# ANALIZA SYSTEMOWA I ZARZĄDZANIE

Książka jubileuszowa
z okazji
50-lecia pracy naukowej

ROMANA KULIKOWSKIEGO

**ISBN 83-85847-34-0**

Druk: "ARGRAF" Agencja Poligraficzno-Wydawnicza, Warszawa
Skład: Barbara Kotuszewska

# CONSTRAINT LOGIC PROGRAMMING – OD PROLOGU DO CHIPU*

*Antoni Niederliński*

*Instytut Automatyki, Politechnika Śląska*

W ostatnim 10-leciu rozwinęła się i osiągnęła dojrzałość efektywna i przyjazna dla użytkownika technologia programistyczna umożliwiająca nowe, deklaratywne podejście do tworzenia programów rozwiązujących trudne kombinatoryczne zagadnienia badań operacyjnych jak harmonogramowanie, planowanie, alokacja resursów. Technologią tą jest *constraint logic programming*. Artykuł omawia rozwój idei leżących u podstaw tej technologii, poczynając od jej najstarszego protoplasty jakim jest Prolog a kończąc na jej współczesnym przedstawicielu, jakim jest CHIP. Zwrócono szczególną uwagę na deklaratywność wynikającą ze stosowania klauzul Horna. Deklaratywność ta w pewnej mierze likwiduje przepaść semantyczną pomiędzy opisem problemu a rozwiązującym go programem przy konwencjonalnym rozwiązaniu. Przedstawiono algorytm *Standard Backtracking* i jego udoskonalenia, rozwiązywanie ograniczeń metodą *propagacji*, efektywność i zwięzłość programu w wyniku stosowania *list* i *rekurencji*. Przedstawiono dwa przykłady ilustrujące efektywność CHIPu.

## 1. Motywacja

Programy komputerowe rozwiązujące skomplikowane zagadnienia kombinatoryczne, w tym również optymalizację kombinatoryczną, dla celów wspomagania decyzji, były opracowywane dotychczas w oparciu o dwa różne podejścia metodologiczne:

- Stosowanie istniejących *solwerów* (np. dla całkowitoliczbowego programowania liniowego), wymagające transformacji rozwiązywanego problemu do postaci akceptowalnej przez solwer. Zaletą tego podejścia jest możliwość korzy-

* Panu Profesorowi Romanowi Kulikowskiemu, pionierowi badań operacyjnych w Polsce, z okazji 50 rocznicy pracy naukowej.

stania z mocnych i wypróbowanych algorytmów, dostarczanych przez solwer, co skraca czas i obniża koszt rozwiązania. Wadami zaś to, że transformacja jest źródłem *przepaści semantycznej* pomiędzy pierwotnym i transformowanym problemem: transformowany problem jest trudno zrozumiały (utrudnia to jego aktualizację i sprzyja błędom), pierwotny problem zostaje niekorzystnie zmodyfikowany (np. liczba zmiennych decyzyjnych może bardzo mocno wzrosnąć). Wymogi solwera mogą bardzo utrudnić lub wręcz uniemożliwić wbudowanie w program dodatkowej wiedzy o pierwotnym problemie, która to wiedza mogłaby znacznie ułatwić rozwiązanie problemu.

- Opracowanie specyficznego algorytmu rozwiązującego problem w sposób optymalny lub przybliżony i kodowanie tego algorytmu za pomocą wypróbowanego języka proceduralnego (FORTRAN, Pascal, C). Zaletami tego podejścia jest możliwość wbudowania w program wszelkiej wiedzy o pierwotnym problemie i możliwość uzyskania bardzo efektywnego programu. Wadami zaś - bardzo długi czas opracowywania i testowania algorytmu oraz programu, konieczność korzystania z wysoko kwalifikowanych i wysoko opłacanych specjalistów, *przepaść semantyczna* pomiędzy pierwotnym problemem a rozwiązującym go programem i związana z tym ograniczona modyfikowalność programu, niemożliwa wręcz do osiągnięcia bez zaangażowania twórców programu.

## 2. Co to jest constraint logic programming?

Wszyscy, którzy borykali się z wymienionymi trudnościami, zapewne zainteresują się nową możliwością rozwiązywania wymienionych problemów, możliwością pozbawioną wad tradycyjnych metodologii. Możliwością tą jest zastosowanie języka realizującego paradygmat *constraint logic programming*. Najbardziej popularnym z tych języków wydaje się być CHIP[†] (Constraint Handling In Prolog). Zawdzięcza ją zarówno swym osiągom, jak i temu, że został bardzo dobrze przedstawiony w znanej książce Van Hentenrycka.

*Constraint logic programming* (CLP) jest paradygmatem programowania łączącym w sobie *programowanie w logice* (Prolog), *rozwiązywanie ograniczeń* oraz *optymalizację*.

### 2.1 Programowanie w logice (Prolog)

Prolog jest deklaratywnym językiem programowania. Deklaratywność oznacza, że opis zadania jest programem rozwiązującym zadanie.

---

[†] CHIP jest produktem firmy COSYTEC S.A., Parc Club Orsay Université, 4, Rue Jean Rostand, F-91893 Orsay Cedex, France.

W odróżnieniu od popularnych języków proceduralnych (Basic, Pascal, C), program w języku Prolog nie służy do zapisu algorytmu rozwiązywania zadania, lecz służy do opisu zadania za pomocą pojęć logiki. Odzwierciedla to nazwa **Prolog,** będąca akronimem słów **Pro**gramming in **Log**ic.

Zadanie opisane programem w Prologu jest rozwiązywane przez *system wnioskujący,* będący częścią kompilatora lub interpretera języka Prolog. *Rozwiązywanie zadania* przez system wnioskujący polega na wyznaczaniu, z opisu zadania, logicznie zasadnych wniosków. Nazywa się to *wnioskowaniem.* Wnioskowanie jest w Prologu *za darmo.* Nie trzeba opracowywać, programować i testować algorytmu wnioskowania: zostało to zrobione (raz i dobrze) przez twórców kompilatora lub interpretera Prologu.

Opis zadania w Prologu jest wykonywany za pomocą klauzul (zdań) Horna. Klauzulami Horna nazywa się *reguły* typu:

Jeżeli jest prawdą a i b i c, to jest prawdą p.

gdzie p jest *wnioskiem* reguły, zaś a, b i c są *warunkami* reguły.

Reguły są zapisywane w Prologu w postaci:

```
p :-
a,
b,
c.
```

Symbol :- jest symbolem strzałki ← , oznaczającej w logice implikację. Stosowane przy powyższym zapisie akapity mają za zadanie uczynienie reguły lepiej czytelną.

Szczególnym przypadkiem reguły jest *fakt,* np.

q jest prawdą.

Fakt jest zapisywany w Prologu w postaci:

```
q.
```

Można wykazać, że dowolne zagadnienie dające się wyrazić w języku logiki, da się zapisać za pomocą klauzul Horna.

Zaletą zaś takiego zapisu jest to, że:

- upraszcza on bardzo automatyzację wnioskowania dla zadań stosujących ten zapis. Oznacza to. że budowa kompilatorów lub interpreterów wnioskujących jest dla tak zapisanych zadań szczególnie prosta, a same kompilatory lub interpretery są bardzo efektywne;

- jego prostota czyni go zrozumiałym i przejrzystym dla programisty.

Warunki i wnioski klauzul Horna w programach prologowych zależą najczęściej od szeregu zmiennych. Charakter tej zależności jest definiowany za pomocą *predykatów*.

*Predykatem* nazywa się *relację* pomiędzy zmiennymi, opatrzoną nazwą. Np. predykatem jest:

```
lubi(Kto,Co).
```

Określa on relację *Kto lubi Co*, zachodzącą pomiędzy zmienną *Kto* i zmienną *Co*. Zmienne w Prologu pisane są z dużej litery, stałe pisane są z małej litery lub – w cudzysłowie – z dużej litery. Łatwo zauważyć, że *predykat nie ma wartości logicznej*: zdanie lubi(Kto,Co) nie jest ani prawdą, ani nieprawdą; jest niejednoznaczne. Predykat taki może służyć do definiowania faktów, np.:

```
lubi(ewa,studia).
lubi(ewa,narty).
```

Można się nim posłużyć również do definiowania reguł. Np. reguła stwierdzająca, że jeżeli Ewa coś lubi, to lubi to również Marek, ma postać:

```
lubi(marek,Co) :-
        lubi(ewa, Co).
```

W oparciu o zadeklarowane dwa fakty i zadeklarowaną regułę, system wnioskujący może np. wnioskować dla celu lubi(marek,Co). W wyniku wnioskowania otrzymuje się:

```
lubi(marek,studia).
lubi(marek,narty).
```

Jeżeli jednak zapytamy się, czy Marek lubi Prolog, formułując cel:

```
lubi(marek,prolog).
```

to system wnioskujący udzieli nam odpowiedzi:

```
No.
```

aczkolwiek nasz Marek może uwielbiać Prolog. System wnioskujący działa bowiem w oparciu o *hipotezę zamkniętego świata*, zgodnie z którą *prawdą jest tylko to, co zostało zadeklarowane jako fakt lub wynika z zadeklarowanych faktów i reguł.*

Mechanizm, za pomocą którego Prolog wnioskuje, nosi nazwę *unifikacji*. Istotą unifikacji jest ukonkretnienie zmiennych występujących w celu na drodze przyrównania tego celu do odpowiedniej klauzuli. Tak więc cel lubi(marek,Co_lubi) jest kolejno przyrównywany do klauzul lubi(ewa,studia) i lubi(ewa,narty). Ponieważ dwie różne stałe (marek i ewa) nie są unifikowalne, kolejno przyrównuje się cel do wniosku reguły lubi(marek,Co). Unifikacja dwóch zmiennych Co i Co_lubi jest możliwa, i celem wnioskowania staje się warunek

reguły lubi(ewa,Co). Nowy cel zostaje kolejno ponownie przyrównywany do klauzul lubi(ewa,studia) i lubi(ewa,narty). Ponieważ zmienna Co jest unifikowalna z dowolną stałą, z pierwszej klauzuli wynika Co=studia, a z drugiej wynika Co=narty.

W kompilator lub interpreter Prologu jest wbudowany nie tylko *system wnioskujący*, lecz również bardzo efektywny algorytm poszukiwania wartości zmiennych spełniających zadane ograniczenia, zwany *standard backtracking* (algorytmem standardowych nawrotów). Algorytm ten bywa również nazywany algorytmem *standardowego poszukiwania w głąb*.

Jego istotę można scharakteryzować następująco:

- Dokonuje ukonkretniania kolejnych zmiennych, dopóty, dopóki ograniczenia są spełniane.
- W przypadku naruszenia ograniczeń przez wartość ostatnio ukonkretnionej zmiennej, wraca (backtracking, nawrót) do najbliższej ukonkretnionej zmiennej, dla której istnieje możliwość innego ukonkretnienia.

Zwraca się uwagę na stosowaną terminologię: *standard backtracking* jest nazwą algorytmu poszukiwań, stosującego *backtracking* jako strategię wycofywania się z niedopuszczalnych rozwiązań cząstkowych. *Backtracking* bywa jednak również stosowany w innych algorytmach poszukiwań jako strategia wycofywania się z dopuszczalnych rozwiązań cząstkowych, nie rokujących jednak pomyślnie dla następnych kroków poszukiwań.

*Standard backtracking* można zilustrować na prostym przykładzie poszukiwania drogi w labiryncie. Zgodnie z tym algorytmem, w każdym rozwidleniu drogi należy skręcić w lewo; jeżeli jednak znajdziemy się w zamkniętym korytarzu, należy wrócić (*backtrack*) do ostatniego rozwidlenia w którym jest jeszcze niewykorzystana droga w prawo, wejść w tą drogę i znów przy następnym rozwidleniu skręcić w lewo, itd.

Z obecnością *backtrackingu* jest ściśle związany *niedeterminizm* Prologu: oznacza to, że jeżeli zadanie sformułowane w Prologu dopuszcza wielość rozwiązań, wszystkie te rozwiązania zostaną w wyniku *backtrackingu* wyznaczone. A więc jeżeli odnaleźliśmy jedną drogę wyprowadzającą nas z labiryntu, możemy poszukiwać następnych dokonując *backtrackingu* od wyjścia z labiryntu.

Podobnie jak w przypadku wnioskowania, również *backtracking* jest w Prologu *za darmo*. Użytkownik nie musi niczego skomplikowanego zrobić, by wywołać *backtracking*: w najbardziej złożonym przypadku wystarczy zakończyć koniunkcję warunków reguły, dla której chcemy *backtrackingu*, słowem kluczowym *fail*. Podstawą zaś jest – jak w przypadku wnioskowania - opis zadania, którego rozwiązanie wymaga *backtrackingu*.

Na deklaratywny charakter programu prologowego składa się nie tylko stosowanie języka klauzul Horna. Składa się nań również *synergiczne* współdziałanie *list*, będących podstawowymi strukturami danych, z *rekurencją*, będącą podstawową formą opisu zależności funkcyjnych za pomocą reguł.

Lista (np. kolejnych 5 liczb całkowitych) jest zapisywana w postaci [1,2,3,4,5]. W Prologu lista jest definiowana jako składająca się z:

- *głowy* (H), będącej pierwszym elementem listy;
- *ogona* (T), będącego listą pozostałą po usunięciu głowy.

Ogólny zapis listy ma postać: [H|T], z kreską pionową | oddzielającą głowę od ogona. Definicja listy jest w istocie swej rekurencyjna: ponieważ *ogon* jest *listą*, ma zapewne *głowę_ogona*, po usunięciu której otrzymamy listę będącą *ogonem_ogona*. Ta lista ma zapewne (przy odpowiednio dużej liczbie elementów list pierwotnej) głowę, którą nazwiemy *głową_ogona_ogona*. Jeżeli ją usunąć, otrzyma się listę będącą *ogonem_ogona_ogona*, itd., aż do otrzymania listy jednoelementowej [E], mającej już tylko głowę. Jeżeli ową głowę usuniemy, powstanie *lista pusta* [], która nie ma już ani głowy, ani tym bardziej ogona.

*Regułą rekurencyjną* nazywa się regułę, której jednym z warunków jest wniosek tej reguły. Zilustrujemy to na prostym przykładzie reguły, w którym rekurencja współdziała z listą: jest to reguła zbudowana w oparciu o predykat element(Element_listy, Lista), a definiująca przynależność elementu do listy:

```
element(G,[G|_]).  % Głowa listy jest elementem listy
element(E,[_|O]) :- % E jest elementem listy, jeżeli jest elementem ogona
element(E,O).
```

Należy zwrócić uwagę na czystą deklaratywność tej definicji. Stosując ją w programie prologowym, nie każe się komputerowi cokolwiek robić, po prostu opisujemy pojęcie „element listy". Opis ten wystarcza jednak każdorazowo kompilatorowi lub interpreterowi Prologu do wyznaczenia wszystkich elementów dowolnej listy lub też rozstrzygnięcia zagadnienia przynależności jakiegoś elementu do listy.

Stosowanie list i rekurencji leży u podstaw bardzo dużej *zwięzłości* programów prologowych. Definicje prologowe są z reguły znacznie krótsze od odpowiednich definicji napisanych w językach proceduralnych.

Prolog był pierwszym językiem programowania, przy stosowaniu którego możliwym się stało rozdzielenie *opisu problemu* od *rozwiązania problemu* i uzyskanie tego, że sam opis problemu wystarcza do jego rozwiązania. Właściwość ta jest ogromna zaletą przy rozwiązywaniu problemów kombinatorycznych i dzięki niej Prolog stał się protoplastą rodziny języków typu *constraint logic programming* (np. CHIP - Constraint Handling In Prolog, Eclipse, Charme, Decision Power), przeznaczonych właśnie do rozwiązywania problemów kombinatorycznych

z ograniczeniami, w tym również problemów optymalizacji kombinatorycznej z ograniczeniami. Języki typu *constraint logic programming*, ze względu na dziedziczoną po Prologu deklaratywność i relacyjną (predykatową) formę reguł, przekroczyły *barierę semantyczną* pomiędzy opisem problemu a rozwiązaniem problemu: umożliwiają one prosty zapis problemu, będący zarazem programem jego rozwiązywania.

## 2.2 Rozwiązywanie ograniczeń

Problem *rozwiązywania ograniczeń* można sformułować następująco: dla danego zbioru zmiennych, zbioru dopuszczalnych wartości tych zmiennych (domen) oraz zbioru ograniczeń dla zmiennych, wyznaczyć wartości zmiennych ze zbioru ograniczeń i domen. Obecność różnorakich ograniczeń ma znaczenie podstawowe w zagadnieniach harmonogramowania, timetablingu, alokacji resursów i szeregu innych zadań badań operacyjnych. Konwencjonalne podejście do tych problemów sprawia, że *im więcej ograniczeń, tym gorzej*: wzrost liczby ograniczeń zwiększa bowiem czas rozwiązywania problemu. Tymczasem na *zdrowy chłopski rozum* powinno być akurat odwrotnie: im więcej ograniczeń, tym mniejszy zbiór rozwiązań dopuszczalnych, a więc tym prostszy problem i tym krótszy czas rozwiązywania. Tą *zdrowochłopską* właściwość ma właśnie rozwiązywanie ograniczeń w językach typu *constraint logic programming*.

Prymitywnym sposobem rozwiązywania ograniczeń równościowych i implikacyjnych w dziedzinie termów prologowych dysponował już Prolog. Rozwiązywanie ograniczeń było w Prologu realizowane przez algorytmy standard backtracking i algorytm unifikacji.

Dziedzictwo prologowe zostało w CHIPie rozwinięte, przede wszystkim w odniesieniu do *standard backtrackingu*, który został zastąpiony bardziej efektywnymi strategiami poszukiwań (*Forward Checking*, *Looking Ahead*) wspomaganymi efektywnymi strategiami numeracji (First Fail), oraz w odniesieniu do *unifikacji*, która została zastąpiona *propagacją ograniczeń*.

*Propagacja ograniczeń* polega na aktualizowaniu domen zmiennych przy każdorazowym uwzględnieniu nowego ograniczenia.

Ilustruje to następujący przykład dla zmiennych o domenach dyskretnych:

```
                    Stan początkowy:
Zmienna X, domena początkowa 1,2,...,10
Zmienna Y, domena początkowa 1,2,...,10
Zmienna Z, domena początkowa 1,2,...,10
                    Ograniczenie 1:  Y<Z
Wynik propagacji ograniczenia 1:
Zmienna X, domena aktualna 1,2,...,10
Zmienna Y, domena aktualna 1,2,...,9
```

```
Zmienna Z, domena aktualna 2,3,...,10
                              Ograniczenie 2: X=Y+Z
Wynik propagacji ograniczenia 2:
Zmienna X, domena aktualna 3,4,...,10
Zmienna Y, domena aktualna 1,2,...,8
Zmienna Z, domena aktualna 2,3,...,9
                              Ograniczenie 3: X=Z+3
Wynik propagacji ograniczenia 3:
Zmienna X, domena aktualna 5,6,...,10
Zmienna Y, domena aktualna 1,2,...,6
Zmienna Z, domena aktualna 2,3,...,7
```

Po uwzględnieniu dowolnego ograniczenia, dla każdej wartości każdej zmiennej istnieją wartości pozostałych zmiennych, spełniające wszystkie dotychczas uwzględnione ograniczenia.

W CHIPie możliwa jest propagacja ograniczeń dla domeny termów prologowych, domeny zmiennej dyskretnej, domeny zmiennej boolowskiej i domeny zmiennej rzeczywistej przy ograniczeniach liniowych.

Udoskonalony algorytmy poszukiwań zastępujące w CHIPie *Standard Backtracking* są następujące:

- *Forward Checking*: po ukonkretnieniu nowej zmiennej, z domen pozostałych zmiennych są usuwane wartości sprzeczne z już ukonkretnionymi zmiennymi.

  Zdarzeniem wyzwalającym *backtracking* nie jest naruszenie ograniczenia (jak w przypadku *Standard Backtracking*), lecz nieuchronność naruszenia ograniczenia w następnym kroku, co sygnalizuje *pusta domena.*

- *Looking Ahead* + *Forward Checking*: po ukonkretnieniu nowej zmiennej, z domen pozostałych zmiennych usuwane są wartości sprzeczne z już ukonkretnionymi zmiennymi (*Forward Checking*). Dodatkowo sprawdza się, czy domeny zmiennych nie ukonkretnionych mają wartości dopuszczalne (*Looking Ahead*).

  Zdarzeniem wyzwalającym backtracking nie jest naruszenie ograniczenia (jak dla *Standard Backtracking*), lecz nieuchronność naruszenia ograniczenia w następnym kroku (jak dla *Forward Checking*) oraz nieuchronność naruszenia ograniczenia w dalszych krokach (*Looking Ahead*), co sygnalizuje obecność domen niepustych lecz bez wartości dopuszczalnych.

  Podobnie jak *Standard Backtracking* w Prologu, *Forward Checking* i *Looking Ahead* + *Forward Checking* są w CHIPie za darmo, tzn. nie wymagają programowania.

Zadaniem *strategii numeracji* jest wybór kolejnych zmiennych ukonkretnianych w trakcie poszukiwań. CHIP udostępnia szereg takich strategii; najczęściej stosowaną jest strategia *First Fail*, zgodnie z którą jako pierwsza jest ukonkretniana zmienna mająca najmniejszą domenę.

Wymienione strategie poszukiwań w ogromny sposób zwiększają efektywność rozwiązań problemów kombinatorycznych.

### 2.3 Optymalizacja

W CHIPie optymalizacja jest specyficznym problem rozwiązywania ograniczeń: warunki opisujące rozwiązywanie ograniczeń (tzn. wyznaczenia zbioru rozwiązań dopuszczalnych) zostały uzupełnione warunkiem minimalizacji/maksymalizacji wskaźnika jakości. Użytkownik CHIPu może korzystać z *Branch-and-Bound* oraz z programowania liniowego dla zmiennych ciągłych.

Obydwie metody są - w postaci predykatów optymalizacyjnych - zintegrowane z pozostałymi predykatami pakietu w sposób umożliwiający ich stosowanie bez doprowadzenia problemu do postaci standardowej. Dzięki temu nie powstaje przepaść semantyczna pomiędzy problemem a rozwiązującym go programem.

Obydwie metody są w CHIPie *za darmo*.

Mocną stroną CHIPu jest optymalizacja kombinatoryczna, realizowana za pomocą *Branch-and-Bound*. *Branch-and-Bound* nie jest algorytmem *sensu stricto*, lecz ogólną ideą optymalizacji kombinatorycznej. Polega ona na:

- znalezieniu dolnego ograniczenia wartości wskaźnika jakości (*bound*),
- odrzuceniu wszystkich tych gałęzi drzewa rozwiązań, które dają wartości gorsze od aktualnego dolnego ograniczenia (*branch*),
- aktualizacji dolnego ograniczenia w przypadku znalezienia rozwiązania lepszego od stosowanego dolnego ograniczenia.

Aczkolwiek *Branch-and-Bound* umożliwia uzyskanie optimum globalnego dla nieliniowego wskaźnika jakości przy ograniczeniach nieliniowych, w CHIPie zaimplementowano je dla liniowych wskaźników jakości przy ograniczeniach liniowych. Wynika to głównie z braku możliwie uniwersalnych metod propagacji ograniczeń nieliniowych. Prace nad tymi metodami stanowią swego rodzaju front badań w zakresie *constraint logic programming*. *Branch-and-Bound* w CHIPie jest wspomagane strategiami poszukiwań *Forward Checking* oraz *Looking Ahead+Forward Checking*, pomocnymi w zakresie propagacji ograniczeń i inicjacji *backtrackingu*:

- dla Branch-and-Bound + Forward Checking inicjowanie backtrackingu następuje:
  a) w wyniku uzyskania wartości wskaźnika jakości gorszego od aktualnego ograniczenia (Branch-and-Bound),
  b) w wyniku uzyskania pustej domeny (Forward Checking).

- dla Branch-and-Bound + Looking Ahead + Forward Checking inicjowanie backtrackingu następuje:
  a) w wyniku uzyskania wartości wskaźnika jakości gorszego od aktualnego ograniczenia (*Branch-and-Bound*),
  b) w wyniku braku wartości dopuszczalnych w domenach niepustych (Looking Ahead),
  c) w wyniku uzyskania pustej domeny (*Forward Checking*).

Programowanie liniowe dla zmiennych ciągłych jest w CHIPie realizowane w oparciu o inkrementalny algorytm Simplex, generujący najbardziej ogólne rozwiązanie. *Inkrementalność* oznacza, że pojawienie się nowego ograniczenia nie wymaga rozwiązywania problemu od początku. *Rozwiązanie najbardziej ogólne* jest zaś rozwiązaniem dopuszczającym zakresy wartości dla pewnych zmiennych decyzyjnych.

Ponadto zmienne decyzyjne przetwarzane przez ten algorytm nie muszą być nieujemne.

## 3. Przykłady

**Przykład 1: optymalizacja kombinatoryczna**

Sposób rozwiązywania kombinatorycznego problemu optymalnego harmonogramowania metodą *Branch-and_Bound* w CHIPie i rozwiązanie ilustruje następujący prosty przykład:

```
Dla 7 zadań dane są czasy trwania i obciążenia wspólnego resursu:

Zadanie     Czas trwania     Obciążenie wspólnego resursu
 1               16                    2
 2                6                    9
 3               13                    3
 4                7                    7
 5                5                   10
 6               18                    1
 7                4                   11

W jakiej kolejności wykonać zadania, by całkowite obciążenie
wspólnego resursu nie przekraczało nigdy 13, a czas wykonania
wszystkich zadań był minimalny?
```

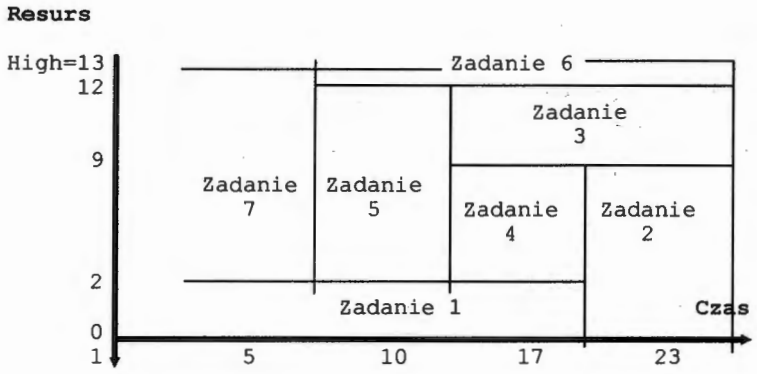
Odpowiedni program w języku CHIP v.4.1 ma postać:

```
top:-
  LO = [O1,O2,O3,O4,O5,O6,O7], %lista czasów startu zadań
  LD = [16, 6,13, 7, 5,18, 4],      %lista czasów trwania zadań
  LE = [E1,E2,E3,E4,E5,E6,E7],  %lista czasów zakończenia zadań
  LR = [2, 9, 3, 7, 10, 1,11],      %lista obciążeń resursu zadaniami


  LO :: 1..100,    %deklaracja domeny czasów startu
  End :: 1..100,   %deklaracja domeny czasu zakończenia
  LE :: 1..100,    %deklaracja domeny czasów zakończenia
  High :: 1..13,   %deklaracja domeny resursu


  nastepuje(LO,LD,LE), % czas zakończenia zadania następuje
                       % po czasie startu + czas trwania
  maximum(End,LE),   % End jest najpóźniejszym czasem
                     % zakończenia zadania z listy LE
  cumulative(LO,LD,LR,LE, unused,High,End,unused),
                     % cudowny predykat standardowy, ograniczający
                     % obciążenie wspólnego resursu


  min_max(labeling(LO),End),  % tak deklaruje się minimalizację
                              % czasu End na drodze doboru
                              % elementów listy LO


  write('LO = '), writeln(LO),     % deklaracja pisania wyników
  write('LE = '), writeln(LE),
  write('High = '), writeln(High),
  write('End = '), writeln(End).


nastepuje([],[],[]).              % rekurencyjna definicja następstwa
nastepuje([S|Ss],[D|Ds],[E|Es]) :-
  S+D #= E,                       % propagowane ograniczenie
  nastepuje(Ss,Ds,Es).
labeling([]).              % rekurencyjna definicja
labeling([X|Y]):-          % numerowania zmiennych
  indomain(X),
  labeling(Y).
```

Rozwiązanie optymalne, uzyskane na PC P200 w czasie tak krótkim, że nie można go było zmierzyć, ma postać:

LO=[ 1,17,10,10, 5, 5,1]
LE=[17,23,23,17,10,23,5]
High=13
End=23

Jego postacią graficzną jest następujący wykres Gantta:




## Przykład 2: optymalizacja ciągła liniowa

Senator może zaciągnąć korzystny kredyt:
100 mil. zł. na 3 lata ze stałym oprocentowaniem 11% rocznie. Inflacja jest prognozowana na pierwszy rok 15%, na drugi rok 14%, a na trzeci rok 12%.
Przyjazny dyrektor banku zaproponował ponadto senatorowi, by sam ustalił raty spłat kredytu, sugerując jedynie, by roczna rata spłaty nie była mniejsza niż 10 mil. zł.
Z prognoz senatora wynika, że pod koniec pierwszego roku może przeznaczyć na spłaty nie więcej niż 30 mil. zł, pod koniec drugiego roku nie więcej niż 60 mil. zł, a pod koniec 3 roku nie więcej niż 80 mil. zł.
Jak senator powinien ustalić raty, by spłacić kredyt przy minimalnych prognozowanych rzeczywistych kosztach spłaty kredytu?
Czemu będzie równy minimalny prognozowany rzeczywisty koszt spłaty kredytu?

Odpowiedni program w języku CHIP v.4.1 ma postać:

```
top :-
  A ^>= 10000000,   % deklarowanie ograniczeń na ratę po 1 roku
  A ^<= 30000000,
  B ^>= 10000000,   % deklarowanie ograniczeń na ratę po 2  roku
  B ^<= 60000000,
  C ^>= 10000000,    % deklarowanie ograniczeń na ratę po 3 roku
    C ^<= 80000000,
  raty([A,B,C], 100000000), % definicja jest poniżej
  Koszt_splaty ^= A*0.85+0.85*0.86*B+0.85*0.86*0.88*C,
  rmin(Koszt_splaty),        % deklaracji minimalizacji wskaźnika
                             % jakości za pomocą inkrementalnej
                             % metody Simplex
  roundup(A,AR),
  roundup(B,BR),
  roundup(C,CR),
  roundup(Koszt_splaty,Koszt),
  write('A = '),write(AR),write(' zł.'),nl,
```

```
  write('B = '),write(BR),write(' zl.'),nl,
  write('C = '),write(CR),write(' zl.'),nl,
  write('Minimalny prognozowany rzeczywisty koszt splaty ='), write (Koszt), write(' zl.')

raty([],Kredyt) :-     % dla kredytu zerowego lista rat jest pusta
       Kredyt^=0.
raty([Rata|Lista_rat],Kredyt) :-
  Kwota_do_zaplacenia^=(1+11/100)*Kredyt-Rata,
  raty(Lista_rat,Kwota_do_zaplacenia).
      % Lista rat [Rata|Lista_rat] odpowiada
      % kredytowi Kredyt, jeżeli Kwota_do_zapłacenia
      % jest dyskontowanym kredytem Kredyt minus najbliższa
      % rata Rata i lista Lista_rat odpowiada
      % kredytowi Kwota_do_zaplacenia.
```

```
Rozwiązanie optymalne, uzyskane na PC P200 w czasie tak krótkim,
że nie można go było zmierzyć, ma postać:

A = 10000000 zł
B = 40037928 zł
C = 80000000 zł
Minimalny prognozowany rzeczywisty koszt splaty = 89230126
```

## 4. Wnioski

Przedstawione przykłady dobrze ilustrują podstawową cechę *CLP* w ogólności, a w szczególności *CHIPu*: zmniejszenie *przepaści semantycznej* pomiędzy sformułowaniem problemu a programem problem rozwiązującym. Deklaratywność programowania sprawia, że *opis (model) problemu* jest *programem problem rozwiązującym.* Wynika z tego szereg niezwykle pożytecznych właściwości np.:

- liczba zmiennych jest i pozostaje taka sama jak pierwotnie w problemie; nie ma bowiem potrzeby sprowadzania problemu do jakiejś postaci kanonicznej, wymaganej przez metodę rozwiązywania, i zwykle pomnażającej liczbę zmiennych;
- liczba i rodzaj ograniczeń mogą być w prosty sposób zmieniane;
- rozmiary programów i ich czasy tworzenia ulegają dużemu skróceniu: szereg ważnych działań (algorytm poszukiwań, propagacja ograniczeń) otrzymuje się bowiem *za darmo*;
- małe rozmiary programów i krótki czas ich tworzenia ułatwiają z kolei eksperymentowanie z różnymi wariantami formułowania i rozwiązywania problemu.

Twórcy CHIP-a – tworząc swój program – stworzyli zarazem język, którym można w precyzyjny sposób *opisywać* złożone problemy kombinatoryczne, w szczególności problemy optymalizacji kombinatorycznej, nawet wtedy, gdy nie-

koniecznie pragnie się je zaraz rozwiązywać. Jest to możliwe dzięki wyjątkowo udanemu doborowi szeregu predykatów-ograniczeń wysokiego poziomu, jak np. cumulative(_,...,_), among(_,..,_), diffn(_,...,_), element(_,...,_) i cycle(_,...,_). Być może predykaty te (lub im podobne) będą w przyszłości, w naszym myśleniu o układach kombinacyjnych, odgrywać podobną rolę jak *dx/dt* w naszym myśleniu o układach ciągłych.

**Literatura**


1. COSYTEC S.A.: *CHIP Primer. V. 4.0.*, 1993.
2. COSYTEC S.A.: *CHIP User's Guide. V. 4.0.*, 1993.
3. COSYTEC S.A.: *CHIP Reference Manual V. 4.0*, 1993.
4. Jaffar, J. and J.-L. Lassez: *Constraint logic programming: A survey*. Journal of Logic Programming, 19/20, 503-581, May/July 1994.
5. Kumar, V.: *Algorithms for Constraiont Satisfaction Problems*. A Survey. AI Magazine, **13**, (1), 32-44, 1992.
6. Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*. The MIT Press, Cambridge, Mass. 1989.

ISBN 83-85847-34-0