*Wrocław, 14. grudnia 2006*

**Zaawansowane Metody Sztucznej Inteligencji**

***Politechnika Wrocławska***
***Wydział Informatyki i Zarządzania***
***V rok studiów***

# Środowisko do implementacji systemów w języku logicznym z ograniczeniami

# MOZART

Autor dokumentu: **STAWARZ Paweł**
Indeks: **125939**
Data wygłoszenia: **30.11.2006**
Data dostarczenia: **14.12.2006**

Prowadzący: **dr Maciej Piasecki**

# Spis treści

# Abstrakt

W niniejszej pracy omówione zostanie przykładowe środowisko do implementacji systemów w języku logicznym z ograniczeniami – Mozart.

W pierwszej części odpowiemy sobie na pytanie „czym właściwie jest Mozart i cóż ciekawego w sobie skrywa?”. Zwrócimy uwagę na jego historię oraz omówimy najważniejsze warunki licencyjne. Zaglądając do jego wnętrza postaramy się krótko przyjrzeć narzędziom, które nam dostarcza. Następnie zobaczymy kilka prostych przykładów sprawnie działających aplikacji i dowiemy się w skrócie o kilku ciekawszych projektach zrealizowanych w środowisku Mozart.

W drugiej części przejdziemy do krótkiego omówienia języka programowania Oz skupiając się przede wszystkim na technikach związanych z programowaniem z ograniczeniami. Pojawi się tutaj trochę teoretycznych pojęć wraz z kilkoma ciekawymi przykładami.

Ostatecznie skupimy się na prezentacji jednego z narzędzi Mozarta – Oz Explorer – na podstawie prostej procedury rozwiązującej przykładowy problem z ograniczeniami.

# MOzart

Jak sama nazwa wskazuje MOzart jest czymś, co pozwala nam wczuć się w rolę wielkiego twórcy dzieł sztuki (*art.*) w kranie Oz. W niniejszej części spróbujemy bacznie się przyjrzeć Mozartowi z każdej jego strony.

## Czym jest?

Okazuje się, że Mozart jest czymś więcej niż zwykłym środowiskiem do implementacji systemów w języku logicznym z ograniczeniami. Mozart jest przede wszystkim zaawansowanym środowiskiem do implementacji inteligentnych i rozproszonych aplikacji, zarówno ogólnego przeznaczenia jak i tych rozwiązujących trudne problemy wymagające wyrafinowanej optymalizacji oraz możliwości wnioskowania. Mozart wykorzystywany jest na przykład do tworzenia aplikacji związanych z rozumieniem języka naturalnego, reprezentacją wiedzy, planowaniem, rozmieszczaniem zasobów, systemami wieloagentowymi, itp. Fundamentem tego środowiska jest zaimplementowany język programowania Oz, o którym powiemy sobie trochę więcej w drugiej części pracy. W związku z językiem Oz głównymi zaletami Mozarta są możliwości przeprowadzania otwartych obliczeń rozproszonych, wnioskowania logicznego oraz wnioskowania logicznego z ograniczeniami jak również współbieżności. Aplikacje tworzone Mozartem cechuje mile widziana wieloplatformowość, tzn. „napisane raz uruchamiają się wszędzie”.

## Historia

Pierwsza wersja Mozarta powstała praktycznie na początku lat 90. za sprawą grupy badawczej Gerta Smolki z Uniwersytetu w Saarlandzie i nazywała się ona wówczas DKFI Oz. Od 1999 roku system (od tej chwili Mozart) był dalej rozwijany przez międzynarodową grupę Mozart Consortium, w której skład wchodziły przede wszystkim trzy główne uczelnie: Saarland University, Swedish Institute of Computer Science, Université catholique de Louvain. Od 2005 roku odpowiedzialność za dalszy rozwój Mozarta została przekazana grupie Mozart Board, w której skład wchodzi 9 osób z wymienionych wcześniej uczelni oraz dodatkowo Laboratoire d`Informatique Fondamentale de Lille, Kungliga Tekniska högskolan, Göteborgs Universitet, Universidad Javeriana. Grupa ta zarządza procesem ulepszania Mozarta, np. poprzez przegłosowywanie propozycji ulepszeń Mozarta (tzw. *Mozart Enhancement Proposals*) zgłaszanych przez wszystkich jego użytkowników oraz poprzez nadawanie odpowiednich praw wolontariuszom społeczności Mozartowskiej, która implementuje wybrane poprawki. Obecnie najświeższą wersją Mozarta jest wersja 1.3.2 wydana 15. czerwca 2006 roku.

## Licencja

Mozart jest całkowicie darmowym środowiskiem implementacyjnym. Z oficjalnej strony Mozarta (www.mozart-oz.org) można ściągnąć jego wersję na dowolną platformę systemową Windows/Unix wraz z kodami źródłowymi ze wskazanego CVS-u. Licencja, zwana ‘X11 style’, w ogólnym sensie pozwala na wykorzystywanie Mozarta w dowolnym celu – nawet komercyjnym. W szczególności licencja przekazuje użytkownikom wszelkie prawa do kopiowania, modyfikowania, rozpowszechniania i licencjonowania

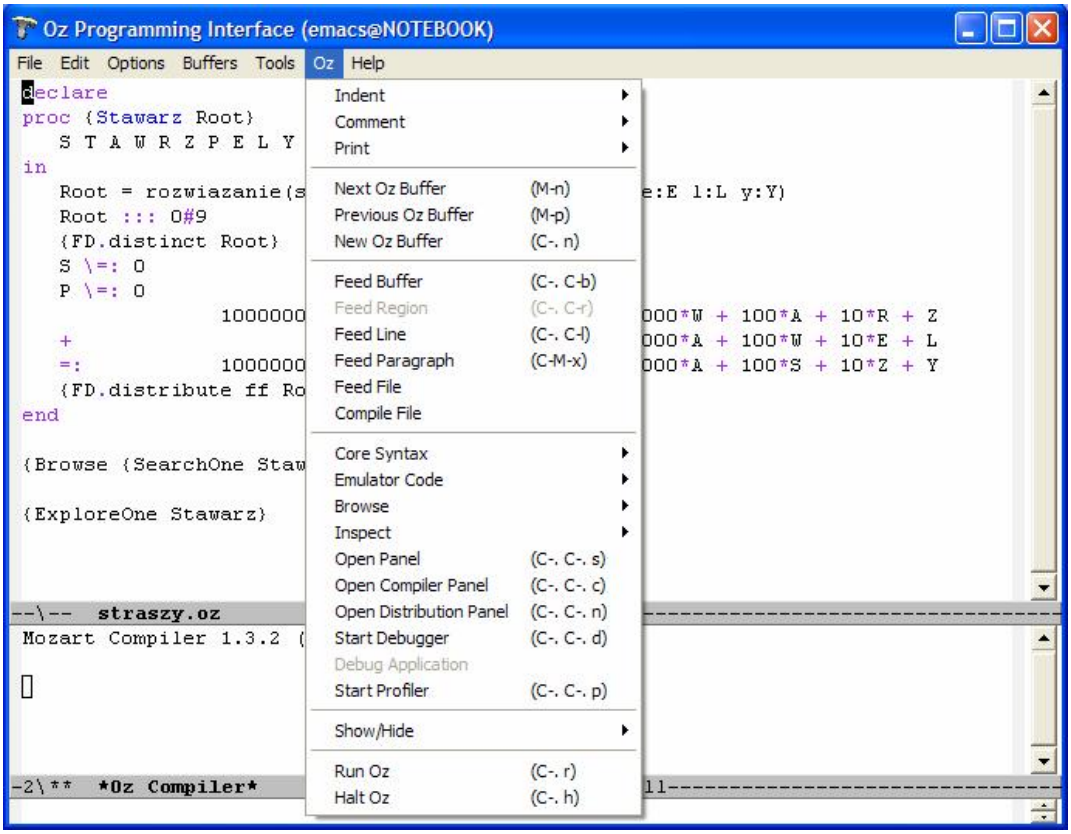
oprogramowania i dokumentacji w dowolnym celu pod warunkiem zachowania uwag dotyczących praw autorskich. Prawa autorskie oprogramowania wraz z dokumentacją spoczywają na Saarland University, Swedish Institue of Computer Science, German Research Center for Artificial Intelligence i innych. Licencja pozwala również nadawać prawa autorskie osobom przyczyniającym się do modyfikacji systemu wraz z wprowadzaniem odrębnych warunków licencyjnych dla poczynionych zmian (pod warunkiem umieszczenia ich w widocznym miejscu). Warto zwrócić dodatkowo uwagę na fakt, iż licencja podkreśla, że ani autorzy systemu, ani jego dystrybutorzy, nie są odpowiedzialni za żadne zniszczenia wynikające z użytkowania zarówno oprogramowania jak i dokumentacji oraz nie są zobowiązani do naprawiania, aktualizowania i ulepszania oprogramowania.

# Narzędzia

Mozart jako zaawansowane środowisko implementacyjne dostarcza szeregu narzędzi ułatwiających pracę przy tworzeniu nowych aplikacji. W niniejszym podrozdziale postaramy się w skrócie przejrzeć ich podstawowe funkcjonalności. W poszukiwaniu ewentualnych szczegółów warto przejrzeć dokumenty [1]-[9].

## Oz Programming Interface (OPI)

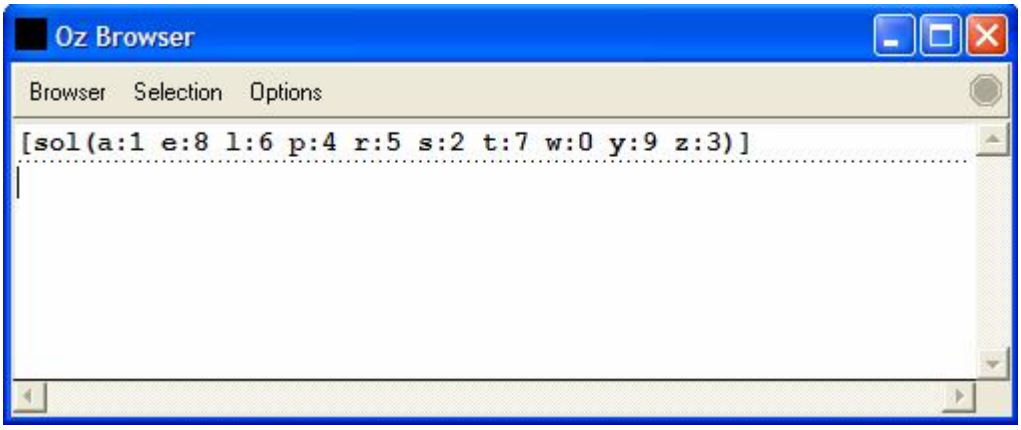
OPI jest głównym narzędziem w interakcji z Mozartem. Podpięty pod Emacsa dostarcza możliwości edycji (kolorowania) kodu Oz oraz udostępnia (poprzez menu oraz skróty klawiszowe) dodatkowe, niżej omówione narzędzia. Wraz z uruchomieniem OPI uruchamiany jest jednocześnie Oz Engine (patrz niżej).




Rysunek 1. Emacs rozszerzony o funkcjonalność OPI.

## Oz Browser

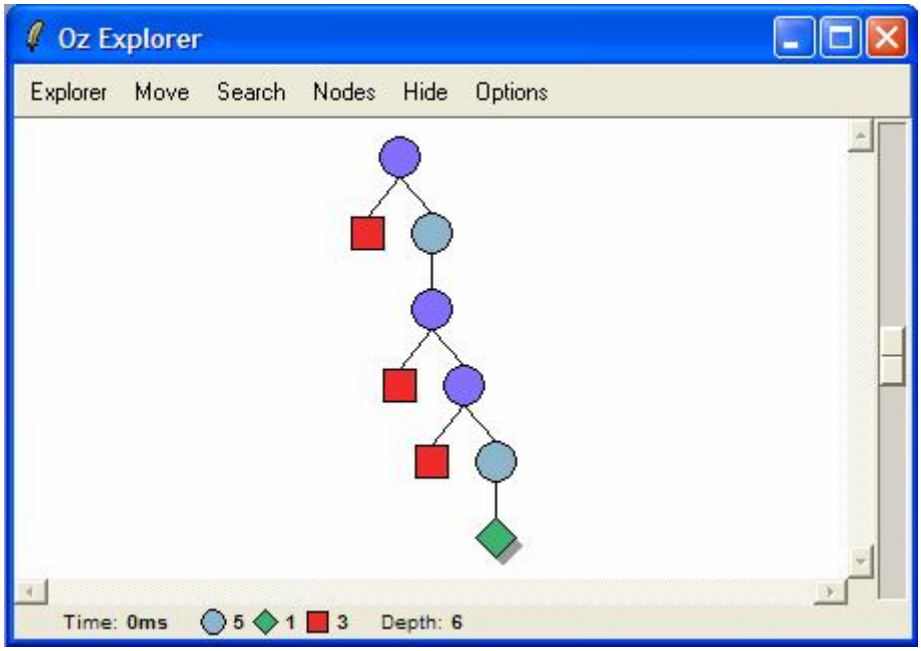
Oz Browser jest współbieżnie działającym wyjściem do wyświetlania informacji o wartościach zmiennych w trakcie działania programu. Może być wykorzystany w aplikacji Oz jako wbudowane narzędzie do przeglądania wyjścia aplikacji.




Rysunek 2. Oz Browser.

## Oz Explorer

Oz Explorer jest bardzo przyjemnym interaktywnym graficznym narzędziem do wizualizacji i analizy drzew przeszukiwań. W sposób bardzo przejrzysty pozwala przeglądać poszczególne przestrzenie rozwiązywanego problemu. Gałęzie powstałego drzewa można odpowiednio zwijać i rozwijać, a także ręcznie można przeszukiwać kolejne nieprzejrzane jeszcze poddrzewa. Pewne szczegóły w użytkowaniu tego narzędzia zostaną przedstawione w ostatniej części tej pracy.



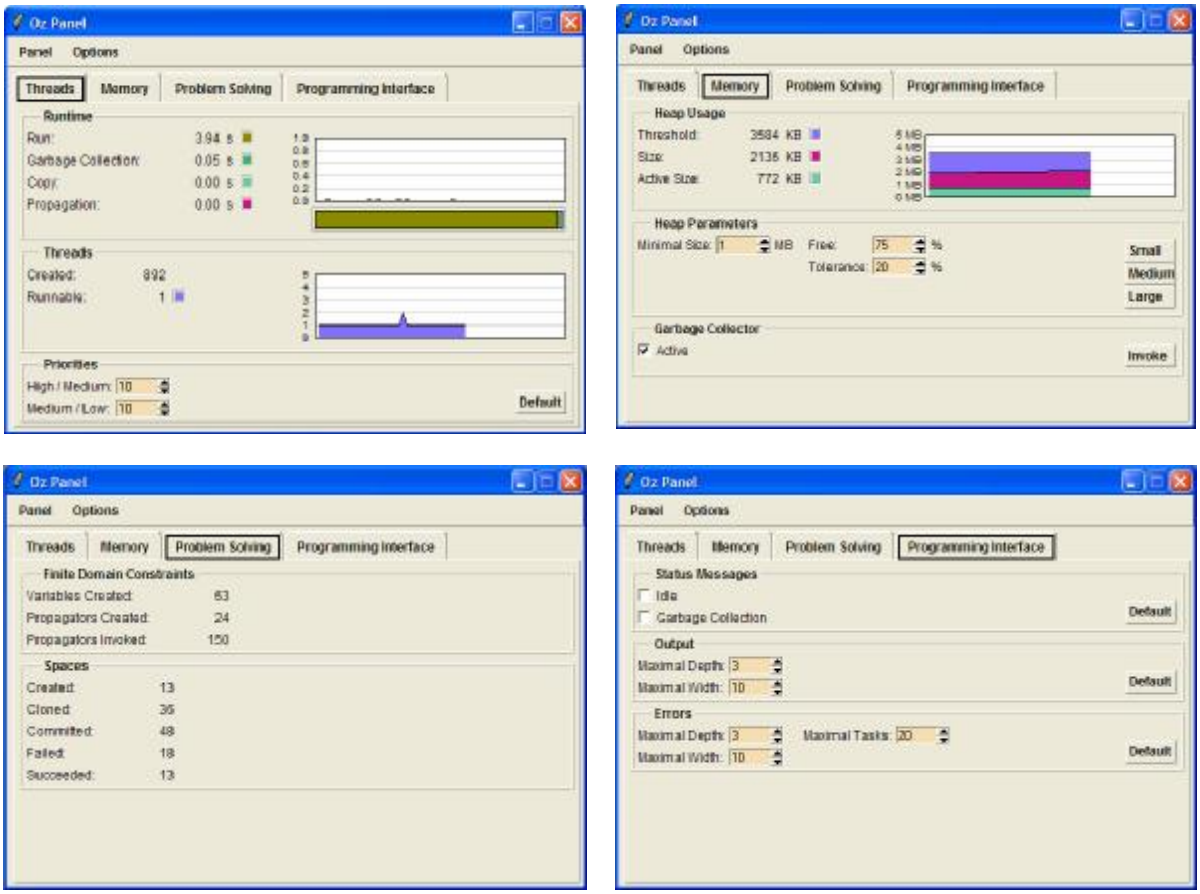

Rysunek 3. Oz Explorer.

## Oz Engine

Oz Engine jest wirtualną maszyną Oz, podobną do tej w Javie, dzięki której osiągana jest wieloplatformowość tworzonych aplikacji.

## Oz Panel

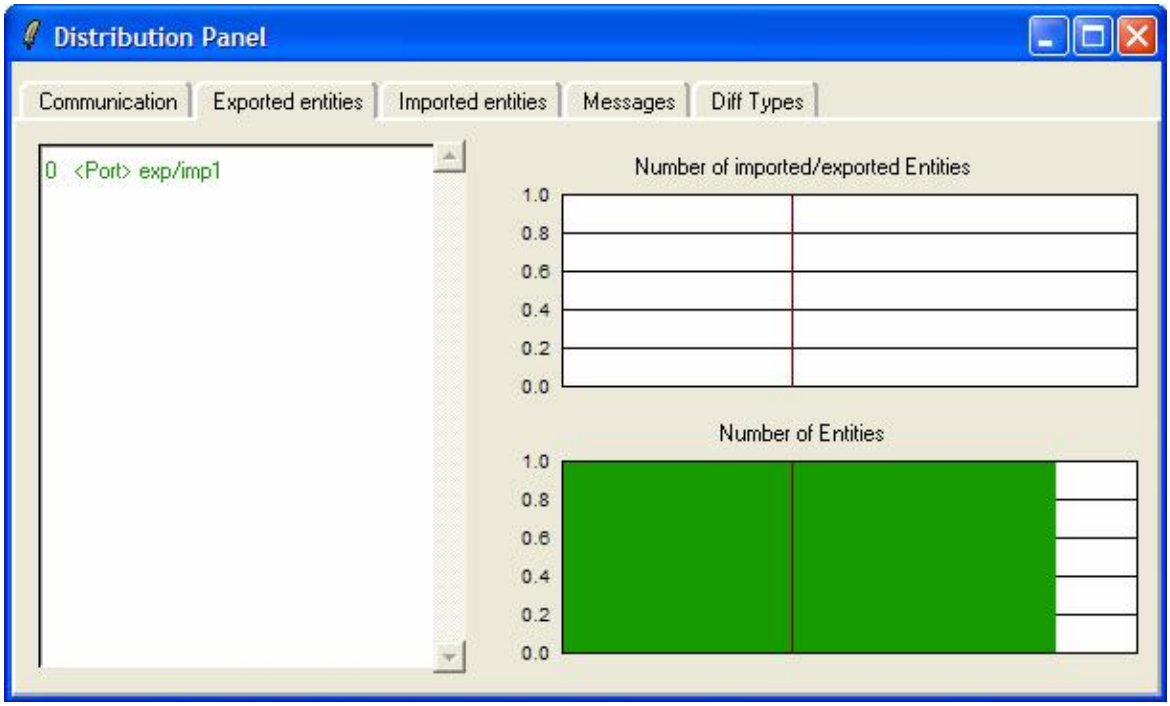
Oz Panel jest graficznym narzędziem do wizualizacji i kontroli najważniejszych parametrów działającego Mozarta, tj. wątki, pamięć, rozwiązywanie problemu, konfiguracja OPI.




Rysunek 4. Oz Panel.

## Distribution Panel

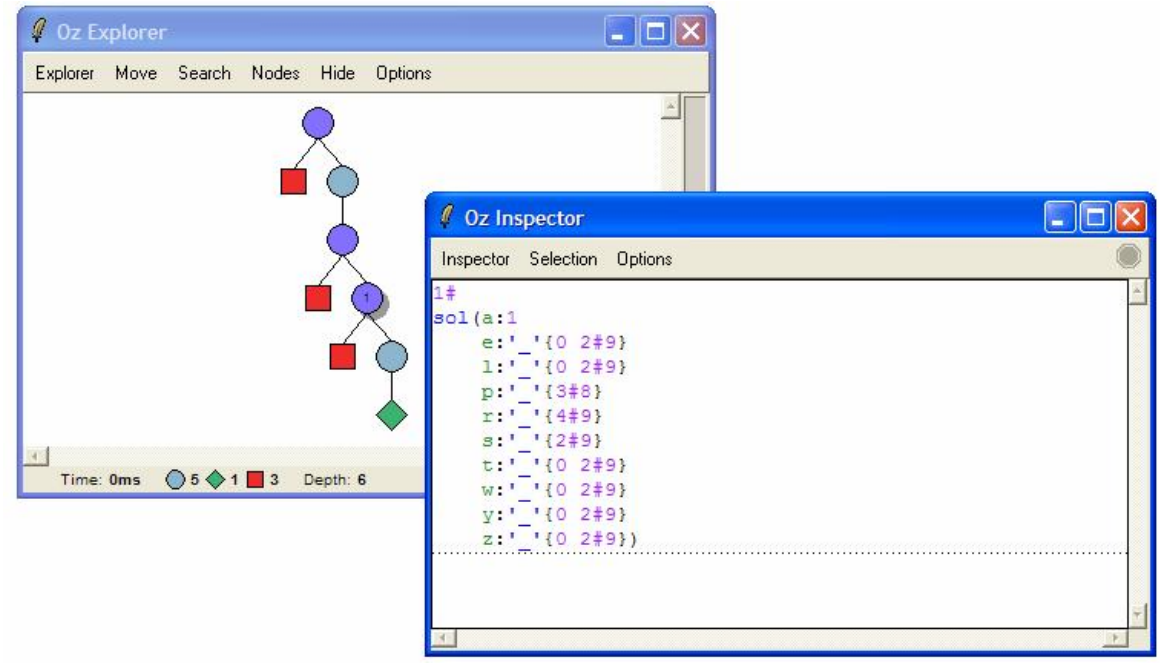
Distribution Panel jest narzędziem do monitorowania zachowania rozproszonej części Mozarta związanej z komunikacją, przesyłaniem wiadomości, eksportowaniem i importowaniem bytów. Narzędzie to jest bardzo przydatne do dostrajania aplikacji i wykrywania jej nieoczekiwanych zachowań.




Rysunek 5. Distribution Panel.

## Oz Inspector

Oz Inspector jest interaktywnym graficznym narzędziem do wyświetlania i badania wartości różnych struktur danych Oz. Wykorzystywany jest np. przez Oz Explorer do podglądania węzłów w drzewie przeszukiwań.




Rysunek 6. Oz Inspector + Oz Explorer.

### Oz Shell Utilities

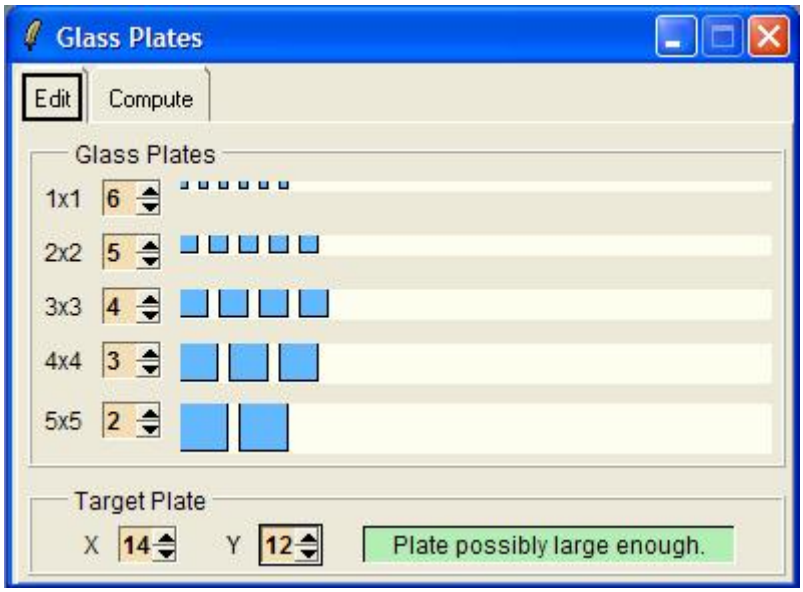
Oz Shell Utilities jest tak naprawdę zbiorem kilku standardowych narzędzi użytkowych, tj. kompilator Oz Compiler, konsolidator Oz Linker, debugger Oz Debugger, a także Oz Profiler pomagający w optymalizacji aplikacji Oz pod względem szybkości i wykorzystania pamięci (np. zlicza wywołania procedur i mierzy ich konsumpcję pamięci), czy też Oz DLL Builder wspierający tworzenie natywnych bibliotek DLL.

## Przykładowe aplikacje

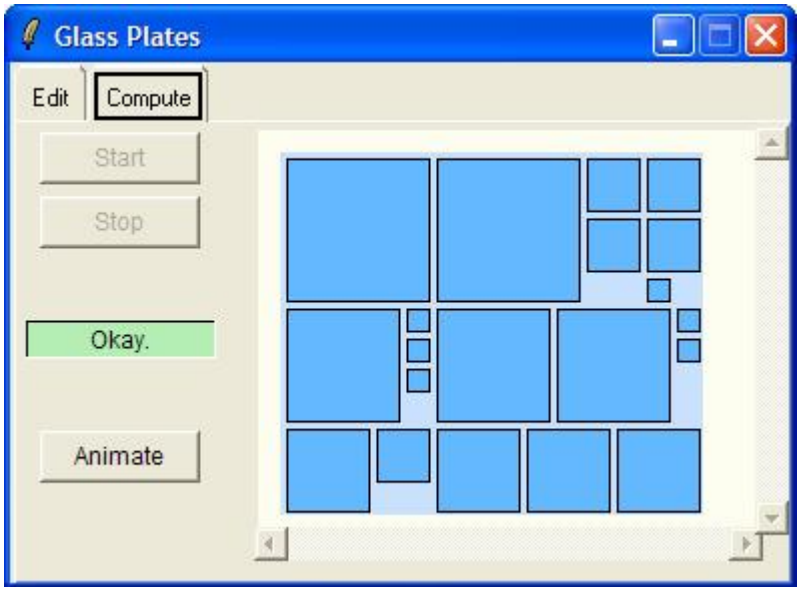
Przyjrzyjmy się teraz kilku przykładowym prostym aplikacjom, które prezentują trzy główne cechy Mozarta: rozwiązywanie problemów z ograniczeniami, współbieżność, rozproszenie. Szczegóły dotyczące omawianych niżej aplikacji znaleźć można w dokumentach [10]-[12].

### Cięcie szkła

Pierwsza aplikacja demonstruje możliwości programowania z ograniczeniami na przykładzie cięcia szkła. Zadanie polega na znalezieniu takiego rozwiązania, które dla dużego szkła o wymiarach $X$ jednostek szerokości na $Y$ jednostek wysokości potrafi wyciąć wybraną liczbę mniejszych szkiełek o wymiarach odpowiednio $1x1$, $2x2$, $3x3$, $4x4$, $5x5$. Aplikacja wyposażona jest we własne GUI, które również zostało zaimplementowane przy pomocy Mozarta.



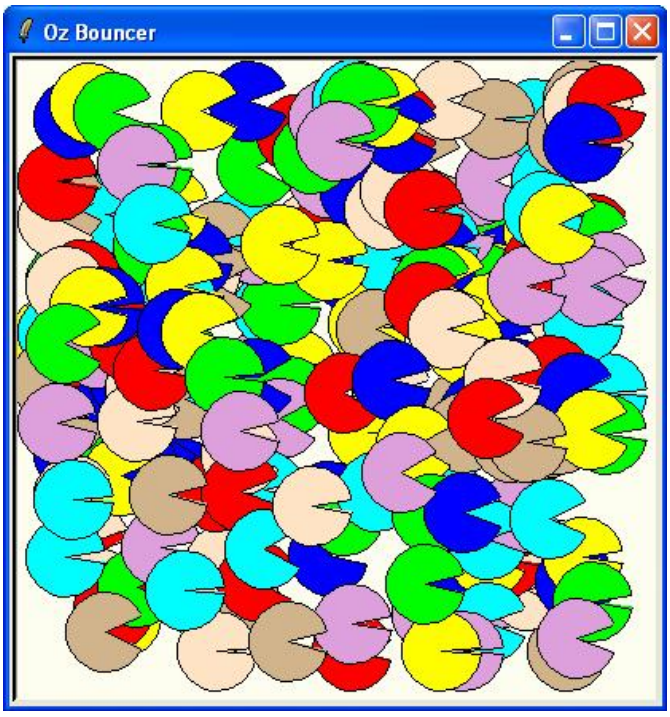

Rysunek 7. Edycja danych do problemu cięcia szkła.




Rysunek 8. Rozwiązanie problemu cięcia szkła.

## Skaczące kulki

Druga aplikacja demonstruje łatwość obsługi wielu wątków na przykładzie współbieżnie skaczących kulek. Z każdą kulką pojawiającą się w oknie aplikacji utożsamiony jest jeden wątek odpowiedzialny za jej przemieszczenie zgodnie z zaimplementowanym sposobem poruszania (każda kulka porusza się w ten sam sposób). Przy pomocy myszy możliwe jest dodawanie i odejmowanie kolejnych kulek. Okazuje się, że aplikacja bardzo dobrze sobie radzi z obsługą setek takich kulek.

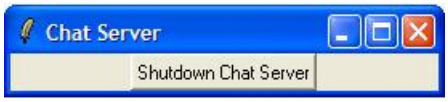



Rysunek 9. Dwieście współbieżnie skaczących kulek.

## Chat

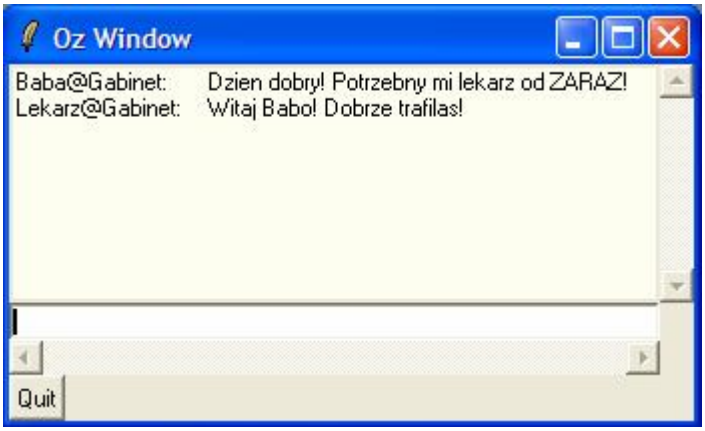
Trzecia aplikacja demonstruje możliwości rozproszenia aplikacji na przykładzie prostego chata zbudowanego na zasadzie klient-serwer. Przykładowo po uruchomieniu serwera „Przychodnia", a następnie uruchomieniu dwóch klientów „Lekarz@Gabinet" oraz „Baba@Gabinet" możemy zaaranżować sytuację, w której „przychodzi baba do lekarza" i następuje znany wszystkim dialog.


Rysunek 10. Uruchamianie serwera.




Rysunek 11. Serwer po uruchomieniu.


Rysunek 12. Uruchamianie klientów.




Rysunek 13. Klient po uruchomieniu.

# Przykładowe projekty

Na swojej oficjalnej stronie Mozart chwali się kilkoma zrealizowanymi dzięki niemu projektami. Warto wspomnieć chociaż parę słów na ten temat.

Na początku 2004 roku wykorzystano Mozarta do zaplanowania sezonu 2003/2004 kobiecej koszykówki konferencji wschodniej (*Big East Women`s Basketball*). Celem projektu było uzyskanie optymalnego rozkładu 122 meczy w ciągu 18 dni dla turnieju, w którym uczestniczy 14 drużyn uwzględniając przy tym dodatkowe ograniczenia typu równomierne rozłożenie gier „w domu" i „na wyjeździe", odległości pomiędzy „miejscem zamieszkania" drużyny a miejscem spotkania, itp.

W tym samym czasie wykorzystano również Mozarta do stworzenia systemu planowania rozkładu zajęć w szkole Nanyang Business School. System miał służyć do optymalnego lokowania zajęć w odpowiednie ramy „czaso-przestrzenne", tzn. odpowiedni czas i miejsce – unikanie kolizji, itp. uwzględniając przy tym dodatkowe ograniczenia, tj. preferencje „czaso-przestrzenne" wykładowców, itp.

W kwietniu 2005 roku Mozart został wykorzystany przez Alenia Aeronautica do stworzenia modułu automatycznie generującego FMEA (*Failure Mode Effects Analysis*) służącego identyfikacji wszelkich potencjalnych błędów jeszcze we wczesnej fazie wytwarzania podzespołów samolotowych, tj. w fazie projektu.

# Język Oz

Język Oz jest językiem wieloparadygmatowym, który unifikuje w sobie wiele różnych podejść programistycznych. Łączy programowanie funkcyjne, obiektowe, logiczne, logiczne z ograniczeniami, współbieżne i rozproszone. Wprowadza dzięki temu pewien szeroki zakres programistycznych abstrakcji, który pozwala w szybki i łatwy sposób tworzyć złożone aplikacje (współbieżne, rozproszone, inteligentne, miękkiego czasu rzeczywistego, itp.). W dalszej części skupimy się jednak na podstawowych możliwościach związanych jedynie z programowaniem z ograniczeniami.

## Programowanie z ograniczeniami

Na początku wprowadzony zostanie pewien zestaw pojęć, który jest nieodłącznie związany z programowaniem z ograniczeniami – po szczegóły warto zwrócić się do dokumentu [14]. Ograniczymy się przy tym jedynie do problemów kombinatorycznych, w których wszelkie występujące zmienne przyjmują wartości ze skończonego zbioru nieujemnych liczb całkowitych. Jak się później okaże problemy te są doskonale rozwiązywane przy pomocy dwóch współdziałających ze sobą technik wnioskowania: propagacji ogarniczeń (*constraint propagation*) oraz dystrybucji ograniczeń (*constraint distribution*). Mówiąc w tej chwili dość hasłowo – propagacja ograniczeń jest mechanizmem wnioskowania opartym na współbieżnie działających propagatorach ograniczeń (*concurrent propagators*) aktualizujących na bieżąco informację w tzw. składzie ograniczeń (*constraint store*), natomiast dystrybucja ograniczeń mechanizmem dzielącym dany problem na dwa uzupełniające się pod-problemy w momencie gdy propagacja ograniczeń staje się niemożliwa.

### Pojęcia

**Skończona dziedzina (*finite domain*)**

Skończona dziedzina jest skończonym zbiorem nieujemnych liczb całkowitych z zakresu od $n$ do $m$ – zapisywanym w postaci $n\#m$.

**Ograniczenie (*constraint*)**

Ograniczenie jest formułą logiki predykatów. Wyróżnia się ograniczenia podstawowe (*basic constraints*) w skład których wchodzą wszelkie równości – np. $X = 3$, $Y = Z$ – jak również, tzw. ograniczenia

dziedzinowe (*domain constraints*) – np. $X \in 11\#30$. Wszystkie inne typy ograniczeń rozumiane są jako nie-podstawowe – np. $(X+Y)^2 > 4 \cdot X \cdot Y$, $X \neq Y$, $X^2 + Y^2 = Z^2$, $X_1, X_2, \mathbf{K}, X_n$ są parami różne.

**Problem skończonej dziedziny (*finite domain problem*)**

Problem skończonej dziedziny rozpatrywany jest jako skończony zbiór $P$ ograniczeń, w którym zawarte są ograniczenia dziedzinowe dla wszystkich zmiennych występujących w dowolnych ograniczeniach zbioru $P$.

**Rozwiązanie (*solution*)**

Rozwiązaniem problemu skończonej dziedziny jest przypisanie zmiennym wartości spełniających wszystkie ograniczenia ze zbioru $P$. Warto przy tym zauważyć, iż każdy problem skończonej dziedziny ma co najwyżej skończenie wiele rozwiązań, ponieważ rozważamy skończoną liczbę zmiennych przyjmujących wartości ze skończonych dziedzin.

**Propagacja ograniczeń (*constraint propagation*)**

Propagacja ograniczeń to reguła wnioskowania dla problemów skończonej dziedziny, która zawęża dziedziny zmiennych. Odbywa się to na dwa sposoby: poprzez propagację dziedzinową (*domain propagation*), która zawęża dziedziny najbardziej jak to tylko możliwe tworząc „dziury" pomiędzy najmniejszą i największą możliwą wartością zmiennej – np. $X \in 1\#10 \rightarrow X \in \{2,4,6,8\}$ – oraz poprzez propagację przedziałową (*interval propagation*), która zawęża dziedziny dzięki zwiększaniu i zmniejszaniu odpowiednio ograniczenia dolnego oraz górnego – np. $X \in 1\#10 \rightarrow X \in 2\#8$. W praktyce propagacja przedziałowa jest o wiele częściej stosowana ze względu na jej szybkość działania i mniejszy koszt obliczeniowy. Język Oz definiuje jednak sytuacje, w których występuje określona propagacja ograniczeń – po szczegóły warto udać się do dokumentacji Mozarta dostępnej na stronie http://www.mozart-oz.org/documentation.

Rozpatrzmy jeszcze tylko prosty przykład propagacji ograniczeń. Załóżmy, że mamy dwie zmienne $Renia$ oraz $Rysiu$ reprezentujące wzrost odpowiednio Reni oraz Rysia objęte ograniczeniami podstawowymi $Renia \in 160\#190$ i $Rysiu \in 150\#180$ oraz ograniczeniem nie-podstawowym $Renia < Rysiu$. W wyniku propagacji ograniczenia nie-podstawowego łatwo zauważyć, iż ograniczenia dziedzinowe wprowadzonych zmiennych zostaną zawężone odpowiednio do $Renia \in 160\#179$ oraz $Rysiu \in 161\#180$ – mówiąc w skrócie propagacja wyeliminowała wartości zmiennych, które na pewno nie spełniałyby rozważanego ograniczenia.

**Skład ograniczeń (*constraint store*)**

Skład ograniczeń przechowuje koniunkcję wszystkich ograniczeń podstawowych.

**Propagator (*propagator*)**

Propagator jest współbieżnie działającym agentem obliczeniowym nałożonym na ograniczenie nie-podstawowe, który stara się zawężać dziedziny wszystkich zmiennych znajdujące się w tym ograniczeniu. Na różne ograniczenia nie-podstawowe nałożone są różne propagatory, które poprzez współdziałanie ze składem ograniczeń realizują omówioną wcześniej propagację ograniczeń. Często zdarza się, że wiele

propagatorów wpływa na zawężanie dziedzin tych samych zmiennych – wówczas dochodzi do przezroczystej komunikacji pomiędzy tymi propagatorami poprzez skład ograniczeń, który naprzemiennie uaktualniają.

Rozpatrzmy prosty przykład komunikacji dwóch propagatorów. Zdefiniujmy zadanie do rozwiązania: „Maurycy i Paweł mają w sumie 46 lat, a za 23 lata Maurycy będzie dwa razy starszy niż Paweł teraz. W jakim wieku jest każdy z chłopców, jeśli Maurycy wygląda na chłopca w wieku od 10 do 30 lat, a Paweł na chłopca – od 20 do 40 lat?". Załóżmy, że mamy dwie zmienne $Maurycy$ oraz $Pawel$ reprezentujące wiek odpowiednio Maurycego oraz Pawła objęte ograniczeniami podstawowymi $Maurycy \in 10\#30$ i $Pawel \in 20\#40$ oraz dwoma ograniczeniami nie-podstawowymi $1: Maurycy + Pawel = 46$ i $2: Maurycy + 23 = 2 \cdot Pawel$. Niezależnie od kolejności propagowanych ograniczeń powinniśmy uzyskać ten sam stan końcowy będący w tym przypadku rozwiązaniem zadania. Załóżmy, że najpierw do akcji przechodzi propagator nr 1. Łatwo zauważyć, że skoro Maurycy ma co najmniej 10 lat, to Paweł nie może mieć więcej niż 36. Natomiast z drugiej strony, skoro Paweł ma co najmniej 20 lat, to Maurycy nie może mieć więcej niż 26. W ten sposób uzyskujemy uaktualnienie składu ograniczeń o nowe spostrzeżenia i w konsekwencji mamy $Maurycy \in 10\#26 \wedge Pawel \in 20\#36$. Ponieważ propagator nr 1 wykonał już swoje zadanie, do akcji przechodzi propagator nr 2. Zauważmy więc, że za 23 lata Maurycy będzie miał parzystą liczbę lat ($2 \cdot Pawel$), w związku z tym teraz Maurycy ma nieparzystą liczbę lat i w zależności od tego jaką technikę propagacji ograniczeń byśmy zastosowali, otrzymalibyśmy $Maurycy \in \{11,13,15,17,19,21,23,25\}$ przy propagacji dziedzinowej lub $Maurycy \in 11\#25$ przy propagacji przedziałowej. Załóżmy, że mamy do czynienia z tym drugim przypadkiem. W takim razie, skoro Maurycy ma co najwyżej 25 lat, to za 23 lata będzie miał 48, a w związku z tym Paweł nie może mieć teraz więcej niż 24 lata. Idąc z drugiej strony możemy zauważyć, że skoro Paweł ma co najmniej 20 lat, to za 23 lata Maurycy będzie miał co najmniej 40 lat, a to oznacza, że w tej chwili ma co najmniej 17. Ostatecznie otrzymujemy nowy stan składu ograniczeń: $Maurycy \in 17\#25 \wedge Pawel \in 20\#24$, a do akcji przechodzi ponownie propagator nr 1. Analizując w ten sposób działanie propagatorów moglibyśmy zaobserwować następującą sekwencję uaktualnień składu ograniczeń:

$$\begin{cases} M \in 10\#30 \\ P \in 20\#40 \end{cases} \overset{1}{\Rightarrow} \begin{cases} M \in 10\#26 \\ P \in 20\#36 \end{cases} \overset{2}{\Rightarrow} \begin{cases} M \in 17\#25 \\ P \in 20\#24 \end{cases} \overset{1}{\Rightarrow} \begin{cases} M \in 22\#25 \\ P \in 21\#24 \end{cases} \overset{2}{\Rightarrow} \begin{cases} M \in 23\#25 \\ P \in 23\#24 \end{cases} \overset{1}{\Rightarrow} \begin{cases} M = 23 \\ P = 23 \end{cases}$$

**Przestrzeń (*space*)**

Przestrzeń utożsamiana jest z architekturą obliczeniową do propagacji ograniczeń, która składa się ze wszystkich propagatorów połączonych ze składem ograniczeń.

**Uaktualnianie składu ograniczeń (*telling a constraint*)**

Dla danego ograniczenia $S$ (ze składu ograniczeń) oraz propagatora $P$ (nałożonego na ograniczenie nie-podstawowe) następuje aktualizacja składu o ograniczenie podstawowe $B$ (równoważne $S \wedge B$), jeśli $S \wedge P \Rightarrow B$ i $B$ dodaje nową spójną informację do składu ograniczeń.

**Spójność składu ograniczeń (*consistent constraint store*)**

Spójność składu ograniczeń jest wymagana w każdej chwili i oznacza, że musi istnieć co najmniej jedno rozwiązanie spełniające wszystkie ograniczenia ze składu ograniczeń.

**Stan propagatora**

Propagator może znajdować się w jednym z trzech stanów: zawiedzenia, zużycia i stabilności.

Propagator zawiedziony (*failed propagator*), to taki propagator, który zdaje sobie sprawę z tego, iż wykonanie pewnego możliwego działania doprowadza do niespójności, tzn. do sytuacji, w której nie istnieje żadne rozwiązanie satysfakcjonujące zarówno skład ograniczeń jak i ograniczenie nałożone przez ten propagator.

Propagator zużyty (*entailed propagator*), to taki propagator, który nie wnosi już i na pewno nie wniesie żadnej dodatkowej informacji do składu ograniczeń, tzn. każde rozwiązanie satysfakcjonujące ograniczenia ze składu ograniczeń satysfakcjonuje również ograniczenie nałożone przez ten propagator – np. wracając do przykładu o Reni i Rysiu, propagator $Renia < Rysiu$ stałby się zużyty, gdybyśmy uzyskali następujący stan składu ograniczeń: $Renia \in 161\#170 \wedge Rysiu \in 171\#180$. Z chwilą gdy propagator staje się zużyty jest on automatycznie usuwany z przestrzeni.

Propagator stabilny (*stable propagator*), to taki propagator, który jest zawiedziony, lub nie może w żaden sposób uaktualnić składu ograniczeń.

**Stan przestrzeni**

Przestrzeń może znajdować się również w jednym z trzech stanów: zawiedzenia, stabilności i rozwiązania.

Przestrzeń zawiedziona (*failed space*), to przestrzeń, w której istnieje co najmniej jeden zawiedziony propagator.

Przestrzeń stabilna (*stable space*), to przestrzeń, w której wszystkie propagatory są stabilne.

Przestrzeń rozwiązana (*solved space*), to przestrzeń, która nie jest zawiedziona i nie istnieje już żaden propagator (wszystkie zostały zużyte).

**Rozwiązanie przestrzeni**

Rozwiązaniem przestrzeni jest dowolne przypisanie wartości zmiennych, które satysfakcjonuje wszystkie ograniczenia ze składu ograniczeń oraz ograniczenia nałożone przez propagatory.

**Dystrybucja ograniczeń**

Przy okazji omawiania propagatora przeanalizowaliśmy przykład o Pawle i Maurycym, w którym pokazaliśmy, iż propagacja ograniczeń wystarczyła do uzyskania jedynego rozwiązania. Niestety w większości przypadków propagacja ograniczeń nie wystarcza do rozwiązania problemu – mówi się o tzw. niekompletności propagacji ograniczeń. Chodzi tutaj o sytuacje, w której uzyskiwana jest przestrzeń stabilna,

ale nadal niewiadomo czy w ogóle istnieje jakiekolwiek jej rozwiązanie. Na przykład problem może być nierozwiązywalny, ale uzyskiwana jest przestrzeń stabilna

$$\begin{array}{lll} X \neq Y & X \neq Z & Y \neq Z \\ X \in 0\#1 & Y \in 0\#1 & Z \in 0\#1 \end{array},$$

albo istnieje dokładnie jedno rozwiązanie, ale uzyskiwana jest również przestrzeń stabilna:

$$\begin{array}{lll} 3 \cdot X + 3 \cdot Y = 5 \cdot Z & X - Y = Z & X + Y = Z + 2 \\ X \in 4\#10 & Y \in 1\#7 & Z \in 3\#9 \end{array}.$$

Dlatego też z pomocą przychodzi tzw. dystrybucja ograniczeń (*constraints distribution*), która uzupełnia powyższe braki. Zestawienie dystrybucji ograniczeń z propagacją ograniczeń pozwala rozwiązać dowolny problem skończonej dziedziny.

Dystrybucja ograniczeń polega na rozdzieleniu problemu $P$ na dwa uzupełniające się pod-problemy względem wybranego ograniczenia $C$, w taki sposób, że rozwiązywany jest dalej problem $P \cup \{C\}$ oraz $P \cup \{\neg C\}$ – jak widać zaczyna się tworzyć drzewo przeszukiwań. Z wyborem odpowiedniej zmiennej i odpowiedniego dla niej ograniczenia związane są różne strategie dystrybucji ograniczeń. W języku Oz mamy do czynienia z dwoma głównymi strategiami wyboru zmiennej: naiwnej oraz najmniejszej liczby możliwych wartości.

Naiwna strategia dystrybucji ograniczeń (*naive distribution strategy*) wybiera nie wyznaczoną jeszcze zmienną leżącą najbardziej na lewo w sekwencji wszystkich zmiennych (sekwencję tę należy utożsamić z porządkiem alfabetycznym nazw zmiennych).

Strategia dystrybucji najmniejszej liczby możliwych wartości (*first-fail distribution strategy*) wybiera nie wyznaczoną jeszcze zmienną leżącą najbardziej na lewo w sekwencji, której ograniczenie dziedzinowe jest najbardziej zawężone. Należy przy tym pamiętać, że strategia *first-fail* tworzy zazwyczaj dużo mniejsze drzewa przeszukiwań niż strategia *naive*.

Mozart pozwala również definiować własne strategie dystrybucji ograniczeń poprzez specjalny rekord *generic*, który w skrócie zostanie omówiony w ostatniej części pracy.

Po odpowiednim wybraniu zmiennej pojawia się jeszcze problem wybrania dla niej ograniczenia $C$, które zostanie odpowiednio dystrybuowane do dwóch kolejnych podprzestrzeni. Jeżeli np. zmienna $X \in n\#m$, to możemy rozdzielić ją na $C \equiv X = n$ (wtedy $\neg C \equiv X \in (n+1)\#m$) lub $C \equiv X = m$ lub $C \equiv X = s$ lub $C \equiv X \leq s$, gdzie $s$ - środkowa wartość przedziału $n\#m$.

# Prezentacja Mozarta

W ostatniej części pracy spróbujemy przeanalizować rozwiązanie prostego problemu skończonej dziedziny bawiąc się jednocześnie narzędziem Oz Explorer do przeglądania drzewa przeszukiwań.

# Problem

W poniższym równaniu jedna litera zastępuje jedną cyfrę, takim samym literom odpowiadają identyczne cyfry, różnym literom odpowiadają różne cyfry, a pierwsza litera z lewej strony każdej z liczb nie jest zerem. Należy podać jakie cyfry zastępowane są przez każdą z liter.

$STAWARZ + PAWEL = STRASZY$ .

# Rozwiązanie

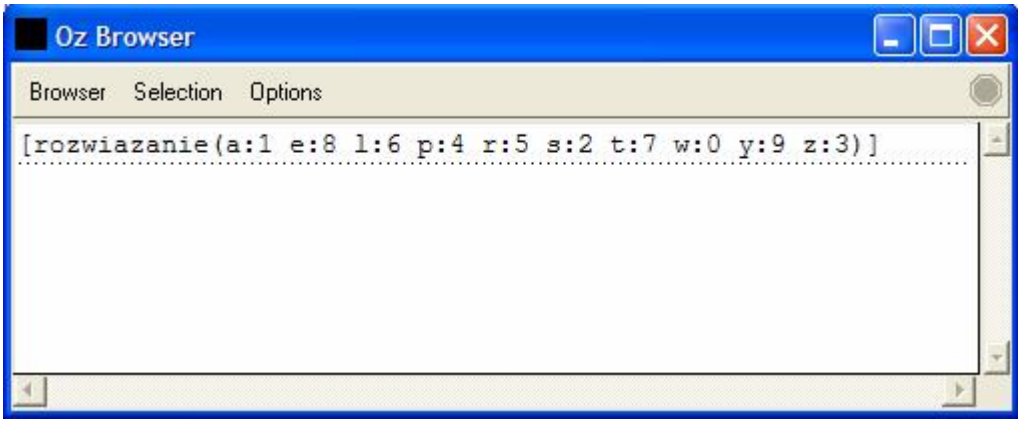
Poszukiwanie rozwiązania powyższego problemu odbędzie się przy pomocy poniższego kodu w języku Oz:

```
declare
proc {Stawarz Root}
   S T A W R Z P E L Y
in
   Root = rozwiazanie(s:S t:T a:A w:W r:R z:Z p:P e:E l:L y:Y)
   Root ::: 0#9
   {FD.distinct Root}
   S \=: 0
   P \=: 0
                1000000*S + 100000*T + 10000*A + 1000*W + 100*A + 10*R + Z
   +                                    10000*P + 1000*A + 100*W + 10*E + L
   =:           1000000*S + 100000*T + 10000*R + 1000*A + 100*S + 10*Z + Y
   {FD.distribute ff Root}
end

{Browse {SearchOne Stawarz}}

{ExploreOne Stawarz}
```

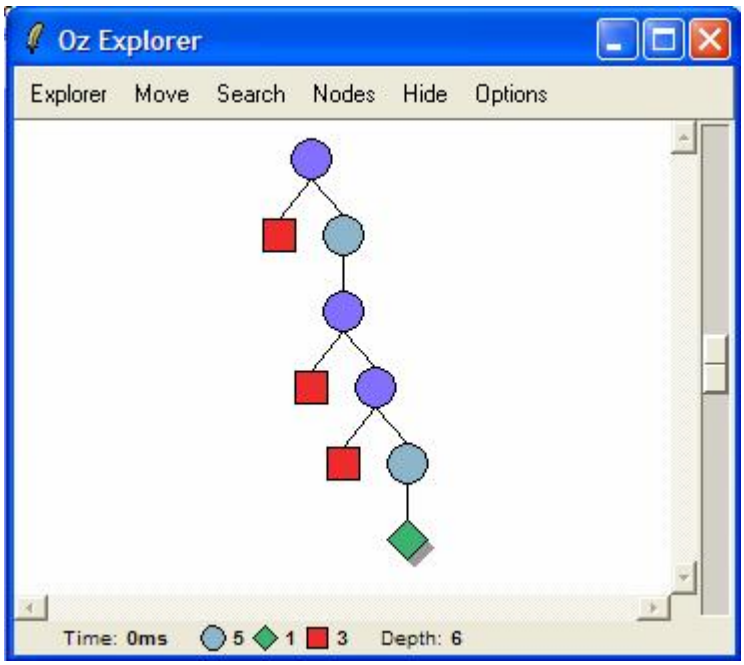
Powyższy kod przedstawia deklarację procedury o nazwie *Stawarz* rozwiązującej zadany problem. Parametr *Root* tej procedury zunifikowany zostaje na samym wstępie z rekordem o nazwie *rozwiazanie*, którego pola o nazwach pisanych małymi literami będą przyjmować odpowiednie wartości zmiennych unifikowanych w trakcie rozwiązywania zadania. Wszystkie zmienne (S, T, A, W, R, Z, P, E, L, Y) zostały również we wstępie zadeklarowane. Instrukcja *Root:::0#9* nakłada ograniczenia dziedzinowe $0\#9$ na każdą zmienną związaną z polami rekordu *rozwiazanie*. Następnie nakładane jest ograniczenie, że wszystkie zmienne są parami różne oraz $S \neq 0$ i $P \neq 0$, po czym następuje definicja arytmetycznego odpowiednika równania $STAWARZ + PAWEL = STRASZY$ . Ostatecznie definiowana jest jeszcze strategia dystrybucji – w tym przypadku *ff* oznacza strategię *first fail*, która po wybraniu odpowiedniej zmiennej dystrybuuje ją dalej z najmniejszą wartością z dziedziny do jednej podprzestrzeni oraz z resztą wartości do drugiej podprzestrzeni.

Wywołanie procedury *Browse* wraz z wywołaniem procedury *SearchOne* dla procedury *Stawarz* powoduje wyświetlenie w narzędziu Oz Browser pierwszego znalezionego rozwiązania dla zadanego w procedurze *Stawarz* problemu (Rysunek 14.).

Wywołanie procedury *ExploreOne* dla procedury *Stawarz* powoduje wyświetlenie w narzędziu Oz Explorer drzewa rozwiniętego do pierwszej rozwiązanej przestrzeni (każdy węzeł drzewa odpowiada pewnej przestrzeni – stabilnej, zawiedzionej, rozwiązanej) (Rysunek 15.).




Rysunek 14. Wywołanie narzędzia Oz Browser.



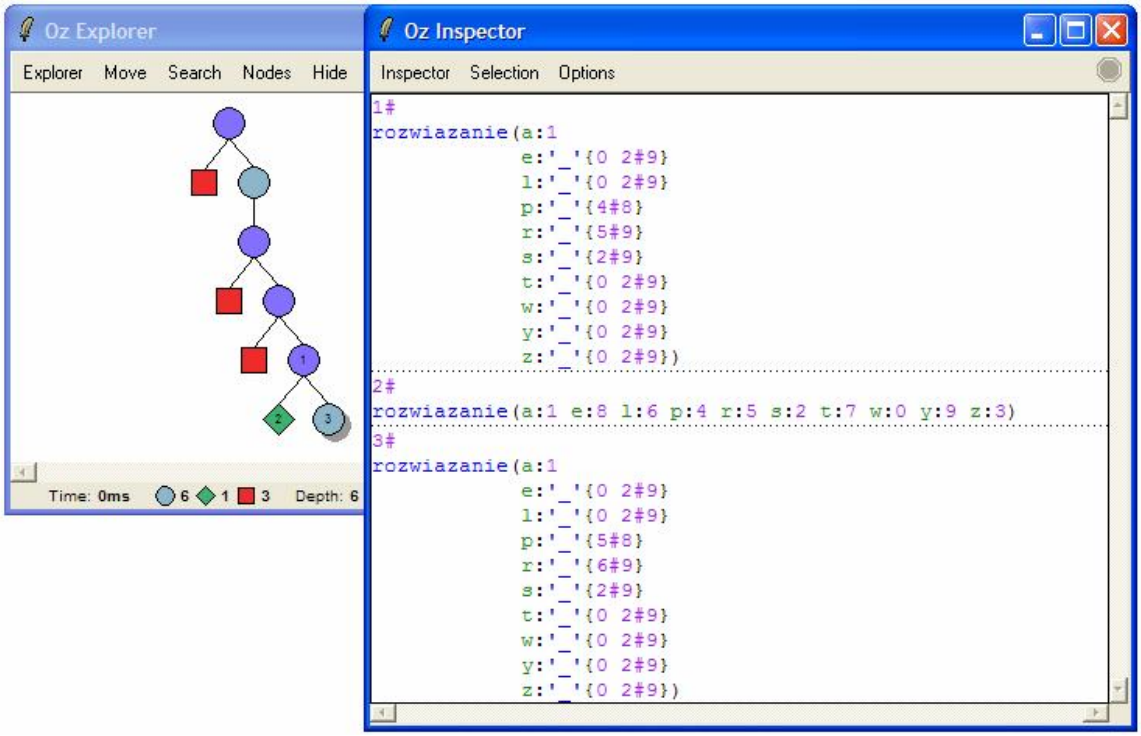

Rysunek 15. Wywołanie narzędzia Oz Explorer.

# Oz Explorer

Przyjrzyjmy się przez chwilę nieco dokładniej bardzo przyjemnemu narzędziu Oz Explorer. Na Rysunku 15. widzimy przykładową prezentację drzewa przeszukiwań do pierwszego znalezionego rozwiązania – jak widać drzewo rozwijane jest w głąb. Każdy węzeł tego drzewa poprzez magiczny symbol kółeczka, kwadracika bądź rombu niesie ze sobą pewną dodatkową informację. Patrząc od góry widzimy ciemnoniebieskie kółeczko będące korzeniem drzewa przeszukiwań. Każdy węzeł będący kółeczkiem reprezentuje pewną stabilną przestrzeń do której doszło po wykonaniu propagacji ograniczeń. Ciemny kolor kółeczka mówi nam dodatkowo, że z danej przestrzeni dystrybuowano dalej już oba możliwe ograniczenia, natomiast jasny – że z danej przestrzeni dystrybuowano tylko jedno ograniczenie. Każdy czerwony kwadracik reprezentuje przestrzenie zawiedzione, w których niemożliwe było znalezienie jakiegokolwiek rozwiązania, a zielone romby wyznaczają przestrzenie rozwiązane. W każdej chwili w pasku statusowym uzyskujemy

dodatkowe informacje o liczbie rozwiniętych przestrzeni stabilnych, zawiedzionych i rozwiązanych wraz z największą głębokością drzewa. Jak widać znalezienie pierwszego rozwiązania problemu „straszącego Stawarza" wymagało rozwinięcia 5 przestrzeni stabilnych, 3 przestrzeni zawiedzionych przy jednoczesnym wejściu na szósty poziom drzewa.

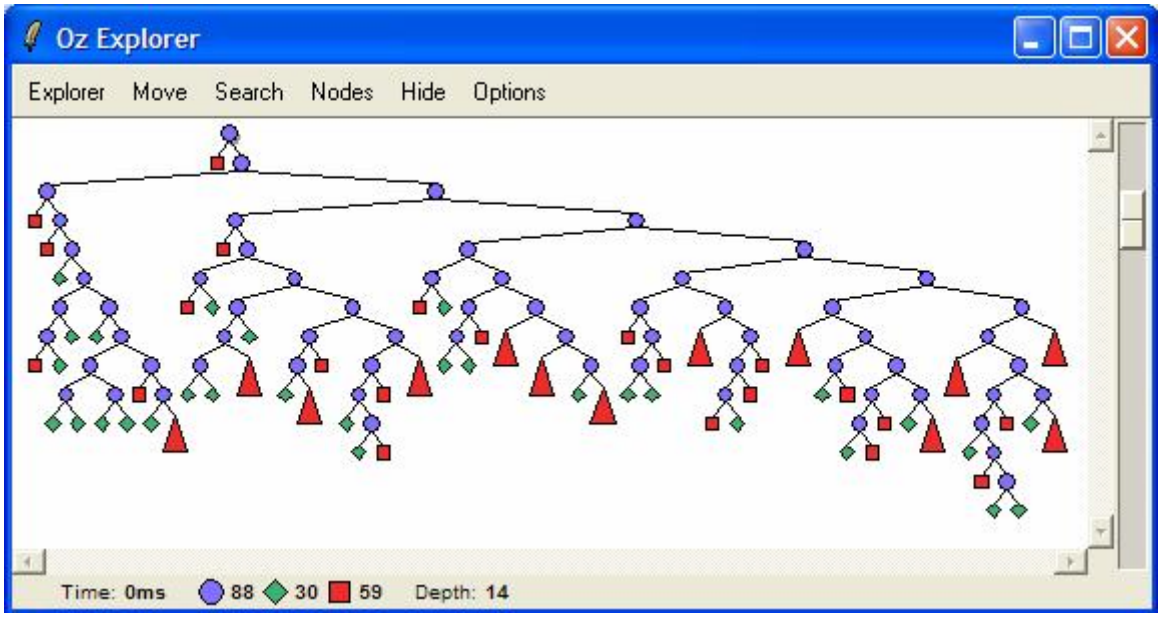
Stan dowolnej przestrzeni stabilnej bądź rozwiązanej można podejrzeć poprzez dwukrotne kliknięcie lewym przyciskiem myszy. Wybierzmy na przykład przestrzeń stabilną będącą poprzednikiem przestrzeni rozwiązanej i rozwińmy ją o jeden krok dalej (w dystrybucji) wciskając klawisz 'o', a następnie podejrzyjmy jej stan oraz jej następników.




Rysunek 16. Podejrzenie stanu trzech przestrzeni.

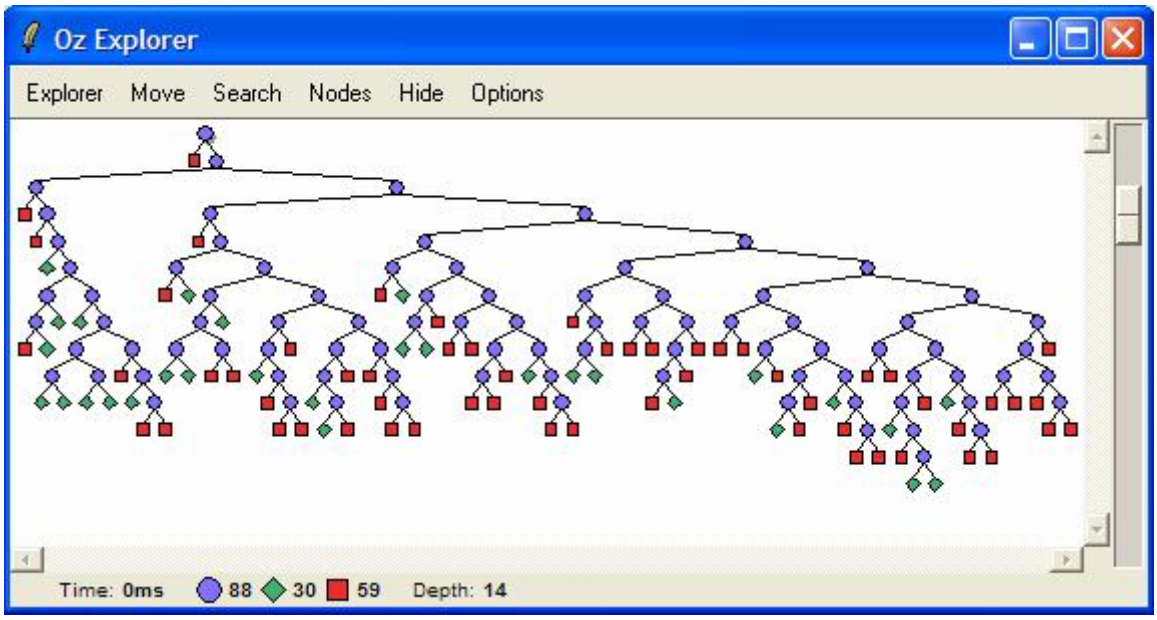
Jak widać z przestrzeni nr 1 nastąpiła dystrybucja względem zmiennej $P$ wybierając najmniejszą jej wartość do lewego poddrzewa i pozostałe jej wartości do prawego poddrzewa. Działanie to nie powinno nas dziwić, ponieważ w istocie jest to najbardziej ograniczona zmienna leżąca najbardziej na lewo w sekwencji (drugą taką zmienną jest $R$, które leży już dalej na prawo).

Wciskając klawisz 'a' dla dowolnego wybranego węzła uzyskujemy całkowite przeszukanie wszystkich poddrzew począwszy od tego węzła. Wciskając zatem klawisz 'a' dla korzenia otrzymamy całe drzewo przeszukiwań (Rysunek 17.). Po wykonaniu tej operacji naszym oczom ukazały się dodatkowe magiczne symbole – duże czerwone trójkąty, w których chowają się poddrzewa, których liśćmi są już same przestrzenie zawiedzione.




Rysunek 17. Widok na całkowicie rozwinięte drzewo przeszukiwań ze schowanymi poddrzewami zawierającymi liście z zawiedzionymi przestrzeniami.

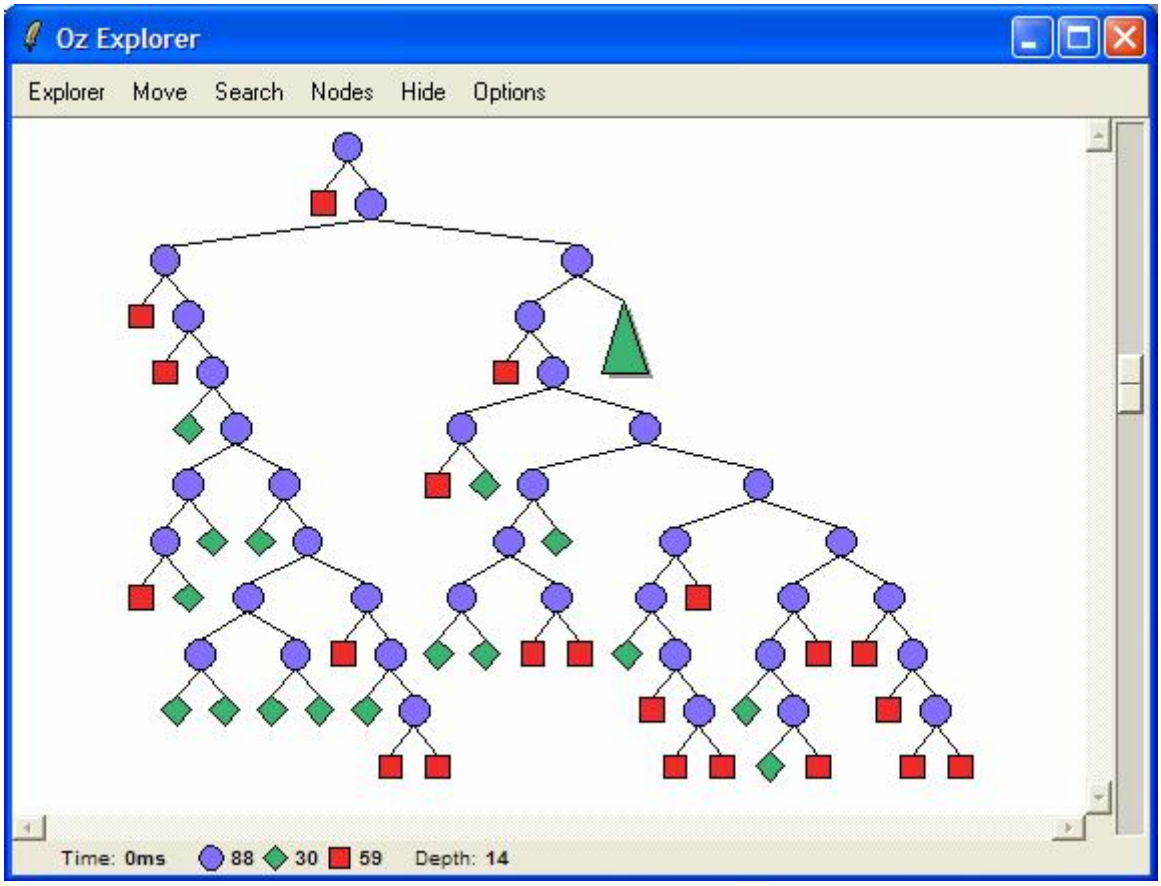
Wybierając w tej chwili dowolny węzeł możemy odkryć wszystkie schowane poddrzewa (począwszy od tego węzła) wciskając kombinację klawiszy ‘Ctrl+u’. Wybierzmy zatem korzeń drzewa i wciśnijmy ‘Ctrl+u’.




Rysunek 18. Widok na całkowicie rozwinięte drzewo przeszukiwań z odkrytymi wszystkimi poddrzewami.

Chowaniu mogą podlegać wszystkie poddrzewa, a nie tylko te ze „zwiędniętymi liśćmi”. Wybierzmy dowolny węzeł naszego drzewa i wciśnijmy klawisz ‘h’ (Rysunek 19.). Naszym oczom ukazuje się kolejny magiczny symbol – tym razem duży ciemnozielony trójkąt, w którym schowane jest poddrzewo, którego co najmniej jeden liść reprezentuje przestrzeń rozwiązaną. W przypadku niecałkowicie rozwiniętego drzewa przeszukiwań możliwe jest pojawienie się jeszcze dużego jasnozielonego trójkąta, w którym schowane jest

niecałkowicie rozwinięte poddrzewo posiadające co najmniej jeden liść reprezentujący przestrzeń rozwiązaną.

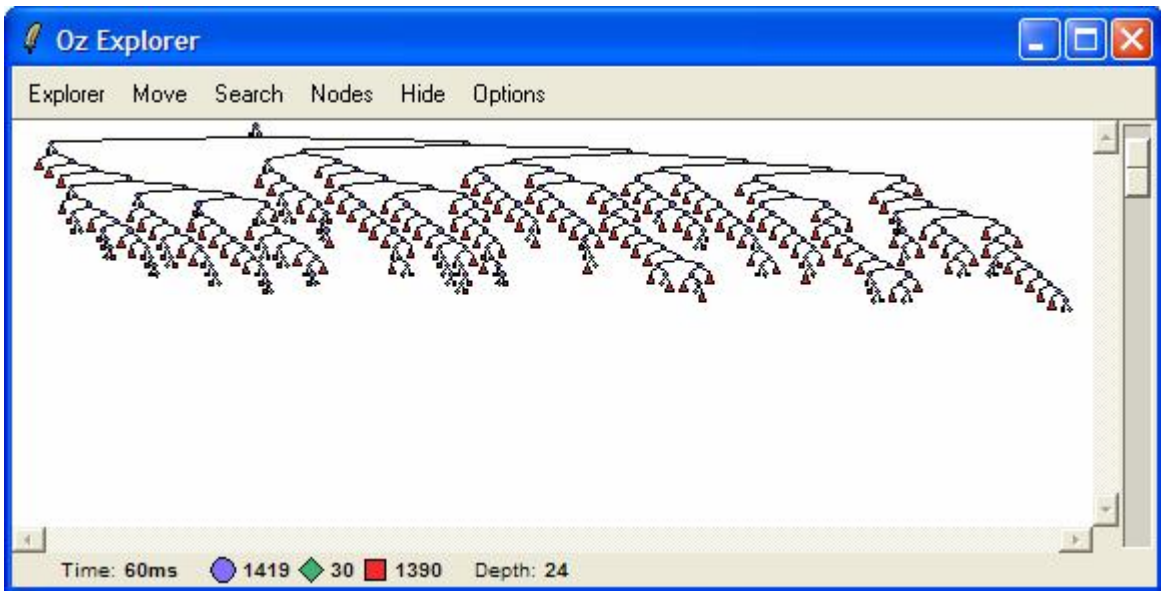



Rysunek 19. Widok na całkowicie rozwinięte drzewo przeszukiwań ze schowanym jednym poddrzewem.

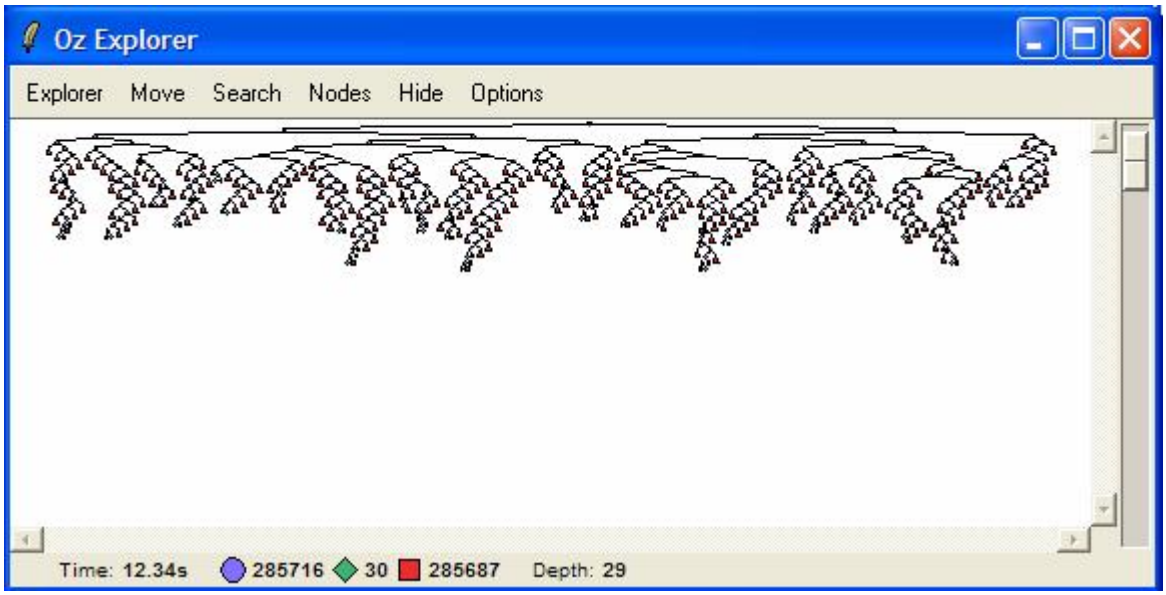
Na koniec zobaczmy jeszcze jak bardzo różni się drzewo przeszukiwań rozwinięte w przypadku korzystania z naiwnej strategii dystrybucji od tego, które analizowaliśmy do tej pory. W tym celu instrukcję *{FD.distribute ff Root}* zamieniamy na *{FD.distribute naive Root}* oraz instrukcję *{ExploreOne Stawarz}* na *{ExploreAll Stawarz}*, aby od razu zobaczyć całkowicie rozwinięte drzewo przeszukiwań (Rysunek 20.). Jak się okazuje w celu znalezienia tych samych 30 rozwiązań potrzebowaliśmy zejść o 10 poziomów głębiej rozwijając przy tym ponad 16 razy więcej przestrzeni stabilnych i dochodząc ponad 23 razy częściej do przestrzeni zawiedzionych. Nie jest to jednak szczyt „zmęczenia problemu". Okazuje się, że korzystając z ręcznej definicji strategii dystrybucji ograniczeń możemy uzyskać jeszcze bardziej okazałe drzewo!

Ręczna definicja strategii dystrybucji ograniczeń odbywa się poprzez rekord *generic*, którego poszczególne pola definiują jej własności – szczegóły można znaleźć w dokumencie [15]. Zdefiniujmy strategię, w której wybierana jest zmienna leżącą najbardziej na lewo w sekwencji, której ograniczenie górne jest największe, a następnie dzielimy dziedzinę na pół - z większymi wartościami na lewo i mniejszymi na prawo. W tym celu instrukcję *{FD.distribute naive Root}* zamieniamy teraz na *{FD.distribute generic(order:max value:splitMax)*

*Root}*. Spójrzmy na powstałe drzewo przeszukiwań (Rysunek 21.), zerknijmy na pasek statusowy i od tej pory pamiętajmy, aby „ostrożnie bawić się zapałkami”.




Rysunek 20. Widok na całkowicie rozwinięte drzewo przeszukiwań ze schowanym „zwiędniętymi” poddrzewami dla naiwnej strategii dystrybucji ograniczeń.




Rysunek 21. Widok na całkowicie rozwinięte drzewo przeszukiwań ze schowanym „zwiędniętymi” poddrzewami dla ręcznie zdefiniowanej strategii dystrybucji ograniczeń.

# Literatura

[1] Leif Kornstaedt, Denys Duchier. *The Oz Programming Interface*. 2006.

[2] Konstantin Popov. *The Oz Browser*. 2006.

[3] Christian Schulte. *Oz Explorer – Visual Constraint Programming Support*. 2006

[4] Christian Schulte. *Oz Panel*. 2006.

[5] Erik Klintskog, Anna Neiderud. *Distribution Panel*. 2006.

[6] Denys Duchier, Leif Kornstaedt, Christian Schulte. *Oz Shell Utilities*. 2006.

[7] Benjamin Lorenz, Leif Kornstaedt. *The Mozart Debugger*. 2006.

[8] Denys Duchier, Benjamin Lorenz, Ralf Scheidhauer. *The Mozart Profiler*. 2006.

[9] Thorsten Brunklaus. *The Oz Inspector*. 2006.

[10] Christian Schulte. *Mozart Demo Applications. Part I: Constraint Applications. 4 Cutting Glass Plates*. 2006.

[11] Michael Mehl, Christian Schulte, Gert Smolka, Jörg Würtz. *Mozart Demo Applications. Part II: Multi Agent and Concurrent Applications. 7 Bounce*. 2006.

[12] Denys Duchier. *Mozart Demo Applications. Part III: Distributed Applications. 11 Chat Server and Client*. 2006.

[13] Seif Haridi, Nils Franzèn. *Tutorial of Oz*. 2006.

[14] Christian Schulte, Gert Smolka. *Finite Domain Constraint Programming in Oz. A Tutorial*. 2006.

[15] Denys Duchier, Leif Kornstaedt, Martin Homik, Tobias Müller, Christian Schulte, Peter Van Roy. *System Modules. (5 Finite Domain Constraints: FD)*. 2006.