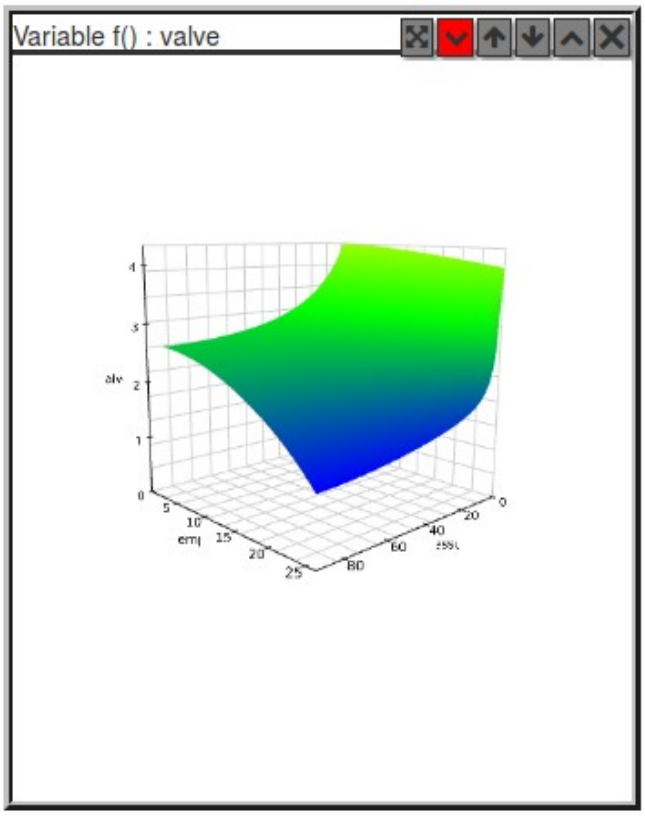
1. nazwa zmiennej wyjściowej w nagłówku
2. przycisk powiększający wykres do szerokości 500px
3. przycisk rozwijający menu dodatkowe (obszar II)
4. wykres funkcji zmiennej wyjściowej
5. wybór zmiennej wejściowej od której będzie rysowany wykres
6. przycisk zatwierdzający wybór i zwijający menu
7. przycisk anulujący wybór i zwijający menu



Rys IV.2.8.8)/1 Okno wykresu funkcji zmiennej wyjściowej

IV.2.8.9) Okno wykresu funkcji zmiennej wyjściowej w przestrzeni dwóch zmiennych wejściowych.

Okno przedstawia trójwymiarowy wykres zależności przestrzennej jednej zmiennej wyjściowej od dwóch zmiennych wejściowych. Podobnie, jak poprzednie okna, posiada przycisk powiększenia do szerokości 500px oraz rozwijane menu wyboru zmiennych wejściowych, dla których rysowany jest wykres. Jeśli zmienna zależy od więcej niż jednej zmiennej wejściowej, pozostałe zmienne, przy wyliczaniu wartości funkcji, zachowują stan ze środowiska. Wykres jest aktualizowany przy każdej zmianie kodu FCL lub wartości dowolnej zmiennej wejściowej.



Rys IV.2.8.9)/1 Okno wykresu wartości zmiennej w przestrzeni dwóch zmiennych

V. Przykłady skryptów FCL

V.1. Zraszanie trawnika

Zraszanie trawnika było już omówione pokrótce w rozdziale II.6. Tutaj będzie przedstawione z punktu widzenia programisty FCL - użytkownika aplikacji.

V.1.1. Opis

System, którego zadaniem jest sterowanie głównym zaworem systemu nawadniania trawnika. Sterowanie ma się odbywać na podstawie przygotowanych przez zawodowych ogrodników danych. Wpływ na siłę zraszania (otwarcie zaworu) mają wilgotność i temperatura powietrza.

V.1.2. Kod FCL

```
FUNCTION_BLOCK Fuzzy_FB
	VAR_INPUT
		wilgotnosc: REAL;
		temperatura : REAL;
	END_VAR
	VAR_OUTPUT
		zraszanie : REAL;
	END_VAR
	FUZZIFY temperatura
		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);
	END_FUZZIFY
	FUZZIFY wilgotnosc
		TERM niska := (0, 1) (20, 1) (35, 0);
		TERM srednia := (10, 0) (30, 1) (60,0);
		TERM duza :=  (40,0) (70, 1) (100, 1);
	END_FUZZIFY
	DEFUZZIFY zraszanie
	    TERM brak := 0;
		TERM slabe := 20;
		TERM srednie := 40;
		TERM silne := 60;
		ACCU : MAX;
		METHOD : COS;
		DEFAULT := 20;
	END_DEFUZZIFY
	RULEBLOCK No1
RULE 1:if ( wilgotnosc is duza or wilgotnosc is srednia or wilgotnosc is niska)
                  and 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;
AND : prod;
	END_RULEBLOCK
END_FUNCTION_BLOCK
```

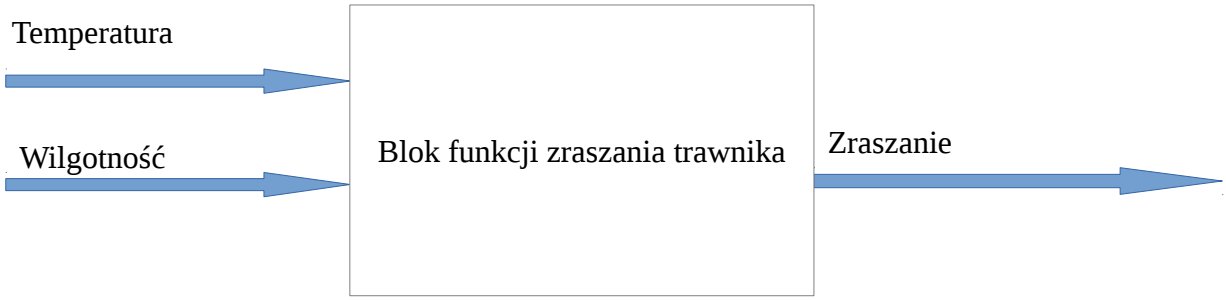
V.1.3. Omówienie

Skrypt definiuje na początku blok funkcji o nazwie Fuzzy_FB. Dalej znajduje się deklaracja zmiennych. Zadeklarowane są dwie zmienne wejściowe:

- temperatura
- wilgotność

oraz jedna zmienna wyjściowa:

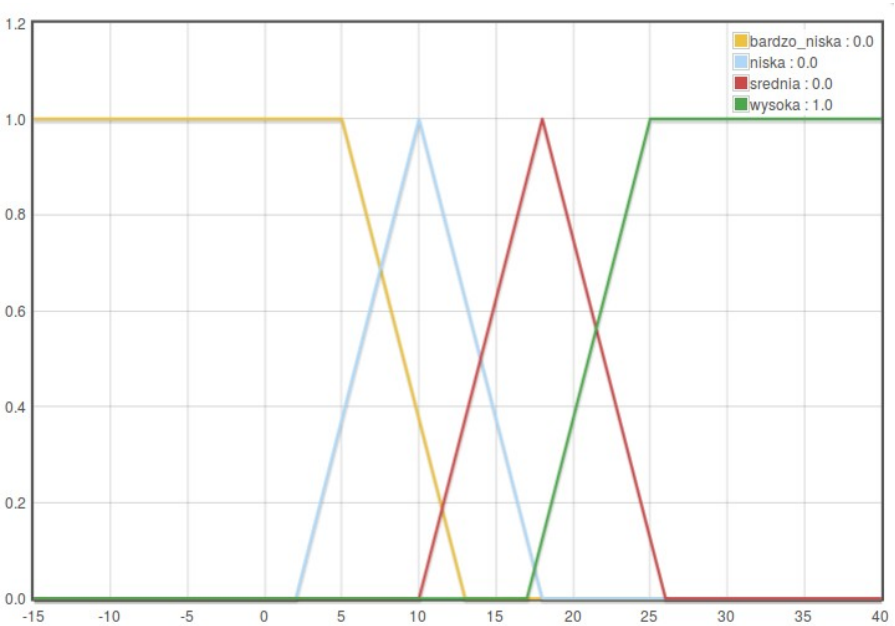
- zraszanie



Rys V.1.3./1 Schemat bloku zraszania trawnika

Kolejnymi elementami skryptu są definicje funkcji przynależności termów zmiennych wejściowych. Zdefiniowane są za pomocą listy punktów, przez które przechodzą kolejne odcinki funkcji. Ich bloki oznaczone są dyrektywą FUZZIFY. Dla zmiennej temperatura tworzone są kolejno termy:

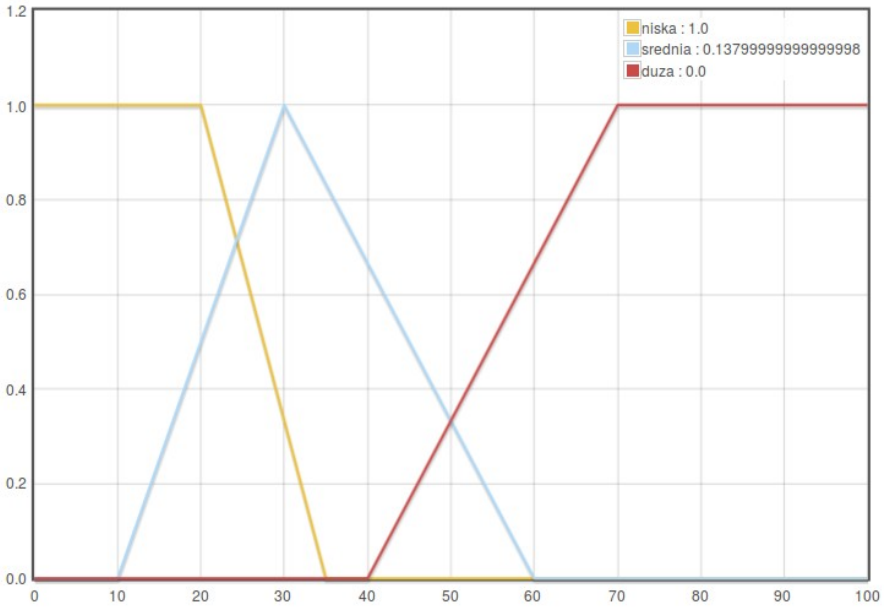
- bardzo niska
- niska
- średnia
- wysoka



Rys V.1.3./2 Wykres termów zmiennej temperatura

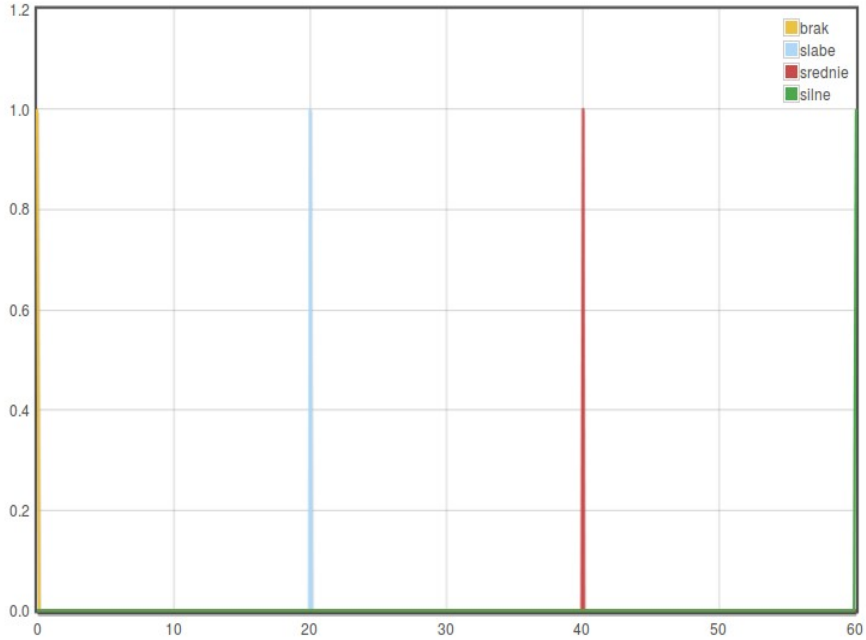
Dla zmiennej wilgotność tworzone są kolejno termy:

- niska
- średnia
- duża



Rys V.1.3./3 Wykres termów zmiennej wilgotność

Kolejny blok zaczyna się słowem kluczowym DEFUZZIFY. Zawiera on definicje parametrów związanych z wyostrzaniem zmiennej wyjściowej zraszanie oraz funkcje przynależności kolejnych termów tej zmiennej.



Rys V.1.3./4 Wykres termów zmiennej zraszanie

Dodatkowo wyszczególnione są:

Metoda akumulacji termów rezultatów poszczególnych reguł – MAX

Metoda defuzyfikacji zmiennej – Center of Singletons

Wartość początkowa – 20

Kolejnym i ostatnim elementem jest blok reguł. Zawiera 10 reguł, które w całości pokrywają przestrzeń zmiennych wejściowych:

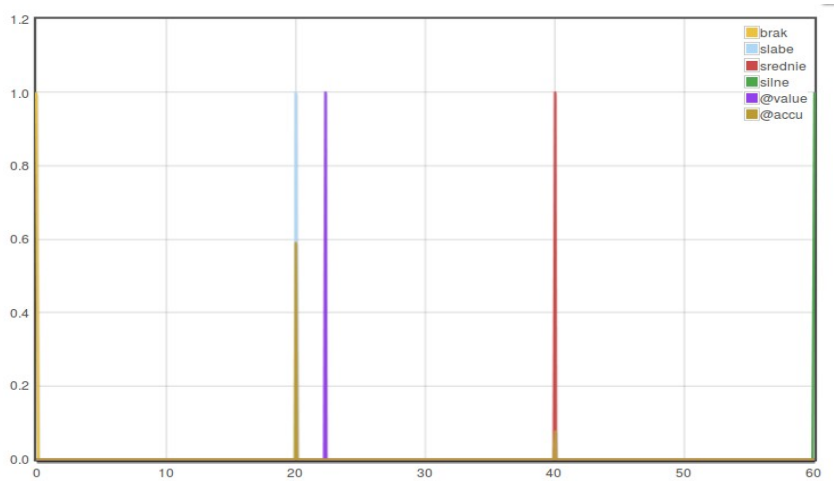
Wilgotność / Temperatura	niska	średnia	duża
bardzo_niska	R1 – brak	R1 – brak	R1 -brak
niska	R8 – slabe	R5 – brak	R2 – brak
średnia	R9 – srednie	R6 – slabe	R3 – brak
wysoka	R10 – silne	R7 – srednie	R4 – slabe

Rys V.1.3./5 Tabela pokrycia przestrzeni zmiennych wejściowych

I odpowiednio reguły:

```
RULE 1:if ( wilgotnosc is duza or wilgotnosc is srednia or wilgotnosc is niska)
                  and 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;
```

Przykład obliczonej przez program wartości zmiennej wraz z obrysem zakumulowanych termów dla zmiennych wejściowych:
wilgotność = 56.4
temperatura = 27.35
zraszanie = 23.6



Rys V.1.3./6 Wykres zakumulowanych termów

Brązowym kolorem oznaczona jest funkcja akumulacji odpalonych reguł, fioletowy przedstawia środek ciężkości tej akumulacji – aktualną wartość zmiennej.

V.2. Auto na odcinku drogi

Problem, który ma rozwiązać ta aplikacja, polega na sterowaniu przyśpieszeniem pojazdu, który ma przejechać możliwie najszybciej, ale bezpiecznie, pewien określony odcinek drogi. Założenia:
Trasa do przejechania 1km
Maksymalne przyśpieszenie auta : 30000 km/h^2
Maksymalne hamowanie auta: 60000 km/h^2
Optymalna prędkość na trasie – 60 km/h

V.2.1. Dane

Zmienne wejściowe:

- prędkość 0 – 100 [km/h]
- odległość od celu 0 – 1[km]

Zmienne wyjściowe:

- przyśpieszenie (-60) – 30 [Mm/h^2]

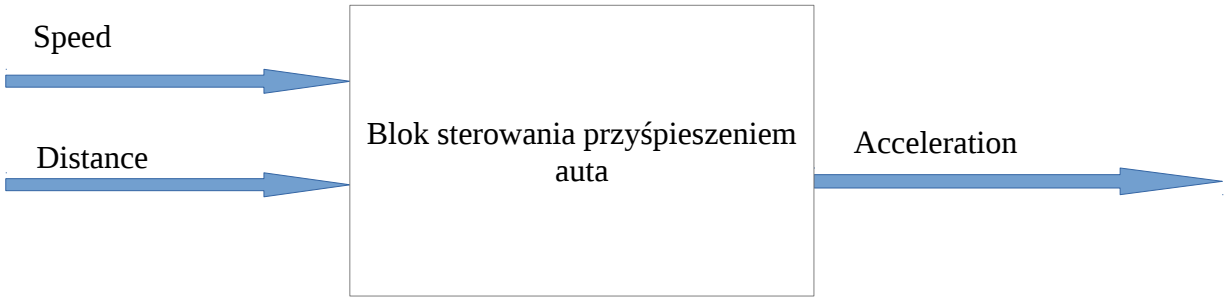
V.2.2. Kod FCL

```
FUNCTION_BLOCK Fuzzy_FB
	VAR_INPUT
		speed: REAL;
		distance : REAL;
	END_VAR
	VAR_OUTPUT
		acceleration : REAL;
	END_VAR
	FUZZIFY speed
	    TERM slow := (0,1) (30,1) (50,0);
	    TERM optimal := (35,0) (60,1) (85,0);
	    TERM fast := (70,0) (90,1) (100,0);
	    TERM too_fast := (90,0) (110,1) (120,1);
	END_FUZZIFY
	FUZZIFY distance
		TERM far := (0, 1) (600, 1) (800, 0);
		TERM near := (600,0) (800,1) (1000, 0);
```

```
            TERM close := (800,0) (900,1) (1000,1);
        END_FUZZIFY
        DEFUZZIFY acceleration
            TERM stop := (-60, 1) (-30,0);
            TERM break := (-40, 0) (-20,1) (0,0);
            TERM none := (-10,0) (0,1) (10,0);
            TERM rush := (0, 0) (10,1) (20,0);
            TERM extreme := (25,0) (30,1);
            ACCU : MAX;
            METHOD : COG;
            DEFAULT := 20;
        END_DEFUZZIFY
        RULEBLOCK No1
         AND : prod;
         rule 1 : if speed is very slow and distance is far then acceleration is extreme;
         rule 2 : if speed is slow and distance is far then acceleration is rush;
         rule 3 : if speed is optimal and distance is far then acceleration is none;
         rule 4 : if speed is fast and distance is far then acceleration is break;
         rule 5 : if speed is too_fast and distance is far then acceleration is stop;

         rule 6 : if speed is very slow and distance is near then acceleration is rush;
         rule 7 : if speed is slow and distance is near then acceleration is none;
         rule 8 : if speed is optimal and distance is near then acceleration is none;
         rule 9 : if speed is fast and distance is near then acceleration is break;
         rule 10 : if speed is too_fast and distance is far then acceleration is stop;

         rule 11 : if speed is slow and distance is close then acceleration is break;
         rule 12 : if speed is optimal and distance is close then acceleration is break;
         rule 13 : if speed is fast and distance is close then acceleration is stop;
         rule 14 : if speed is too_fast and distance is far then acceleration is stop;
        END_RULEBLOCK
END_FUNCTION_BLOCK
```

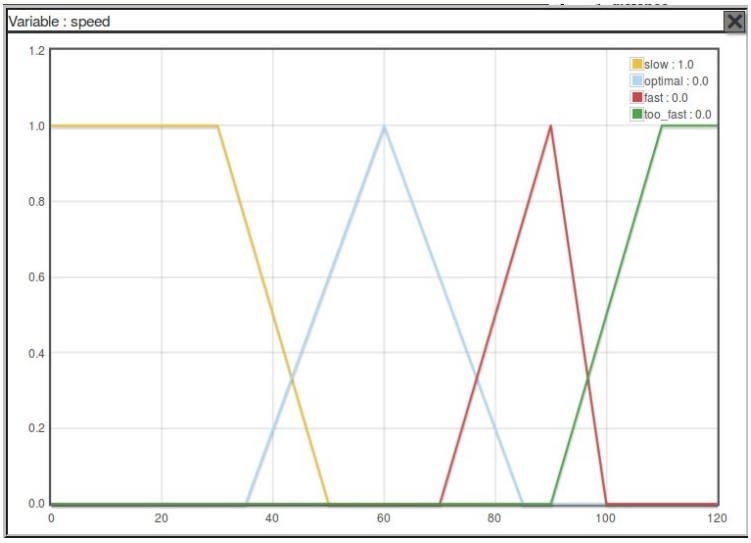
V.2.3. Omówienie

Zmienne skryptu pokrywają się ze zmiennymi wynikającymi z założeń problemu. Struktura blokowa systemu przedstawia się następująco:

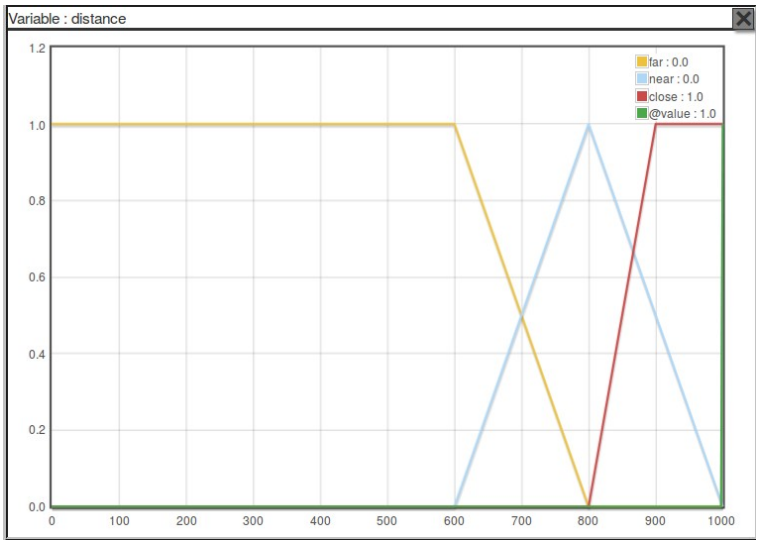


Rys V.2.3./1 Schemat bloku sterowania przyśpieszeniem auta

Zmienne wejściowe charakteryzują następujące funkcje przynależności termów:

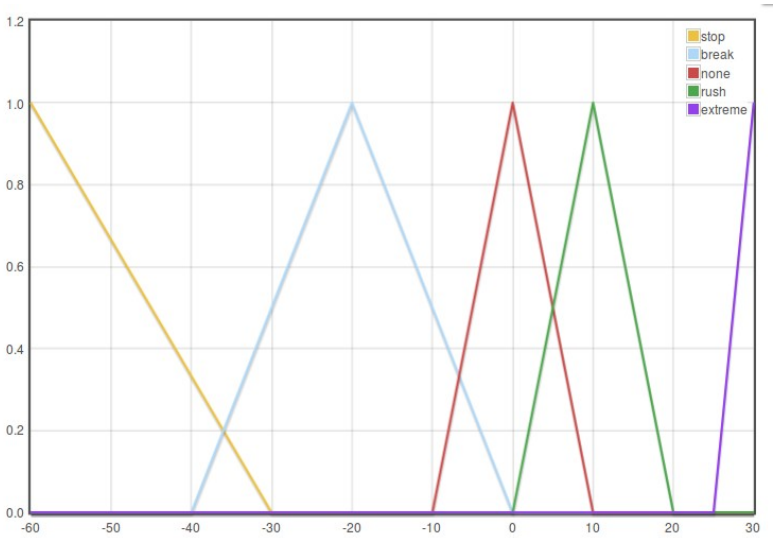


Rys V.2.3./2 Funkcje przynależności zmiennej speed



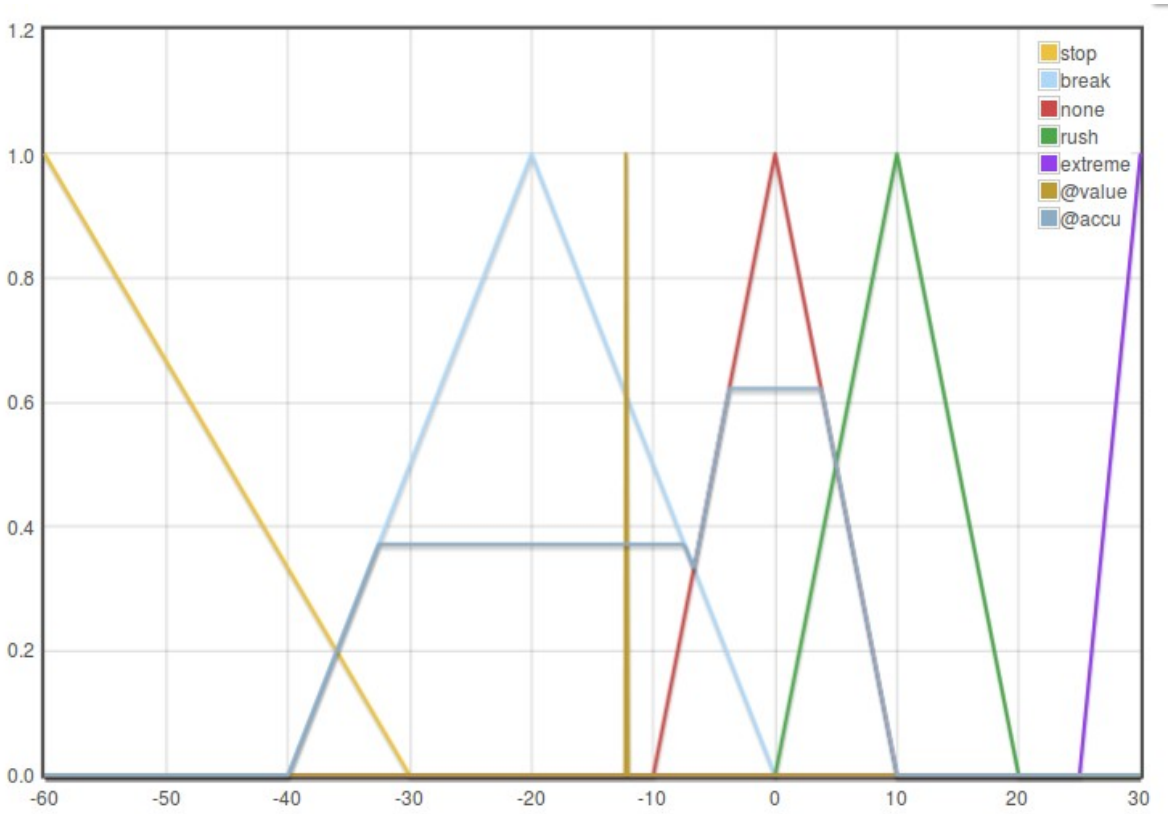
Rys V.2.3./3 Funkcje przynależności zmiennej distance

Zmienna wyjściowa przyśpieszenie posiada następujące funkcje przynależności termów:



Rys V.2.3./4 Funkcje przynależności zmiennej acceleration

W tym przypadku termy zmiennej wyjściowej są funkcjami liniowymi, do defuzyfikacji wykorzystano metodę center of gravity, w celu obliczenia środka ciężkości zakumulowanych termów. Przykład działania tej metody:



Rys V.2.3./5 Akumulacja i defuzyfikacja przyśpieszenia

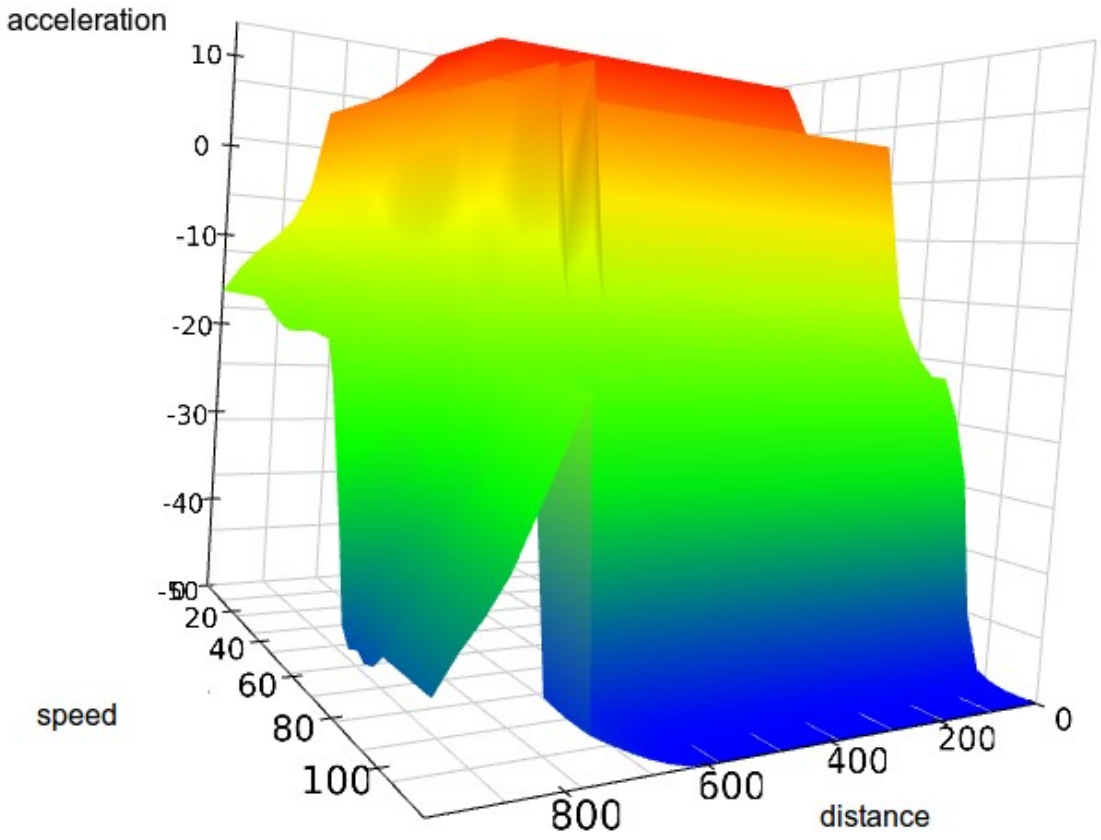
Kolorem ciemnoniebieskim oznaczono obrys zakumulowanych termów, wynikających z odpalonych reguł, kolor brązowy określa położenie środka ciężkości wykresu.

Reguły systemu w pełni pokrywają przestrzeń zmiennych wejściowych:

Speed / Distance	far	near	close
very slow	R1 - extreme	R6 - rush	N/A
slow	R2 - rush	R7 - none	R11 - break
optimal	R3 - none	R8 - none	R12 - break
fast	R4 - break	R9 - break	R13 - stop
too_fast	R5 - stop	R10 - stop	R14 - stop

Rys V.2.3./6 Tabela pokrycia przestrzeni zmiennych wejściowych

Tak przedstawia się wykres funkcji zmiennej wyjściowej acceleration, w zależności od zmiennych speed i distance:



Rys V.2.3./7 Wykres zmiennej wyjściowej acceleration

VI. Wnioski

Celem pracy było stworzenie środowiska programistycznego języka FCL, które można wykorzystać w celach edukacyjnych. Części wymagań funkcjonalnych aplikacji dotyczących perezystencji nie udało się zrealizować. Pozostałe wymagania zostały zrealizowane i pozwalają skutecznie edytować i badać na bieżąco kreowane środowisko rozmyte.

Praca nad aplikacją pozwoliła mi zrealizować moje cele – rozwinięcie umiejętności obiektowej reprezentacji algorytmów, głębsze poznanie języka programowania – Javy, w najnowszej na ten czas wersji – 1.8 wraz z niektórymi jej nowymi możliwościami takimi, jak Stream API i lambdy. Pomogła również w poznaniu podstaw JavaScript-u, który obsługuje część frontendową aplikacji.

Dodatkowo rozwinąłem swoje umiejętności dokumentowania kodu – elementu niezbędnego w programowaniu, a jednak często pomijanego przez programistów z powodu „braku czasu”. Niestety czas właśnie pokazuje, że brak dokumentacji przyczynia się do ostrej krzywej nauki programistów, którzy po raz pierwszy stykają się z projektem. Mam nadzieję, że ta umiejętność bardzo mi się przyda przy kolejnych projektach.

VII. Bibliografia

1) https://pl.wikipedia.org/wiki/Logika_tr%C3%B3jwarto%C5%9Bciowa
2) https://en.wikipedia.org/wiki/De_Morgan%27s_laws
3) RescherN. : Many-ValuedLogic; New York, McGraw-Hill, 1969
4) Rutkowski L.: Metody i techniki sztucznej inteligencji
5) kolegia.sgh.waw.pl/pl/KZiF/publikacje/koik/.../2013_NR_5(158).pdf
6) www.mimuw.edu.pl/~son/sysdec/materials/Sterowanie%20rozmyte.pdf
7) home.agh.edu.pl/~ligeza/wiki/_media/ke:fuzzy-control-tecza.pdf?...
8) www.cs.put.poznan.pl/mhapke/OEiG1.pdf
9) Mamdani, E. H., Application of fuzzy algorithms for the control of a simple dynamic plant. In Proc IEEE (1974)
10) Andrzej Piegat; „Modelowanie i sterowanie rozmyte”; Akademicka Oficyna Wydawnicza EXIT, Warszawa 1999
11) http://www.fuzzytech.com/binaries/ieccd1.pdf
12) https://en.wikipedia.org/wiki/Fuzzy_Control_Language