UMCS

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: Informatyka

Martyna Filiks
nr albumu: 246045

Szkieletowy system ekspertowy z możliwością rozwiązywania konfliktów między regułami

Expert system shell implementing mechanism which solves conflicts between rules

Praca licencjacka
napisana w Zakładzie Układów Złożonych i Neurodynamiki
pod kierunkiem dr. Tomasza Żurka

Lublin rok 2017

Spis treści

Wstęp **5**

1 Systemy ekspertowe i argumentacja **7**

1.1 Klasyczne systemy ekspertowe 7

1.1.1 Definicja systemu ekspertowego 7

1.1.2 Baza wiedzy 9

1.1.3 Mechanizm wnioskowania 12

1.1.4 Spójność bazy wiedzy 15

1.2 Systemy z argumentacją podważalną 15

1.2.1 System I: Priorytety stałe 16

1.2.2 System II: Priorytety podważalne 26

2 Koncepcja systemu **27**

2.1 Sposób zapisu wiedzy 27

2.2 Mechanizm wnioskowania 29

2.2.1 Metoda wnioskująca 30

2.2.2 Spełnienie reguły 31

2.2.3 Uaktywnienie reguły 32

2.2.4 Rozwiązanie konfliktów między regułami 33

2.2.5 Atak reguł 35

2.3 Interfejs 36

2.4 Test 37
2.5 Podsumowanie 40

Bibliografia **43**

Spis rysunków i tabel **45**

Spis listingów **47**

Wstęp

Systemy ekspertowe są narzędziami wspomagającymi podejmowanie decyzji, diagnostykę i zarządzanie wiedzą, które od dawna są wykorzystywane w praktyce. System ekspertowy jest to program komputerowy, w którym wiedza dziedzinowa oddzielona od systemu jest przetwarzana za pomocą wbudowanego mechanizmu wnioskującego. Większość systemów zakłada, że wiedza wprowadzona do bazy jest spójna, tzn. wnioskowanie na jej podstawie nie może doprowadzić do sprzeczności. Istnieją jednak dyscypliny, w których wiedza często nie jest spójna, np. medycyna, prawo. Dlatego niezbędny jest system pozwalający na poprawne wnioskowanie pomimo niespójności w bazie wiedzy.

Celem niniejszej pracy jest realizacja szkieletowego systemu ekspertowego, który umożliwiałby wnioskowanie przy niespójnej bazie wiedzy. W pracy omówiono podstawy tematyki systemów ekspertowych oraz argumentacji podważalnej, która stanowi podstawę wykrywania i rozwiązywania sprzeczności między regułami.

Rozdział pierwszy jest zorganizowany następująco: najpierw zdefiniowano system ekspertowy, podano podstawowe elementy budowy systemu, opis reprezentacji wiedzy w postaci reguł, rodzaje mechanizmów wnioskowania oraz zasady ich działania. W podrozdziale drugim wyjaśniono, na czym polega argumentacja podważalna w połączeniu z priorytetami oraz przedstawiono dwa systemy z taką argumentacją. Kolejno zaprezentowano typy ataków na argument — obalanie

i podkopanie. W celu łatwiejszego zrozumienia pojęć argumentacji, zilustrowano je przykładami.

Rozdział drugi przedstawia projekt szkieletowego systemu ekspertowego opartego na regułowej reprezentacji wiedzy, napisany w języku C#. W rozdziale tym podano i wyjaśniono sposób zapisu wiedzy w standardzie XML oraz najważniejsze elementy implementacji mechanizmu wnioskowania w przód.

Rozdział 1

Systemy ekspertowe i argumentacja w literaturze

1.1 Klasyczne systemy ekspertowe

Ten podrozdział będzie dotyczył podstawowych wiadomości o klasycznych systemach ekspertowych i będzie oparty o następującą literaturę: [1, 5, 6, 9].

1.1.1 Definicja systemu ekspertowego

Na początek wyjaśnijmy co to jest system ekspertowy. Nazwa *system ekspertowy* ma pochodzenie od słowa ekspert, które oznacza człowieka posiadającego wiedzę dziedzinową i umiejętność wnioskowania z tej wiedzy, oznacza to że potrafi on stosować zdobytą wiedzę w rozwiązywaniu problemów z tej dziedziny.

Według [1] system ekspertowy (ang. *expert system*) jest to program komputerowy wykorzystujący procedury wnioskowania do rozwiązywania problemów, które są na tyle trudne, że wymagają znaczącej ekspertyzy specjalistów. Taki program ułatwia podejmowanie decyzji dotyczących danej dziedziny.

Aby móc zbudować system ekspertowy musimy poznać jego schemat budowy. Rysunek 1.1 przedstawia architekturę systemu i został zaczerpnięty z [9]. System ekspertowy składa się z podstawowych elementów takich jak [6]:

1. **Baza wiedzy**

   Zawiera wiedzę dziedzinową eksperta, istotną dla rozwiązania określonego problemu. Składa się ze zbioru reguł i faktów zadeklarowanych przez użytkownika.

2. **Mechanizm wnioskujący**

   Jest to program wykonywalny, który wnioskuje na podstawie bazy wiedzy, w celu wyznaczenia nowych faktów oraz rozwiązania zadanego problemu.

3. **Edytor bazy wiedzy**

   Służy do tworzenia, modyfikowania i czytania bazy wiedzy.

4. **Pamięć podręczna**

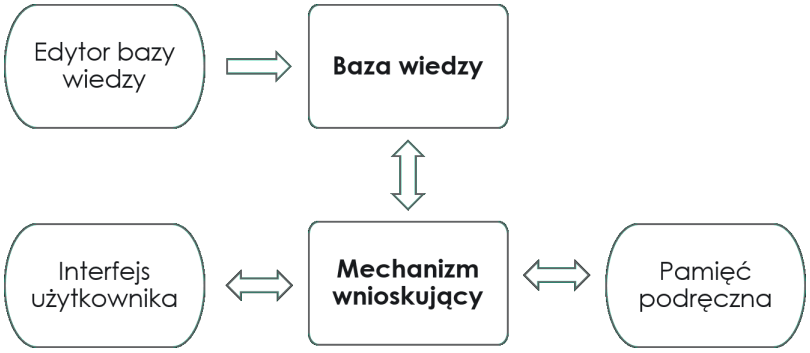
   Zawiera fakty zadeklarowane przez użytkownika oraz cząstkowe wyniki wnioskowania. Jest dostępna wyłącznie dla mechanizmu wnioskowania.

5. **Interfejs użytkownika**

   Umożliwia użytkownikowi korzystanie z systemu.

Według pozycji [9] podstawowymi cechami systemu ekspertowego są:

- Wiedza jest oddzielona od mechanizmu wnioskowania.
- Łatwo jest rozbudować bazę wiedzy.
- Możliwość modyfikacji bazy wiedzy i jej uaktualnianie.



Rysunek 1.1: Architektura systemu ekspertowego

- Możliwość tworzenia systemów z pustymi bazami wiedzy, tzw. systemy szkieletowe lub skorupowe (ang. *expert system shell*). Za pomocą edytora bazy wiedzy, użytkownik może sam wprowadzić wiedzę z wybranej przez siebie dziedziny.

- Sposób zakodowania wiedzy w bazie jest w miarę prosty, ponieważ nie wymaga to specjalnego przygotowania.

1.1.2 Baza wiedzy

Baza wiedzy jest jednym z głównych elementów systemu ekspertowego. W trakcie jej tworzenia należy wybrać odpowiedni sposób reprezentacji wiedzy, ponieważ znacząco wpływa to na działanie systemu. Zatem zdefiniujmy pojęcie wiedzy i jej reprezentacji.

Wiedza jest symbolicznym opisem istniejącego świata rzeczywistego, składa się ona z faktów, relacji pomiędzy nimi oraz procedur. *Reprezentacja wiedzy* jest procesem przetwarzania pozyskanej od eksperta wiedzy do odpowiedniej formy, którą będzie można dalej interpretować. Wyróżnia się następujące sposoby reprezentacji wiedzy:

- ramy,

- sieci semantyczne,
- reguły.

Do powyższej listy można również dodać sposoby reprezentacji faktów występujących w bazie wiedzy, np:

- za pomocą logiki dwuwartościowej, czyli prawda-fałsz,
- za pomocą logiki wielowartościowej, polegającej na strukturze Obiekt-Atrybut-Wartość,
- za pomocą logiki rozmytej.

Systemy ekspertowe oparte na regułowej reprezentacji wiedzy nazywamy regułowymi systemami ekspertowymi (ang. *rule-based expert systems*). Do dalszej charakterystyki wybierzemy reguły.

Reguły

Reguły pozwalają w sposób przejrzysty i łatwy przedstawić wiedzę specjalisty. Reguła może mieć następującą postać:

$$wniosek \textbf{ jeżeli } warunek_1 \textbf{ i } warunek_2 \textbf{ i } warunek_3 \textbf{ i } \ldots,$$

gdzie wniosek jest zazwyczaj jeden, a warunków może być wiele (część warunkowa). Wniosek będziemy nazywać również konkluzją, rezultatem, natomiast warunek — przesłanką lub założeniem. Mówimy, że wniosek jest prawdziwy wtedy, gdy koniunkcja wszystkich warunków jest prawdziwa. Wówczas taka reguła jest spełniona i można ją uaktywnić (ang. *firing*), aby jej wniosek mógł zostać dodany do bazy wiedzy.

W przypadku, gdy mamy kilka reguł o takim samym wniosku, to prawdziwość warunków jednej z nich wystarczy, aby uznać wniosek za prawdziwy. Na przykład:

$$\textbf{reguła}_\mathbf{1}\text{: } \textit{wniosek} \textbf{ jeżeli } A \textbf{ i } B,$$

$$\textbf{reguła}_\mathbf{2}\text{: } \textit{wniosek} \textbf{ jeżeli } C \textbf{ i } D.$$

Reguły można *zagnieżdżać*, polega to na tym, że wniosek jednej reguły może być warunkiem innej reguły:

$$\textbf{reguła}_\mathbf{1}\text{: } A \textbf{ jeżeli } B \textbf{ i } C,$$

$$\textbf{reguła}_\mathbf{2}\text{: } B \textbf{ jeżeli } D \textbf{ i } E.$$

Odwrotnym zjawiskiem do zagnieżdżania jest *spłaszczanie* reguł, co odpowiada zapisowi:

$$\textbf{reguła}_\mathbf{1}\text{: } A \textbf{ jeżeli } D \textbf{ i } E \textbf{ i } C.$$

Przy większej liczbie takich spłaszczonych reguł, czytelność i analiza takiego zapisu jest utrudniona. Zaleca się stosować zagnieżdżanie reguł.

Istotnym założeniem klasycznych systemów ekspertowych jest *zasada zamkniętego świata*. Na podstawie [9], jest to ograniczenie systemu, mówiące o tym, że:

Wszystko co nie zostało zadeklarowane wprost, bądź nie zostało wywnioskowane z reguł traktuje się jako nieprawdę.

Oznacza to, że twórca wyposażył bazę wiedzy w kompletną wiedzę i jeśli w trakcie wnioskowania nie da się uznać czegoś za prawdę, to należy uznać to za nieprawdę.

Może się zdarzyć, że w trakcie tworzenia bazy wiedzy niektóre warunki reguły *wzajemnie się wykluczają*. Wówczas oba warunki nie mogą być jednocześnie prawdzie, ponieważ jeśli jeden z nich będzie prawdziwy, to drugi musi być nieprawdziwy. System powinien w takiej sytuacji odpowiednio zareagować, to znaczy,

że gdy użytkownik zadeklaruje pewien warunek jako prawdziwy, to wszystkie warunki wzajemnie wykluczające się z tym warunkiem, powinny zostać uznane za fałszywe.

1.1.3 Mechanizm wnioskowania

System ekspertowy na podstawie bazy wiedzy powinien generować nowe fakty, a tym zadaniem zajmuje się mechanizm wnioskowania. Wyróżnia się dwa podstawowe rodzaje mechanizmów wnioskowania:

- Wnioskowanie w przód (ang. *forward chaining*).
- Wnioskowanie wstecz (ang. *backward chaining*).

Wnioskowanie w przód

Wnioskowanie w przód przebiega od warunków do wniosku, tzn. zgodnie z kierunkiem implikacji reguły. Na początek należy zdefiniować założenia oraz cel wnioskowania. Założeniami będą:

1. System posiada zdefiniowaną bazę wiedzy będącą zbiorem reguł oraz fakty zadeklarowane przez użytkownika.
2. Fakty przechowywane są w pamięci podręcznej.

Celem wnioskowania jest znalezienie wszystkich możliwych wniosków.

Pierwszym krokiem jest sprawdzenie, czy warunki pierwszej reguły są prawdziwe, to znaczy, że są zadeklarowane w pamięci podręcznej.

- Jeżeli wszystkie warunki są spełnione, to wniosek reguły zostaje dodany do pamięci podręcznej.
- W przeciwnym wypadku system pomija regułę i sprawdza kolejną.

W momencie, kiedy system sprawdzi wszystkie reguły, powtarza procedurę od początku. Wnioskowanie kończy się, gdy nie da się wyprowadzić więcej wniosków.

Przykład 1.1. Niech będą dane następujące fakty: $A, \neg K, S, P$ oraz reguły:

$$\textbf{reguła}_\textbf{1}: C \wedge S \wedge Y \Rightarrow \neg W$$

$$\textbf{reguła}_\textbf{2}: A \Rightarrow C$$

$$\textbf{reguła}_\textbf{3}: \neg K \wedge C \Rightarrow R$$

$$\textbf{reguła}_\textbf{4}: P \Rightarrow Y$$

$$\textbf{reguła}_\textbf{5}: \neg W \wedge Z \Rightarrow M$$

$$\textbf{reguła}_\textbf{6}: Y \Rightarrow Z$$

Przeprowadzimy proces wnioskowania w przód, aby wyznaczyć wszystkie możliwe wnioski. Na początek sprawdzamy regułę nr 1, tylko warunek S znajduje się w faktach, stąd reguła pozostaje nieokreślona. Przechodzimy do kolejnej, warunek A jest zadeklarowany jako prawda, zatem wniosek C zostaje dodany do faktów. Z reguły nr 3 wynika, że na podstawie warunków $\neg K, C$ otrzymujemy wniosek R. Na podstawie reguły nr 4 zapisujemy wniosek Y. Reguła nr 5 pozostaje nieokreślona, natomiast z reguły nr 6 dodajemy wniosek Z do pamięci podręcznej. Powtarzamy sprawdzanie reguł, od początku. W drugim cyklu reguła 1 jest już spełniona, zatem wniosek $\neg W$ zostaje zapisany do bazy faktów. Wnioski reguł: 2, 3, 4 zostały już dodane i je pomijamy. Przechodzimy do reguły nr 5, jej warunki są faktami, stąd wniosek M można dopisać do faktów. Reguła 6 jest pomijana, bo jej wniosek jest już dodany. Ponieważ w tym cyklu zostały dodane dwa wnioski, to rozpoczynamy nowy cykl. Podczas trzeciego cyklu nie zostaną dodane już żadne fakty, zatem kończymy proces wnioskowania z ostateczną bazą faktów: $A, \neg K, S, P, C, R, Y, Z, \neg W, M$.

Wnioskowanie wstecz

Wnioskowanie wstecz przebiega od hipotezy do wniosku, tzn. przeciwnie do kierunku implikacji reguły. Najpierw definiuje się pewną hipotezę. Celem wnioskowania jest udowodnienie tej hipotezy.

W pierwszym kroku system sprawdza, czy postawiona hipoteza nie jest zadeklarowana w pamięci podręcznej.

- Jeśli tak, to hipoteza zostaje potwierdzona i algorytm kończy działanie sukcesem.
- W przeciwnym wypadku, system bada, czy w bazie wiedzy są reguły, których wnioskiem jest hipoteza.
  - Jeśli tak, to algorytm stara się warunki tej reguły potwierdzić, traktując je jako kolejne hipotezy. Gdy potwierdzenie jednej z hipotez zawiedzie, system szuka innych reguł, których wnioskiem jest badana hipoteza, aby udowodnić jej prawdziwość. Algorytm kończy się klęską, gdy próba zawiedzie dla wszystkich reguł.
  - Jeśli nie, to algorytm nie może udowodnić hipotezy i kończy działanie.

Przykład 1.2. Załóżmy, że nasza baza wiedzy ma postać jak z przykładu 1.1. Fakty: $A, \neg K, S, P$ oraz reguły:

$$\textbf{reguła}_\textbf{1}: C \wedge S \wedge Y \Rightarrow \neg W$$

$$\textbf{reguła}_\textbf{2}: A \Rightarrow C$$

$$\textbf{reguła}_\textbf{3}: \neg K \wedge C \Rightarrow R$$

$$\textbf{reguła}_\textbf{4}: P \Rightarrow Y$$

$$\textbf{reguła}_\textbf{5}: \neg W \wedge Z \Rightarrow M$$

$$\textbf{reguła}_\textbf{6}: Y \Rightarrow Z$$

Będziemy chcieli udowodnić hipotezę Z, dlatego przeprowadzimy wnioskowanie wstecz, w celu potwierdzenia jej na podstawie faktów. Najpierw sprawdzamy, czy badana hipoteza jest wśród faktów, okazuje się, że nie ma. Dalej szukamy reguły, której wnioskiem jest hipoteza, taką regułą jest reguła nr 6. Teraz naszą hipotezą pomocniczą będzie Y, będące wnioskiem reguły nr 4. Warunek P jest zadeklarowany jako prawda w bazie faktach, a więc Y zostaje potwierdzone. Tym samym możemy potwierdzić hipotezę Z z reguły 6, co kończy wnioskowanie.

1.1.4 Spójność bazy wiedzy

Baza wiedzy powinna być spójna, to znaczy, że nie zawiera sprzecznych reguł. Niestety w sytuacjach życiowych tak nie jest. Takim przykładem jest dziedzina prawa, gdzie w przypadku kolizji reguł prawnych wybór jednej skutkuje nieważność drugiej. Dzieje się tak, ponieważ nie można jednocześnie wnioskować z dwóch przeciwstawnych sobie reguł, jedna z nich traci moc obowiązującą. Taka kolizja może występować, gdy ta sama sprawa podlega regulacji kilku różnym przepisom (np. nowa ustawa uchyla ustawę obowiązującą poprzednio).

W systemach ekspertowych sprzeczność reguł może prowadzić do zawieszenia się procesu wnioskowania. Jednak ratunkiem na niespójność wiedzy w systemie będzie wprowadzenie do mechanizmu wnioskowania pewnych elementów z argumentacji podważalnej.

1.2 Systemy z argumentacją podważalną

Ten podrozdział został opracowany na podstawie artykułów [3, 4, 7, 8]. Nasze rozważania zaczniemy od wyjaśnienia pojęcia argumentacji w dwóch aspektach: formalnym i nieformalnym. Argumentacja nieformalna jest sposobem jakim posługują się ludzie, by rozwiązywać konflikty oraz likwidować niezgodność opi-

nii w codziennym życiu, prawie, medycynie itp. W literaturze [2] argumentacja jest zdefiniowana jako działalność słowna, społeczna i racjonalna, mającą na celu uzasadnienie określonego punktu widzenia. Z kolei model argumentacji formalnej próbuje opisać ów nieformalny, ludzki sposób rozumowania za pomocą formalnych metod, które dadzą się zaimplementować w systemie komputerowym.

W argumentacji podważalnej argument może być wspierany przez inne argumenty lub też może zostać zaatakowany przez nie. Podważalność wynika z faktu, że argumenty mogą zostać pokonane przez mocniejsze kontrargumnty. Rozumowanie prawne łączy w sobie wykorzystanie priorytetów do wyboru pomiędzy sprzecznymi regułami oraz użycie założeń lub słabej negacji wewnątrz reguły, aby spowodować jej niestosowalność w pewnych sytuacjach.

Zaprezentujemy system w dwóch przypadkach. W pierwszym priorytety określone przez użytkownika są stałe, natomiast w drugim przypadku będą one pochodziły z samego systemu.

1.2.1 System I: Priorytety stałe

Systemy z argumentacją podważalną zawierają takie elementy jak: język formalny oraz argument.

Język

Język posiada dwa rodzaje negacji: słabą (*negation as failure*) i klasyczną. Formuła atomowa pierwszego rzędu jest *literałem pozytywnym*. Jeśli literał pozytywny poprzedzony jest symbolem $\neg$, to nazywamy go *literałem negatywnym*. Literał pozytywny lub negatywny jest słaby wtedy, gdy jest poprzedzony symbolem $\sim$, w przeciwnym wypadku jest mocnym literałem. Dla dowolnego atomu $P(x)$ mówimy, że literały $P(x)$ i $\neg P(x)$ są *komplementarne*. Dopełnienie języka L będziemy oznaczali za pomocą $\overline{L}$.

Reguła jest wyrażeniem postaci:

$$r\colon L_0 \wedge \ldots \wedge L_j \wedge \sim L_k \wedge \ldots \wedge \sim L_m \Rightarrow L_n,$$

gdzie r jest nazwą reguły, a każde L_i $(0 \leqslant i \leqslant j)$ jest literałem mocnym. Koniunkcja po lewej stronie strzałki jest poprzednikiem, a literał po prawej stronie jest następnikiem reguły.

W informacji wejściowej system dostaje reguły oraz priorytety. Wejście nazywamy *teorią uporządkowaną*, która jest parą $(T, <)$, gdzie T jest zbiorem reguł, natomiast $<$ jest porządkiem w zbiorze T. Zapis $r < r'$ oznacza, że r' ma pierwszeństwo nad r. Reguła może być podważalna na dwa sposoby:

- może zawierać bezpośrednio założenia;
- może być przesłonięta mocniejszymi regułami ze sprzecznym następnikiem, nawet jeśli sama nie zawiera założeń.

W sytuacji, gdy poprzednik reguły jest pusty, to taka reguła wyraża fakt. Fakty też mogą być przedmiotem ataku.

Argumenty

Podstawowym pojęciem systemu z argumentacją podważalną jest argument.

Definicja 1.3. *Argument* jest skończonym ciągiem reguł $[r_n, \ldots, r_m]$ takim, że:

1. dla każdego i, gdzie $n \leqslant i \leqslant m$, oraz dla każdego literału pozytywnego lub negatywnego L w poprzedniku reguły r_i, istnieje $j < i$ takie, że L jest następnikiem r_j;
2. następnik żadnej reguły r_i nie jest następnikiem reguły r_j $(j < i)$.

Dla dowolnej teorii uporządkowanej Γ zbiór wszystkich argumentów pochodzących z Γ oznaczamy za pomocą $Args_\Gamma$. Podobnie dla dowolnego zbioru reguł T, zbiór $Args_T$ oznacza zbiór wszystkich argumentów, które składają się wyłącznie z reguł pochodzących z T.

Przydatna będzie znajomość następujących pojęć:

- Dla każdego argumentu A, argument A' jest *(właściwym) podargumentem* A wtedy i tylko wtedy, gdy A' jest (właściwym) podciągiem A.
- Literał L jest *wnioskiem argumentu* A wtedy i tylko wtedy, gdy jest następnikiem pewnej reguły w A.
- Literał L jest *założeniem argumentu* A wtedy i tylko wtedy, gdy $\sim \overline{L}$ występuje w pewnej regule argumentu A.

Przykład 1.4. Następujący przykład przedstawia te pojęcia. Dla argumentu:

$$A = [r_1 \colon \sim \neg a \Rightarrow b, \quad r_2 \colon b \wedge \sim d \Rightarrow c]$$

wnioskami są $\{b, c\}$, jego podargumentami są $\{[\,], [r_1], [r_1, r_2]\}$, natomiast jego założeniami są $\{a, \neg d\}$.

Relacje między argumentami

Dwa różne rodzaje negacji mogą prowadzić do dwóch sposobów ataku argumentu. Poniższa definicja mówi o tym, które argumenty są ze sobą w konflikcie.

Definicja 1.5. Argument A *atakuje* (lub jest kontrargumentem) argument B wtedy i tylko wtedy, gdy dany wniosek A jest dopełnieniem pewnego wniosku lub założenia B. Jeśli jeden argument atakuje drugi, to mówimy że oba są w konflikcie ze sobą.

Kontrargument może zaatakować sam siebie lub jeden ze swoich podargumentów, aby pośrednio zaatakować cały argument. Tak naprawdę każdy argument jest podargumentem siebie, dlatego definicja mówi, że A atakuje B wtedy i tylko wtedy, gdy podargument z A atakuje podargument z B. Zatem jeśli mamy:

$$A = [r_1 \colon \Rightarrow a, \quad r_2 \colon a \Rightarrow b],$$
$$B = [r_3 \colon \Rightarrow \neg b, \quad r_4 \colon \neg b \Rightarrow \neg a],$$

to ataki będą występować pomiędzy podargumentami:

$$[r_1, r_2] \text{ i } [r_3],$$
$$[r_3, r_4] \text{ i } [r_1],$$
$$[r_1, r_2] \text{ i } [r_3, r_4].$$

Koncepcja ataku (kontrargumentu) jest bardzo istotna, ponieważ każdy system z argumentacją podważalną powinien stwierdzić, że jeśli dwa argumenty są ze sobą w konflikcie, to nie powinny one zostać uzasadnione. Musimy jeszcze określić kiedy argument jest spójny a kiedy zbiór argumentów jest bezkonfliktowy.

Definicja 1.6. Argument jest *spójny* wtedy i tylko wtedy, gdy nie atakuje sam siebie.

Przykład 1.7. Przedstawimy dwa przykłady niespójności argumentów. Pierwszy:

$$A = [r_1 \colon \Rightarrow a, \quad r_2 \colon a \Rightarrow \neg a].$$

Drugim przykładem jest:

$$B = [r_1 \colon \sim a \Rightarrow b, \quad r_2 \colon b \Rightarrow a].$$

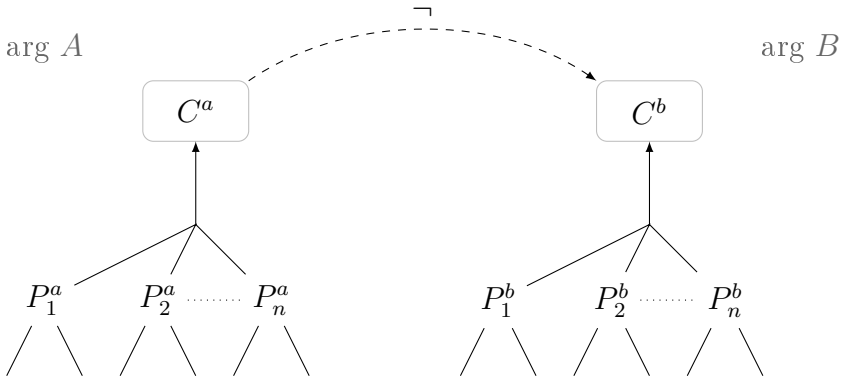
Definicja 1.8. Zbiór argumentów $Args$ jest bezkonfliktowy wtedy i tylko wtedy, gdy żaden argument w $Args$ nie atakuje argumentu w tym zbiorze.

Teraz kiedy już wiemy, które argumenty są ze sobą w konflikcie, to kolejnym etapem jest porównanie sprzecznych argumentów za pomocą priorytetów. W tym celu zdefiniujmy pojęcie *porażki* w zależności od tego, czy atak jest na wniosek czy założenie argumentu.

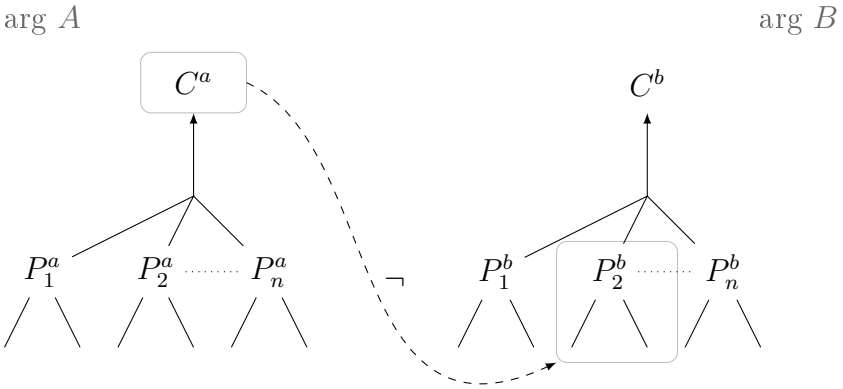
Definicja 1.9. Niech A i B będą dwoma argumentami i niech będzie dana para reguł: $r_A \in A$ i $r_B \in B$. Wtedy:

- A *obala* B wtedy i tylko wtedy, gdy:
  - reguły r_A i r_B mają sprzeczne wnioski,
  - $r_A \not< r_B$ (ang. *rebutting attack*).
- A *podkopuje* B wtedy i tylko wtedy, gdy:
  - wnioskiem r_A jest negacja jakiegoś założenia r_B,
  - $r_A \not< r_B$ (ang. *undermining attack*).

Sposoby ataku zdefiniowane powyżej zostały zilustrowane na rysunkach 1.2, 1.3, gdzie C jest wnioskiem danego argumentu, a $P_1, P_2, \ldots, P_n$ są warunkami.



Rysunek 1.2: Atak obalania



Rysunek 1.3: Atak podkopania

W przypadku wystąpienia ataku obalania lub podkopania konflikt reguł rozwiązuje się poprzez porównanie priorytetów tych reguł. Zauważmy, że jeśli A obala lub podkopuje B, to A atakuje B. Nie zachodzi następujący przypadek: jeśli A podkopuje B, to B podkopuje A.

Przykład 1.10. Niech będą dane argumenty:

$$A = [r_1 \colon \sim b \Rightarrow a],$$
$$B = [r_2 \colon \sim a \Rightarrow b],$$

wtedy argument A podkopuje B. Wniosek argumentu A zostanie uznany, z kolei wniosek argumentu B już nie, ponieważ jego warunek nie jest spełniony.

Nie zachodzi również przypadek: jeśli A obala B, to B obala A.

Przykład 1.11. Załóżmy, że mamy:

$$A = [r_1 \colon \Rightarrow a],$$
$$B = [r_2 \colon \Rightarrow \neg a],$$
$$r_1 < r_2.$$

Wtedy argument B obala A, ale nie na odwrót, ponieważ priorytet reguły argumentu B jest większy niż argumentu A.

Porażka argumentu może być:

- bezpośrednia lub
- pośrednia, czyli pokonanie jednego z jego podargumentów właściwych.

Argument jest tak mocny jak jego najsłabszy podargument. W przypadku, gdy jeden argument podkopuje drugi, a drugi go nie podkopuje, ale tylko obala pierwszy, to ostatecznie drugi nie pokonuje pierwszego.

Definicja 1.12. Niech A i B będą dwoma argumentami. Wówczas A *pokonuje* B wtedy i tylko wtedy, gdy:

- A jest pusty oraz B jest niespójny; lub
- A podkopuje B; lub
- A obala B i B nie podkopuje A.

Mówimy, że A *ściśle pokonuje* B wtedy i tylko wtedy, gdy A pokonuje B oraz B nie pokonuje A.

Wniosek 1.13. Jeśli A atakuje B, to A pokonuje B lub B pokonuje A. Natomiast jeśli A pokonuje B, to A atakuje B.

Przykład 1.14. Następujący przykład pokazuje, że podkopanie argumentu jest mocniejsze niż jego obalanie. Rozważmy reguły:

$$r_1 \colon \sim \neg Niewinny(OJ) \Rightarrow Niewinny(OJ),$$
$$r_2 \colon \Rightarrow \neg Niewinny(OJ)$$

oraz załóżmy, że $<= 0$. Wtedy mimo, że $[r_1]$ obala $[r_2]$, to $[r_1]$ nie pokonuje $[r_2]$, ponieważ $[r_2]$ podkopuje $[r_1]$. Zatem $[r_2]$ ściśle pokonuje $[r_1]$.

W przypadku ataku obalania i podkopania, gdy porównujemy odpowiednie reguły, wystarczy relacja $\not<$. Wynika z tego, że pokonujący argument nie może być jedynie słabszy niż ten, który ma być pokonany.

Przykład 1.15. Teraz zilustrujemy relację pomiędzy atakiem obalania a podkopania. Niech:

$$A = [r_0\colon\ \sim Bogaty \Rightarrow \neg Ma\_Porsche],$$
$$B = [r_1\colon\ \Rightarrow Ma\_Porsche, \quad r_2\colon Ma\_Porsche \Rightarrow Bogaty].$$

Załóżmy, że $r_1 < r_0$. Argument A obala B, ale go nie pokonuje, ponieważ B podkopuje A. Skutkuje to tym, że argument B jest uzasadniony oraz jego podargument $B' = [r_1]$ jest uzasadniony pomimo tego, że B' jest pokonany przez A. B' zostaje przywrócony, ponieważ B ściśle pokonuje kontrargument A.

Status argumentów

Ponieważ pokonujące argumenty mogą same zostać pokonane przez inne argumenty, to porównywanie wyłącznie par argumentów nie jest wystarczające. Potrzebna jest również definicja, która określa status argumentów na podstawie wszystkich przypadków ataku, w których one występują. W szczególności, definicja powinna pozwalać na przywrócenie pokonanych argumentów, jeśli argument pokonujący sam jest (ściśle) pokonany przez inny argument. Ta definicja jest kluczowym elementem systemu. Na wejście systemu bierzemy zbiór wszystkich możliwych argumentów i ich wzajemne relacje ataku, natomiast na wyjściu otrzymujemy podział argumentów na trzy klasy:

- argumenty z konfliktem wygranym,
- argumenty przegrane,
- argumenty, które pozostawiają konflikt nierozstrzygnięty.

W związku z powyższym, argumenty wygrywające powinny być wyłącznie tymi argumentami, które ze względu na przesłanki są ponad wszelką wątpliwość. Jedyny sposobem, aby poddać w wątpliwość tych argumentów jest zapewnienie nowych przesłanek, dając początek nowym, pokonującym kontrargumentom. Zatem chcemy, aby zbiór argumentów uzasadnionych był unikalny i bezkonfliktowy.

Zbiór argumentów *uzasadnionych* jest konstruowany krok po kroku (zgodnie z domyślnie założoną teorią wejściową).

1. Najpierw do zbioru $JustArgs_1$ zbieramy wszystkie argumenty, które są bezpośrednio uzasadnione ze względu na ich moc: to takie, które nie są pokonane przez żaden kontrargument.

2. Następnie dodajemy wszystkie argumenty, które są pośrednio uzasadnione za pomocą argumentów z $JustArgs_1$. A mianowicie każdy argument, który ma pokonujące kontrargumenty jest dodany do $JustArgs_1$, jeśli wszystkie te kontrargumenty są ściśle pokonane przez argument z $JustArgs_1$. Powstały zbiór jest zbiorem $JustArgs_2$.

3. Powtarzamy powyższy krok dopóki otrzymamy zbiór $JustArgs_n$, do którego żaden nowy argument nie może być już dodany, który następnie jest zbiorem wszystkich argumentów uzasadnionych.

Po zdefiniowaniu argumentów uzasadnionych możemy określić:

- argumenty *odrzucone* lub przegrywające — jako te, które zostały zaatakowane przez argumenty uzasadnione;

- argumenty *nierozstrzygające* lub możliwe do obronienia — jako te argumenty, które nie są ani uzasadnione ani odrzucone.

Definicja 1.16. Argument A jest *akceptowalny* względem zbioru argumentów $Args$ wtedy i tylko wtedy, gdy każdy argument pokonujący argument$A$ jest ściśle pokonany przez argument ze zbioru $Args$.

Pokażemy na poniższym przykładzie proces tworzenia zbioru argumentów uzasadnionych.

Przykład 1.17. Niech będą dane argumenty:

$$\begin{aligned} A &= [r_0\colon \Rightarrow a, \quad r_1\colon a \Rightarrow b], \\ B &= [r_2\colon \sim b \Rightarrow c], \\ C &= [r_3\colon \Rightarrow \neg a], \\ \text{gdzie} \quad & r_0 < r_3. \end{aligned}$$

Najpierw określimy relacje ataku. Argument A pokonuje argument B, ponieważ go podkopuje. Ponadto argument C pokonuje A poprzez obalenie jego właściwego podargumentu $A_0 = [r_0]$, a tym samym obalenie argumentu A. Na podstawie tych relacji możemy skonstruować zbiór argumentów uzasadnionych w następujący sposób. Argument C nie jest pokonany przez żaden argument, ponieważ jego kontrargument jest A jest za słaby. Zatem $JustArgs_1 = \{C\}$, z kolei A nie można dodać do zbioru, dlatego że jest on pokonany przez C. Również A_0 nie może zostać dodany z tych samych powodów. W przypadku argumentu B, mimo że jest on pokonany przez kontrargument A, to możemy go dodać do zbioru $JustArgs$, ponieważ A jest ściśle pokonany przez argument ze zbioru $JustArgs_1$, a mianowicie przez C. Stąd C przywraca B i mamy, że $JustArgs_2 = \{C, B\}$. Powtórzenie tego procesu nie powoduje dodanie nowych argumentów, więc możemy tu zatrzymać: $JustArgs_3 = JustArgs_2$. Stąd dostajemy, że argumenty B, C są uzasadnione, podczas gdy A_0, A są odrzucone.

1.2.2 System II: Priorytety podważalne

Do tej pory zakładaliśmy, że istnieje porządek reguł. Teraz zbadamy sytuację, gdzie ten porządek można wyznaczyć podczas wnioskowania. W tym celu załóżmy, że nasz język zawiera predykat $\prec$, który może być wnioskiem reguły. To sprawia, że składnik porządku teorii uporządkowanej jest zbędny, zatem teoria uporządkowana jest od teraz tylko zbiorem reguł.

Definicja 1.18. Niech A i B będą dwoma argumentami. Wtedy:

1. A obala B wtedy i tylko wtedy, gdy
   - dla pewnej pary reguł $r_1 \in A$ i $r_2 \in B$:
     - r_1 i r_2 mają komplementarne następniki,
     - $r_1 \not\prec r_2$;
   - lub dla ciągu reguł $r_1, \ldots r_n \in A$ oraz pewnej reguły $r_m \in B$:
     - wniosek reguły r_m jest $x \prec y$ oraz wnioski reguł $r_1 \ldots . r_n$ są łańcuchem $y \prec z, \ldots, z' \prec x$,
     - dla każdego r_i $(1 \leqslant i \leqslant n)$: $r_i \not\prec r_m$.
2. A podkopuje B wtedy i tylko wtedy, gdy
   - wniosek A jest dopełnieniem jakiegoś założenia w B; lub
   - argument B zawiera założenie $x \prec y$ oraz A ma łańcuch wniosków $y \prec z, \ldots, z' \prec x$.

Opisany w następnym rozdziale system opierał się będzie tylko na mechanizmie bazującym na stałych priorytetach.

Rozdział 2

Koncepcja systemu

System ekspertowy został napisany w języku C# przy użyciu środowiska *Microsoft Visual Studio 2017* i platformy *.NET Framework 4.5.2*. Aby uruchomić program należy na danym komputerze mieć zainstalowane środowisko *.NET Framework*, w którym będzie on wykonywany.

2.1 Sposób zapisu wiedzy

Baza wiedzy systemu ekspertowego będzie oparta na regułowej reprezentacji wiedzy, natomiast fakty będą mieć reprezentację logiki trójwartościowej o strukturze Atrybut–Wartość. Baza wiedzy będzie zapisana w języku XML. W szczególności fakty będą postaci jak na listingu 2.1.

Listing 2.1: Zapis faktu w języku XML

```
1 <Fact>
2   <Attribute>Atrybut_faktu</Attribute>
3   <Value>1</Value>
4 </Fact>
```

Napis `Atrybut_faktu` pomiędzy znacznikami `<Attribute>` jest zmienną łańcuchową, a wartość atrybutu pomiędzy `<Value>` jest liczbą całkowitą i może przyj-

mować wartości:

$$1 \quad : \textit{prawda},$$
$$0 \quad : \textit{nie wiadomo},$$
$$-1 \quad : \textit{fałsz}.$$

Z kolei reguły będą miały postać jak na Listingu 2.2.

Listing 2.2: Zapis reguły w języku XML

```
<Rule>
  <ConditionList>
    <Condition>
      <Attribute>Warunek_1</Attribute>
      <Value>1</Value>
    </Condition>
   <Condition>
      <Attribute>Warunek_2</Attribute>
      <Value>-1</Value>
    </Condition>
  </ConditionList>
  <Conclusion>
    <Attribute>Wniosek</Attribute>
    <Value>1</Value>
  </Conclusion>
  <Priority>1</Priority>
</Rule>
```

Pomiędzy tagami `<ConditionList>` znajduje się lista warunków reguły, a każdy warunek określony jest w `<Condition>`. Znacznik `<Conclusion>` definiuje wniosek reguły, a `<Priority>` określa priorytet reguły i jest liczbą zmiennoprzecinkową typu `decimal`. Atrybuty oraz wartości warunków i wniosku określamy podobnie jak dla faktu. Na listingu 2.3 zamieszczono przykładową bazę wiedzy.

Listing 2.3: Przykładowa baza wiedzy zapisana w języku XML

```
<?xml version="1.0" encoding="utf-8"?>
<KnowledgeBase
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  name="Przykładowa baza wiedzy">

```

```
7   <FactList>
8     <Fact>
9       <Attribute>A</Attribute>
10      <Value>1</Value>
11    </Fact>
12  </FactList>
13
14  <RuleList>
15    <Rule>
16      <ConditionList>
17        <Condition>
18          <Attribute>A</Attribute>
19          <Value>1</Value>
20        </Condition>
21        <Condition>
22          <Attribute>B</Attribute>
23          <Value>0</Value>
24        </Condition>
25      </ConditionList>
26
27      <Conclusion>
28        <Attribute>W</Attribute>
29        <Value>1</Value>
30      </Conclusion>
31
32      <Priority>0,2</Priority>
33    </Rule>
34  </RuleList>
35
36 </KnowledgeBase>
```

2.2 Mechanizm wnioskowania

Szkieletowy system ekspertowy będzie posiadał mechanizm wnioskowania w przód. Przypomnijmy, że celem wnioskowania w przód jest wyznaczenie wszystkich możliwych wniosków reguł na podstawie zadeklarowanych faktów. Proces wnioskowania kończy działanie wówczas, gdy nie da się już wyprowadzić nowych faktów. U podstaw wnioskowania leży *zasada zamkniętego świata*, czyli system za prawdę uzna tylko to, co wynika z reguł i co zostało zadeklarowane przez użytkownika w postaci faktów.

2.2.1 Metoda wnioskująca

Teraz możemy przejść do implementacji mechanizmu wnioskowania. Na listingu 2.4 zamieszczono przykład implementacji wnioskowania — metoda `ExecuteReasoning()`. Przeglądamy wszystkie reguły, proces powtarzamy do momentu, gdy w ciągu danego cyklu nie został dodany żaden nowy wniosek (pętla `while`). Zdefiniowany typ wyliczeniowy `GenerateNewFact` służy jako informacja, czy systemowi udało się wygenerować nowy fakt. Metody wykorzystane w poniższym kodzie będziemy wyjaśniać w kolejnych paragrafach.

Listing 2.4: Metoda wnioskująca

```
1  public enum GeneratedNewFact {
2    True = 1,
3    False = 0
4  }
5  public enum TypeOfValue {
6    [Description("prawda")]
7    True = 1,
8    [Description("nie wiadomo")]
9    Unknown = 0,
10   [Description("fałsz")]
11   False = -1
12 }
13
14 public void ExecuteReasoning() {
15
16   List<GeneratedNewFact> results = new List<GeneratedNewFact>();
17   do {
18     results.Clear();
19
20     foreach (Rule r in Rules) {
21       if (RuleIsPositivelyResolved(r)) {
22         if (r.Conclusion.Value != TypeOfValue.Unknown)
23           results.Add(FireRule(r));
24       }
25     }
26   }
27   while (results.Contains(GeneratedNewFact.True));
28
29   ResolveConflicts(Conflict.TypeOfAttack.Undermining);
30   ResolveConflicts(Conflict.TypeOfAttack.Rebutting);
31 }
```

2.2.2 Spełnienie reguły

Przyjmujemy, że atrybut literału może mieć trzy stany: 1, 0, -1. Wartość 1 oznacza prawdę, zapis negacji klasycznej będzie odpowiadał wartości -1 (fałsz), natomiast dla słabej negacji będzie to 0 (nie wiadomo).

W tabeli 2.1 przedstawiono zasady akceptacji przez system wniosków, które będzie można dodać do bazy faktów.

1. Wniosek b dodajemy do faktów, jeśli warunek a jest faktem o wartości fałsz lub gdy brakuje o nim informacji.

2. Wniosek b uznajemy za prawdę, jeśli warunek a jest faktem o wartości fałsz.

3. Wniosek b uznajemy za prawdę, jeśli a występuje w faktach jako prawda.

4. Wniosek b dodajemy do faktów o wartości fałsz, gdy a istnieje w faktach jako prawda.

Zakładamy, że nasz system nie obsługuje warunków postaci $\sim \neg a$.

Tabela 2.1: Akceptacja wniosku

		reguła		wniosek		warunek
1.	jeśli	$\sim a \rightarrow b,$	to	$b = 1,$	gdy	$a = 0 \ \vee \ a = -1$
2.	jeśli	$\neg a \rightarrow b,$	to	$b = 1,$	gdy	$a = -1$
3.	jeśli	$a \rightarrow b,$	to	$b = 1,$	gdy	$a = 1$
4.	jeśli	$a \rightarrow \neg b,$	to	$b = -1,$	gdy	$a = 1$

Na listingu 2.5 metoda `RulePositivelyResolved()` sprawdza, czy dana reguła jest spełniona. Reguła będzie spełniona, jeśli koniunkcja wszystkich warunków będzie prawdziwa, zatem będziemy sprawdzać, czy warunki reguły znajdują się w faktach.

Listing 2.5: Metoda sprawdzająca spełnienie reguły

```
1  private bool RuleIsPositivelyResolved(Rule r) {
2
3    foreach (Literal c in r.Conditions) {
4      if (c.Value != TypeOfValue.Unknown) {
5        if (! CFacts.Contains(c))
6          return false;
7      }
8      else {
9        Literal c_false = new Literal(c.Attribute, TypeOfValue.False);
10       if (CFacts.Contains(c.Attribute) && ! CFacts.Contains(c_false))
11         return false;
12     }
13   }
14   return true;
15 }
```

2.2.3 Uaktywnienie reguły

Podczas gdy reguła jest spełniona, można ją uaktywnić (odpalić, ang. *fire*). Metoda `FireRule()` na listingu 2.6 ma na celu zweryfikowanie, czy uznany wniosek można dopisać do bazy faktów. Dla analizowanej reguły system sprawdza obecność wniosku w bazie faktów. Jeżeli lista faktów nie zawiera indeksu danej reguły oraz jej wniosek nie jest sprzeczny ze wstępnie zadeklarowanymi faktami (o indeksie -1), to wtedy taki wniosek zostanie dopisany. Metoda zwraca informację, czy udało się wygenerować nowy fakt, która jest typu wyliczeniowego.

Listing 2.6: Metoda uaktywniająca regułę

```
1  private GeneratedNewFact FireRule(Rule r) {
2
3    Fact C = new Fact(r.Conclusion, r.Id);
4    Literal neg_C = new Literal(C.Attribute,
5                          (TypeOfValue) Convert.ToInt32(-1 * (int) C.Value));
6
7    if (! CFacts.Contains(C.IdRule) && ! CFacts.Contains(new Fact(neg_C,-1))) {
8      CFacts.AssertConclusion(C);
9      return GeneratedNewFact.True;
10   }
11   return GeneratedNewFact.False;
12 }
```

2.2.4 Rozwiązanie konfliktów między regułami

Po tym, jak dodano wnioski spełnionych reguł, należy jeszcze sprawdzić wystąpienie konfliktów między tymi regułami. Służy do tego metoda `ResolveConflicts()` przedstawiona na listingu 2.7.

Listing 2.7: Metoda rozwiązująca konflikty reguł

```
1  private void ResolveConflicts(Conflict.TypeOfAttack analyzed) {
2
3    Rule r_ext, r_int;
4    int i_ext = 0;
5
6    while (i_ext < CFacts.Count) {
7      Fact f_ext = CFacts[i_ext] as Fact;
8      if (f_ext.IdRule != -1) {
9        r_ext = Rules[f_ext.IdRule];
10       int i_int = 0;
11
12       while (i_int < CFacts.Count) {
13         Fact f_int = CFacts[i_int] as Fact;
14         if (f_int.IdRule != -1 && f_ext.Id != f_int.Id) {
15           r_int = Rules[f_int.IdRule] as Rule;
16           Conflict conflict = new Conflict(CFacts, Rules);
17           Conflict.TypeOfAttack occuring = conflict.Check(r_ext, r_int);
18           Attack attack = null;
19
20           if (analyzed == Conflict.TypeOfAttack.Undermining &&
21               occuring == Conflict.TypeOfAttack.Undermining) {
22             attack = new Undermining(CFacts, Rules);
23           }
24           else if (analyzed == Conflict.TypeOfAttack.Rebutting &&
25               occuring == Conflict.TypeOfAttack.Rebutting) {
26             attack = new Rebutting(CFacts, Rules);
27           }
28           if (attack != null) {
29             attack.Execute(r_ext, r_int);
30             i_int = CFacts.Count;
31             i_ext = 0;
32           }
33         }
34         ++i_int;
35       } // while
36     }
37     ++i_ext;
38   } // while
39 }
```

Klasa `Conflict` (Listing 2.8) posiada metodę `Check()` sprawdzającą konflikt między regułami. Metoda może zwrócić odpowiedni typ ataku, a mianowicie:

- `Rebutting`, jeśli wnioski obu reguł są sprzeczne:

$$\begin{array}{ccc} wniosek_{r1} & & wniosek_{r2} \\ 1 & i & -1, \\ -1 & i & 1. \end{array}$$

- `Undermining`, jeśli wniosek rozstrzyganej reguły jest sprzeczny z warunkiem drugiej reguły:

$$\begin{array}{ccc} wniosek_{r1} & & warunek_{r2} \\ 1 & i & 0. \end{array}$$

albo zwrócić wartość `None`, gdy nie występuje żaden konflikt.

Listing 2.8: Klasa odpowiedzialna za wystąpienie konfliktu

```
public class Conflict {

  public enum TypeOfAttack {
    None = 0,
    Undermining = 1,
    Rebutting = 2
  }
  public TypeOfAttack Check(Rule r1, Rule r2) {

    Literal neg_Conclusion = new Literal(r2.Conclusion.Attribute,
        (TypeOfValue)Convert.ToInt32(-1 * (int)r2.Conclusion.Value));

    if (r1.Conclusion.Equals(neg_Conclusion))
      return TypeOfAttack.Rebutting;

    int index = r2.Conditions.IndexOf(r1.Conclusion.Attribute);
    if (index != -1) {
      Literal r2_cond = r2.Conditions[index] as Literal;
      if (r1.Conclusion.Value == TypeOfValue.True &&
          r2_cond.Value == TypeOfValue.Unknown) {
        return TypeOfAttack.Undermining;
      }
    }
    return TypeOfAttack.None;
  }
}
```

2.2.5 Atak reguł

Klasa `Attack` jest klasą abstrakcyjną i zawiera metodę abstrakcyjną `Execute()`, którą klasa dziedzicząca powinna zaimplementować. Metoda ta ma za zadanie przeprowadzić atak (Listing 2.9).

Listing 2.9: Klasa odpowiedzialna za atak

```
abstract public class Attack {
  abstract public void Execute(Rule r1, Rule r2);
}
```

Listing 2.10 przedstawia implementację ataku obalania. W metodzie `Execute()` sprawdzone zostają priorytety sprzecznych reguł. Jeżeli priorytet rozstrzyganej reguły jest większy niż drugiej, to wniosek pokonanej reguły `r2` zostaje odrzucony razem ze wszystkimi wnioskami podargumentów. Natomiast jeżeli priorytet jest mniejszy, to wniosek rozważanej reguły `r1` zostaje usunięty ze wszystkimi innymi, które zostały dodane dzięki niemu. W przypadku gdy priorytety są równe, to reguły się obalają i system odrzuca ich wnioski.

Listing 2.10: Atak obalania

```
class Rebutting : Attack {

  public override void Execute(Rule r1, Rule r2) {

    if (r1.Priority > r2.Priority) {
      Facts.RetractConclusion(Facts.IndexOf(new Fact(r2.Conclusion, r2.Id)));
      Facts.RetractAllConclusionsGeneratedByTheRule(r2, Rules);
    }
    else if (r1.Priority < r2.Priority) {
      Facts.RetractConclusion(Facts.IndexOf(new Fact(r1.Conclusion, r1.Id)));
      Facts.RetractAllConclusionsGeneratedByTheRule(r1, Rules);
    }
    else if (r1.Priority == r2.Priority) {
      Facts.RetractConclusion(Facts.IndexOf(new Fact(r1.Conclusion, r1.Id)));
      Facts.RetractAllConclusionsGeneratedByTheRule(r1, Rules);
      Facts.RetractConclusion(Facts.IndexOf(new Fact(r2.Conclusion, r2.Id)));
      Facts.RetractAllConclusionsGeneratedByTheRule(r2, Rules);
    }
  }
}
```

Podobnie jak w przypadku ataku obalania, klasa `Undermining` powinna zaimplementować metodę `Execute()` przedstawioną na listingu 2.11. Rozpatrywana reguła `r1` pokonuje regułę `r2`, ponieważ warunkiem `r2` jest słaby literał poprzedzony znakiem ∼.

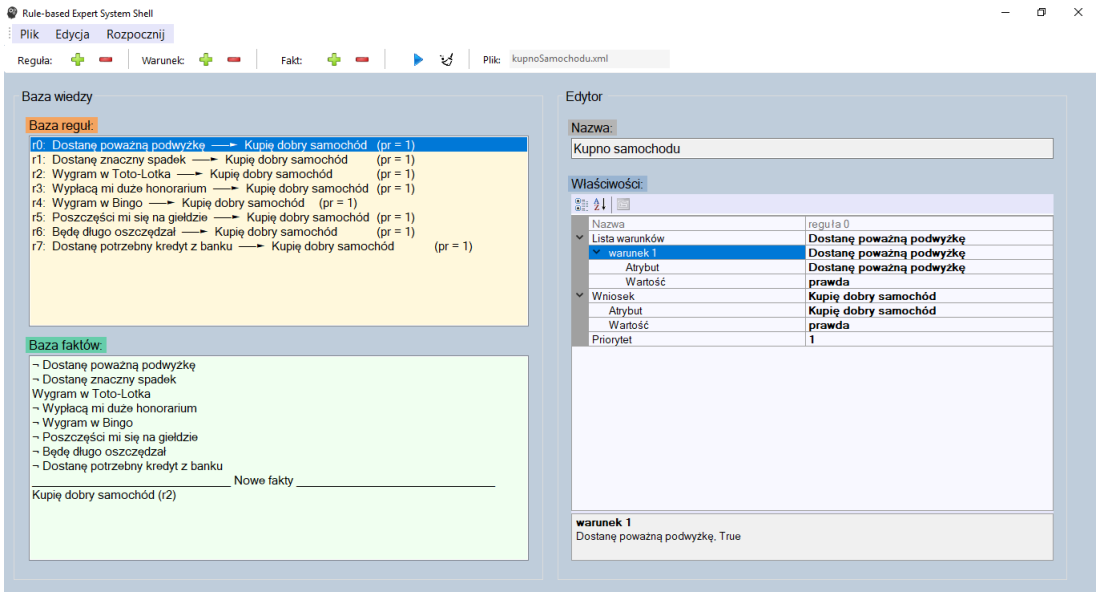
Listing 2.11: Atak podkopania

```
1 class Undermining : Attack {
2
3   public override void Execute(Rule r1, Rule r2) {
4
5     Facts.RetractConclusion(Facts.IndexOf(new Fact(r2.Conclusion, r2.Id)));
6     Facts.RetractAllConclusionsGeneratedByTheRule(r2, Rules);
7   }
8 }
```

2.3 Interfejs

Szkieletowy system ekspertowy ma możliwość wnioskowania z dowolnej bazy wiedzy. Użytkownik projektując bazę wiedzy za pomocą edytora, może dodawać lub usuwać reguły oraz fakty, zatem sam proces tworzenia jest bardzo prosty i intuicyjny. Powstałą bazę wiedzy można zapisać do pliku XML, a potem ją załadować do programu. Oczywiście można utworzyć bazę wiedzy poza wbudowanym edytorem, tylko wtedy należy to zrobić według zaleceń w podrozdziale 2.1. Interfejs systemu został przedstawiony na rysunku 2.1.

Podczas wnioskowania system poinformuje o sytuacjach, w których występuje nierozstrzygnięty konflikt reguł. Dodatkowo w celu ułatwienia zrozumienia działania mechanizmu wnioskowania, użytkownik może podejrzeć plik `LogFile.txt`, w którym został wygenerowany przebieg wnioskowania.



Rysunek 2.1: Interfejs użytkownika

2.4 Test

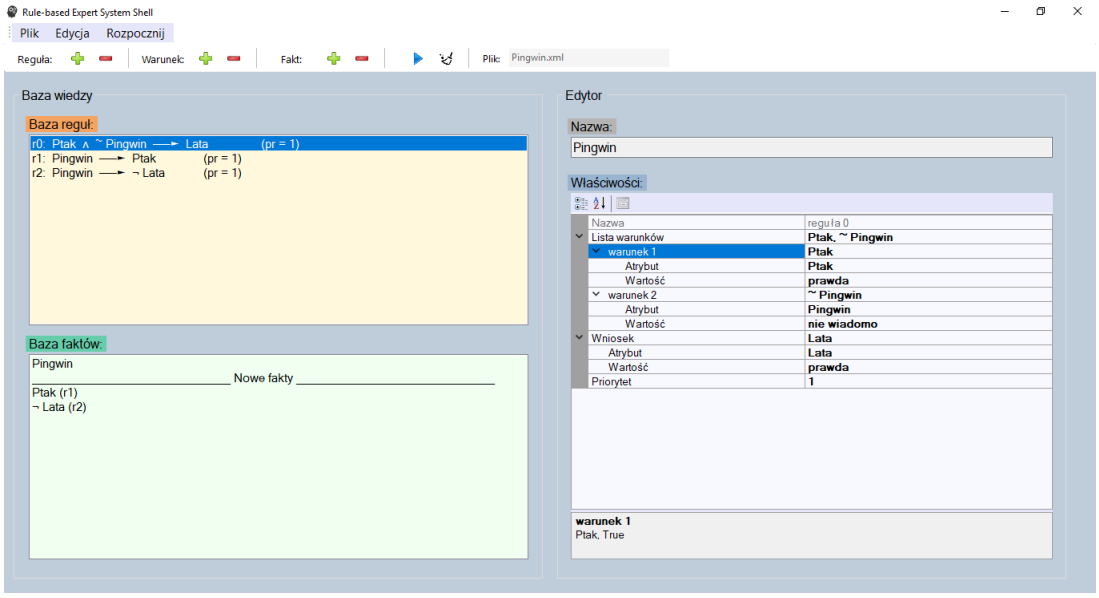
Poniższe zrzuty to przykłady typowych wnioskowań podważalnych (dyskutowane w literaturze) służących do ilustracji właściwości tego rodzaju rozumowań. Wyniki otrzymane z systemu są zgodne z wynikami zalecanymi w literaturze i z intuicją.

Na rysunku 2.2 system wywnioskował, że „pingwin nie lata" pomimo tego, że jest ptakiem. Mamy tu do czynienia z podważeniem ogólnej zasady, że „ptaki latają", ponieważ pojawił się szczególny przypadek nielatającego ptaka.

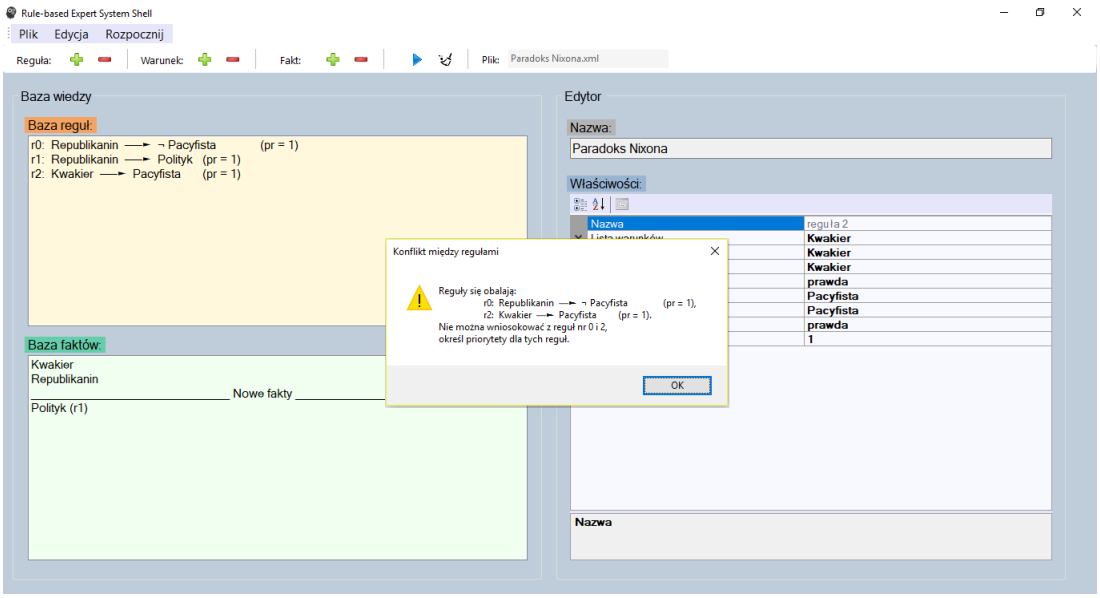
Na następnym zrzucie (Rysunek 2.3) został przedstawiony paradoks Nixona. Polega on na tym, że prezydent Nixon będąc politykiem nie powinien być pacyfistą, z kolei będąc również kwakrem, pacyfistą być powinien. W takim wypadku system informuje o zaistniałym konflikcie, ponieważ wnioski obu reguł się obalają.

Rysunek 2.3 prezentuje system z abstrakcyjną bazą wiedzy. Do listy faktów dodano wniosek b, ponieważ warunek a nie jest faktem i traktuje się go jako fałsz.

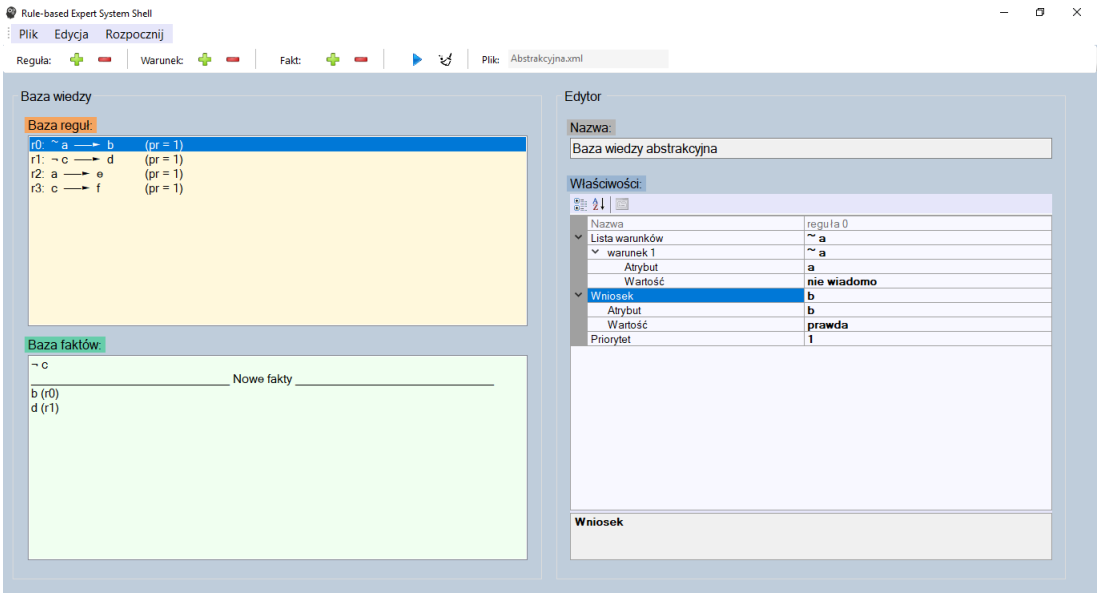
Również dopisano wniosek d , gdyż warunek $\neg c$ jest zadeklarowanym faktem. Kolejne zrzuty 2.5 i 2.6 ilustrują omówiony przykład 1.15.



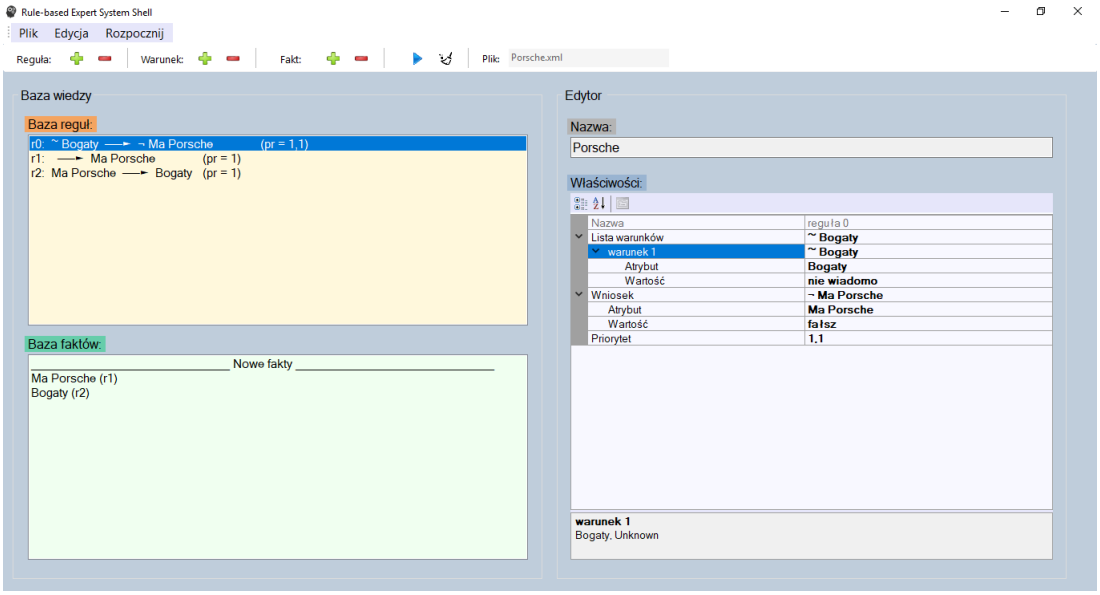
Rysunek 2.2: Pingwin nie lata



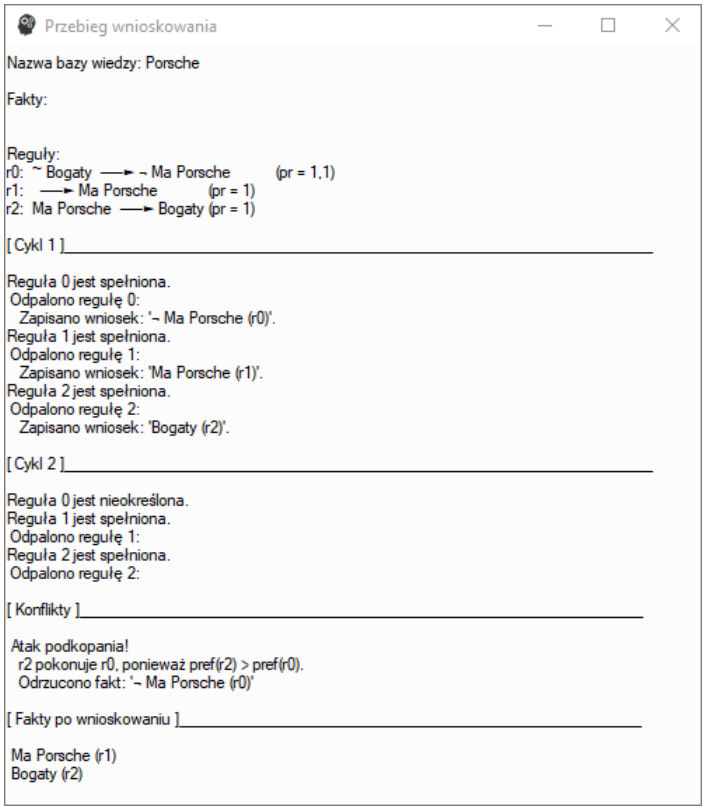
Rysunek 2.3: Paradoks Nixona



Rysunek 2.4: System z abstrakcyjną bazą wiedzy



Rysunek 2.5: Mam porsche



Rysunek 2.6: Plik z przebiegiem wnioskowania

2.5 Podsumowanie

Celem pracy było stworzenie skorupowego systemu ekspertowego pozwalającego na wprowadzanie niespójnej wiedzy i umożliwiającego rozwiązywanie konfliktów występujących w tej bazie wiedzy.

Poruszona w pracy argumentacja podważalna z użyciem stałych priorytetów została wykorzystana w systemie do rozwiązywania sprzeczności reguł. W systemie został zaimplementowany mechanizm wnioskowania w przód, który rozpoznaje i obsługuje dwa typy konfliktów: obalanie i podkopanie. Dzięki tej własności użytkownik będzie mógł korzystać z systemu bez obawy, że dodane wnioski w trakcie wnioskowania będą sprzeczne z innymi.

Stworzony system różni się od klasycznego systemu ekspertowego przede wszystkim tym, że ma zdolność rozwiązywania konfliktów. W systemie dodano możliwość zaprzeczeń poprzez zastosowanie dwóch rodzajów negacji. Negacja klasyczna służy do zaprzeczania warunku i deklarowania, że jakiś fakt jest nieprawdą, natomiast słaba negacja skutkuje tym, że brak informacji o prawdziwości warunku oznacza, że jest on fałszywy. W dodatku wykorzystano standard XML do reprezentacji wiedzy w sposób strukturalny.

Praca zakończona jest serią testów, które pokazują, że wyniki otrzymane z systemu są poprawne a cel postawiony w pracy został osiągnięty.

Bibliografia

[1] W. Duch. Sztuczna inteligencja i systemy ekspertowe. `http://www.is.umk.pl/~duch/Wyklady/AI_plan.html`. [dostęp 17.06.2017].

[2] F. H. van Eemeren, R. Grootendorst. *A Systematic Theory of Argumentation. The pragma-dialectical approach.* Cambridge University Press, 2014.

[3] L. Longo. Argumentation for Knowledge Representation, Conflict Resolution, Defeasible Inference and Its Integration with Machine Learning. *Machine Learning for Health Informatics*. 2016.

[4] S. Modgil, H. Prakken. The ASPIC+ framework for structured argumentation: A tutorial. *Argument & Computation*, vol. 5, 2014.

[5] J. J. Mulawka. *Systemy ekspertowe*. WNT, Warszawa, 1996.

[6] A. Niederliński. *Regułowo-modelowe systemy ekspertowe RMSE*. WPKJS, Gliwice, 2013.

[7] H. Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, vol. 1, 2010.

[8] H. Prakken, G. Sartor. A System for Defeasible Argumentation, with Defeasible Priorities. *Practical Reasoning*, 1996.

[9] T. Żurek. *Metody sztucznej inteligencji*. Instytut Informatyki UMCS, Lublin, 2011.

Spis rysunków i tabel

Rysunek 1.1: Architektura systemu ekspertowego 9
Rysunek 1.2: Atak obalania 20
Rysunek 1.3: Atak podkopania 21

Rysunek 2.1: Interfejs użytkownika 37
Rysunek 2.2: Pingwin nie lata 38
Rysunek 2.3: Paradoks Nixona 38
Rysunek 2.4: System z abstrakcyjną bazą wiedzy 39
Rysunek 2.5: Mam porsche 39
Rysunek 2.6: Plik z przebiegiem wnioskowania 40

Tabela 2.1: Akceptacja wniosku 31

Spis listingów

Listing 2.1: Zapis faktu w języku XML 27
Listing 2.2: Zapis reguły w języku XML 28
Listing 2.3: Przykładowa baza wiedzy zapisana w języku XML . . . 28
Listing 2.4: Metoda wnioskująca 30
Listing 2.5: Metoda sprawdzająca spełnienie reguły 32
Listing 2.6: Metoda uaktywniająca regułę 32
Listing 2.7: Metoda rozwiązująca konflikty reguł 33
Listing 2.8: Klasa odpowiedzialna za wystąpienie konfliktu 34
Listing 2.9: Klasa odpowiedzialna za atak 35
Listing 2.10: Atak obalania 35
Listing 2.11: Atak podkopania 36