STUDIA INFORMATICA 2010
Volume 31 Number 2A (89)

Przemysław KUDŁACIK
Uniwersytet Śląski, Instytut Informatyki

STRUKTURA BAZY WIEDZY W BIBLIOTECE FUZZLIB

Streszczenie. Biblioteka FUZZLIB to zbiór narzędzi pozwalających tworzyć i zarządzać rożnymi systemami rozmytymi za pomocą prostego w użyciu interfejsu. Szczególnie ułatwiono konfigurację i zarządzanie regułowej bazy wiedzy opartej na strukturach dynamicznych. Niniejszy artykuł opisuje zastosowane rozwiązania oraz interfejs programistyczny.

Słowa kluczowe: regułowe systemy rozmyte, baza wiedzy, wnioskowanie przybliżone

STRUCTURE OF KNOWLEDGE BASE IN FUZZLIB LIBRARY

Summary. FUZZLIB library provide a set of tools that let to create and manage diverse fuzzy systems with an easy to use interface. Configuration and management of a rule base, based on dynamic structures, is especially simplified. Article describes developed solutions and program interface.

Keywords: rule based fuzzy systems, knowledge base, approximate reasoning

1. Wprowadzenie

Biblioteka FUZZLIB [2] jest uniwersalnym zestawem modułów programistycznych projektowanych z myślą o implementacji w popularnych obiektowych językach programowania[1]. Głównym przeznaczeniem biblioteki jest tworzenie regułowych systemów rozmytych wykorzystujących różne rozwiązania wnioskowania rozmytego[2] [8 i 9]. Jej cechą szczególną

[1] Obecnie istnieje implementacja w języku C++. Trwają jednak prace nad udostępnieniem modułów biblioteki w ramach języków Java oraz C#.

[2] W chwili obecnej dostępne są podejścia z logiczną i koniunkcyjną interpretacją reguły jeżeli-to wprowadzone przez Zadeha [7], Mamdaniego [4] oraz Baldwina [1].

jest odcinkowo-liniowa reprezentacja funkcji przynależności zbiorów rozmytych [2]. Pozwala ona w prosty sposób definiować funkcje o dowolnym przebiegu z określoną dokładnością. Ze względu na wektorową postać opisu funkcji większość algorytmów charakteryzuje logarytmiczna i liniowa czasowa złożoność obliczeniowa [2].

Niniejszy artykuł opisuje rozwiązanie problemu formatu danych przechowujących bazę wiedzy, którą zorganizowano w ramach dynamicznej struktury pojedynczych reguł oraz dynamicznej struktury ich zbioru.

2. Struktura informacji w rozmytym systemie regułowym

Najważniejszym elementem wielu systemów rozmytych, z użytkowego punktu widzenia, jest baza reguł, czyli „wiedza" systemu, która definiuje jego działanie. Stanowi ją zbiór reguł jeżeli-to, które mogą być określone bezpośrednio np. przez eksperta, bądź utworzone automatycznie (wydobyte z danych), co ma miejsce w przypadku systemów uczących się. Bazę reguł **R** w postaci kanonicznej dla systemu z N wejściami, składającą sie z I rozmytych reguł jeżeli-to, opisuje się następująco [3]:

$$\mathbf{R} = \left\{R^{(i)}\right\}_{i=1}^{I} = \left\{ jezeli \left(\bigwedge_{n=1}^{N} \mathbf{x}_n \ jest \ A_n^{(i)} \right), to \ \mathbf{y} \ jest \ B^{(i)} \right\}_{i=1}^{I}, \tag{1}$$

gdzie **x** i $A^{(i)}$ reprezentują odpowiednio zmienne wejściowe oraz przypisane im w (i)-tej regule przesłanki. Natomiast **y** i $B^{(i)}$ określają odpowiednio wyjście systemu oraz konkluzję reguły (i). Symbol $\wedge$ oznacza spójnik „I" kolejnych przesłanek. W ogólnym przypadku przesłanki mogą być również łączone za pomocą spójnika „LUB", dana reguła może być określona dla mniejszej liczby wejść oraz system może posiadać więcej wyjść. Jednak każdy z tych przypadków można sprowadzić do postaci kanonicznej odpowiednio poprzez rozbicie reguł i wprowadzenie dodatkowych, dodanie odpowiednio opisanych przesłanek oraz dekompozycję systemu z wieloma wyjściami na większą liczbę systemów [3].

Projektowane rozwiązanie powinno uwzględnić możliwość tworzenia reguł najbardziej rozbudowanych, w których można określić dowolną liczbę przesłanek w każdej regule oraz stosować różne spójniki. Zatem równanie (1) przyjmie w tym przypadku następującą postać:

$$\mathbf{R} = \left\{R^{(i)}\right\}_{i=1}^{I} = \left\{ jezeli \left(\mathop{*}_{k=1}^{N^{(i)}} \mathbf{x}_n \ jest \ A_n^{(i)} \right), to \left(\bigwedge_{m=1}^{M} \mathbf{y}_m \ jest \ B_m^{(i)} \right) \right\}_{i=1}^{I}, \tag{2}$$

gdzie symbol * oznacza dowolny spójnik („I" bądź „LUB"*)*, natomiast parametr M odpowieda liczbie wyjść systemu. Parametry $N^{(i)} \le N$ oraz wybór $n \in \{1,2,\ldots,N\}$ pozwalają dostosować liczbę przesłanek przypisanych do wybranych wejść systemu dla każdej z reguł w bazie. Wyjścia odpowiadają różnym zmiennym, dlatego zostały złączone spójnikiem „I".

3. Dynamiczna struktura bazy wiedzy w zastosowanym rozwiązaniu

Pewnymi wspólnymi elementami opisu bazy wiedzy, które mogą zostać wielokrotnie wykorzystane w regułach, są lingwistyczne określenia przesłanek i konkluzji, reprezentowane za pomocą zbiorów rozmytych. Zbiór rozmyty w bibliotece *FUZZLIB* implementuje klasa *FuzzySet*, której podstawowe cechy i funkcjonalność opisano dokładniej w publikacji [2]. Zatem przed tworzeniem reguł można zdefiniować zestaw wykorzystywanych zbiorów, natomiast w regułach umieszczać jedynie odwołania. To podejście pozwala zaoszczędzić znaczną ilość pamięci operacyjnej, ponieważ nie zachodzi redundancja przechowywanych informacji. Jest to szczególnie ważne w przypadku dużej liczby reguł. Ponadto zbiory przypisywane przesłankom i konkluzjom nie są modyfikowane w procesie wnioskowania (tylko odczyt), zatem rozwiązanie nie będzie wymagało zastosowania wyszukanych mechanizmów współdzielenia dostępu do pamięci oraz synchronizacji w możliwych rozwiązaniach wielowątkowych/wieloprocesowych.

Omawiane rozwiązanie rozdziela zbiory przesłanek oraz konkluzji gromadząc je w dwóch różnych kolekcjach dynamicznych[3], które najczęściej są tworzone jeszcze przed definiowaniem reguł, ponieważ przy tworzeniu reguły muszą już istnieć opisy wykorzystywanych przez nią przesłanek i konkluzji.

Innym stałym elementem konfiguracji systemu, z punktu widzenia opisu, są jego zmienne wejściowe i wyjściowe, które utożsamia się odpowiednio z wejściami oraz wyjściami. Oczywiście podczas pracy systemu mogą one przyjmować różne wartości, jednak ich konfiguracja, a szczególnie liczba, nie zmieniają się. W omawianej bibliotece zmienne wejściowe i wyjściowe reprezentują odpowiednio klasy *InVar* oraz *OutVar*. Ich obiekty są również przechowywane w ramach dwóch dynamicznych kolekcji.

W obiektach klas *InVar* i *OutVar* zachodzi etap rozmywania i wyostrzania. Odpowiednio określony zbiór rozmyty może zostać przechowany w ramach obiektu zmiennej wejściowej i użyty, kiedy zajdzie potrzeba rozmycia wprowadzonej wartości numerycznej [2].

Obiekty reprezentujące zmienne wyjściowe również zawierają zbiory rozmyte, które pełnią w tym przypadku dwie role. Po pierwsze określają rozmyte wyjścia systemu, a po drugie są wykorzystywane jako bufor wyników tymczasowych w procesie wnioskowania podczas agregacji rezultatów. Numeryczne wyjście systemu uzyskuje się poprzez wyostrzenie rozmytych wyników, co jest standardową funkcjonalnością klasy *FuzzySet* [2].

Przedstawione elementy pozwalają w pełni opisywać reguły łącząc przesłanki i konkluzje z wejściami i wyjściami systemu, minimalizując przy tym użycie pamięci operacyjnej.

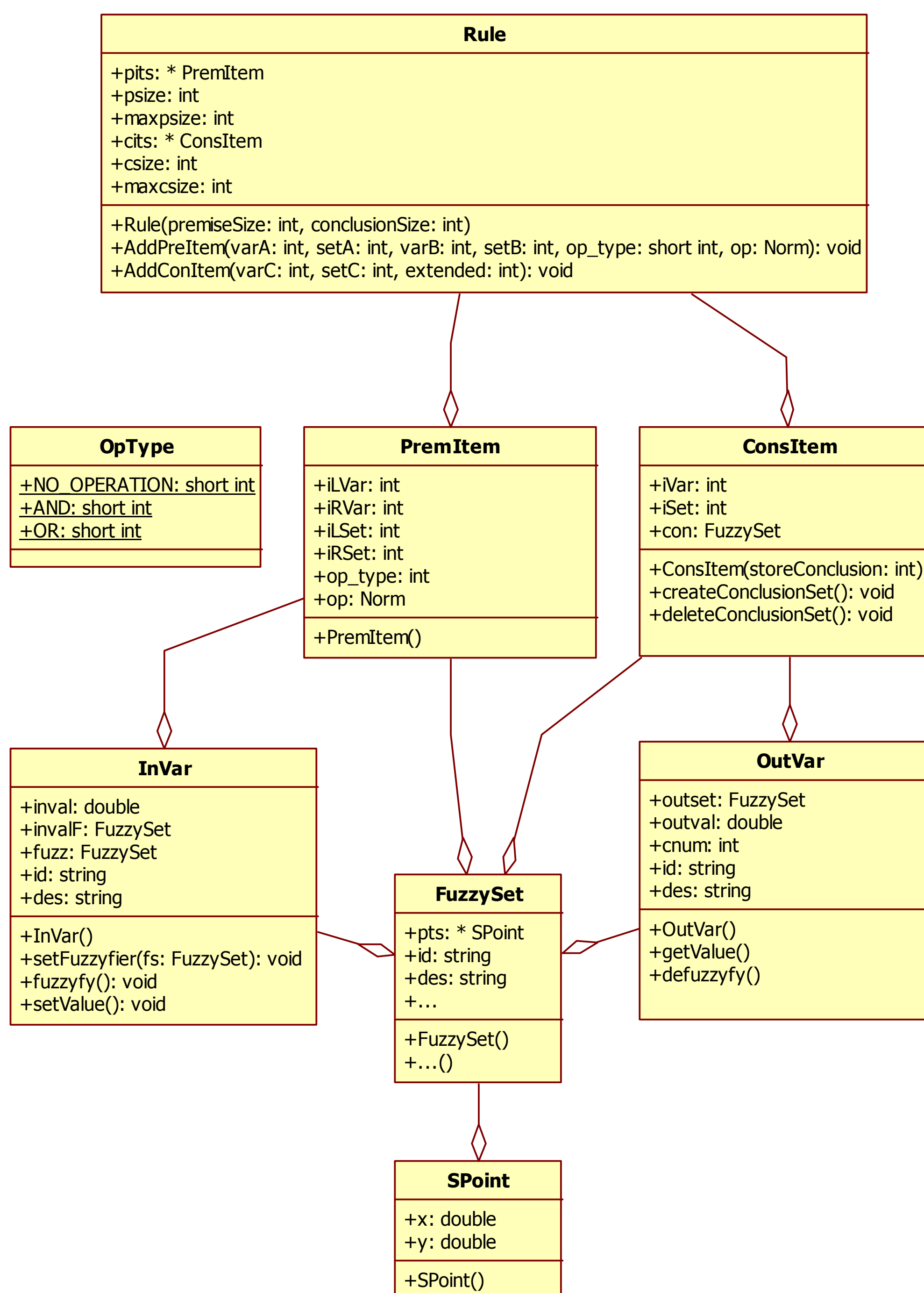
[3] W dotychczasowym rozwiązaniu w ramach języka C++ są to tablice dynamiczne.

Pozostałe elementy są bezpośrednio związane ze strukturą bazy wiedzy. Składową najbardziej ogólną w tym zbiorze jest reguła, reprezentowana w omawianej bibliotece przez klasę *Rule*. Agreguje ona złożony poprzednik oraz złożony następnik reguły rozmytej. Do tego celu utworzono dwie dodatkowe klasy: *PremItem* oraz *ConsItem*, rozbijające złożony poprzednik i następnik na pojedyncze elementy składowe.

Zgodnie z (2) następnik reguły może utworzyć ciąg stwierdzeń „**y** *jest B*", określający stan wyjść w danym przypadku. Zatem element *ConsItem* łączy jeden obiekt *OutVar* z obiektem *FuzzySet*, określającymi odpowiednio zmienną wyjściową i konkluzję. Natomiast złożony następnik tworzy kolekcja obiektów *ConsItem*, która jest zawierana w obiektach klasy *Rule*.

Poza elementami konkluzji reguła zawiera kolekcję obiektów *PremItem*, które ze względu na zastosowanie różnych spójników oraz określanie priorytetów kolejnych złączeń są bardziej rozbudowane. Założono, iż podstawowy element poprzednika (*PremItem*) pozwoli określić złączenie dwóch części przesłanki złożonej za pomocą określonego operatora. Rodzaj zastosowanego spójnika pomagają określić wartości statyczne w pomocniczej klasie *OpType*. Łączyć można zarówno dwie przesłanki proste, jak i dwa rezultaty otrzymane z operacji wcześniejszych, których wyniki zapamiętano na stosie wartości. Zatem obiekt klasy *PremItem* pozwala złączyć elementy umieszczone po lewej i prawej stronie określonego operatora, a elementami tymi mogą być przesłanki proste bądź wyniki poprzednich złączeń. Tym sposobem zapewniono warunki przedstawione w równaniu (2), uzyskując dodatkowo możliwość łączenia przesłanek prostych w określonej kolejności.

Na rysunku 1 przedstawiono skrótowy diagram wymienionych dotychczas klas, ze szczególnym uwzględnieniem relacji agregacji. Schemat wyraźnie obrazuje strukturę zawierania kolejnych obiektów. Zbiór rozmyty określający przesłanki i konkluzje (klasa *FuzzySet*) jest kolekcją punktów opisu (klasa *SPoint*). Dalej obiekty zbiorów są umieszczane w ramach zmiennych wejściowych/wyjściowych oraz elementów przesłanek i konkluzji, które ostatecznie tworzą całą regułę. Rozbudowana baza wiedzy zawiera oczywiście całą kolekcję tak określonych reguł, którą w istniejącej implementacji oparto na dynamicznej liście obiektów klasy *Rule*.



Rys. 1. Diagram klas biorących udział w tworzeniu reguły
Fig. 1. Diagram of classes taking part in rule creation

Warto zwrócić uwagę na wspomnianą wcześniej oszczędność pamięci poprzez przechowywanie jedynie referencji do jednokrotnie opisanych zmiennych (*InVar*, *OutVar*) oraz przesłanek i konkluzji (*FuzzySet*) w obiektach klas *PremItem* oraz *ConsItem* (pola *iVar* – referencja na zmienną oraz *iSet* – referencja na zbiór rozmyty). Całe obiekty zbiorów pojawiają się w klasach *InVar*, *OutVar* oraz *ConsItem*. Są to pola *invalF* (rozmyta wartość wejściowa), *fuzz* (zbiór rozmywający [2]), *outset* (rozmyte wyjście) oraz *con* (wniosek wynikający z reguły).

4. Interfejs programistyczny

Diagram przedstawiony na rysunku 1 pokazuje kilka prostych metod umieszczonych w ramach klasy *Rule*, pozwalających zarządzać strukturą bazy na poziomie reguły. Za pomocą tych metod modyfikuje się regułę dodając jej kolejne elementy składowe. Jednak większość programistycznego interfejsu zaprojektowanego do budowania bazy wiedzy znajduje się w klasie *FuzzySystem*, reprezentującej system rozmyty, zawierający całą kolekcję reguł.

Głównym założeniem biblioteki *FUZZLIB* jest maksymalne uproszczenie interfejsu tak, aby każdy użytkownik znający podstawy programowania obiektowego potrafił z niej korzystać. Elementem tego założenia jest również uproszczenie zasad konfiguracji systemu rozmytego poprzez wykorzystanie określeń lingwistycznych. Jak można zauważyć na rysunku 1, klasy *FuzzySet*, *InVar* oraz *OutVar* zawierają dwie składowe opisowe typu *string*: identyfikator *id* oraz opis *des*. Szczególnie ważne są pola identyfikatorów, które odpowiadają skrótowym nazwom lingwistycznym przypisanym do zbiorów rozmytych oraz zmiennych systemu. W ten sposób kod odpowiedzialny za definicje reguł jest bardzo łatwy w interpretacji.

Konfigurację struktury najlepiej rozpocząć od utworzenia obiektów zbiorów rozmytych oraz przypisania im identyfikatorów lub opisów. Zbiory te po przypisaniu do systemu staną się definicjami jego przesłanek i konkluzji. Przypisania dokonuje się za pomocą metod *addPremiseSet(FuzzySet & fs)* oraz *addConclusionSet(FuzzySet & fs)*, składowych klasy *FuzzySystem*.

W części przygotowawczej należy również skonfigurować i przypisać identyfikatory obiektom zmiennych wejściowych i wyjściowych systemu. Utworzenie zmiennych jest realizowane za pomocą metod *setInputWidth(int n)* oraz *setOutputWidth(int m)*. Później przypisywane są identyfikatory każdej z n zmiennych wejściowych i m wyjściowych metodami:

- *describeInputVar(int var, string id, string des)*,
- *describeOutputVar(int var, string id, string des)*.

Następnie można przejść do definiowania struktury bazy wiedzy poprzez dodawanie i konfigurację kolejnych reguł.

Metoda *addRule()* dodaje nową regułę, której kolejne elementy definiuje się za pomocą:

- *addRuleItem(string varA, string setA, string operator, string varB, string setB)*,
- *addRuleItem(string varA, string setA)*,
- *addRuleConclusion(string varC, string setC)*,

gdzie pierwsza dodaje element przesłanki złożonej, druga wykorzystywana jest do definiowania przesłanek prostych, natomiast trzecia dodaje element konkluzji. Należy zwrócić uwagę na opisowe typy parametrów, które odpowiadają identyfikatorom dodanych wcześniej zmiennych wejściowych oraz przesłanek i konkluzji.

Przyjęto następujące tekstowe określenia dla operatorów stanowiących spójniki przesłanek: ”AND” i ”OR”, odpowiednio dla spójników „I” oraz „LUB”.

W trakcie pracy systemu wynik każdego złączenia, określonego za pomocą obiektów *RuleItem*, zapisywany jest na stosie. Na etapie konfiguracji reguły istnieje możliwość określenia czy dana strona złączenia powinna być pobrana ze stosu. Do tego celu służy ciąg tekstowy ”STACK” podany w miejsce identyfikatora zmiennej wejściowej. W tym przypadku identyfikator przesłanki jest pomijany i może przyjąć dowolną wartość. Jak wcześniej wspomniano, takie rozwiązanie pozwala w prosty sposób definiować dowolnie złożone reguły z określonym priorytetem złączeń. Przykładowo, regułę opisującą pracę pewnego urządzenia osuszającego, zapisaną lingwistycznie w postaci:

*jeśli (***powierzchnia** *jest* ***mokra*** *i* **temperatura** *jest* ***niska****) lub*
*(***powierzchnia** *jest* ***bardzo mokra*** *i* **temperatura** *jest* ***średnia****),*
to **poziom osuszania** *jest* ***bardzo intensywny***,

można utworzyć za pomocą następującego ciągu instrukcji:

```
system.addRule();
system.addRuleItem(”surface”, ”wet”, ”AND”, ”temp”, ”low”);
system.addRuleItem(”surface”, ”very_wet”, ”AND”, ”temp”, ”medium”);
system.addRuleItem(”STACK”, ””, ”OR”, ”STACK”, ””);
system.addRuleConclusion(”drying_level”, ”high”);
```

gdzie założono istnienie definicji zmiennych wejściowych, przesłanek i konkluzji o podanych identyfikatorach. Przykładowa definicja zbioru rozmytego przesłanki o gaussowskiej funkcji przynależności w języku C++ wygląda następująco:

```
FuzzySet medium;
medium.newGaussianFast(m,q);
```

gdzie parametry m oraz q określają odpowiednio środek oraz szerokość funkcji Gaussa.

Natomiast postać trójkątna lub trapezoidalna może zostać utworzona przez proste dodawanie kolejnych punktów:

```
FuzzySet medium;
medium.addPoint(0.0, 0.0); medium.addPoint(2.0, 1.0); …
```

Po skonfigurowaniu systemu jest on gotowy do działania. Wnioskowanie przeprowadza się wywołując metodę *process()* obiektu klasy *FuzzySystem*.

5. Podsumowanie

Podsumowując warto wymienić najważniejsze cechy zaproponowanego rozwiązania. Zapewne mechanizm definiowania praktycznie dowolnie złożonych reguł oferuje duże możliwości konfiguracji bazy wiedzy, w zależności od potrzeb i upodobań użytkownika. Ponadto

stosowanie spójników „LUB” w przesłankach złożonych pozwoli w niektórych przypadkach[4] zmniejszyć liczbę reguł przez ich scalanie.

Zaproponowana struktura, oparta w większości na dynamicznych tabelach, pozwala szybko odwołać się do adresowanych obiektów. Niestety, takie rozwiązanie może się wiązać z większą złożonością obliczeniową etapu wstawiania nowego elementu do kolekcji. Problem powstaje, gdy rozmiar tabeli jest zbyt mały i trzeba ją powiększyć lub gdy wstawiany element powinien się znaleźć w środku kolekcji, co wiąże się z koniecznością przesunięcia innych. Należy jednak podkreślić, iż etap tworzenia struktury jest wykonywany jednokrotnie, dlatego w tym podejściu ważniejsze są zalety wykorzystania tabel podczas pracy systemu.

Zaletą jest również możliwość lingwistycznej konfiguracji kolekcji reguł. Zapewne rozwiązaniem idealnym byłoby umożliwienie definiowania bazy wiedzy za pomocą tekstowego skryptu. Implementacja tej funkcjonalności jest kolejnym planowanym krokiem rozwoju biblioteki *FUZZLIB*. Najważniejszym elementem takiego rozwiązania jest określenie odpowiedniej gramatyki, szczególnie w zakresie definicji zbiorów rozmytych. Możliwość definiowania funkcji przynależności o dowolnym kształcie jest atutem omawianej biblioteki i projektowane rozszerzenia nie mogą ograniczać dostępu do bogatego zbioru jej funkcji.

Głównym założeniem, które przyjęto podczas rozwoju biblioteki *FUZZLIB*, jest projektowanie przystępnego interfejsu programistycznego. Powinien on z jednej strony umożliwić szeroką konfigurację budowanych systemów rozmytych zaawansowanym użytkownikom, a z drugiej zachęcać wszystkich swoją prostotą. Autor ma nadzieję, iż zaprezentowane w niniejszym artykule rozwiązanie obiektowe jest zgodne z przyjętym założeniem i pozwoli na łatwe wykorzystanie biblioteki w wielu różnych zastosowaniach.

BIBLIOGRAFIA

1. Baldwin J.F.: A new approach to approximate reasoning using a fuzzy logic. Fuzzy Sets and Systems Vol. 2, 1979, s. 309÷325.
2. Kudłacik P.: Operacje na zbiorach rozmytych z odcinkowo-liniową funkcją przynależności. Studia Informatica Vol. 29(3A), Gliwice 2008, s. 91÷111.
3. Łeski J.: Systemy Neuronowo-Rozmyte. Wydawnictwa Naukowo-Techniczne, Warszawa 2008.
4. Mamdani E.H., Assilan S.: An experiment in linguistic synthesis with a fuzzy logic controller. International Journal of Man-Machine Studies Vol. 20(2), 1975, s. 1÷13.
5. Rutkowski L.: Metody i Techniki Sztucznej Inteligencji. Wydawnictwo Naukowe PWN, Warszawa 2006.

[4] Gdy konkluzje reguły są identyczne bądź podobne.

6. Zadeh L.A.: Fuzzy sets. Information and Control Vol. 8, 1965, s. 338÷353.
7. Zadeh L.A.: Outline of a new approach to the analysis of complex systems and decision processes. IEEE Transactions on Systems, Man and Cybernetics Vol. 3(1), 1973, s. 28÷44.
8. Cho Y.I.: Fuzzy Inference Method for Intelligent Artificial System. Journal of Medical Informatics & Technologies, Vol. 13, 2009, ISSN 1642-6037, s. 11÷14.
9. Przybyła T.: Hybrid Fuzzy Clustering Method. Journal of Medical Informatics & Technologies Vol. 10, 2006, ISSN 1642-6037, s. 143÷150.

Recenzenci: Dr inż. Bożena Małysiak-Mrozek,
Dr hab. inż. Adam Pelikant, prof. Pol. Łódzkiej

Wpłynęło do Redakcji 31 stycznia 2010 r.

Abstract

FUZZLIB library provides easy to use tools designed for developing fuzzy systems of diverse complexity. Article describes an object oriented approach to a knowledge base structure and methods representing an interface designed for creating rules.

Canonical form of a rule (1) does not consider “OR” operators in compound premise and does not consider compound conclusion [3]. Presented approach allows both “AND” and “OR” operators, as well as compound conclusion joined with “AND” operator (equation (2)).

Solution stores common elements of a system description in one place. This considers objects for input and output variables and fuzzy sets, such as premises and conclusions. Objects directly involved in creating a structure of a knowledge base (rules) use only references to common elements.

Figure 1 presents diagram of classes used in rule representation. The whole knowledge base is a simple collection of such rules. Class *Rule* allows to define more than one part of a premise (*PremItem*), which is used in compound statements. Also more than one object of a *ConsItem* class is allowed, which enables to define compound conclusions. Classes aggregated below represent common elements described earlier, like system variables (*InVar*, *OutVar*), premises and conclusions (*FuzzySet*). For these, only references are stored in *PremItem* and *ConsItem*.

Configuration and management of a knowledge base is very simplified, because of a linguistic object description. Such human-like approach makes a code of a program very

clear. The author hopes that this advantage of the library will help it find many different applications.

Adres

Przemysław KUDŁACIK: Uniwersytet Śląski, Instytut Informatyki, ul. Będzińska 39,
41-200 Sosnowiec, Polska, przemyslaw.kudlacik@us.edu.pl .