Państwowa Wyższa Szkoła Zawodowa
w Tarnowie

Instytut Politechniczny

Kierunek: Informatyka
Specjalność: Informatyka stosowana
2018/2019

Łukasz Sumara

PRACA INŻYNIERSKA

Analiza systemu MiniZinc i opracowanie zestawu przykładowych aplikacji z zakresu programowania z ograniczeniami

Promotor pracy
Prof. dr hab. inż. Antoni Ligęza

Tarnów 2018

O Ś W I A D C Z E N I E

Łukasz Sumara

Nr albumu: **27184**

Oświadczam, że moja praca pt.:

Analiza systemu MiniZinc i opracowanie zestawu przykładowych aplikacji z zakresu programowania z ograniczeniami

a. została przygotowana przeze mnie samodzielnie,*
b. nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2006 r., Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem,
c. nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,
d. nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie ani innej osobie.

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

Tarnów, dnia ....

/podpis studenta/

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

Spis Treści

Wstęp ... 7
1. Podstawowa struktura modelu ... 10
1.1. Element dołączenia ... 12
1.2. Element deklaracji zmiennej ... 12
1.3. Element przypisania ... 14
1.4. Element ograniczenia ... 14
1.5. Element rozwiązania ... 15
1.6. Element wyjścia ... 15
1.7. Element adnotacji ... 17
1.8. Element operacji zdefiniowanej przez użytkownika ... 17
1.9. Pliki danych ... 18
2. Typy danych ... 19
2.1. Integer ... 19
2.2. Float ... 19
2.3. Boolean ... 20
2.4. String ... 20
2.5. Enumerated ... 21
2.6. Set ... 22
2.7. Array ... 23
2.8. Option ... 26
2.9. Annotation ... 27
2.10. Typy ograniczone zakresem ... 28
2.11. Konwersje typów ... 28
3. Tworzenie złożonych modeli ... 31
3.1. Wyrażenie warunkowe ... 31
3.2. Zmienne lokalne, wyrażenie *let* ... 31
3.3. Tablice, listy, zestawy ... 32
3.4. Generowanie tablic i zestawów ... 34
3.5. Złożone ograniczenia ... 35
3.6. Ograniczenia globalne ... 38
3.7. Funkcje refleksji ... 40
3.8. Adnotacje przeszukiwania ... 40
3.9. Adnotacje przeszukiwań dla typów *bool*, *int*, *float*, *set* ... 42
4. Zintegrowane środowisko programistyczne ... 44
4.1. Instalacja MiniZinc w systemie Linux ... 44

4.2. Edytowanie plików 44
4.3. Wybór i zastosowania solverów 46
4.4. Konfiguracja solvera 48
4.5. Korzystanie z projektów 52
4.6. Dodanie nowego solvera do MiniZinc IDE 52
4.7. MiniZinc w linii poleceń 56
5. Przykładowe modele w języku MiniZinc, ciekawe strony internetowe powiązane z MiniZinc 58
5.1. Problem mostu i pochodni 58
5.2. Układanie domina 64
5.3. Problem kolejnych cyfr 70
5.4. Inwestycje – problem plecakowy 73
5.5. Planowanie produkcji 78
5.6. Zagadka Einsteina 83
5.7. Ciekawe strony internetowe powiązane z MiniZinc 86
Zakończenie 88
Bibliografia 89

Wstęp

Przełom wieku XX i XXI był czasem znacznych postępów w dziedzinach informatyki, dla których wcześniej głównym ograniczeniem była niewystarczająca wydajność sprzętu komputerowego. Jedną z takich dziedzin jest programowanie z ograniczeniami (ang. *Programming with Constraints, Constraint Programming*), jak można przeczytać w pracy [1] zwiększenie wydajności sprzętu komputerowego na przestrzeni lat otworzyło nowe możliwości rozwoju oprogramowania.

Artykuł [2] określa programowanie z ograniczeniami jako paradygmat programowania, który należy do rodziny paradygmatów zwaną programowaniem deklaratywnym. Programowanie deklaratywne jest alternatywą dla programowania imperatywnego, które opisuje dokładnie jakie instrukcje mają zostać wykonane oraz ich kolejność. W programowaniu deklaratywnym nie zapisuje się sposobu rozwiązania zadania, a jedynie dokonuje się specyfikacji wiedzy dziedzinowej oraz formułuje się warunki, które muszą zostać spełnione przez rozwiązanie danego problemu.

Paradygmat programowania z ograniczeniami przedstawia zależności między zmiennymi w postaci ograniczeń. Oczekiwanym wynikiem jest konkretny zestaw wartości dla zmiennych, który spełnia zadane ograniczenia. Programowanie z ograniczeniami często jest uzupełnieniem dla innych paradygmatów programowania.

Jedną z najważniejszych cech programowania z ograniczeniami jest jego deklaratywność. Jest to związane z rozdzieleniem definicji problemu od sposobu jego rozwiązania. W programowaniu z ograniczeniami definicja rozwiązania ograniczeń oraz sposób rozwiązania różnią się w zależności od ograniczanego obszaru. W związku z tym dla każdego obszaru ograniczeń musi być zapewniony odpowiedni solver, przykładowo kiedy problem obejmuje liczby zmiennoprzecinkowe nie można zastosować solvera, który obsługuje jedynie liczby całkowite. Cechą immanentną programowania z ograniczeniami jest to, że wymaga przeszukiwania szerokiego obszaru za pomocą rozwiązywania ograniczeń co wymaga dużej mocy obliczeniowej oraz dostępnej pamięci.

Początki historii programowania z ograniczeniami sięgają wczesnych lat 70. XX wieku. Ivan Sutherland przedstawia SketchPad, opisany w artykule [3], pierwszy

program komputerowy korzystający z systemu ograniczeń. Rozwój badań nad programowaniem z ograniczeniami przypada na lata 80. Powstają pierwsze wdrożenia nowej idei: „programowanie logiczne z ograniczeniami". Artykuł [4] wymienia Prolog III (rozszerzenie języka Prolog) oraz CHIP (Constraint Handling in Prolog), który używa interfejsu języka Prolog, jako implementacje programowania logicznego z ograniczeniami.

Początki praktycznego wykorzystania programowania z ograniczeniami miały miejsce w pierwszej połowie lat 90. XX wieku. Rozwinięto i sprzedano pierwsze systemy przetwarzania ograniczeń dla celów biznesowych. W pracy [5] wymienione jest kilka systemów, które jako pierwsze korzystały z ograniczeń, są to między innymi: CHARME – system, rozwinięty przez BULL, bazujący na ograniczeniach, korzystający ze składni przypominającej język C oraz SNI-Prolog (później IF-Prolog) używany przez Siemens. Były to środowiska do tworzenia oprogramowania. Za ich pomocą tworzono głównie aplikacje do harmonogramowania np. produkcji lub transportu.

Programowanie z ograniczeniami rozwinęło się jako podstawa dla budowy systemów planowania. Znajduje zastosowanie przy tworzeniu i rozwijaniu systemów APS (ang. *Advenced Planning and Scheduling*), które są rozwinięciem dla systemów ERP (ang. *Enterprise Resource Planning*). Programowanie z ograniczeniami oferuje narzędzia do modelowania problemów, używane np. do weryfikacji systemów kontroli czasu rzeczywistego, propagacji ograniczeń, używane np. w aplikacjach uwzględniających informacje zwrotne od użytkowników oraz narzędzia do przeszukiwania, używane np. w harmonogramowaniu i zarządzaniu zasobami, które zawierają problemy kombinatoryczne.

Język MiniZinc został zaprezentowany w pracy [6]. W tej pracy dyplomowej system MiniZinc jest omawiany na podstawie wersji 2.2.1. Jest to wysokopoziomowy język używany do tworzenia modeli ograniczeń. MiniZinc jest darmowy, udostępniany na zasadzie oprogramowania otwarto-źródłowego. Można używać go do modelowania problemów optymalizacji oraz satysfakcji ograniczeń. Posiada dużą bibliotekę predefiniowanych ograniczeń. Zapisany w języku MiniZinc model, wraz z plikiem danych, jest kompilowany do modelu FlatZinc, który zawiera deklaracje zmiennych, definicje ograniczeń oraz definicję funkcji celu, jeśli jest to problem optymalizacyjny.

Następnie do rozwiązania modelu w języku FlatZinc można użyć solverów, które posiadają implementację interfejsu dla tego języka. Aktualna lista z solverami posiadającymi interfejs dla języka FlatZinc znajduje się na oficjalnej stronie MiniZinc. Dotychczas obsługa modeli FlatZinc została zaimplementowana w: Gecode/FlatZinc[7], Chuffed[8], Choco[9], ECLiPSe[10], HaifaCSP[11], JaCoP[12], MinisatlD[13], Mistral 2.0[14], OR-Tools[15], OscaR/CBLS[16], Picat CP/SAT[17], SICStus Prolog[18], SCIP[19], Yuck[20].

1. Podstawowa struktura modelu

Do przedstawienia struktury prostego modelu dobrze jest posłużyć się przykładem. Widoczny poniżej model przedstawia problem wydawania reszty. Dla uproszczenia założono, że reszta wydawana jest w monetach o wartości od jednego do pięćdziesięciu groszy i nie jest ona większa od jednej złotówki.

```
% wydawanie reszty - model przedstawia problem wydawania reszty
% przy użyciu jak najmniejszej ilości monet
% reszta_1.mzn

int: kwota = 88;
var 0..100: g1;
var 0..100: g2;
var 0..100: g5;
var 0..100: g10;
var 0..100: g20;
var 0..100: g50;

constraint (g1*1 + g2*2 + g5*5 + g10*10 + g20*20 + g50*50)=kwota;

var int: monety = sum([g1,g2,g5,g10,g20,g50]);

solve satisfy;

output ["1 grosz * \(g1)\n",
"2 grosze * \(g2)\n",
"5 groszy * \(g5)\n",
"10 groszy * \(g10)\n",
"20 groszy * \(g20)\n",
"50 groszy * \(g50)\n",
"Łącznie " ++ show(monety) ++ " monet\n"];
```

Na początku zdefiniowane są zmienne, które będą używane w modelu, są to elementy deklaracji zmiennych. Zmienna *kwota* oznacza resztę do wydania. Zmienne *g1, g2, (...), g50* oznaczają liczbę monet o odpowiedniej wartości, za pomocą, których zostanie wydana pożądana reszta.

Następnie model zawiera element ograniczeń oznaczany słowem kluczowym *constraint*. Ograniczenie to wskazuje, że łączna wartość monet, które posłużą do wydania musi być równa pożądanej reszcie.

Po elemencie ograniczenia ponownie występuje element deklaracji zmiennej. Zmienna *monety* przedstawia łączoną ilość monet wykorzystaną do wydania reszty.

Kolejnym elementem jest element rozwiązania, który może wystąpić tyko jeden raz w modelu. W tym wypadku użyto *solve satisfy* co oznacza, że zadaniem solvera jest znalezienie dowolnej kombinacji wartości zmiennych, które spełnią wszystkie ograniczenia.

Na końcu modelu umieszczono element wyjścia, który definiuje w jaki sposób ma zostać zwrócony wynik otrzymany od solvera, po rozwiązaniu modelu. Przykładowe wyniki zwrócone po uruchomieniu powyższego modelu będą miały postać:

1 grosz * 8 2 grosze * 5 5 groszy * 10 10 groszy * 2 20 groszy * 0 50 groszy * 0 Łącznie 25 monet	1 grosz * 51 2 grosze * 6 5 groszy * 1 10 groszy * 2 20 groszy * 0 50 groszy * 0 Łącznie 60 monet	1 grosz * 16 2 grosze * 1 5 groszy * 14 10 groszy * 0 20 groszy * 0 50 groszy * 0 Łącznie 31 monet

Jednak w problemie wydawania reszty zadanie polega zazwyczaj na wybraniu kombinacji, która umożliwi wydanie reszty przy użyciu jak najmniejszej liczby monet. W przypadku tego modelu wystarczy zmienić element rozwiązania, czyli *solve satisfy,* na *solve minimize monety* co spowoduje, że zadaniem solvera będzie nie tylko znaleźć kombinację monet, która będzie równa wartości pożądanej reszty, ale również będzie wymagało najmniejszej możliwej wartości zmiennej *monety*. Po zmianie elementu rozwiązania otrzymany wynik będzie miał postać:

```
1 grosz * 1
2 grosze * 1
5 groszy * 1
10 groszy * 1
20 groszy * 1
50 groszy * 1
Łącznie 6 monet
```

Problem przedstawiany przy użyciu MiniZinc jest podzielony na dwie części. Pierwszą z nich jest model, który przedstawia założenia dla pewnego typu problemów. Druga część to dane dla modelu, które określają konkretny przypadek w ramach danego typu problemów.

Model utworzony przy użyciu MiniZinc składa się z wielu elementów. Na końcu każdego elementu znajduje się znak średnika. W ramach jednego modelu dany typ

elementu może występować wiele razy, z wyjątkiem elementu rozwiązania, który występuje tylko jeden raz w modelu. MiniZinc nie określa kolejności, w której muszą występować poszczególne elementy, przykładowo identyfikator zmiennej może być używany zanim jeszcze zostanie zadeklarowany.

1.1. Element dołączenia

Element ma postać:

include ”*nazwa_pliku*”;

Przykład użycia:

```
include "alldifferent.mzn";
```

nazwa_pliku – jest to ciąg znaków wskazujący ścieżkę do pliku, wraz z jego nazwą, który będzie dołączony do tego modelu. Ze względu na to, że jest to ciąg znaków konieczne jest umieszczenie ścieżki w cudzysłowach (” ”).

Element ten pozwala na zawarcie treści innego pliku w tworzonym modelu. Przydaje się to zwłaszcza w bardziej skomplikowanych i rozbudowanych modelach, które można podzielić na mniejsze pliki i później połączyć je za pomocą tego elementu. W przypadku kiedy model wymaga skorzystania z ograniczenia globalnego wymagane jest zawarcie pliku, które pozwoli na jego użycie (można również zawrzeć plik *globals.mzn*, który pozwala używać wszystkich dostępnych ograniczeń globalnych). Pełna lista ograniczeń globalnych jest dostępna w podręczniku MiniZinc [21]. System MiniZinc nie posiada żadnego rodzaju systemu modułów, treść wszystkich plików dołączonych poprzez *include* zostaje dodana do siebie i jest traktowana jako pojedynczy plik.

1.2. Element deklaracji zmiennej

Element ma postać:

typ_instancji typ_zmiennej: *nazwa_zmiennej* [= *wyrażenie*];

Przykład użycia:

```
var float: x;
```

typ_instancji – określa jakiego typu będzie instancja danej zmiennej. W języku MiniZinc rozróżnia się dwa typy instancji zmiennych. Jednym z typów jest parametr wskazywany przez słowo *par*, drugi to zmienna decyzyjna, wskazywana przez *var*. Jeśli nie sprecyzuje się typu zmiennej, wtedy przyjmuje się parametr jako typ domyślny. Różnica między nimi polega na tym, że zmienna parametryczna ma ustaloną wartość w modelu, a wartość zmiennej decyzyjnej ustalana jest dopiero w czasie rozwiązywania modelu przez solver. Dla parametru dozwolone typy zmiennych to: *float* (liczby zmiennoprzecinkowe), *integer* (liczby całkowite), *string* (ciągi znaków), *boolean* (typ logiczny), *array* (tablica wartości), *set* (zestaw wartości) oraz *ann* (typ adnotacji). Dla zmiennej decyzyjnej dostępne są tylko typy: *integer*, *float*, *bool* oraz *set* złożony ze zmiennych typu *integer*.

typ_zmiennej – ta część determinuje jakiego typu dane będzie przedstawiać dana zmienna. Wśród dostępnych typów danych można rozróżnić typy proste, do których należą: *integer*, *float*, *boolean*, *string* oraz typy złożone: *enum*, *set*, *array*, *ann* (typy zmiennych zostały opisane w rozdziale 2).

nazwa_zmiennej – jest to identyfikator zmiennej. Identyfikator zmiennej musi zaczynać się od litery i może być kombinacją liter, cyfr oraz znaków podkreślenia (‘_’). Niedozwolone jest używanie, jako nazw zmiennych, słów kluczowych języka MiniZinc oraz języka Zinc. Nie można również użyć operatora języka MiniZinc jako nazwy zmiennej (błędną nazwą będzie np. *union*).

wyrażenie – jest opcjonalną częścią tego elementu. Pozwala na nadanie wartości zmiennej w czasie jej deklaracji. Przykład:

```
par float: x = 10;
```

Taki zapis oznacza, że zmienna parametryczna typu *float* o nazwie *x* będzie miała przypisaną wartość *10*.

Element ten służy do deklaracji zmiennych decyzyjnych oraz parametrów, które będą używane w ramach danego modelu. Wszystkie identyfikatory zmiennych w MiniZinc należą do globalnej przestrzeni nazw (każdy identyfikator musi być unikalny w ramach modelu). Dodając *wyrażenie* można zainicjalizować zmienną z konkretną wartością.

1.3. Element przypisania

Element ma postać:

zmienna = *wyrażenie*;

Przykład użycia:

```
x = 10 * 2;
```

zmienna – określa identyfikator zmiennej, do której ma zostać zapisana wartość.

wyrażenie – jest to wyrażenie, którego wynik zostanie przypisany do podanej zmiennej.

Element ten służy przypisaniu wartości z wyrażenia do określonej zmiennej. W przypadku kiedy podana nazwa zmiennej należy do zmiennej decyzyjnej wartość wyrażenia jest traktowana jako ograniczenie. Zapis:

zmienna_decyzyjna = *wyrażenie*;

Jest równoznaczny z zapisem:

constraint *zmienna_decyzyjna* = *wyrażenie*;

1.4. Element ograniczenia

Element ma postać:

constraint *wyrażenie*;

Przykład użycia:

```
constraint x*5 > 10;
```

wyrażenie – wyrażenie, które ma być ograniczeniem musi być typu *bool*, może być to zarówno parametr jak i zmienna decyzyjna, jednak fakt, że parametr musi być ustalony przed próbą rozwiązania sprawia, że ciężko znaleźć zastosowanie dla ograniczenia typu *par bool*.

Jest to najważniejszy element modelu. To ograniczenia nadają kształt modelu. Rozwiązanie modelu otrzymuje się kiedy wszystkie ograniczenia zostaną spełnione.

1.5. Element rozwiązania

Element może mieć jedną z trzech postaci:

solve satisfy;

solve minimize *wyrażenie_arytmetyczne*;

solve maximize *wyrażenie_arytmetyczne*;

Przykład użycia:

```
solve satisfy;
solve minimize 5*x + 3*y;
solve maximize 5*x + 3*y;
```

wyrażenie_arytmetyczne – jest to wyrażenie określające funkcję celu, według której solver będzie prowadził optymalizację. Wyrażenie może być typu *integer* lub *float*.

Pierwsza postać elementu wskazuje na tzw. problem satysfakcji ograniczeń, zadaniem solvera w takim przypadku jest ustalenie wartości dla zmiennych decyzyjnych tak, aby spełniły ograniczenia modelu. Każdy zestaw wartości, który spełnia ograniczenia traktowany jest na równi. Druga oraz trzecia postać elementu oznacza, że zadaniem solvera będzie optymalizacja funkcji celu. W przypadku *minimize* funkcja celu będzie minimalizowana, natomiast *maximize* prowadzi do maksymalizacji wartości funkcji celu.

1.6. Element wyjścia

Element ma postać:

output [”*ciąg_znaków*”, … , ”*ciąg_znaków*”];

Przykład użycia:

```
output [”Wynikiem optymalizacji modelu jest x = ”, show(x), ”\n”];
```

ciąg_znaków – ciągiem znaków może być tekst ujęty w cudzysłów, zmienna typu *string*, lub wyrażenie w formie *show(x)*, gdzie *x* oznacza wyrażenie języka MiniZinc.

Element wyjścia służy do przejrzystego prezentowania wyników zwróconych przez solver po rozwiązaniu danego modelu. Jeśli element wyjścia nie zostanie użyty w modelu jako wynik wyświetlane są wszystkie zmienne decyzyjne wraz z ich wartościami. Dla kodowania znaków specjalnych używana jest notacja przypominająca tę z języka C, np. ”\n” dla nowej linii lub ”\t” dla tabulacji. Pojedynczy ciąg znaków musi zmieścić się w jednej linii. W przypadku kiedy jest zbyt długi można go rozdzielić na wiele linii stosując operator konkatenacji ciągu znaków np.:

```
output [”Zła wartość zmiennej Y, zmienna musi być większa niż 0”];
```

Można zapisać też jako:

```
output [”Zła wartość zmiennej Y,” ++
        ” zmienna musi być większa niż 0”];
```

Wyrażenia *show(x)* mogą być zawarte bezpośrednio w ciągu znaków. Zapis:

```
output [”x=” ++ show (x) ++ ”\n”];
```

można skrócić przy użyciu interpolacji ciągu znaków do postaci:

```
output [”x=\(x)\n”];
```

W tym przypadku zapis \(*x*) wewnątrz ciągu znaków rozumiany jest jako *show*(*x*).
Istnieją dwa warianty funkcji *show* pozwalające na formatowanie wyświetlanej liczby:

show_int(*n*,*X*) – wartość zmiennej *X* jest wyświetlana przy użyciu co najmniej *n* znaków.

show_float(*n*,*d*,*X*) – wartość zmiennej *X* wyświetlana jest przy użyciu co najmniej *n* znaków z *d* znakami po przecinku.

W obu przypadkach wartość $n > 0$ oznacza, że liczby będą wyrównane do prawej, w przeciwnym wypadku będą wyrównane do lewej.

Model może zawierać jedynie jeden komunikat wyjścia. W przypadku kiedy w ramach modelu występuje wiele elementów wyjścia wyświetlanym komunikatem jest konkatenacja elementów wyjścia według kolejności ich występowania w modelu.

1.7. Element adnotacji

Element ma postać:

annotation *identyfikator*(*argumenty*);

Przykład użycia:

```
annotation solver(int: rodzaj);
```

identyfikator – oznacza nazwę adnotacji, identyfikator adnotacji należy do globalnej przestrzeni nazw, tak jak nazwy zmiennych, więc musi być unikalny w ramach konkretnego modelu.

argumenty – są argumentami deklarowanej adnotacji

Element adnotacji służy do zadeklarowania nowej adnotacji. Więcej informacji o adnotacjach jest w rozdziale drugim.

1.8. Element operacji zdefiniowanej przez użytkownika

Istnieją trzy rodzaje operacji, które może zdefiniować użytkownik:

predicate *identyfikator*(*argumenty*) = *wyrażenie*;
test *identyfikator*(*argumenty*) = *wyrażenie*;
function *typ_funkcji*: *identyfikator*(*argumenty*) = *wyrażenie*;

Przykłady użycia:

```
predicate nieparzysta(var int: a) = x mod 2 = 1;
test nieparzysta(int: a) = a mod 2 = 1;
function int: reszta(int: a, int: b) = a mod b;
```

identyfikator – jest to nazwa konkretnej operacji i musi być unikalna w obrębie modelu.

argumenty – są to deklaracje argumentów, które przyjmuje dana operacja. Poszczególne zmienne są oddzielane przecinkiem.

wyrażenie – jest to wyrażenie, które będzie wykonane kiedy zostanie wywołana dana operacja

Poprzez zastosowanie operacji definiowanych przez użytkownika można zwiększyć poziom abstrakcji tworzonych modeli, wprowadzić do nich modularność oraz zwiększyć przejrzystość kodu. Operacje mogą być definiowane w oddzielnych plikach, dzięki czemu można tworzyć własne biblioteki ograniczeń oraz funkcji i później używać ich podczas pisania modeli przy wykorzystaniu elementów dołączeń.

Właściwością, która charakteryzuje poszczególne operacje zdefiniowane przez użytkownika jest typ zwracanych danych. Predykaty mają niejawnie zadeklarowany typ danych *var bool* dla zwracanych wartości. Najczęściej używa się ich do zapisu skomplikowanych ograniczeń. Testy mają niejawnie zadeklarowany typ danych *par bool* dla zwracanych wartości. Testy mogą zawierać jedynie zmienne parametryczne, więc ich użycie może być pomocne przy zapisie warunków dla wyrażeń warunkowych. Funkcje różnią się od testów i predykatów tym, że typ zwracanych danych jest deklarowany jawnie przez użytkownika. Funkcje są używane do zapisu skomplikowanych wyrażeń, których używa się wiele razy w modelu.

1.9. Pliki danych

Pliki danych to pliki o rozszerzeniu *.dzn*, które pozwalają na inicjalizację zmiennych modelu. Pozwalają one zaoszczędzić nakład pracy w przypadku, gdy dla jednego modelu pożądane jest rozwiązanie dla więcej niż jednego zestawu wartości zmiennych. Zamiast przypisywać wartości zmiennych bezpośrednio w kodzie modelu i później modyfikować go w celu zmiany tych wartości, można stworzyć dla każdego zestawu oddzielny plik danych, a następnie rozwiązać model przy użyciu różnych plików danych.

Nazwa lub nazwy plików danych dla modelu nie są w żaden sposób umieszczane w jego kodzie. Zadaniem solvera jest umożliwienie dołączenia dowolnej liczby plików danych do rozwiązywanego modelu. Pliki danych składają się tylko i wyłącznie z elementów przypisania. Każda zmienna może mieć tylko jedno przypisanie wartości wewnątrz wszystkich plików danych dołączonych do konkretnego rozwiązania modelu. Przykład pliku danych:

```
% deklaracja zmiennych w modelu
array[1..3] of int: zmienna1;
float: zmienna2;
```

```
% dane1.dzn
zmienna1=[1,5,10];
zmienna2=10.5;
```

```
% dane2.dzn
zmienna1=[1,2,3];
zmienna2=3.4;
```

2. Typy danych

Zmienne w MiniZinc poza typem przechowywanej wartości posiadają również typ instancji tej wartości. Rozróżnia się dwa typy instancji: ustalona (ang. *fixed*) oraz nieustalona (ang. *unfixed*). Wartości ustalone to parametry modelu, natomiast wartości nieustalone to zmienne decyzyjne modelu. Zmienne parametryczne muszą mieć nadaną wartość w czasie tworzenia modelu. Wartości zmiennych decyzyjnych nadawane są w trakcie rozwiązywania modelu.

2.1. Integer

Typ *integer* reprezentuje liczby całkowite. Język MiniZinc nie określa zakresu wartości tego typu. W specyfikacji języka MiniZinc [22] można przeczytać, że kiedy przekroczony zostanie zakres wartości obsługiwany przez solver powinien on przerwać działanie. Typ *integer* może występować w postaci ustalonej, jako parametr (*int* lub *par int*) oraz nieustalonej, jako zmienna decyzyjna (*var int*). Nie jest to typ skończony. Język MiniZinc dopuszcza zapis dla liczb typu *integer* w postaci dziesiętnej, szesnastkowej oraz ósemkowej:

```
int: x = 10;
int: y = 0xA;
int: z = 0o12;
```

W podręczniku MiniZinc [21] wymienione są obsługiwane operatory arytmetyczne dla typu *integer*: dodawanie (+), odejmowanie (–), mnożenie (*), dzielenie całkowite (*div*), reszta z dzielenia całkowitego (*mod*). Operatory ”+” oraz ”–” są dostępne również jako operatory jednoargumentowe. Dla tego typu dostępne są funkcje wartości absolutnej (*abs*), oraz potęgowania (*pow*).

2.2. Float

Typ *float* reprezentuje liczby rzeczywiste. MiniZinc nie określa zakresu tego typu. W specyfikacji języka MiniZinc [22] można przeczytać, że kiedy solver przekroczy obsługiwany zakres wartości, lub będzie potrzeba przeprowadzenia operacji wyjątkowej na typie *float* (np. nadanie zmiennej wartości *NaN*) solver powinien przerwać działanie. Typ *float* może występować w postaci ustalonej, jako parametr (*float* lub *par float*) oraz nieustalonej, jako zmienna decyzyjna (*var float*).

Nie jest to typ skończony. Zmienna zadeklarowana jako zakres typu *float* również nie jest typem skończonym. Typ *float* może być typem skończonym, jeżeli zostanie ograniczony poprzez zestaw wartości (*set*). W podręczniku MiniZinc [21] wymienione są obsługiwane operatory arytmetyczne dla typu *float* : dodawanie (+), odejmowanie (–), mnożenie(*), dzielenie zmiennoprzecinkowe (/). Operatory ”+” oraz ”–” są dostępne również jako operatory jednoargumentowe.

Język MiniZinc zawiera zestaw funkcji operujących na liczbach zmiennoprzecinkowych, które zostały wymienione w podręczniku MiniZinc [21]: wartość absolutna (*abs*), pierwiastek kwadratowy (*sqrt*), logarytm naturalny (*ln*), logarytmy o podstawach dwa oraz dziesięć (*log2*, *log10*), eksponenta (*exp*), zestaw funkcji trygonometrycznych (*sin, cos, tan*), funkcje odwrotne do trygonometrycznych (*asin, acos, atan*), funkcje hiperboliczne (*sinh, cosh, tanh, asinh, acosh, atanh*), potęgowanie (*pow*).

Typ *float* pozwala na zapis wartości na kilka sposobów:

```
float: x = 10.5;
float: y = 1.2e-5;
float: z = 2e+2;
```

W zapisie naukowym można naprzemiennie używać *e* oraz *E*.

2.3. Boolean

Typ *bool* reprezentuje prawdę lub fałsz. Wartość *false* jest traktowana jako mniejsza od *true*. Typ *bool* może występować w postaci ustalonej jako parametr (*bool* lub *par bool*) oraz nieustalonej, jako zmienna decyzyjna (*var bool*). Typ *bool* należy do typów skończonych. Domeną typu *bool* jest {*false, true*} (fałsz, prawda). Podręcznik MiniZinc [21] wymienia następujące operatory dla typu *bool*: koniunkcja (/\), alternatywa (\/), implikacja (–> lub <–), równoważność (<–>) i negacja (*not*).

2.4. String

Typ *string* jest używany w elementach wyjścia, jako argument operacji zdefiniowanych przez użytkownika oraz w funkcji *assert*. W specyfikacji języka MiniZinc [22] znajduje się informacja, że typ *string* jest typem prostym, co oznacza, że nie jest to lista złożona ze znaków. Typ *string* występuje tylko w postaci ustalonej, jako parametr (*string* lub *par string*), nie należy do grupy typów skończonych. W przypadku

porównań typ *string* przyjmuje porządek leksykograficzny na podstawie kodów reprezentujących rozważane znaki.

Dla typu *string* dostępny jest operator konkatenacji (++), który łączy dwa ciągi znaków w jeden. Dostępna jest również funkcja *concat*, służy ona do konkatenacji w ciąg znaków elementów typu *string* zawartych w tablicy podanej jako argument funkcji. Za pomocą funkcji *join* można połączyć w jeden ciąg znaków tablicę z elementami typu *string*, które będą rozdzielone określonym separatorem. Literałem typu łańcuchowego w języku MiniZinc jest ciąg znaków zawarty w znakach cudzysłowu:

```
string: tekst = "Przykładowy literał łańcuchowy\n"
                ++ "\tzawierający \(1+1+1) wiersze tekstu\n"
                ++ "oraz interpolację ciągów znakowych";
output [tekst, "\n"];
```

Wynikiem tej części kodu jest:

```
Przykładowy literał łańcuchowy
      zawierający 3 wiersze tekstu
oraz interpolację ciągów znakowych
```

2.5. Enumerated

Typy wyliczeniowe (ang. *enumerated*) są zbiorem wartości, które dany typ może przyjąć. Pojedynczy element zbioru jest nazywany przypadkiem wyliczeniowym (ang. *enum case*). Identyfikatory używane do nazywania przypadków są określane jako nazwy przypadków wyliczeniowych (ang, *enum case name*). Nazwy przypadków, podobnie jak identyfikatory zmiennych, należą do globalnej przestrzeni nazw, więc muszą być unikalne w obrębie modelu. Typ wyliczeniowy może być użyty zamiast typu *integer* w celu ograniczenia możliwych wartości. Zmienne typu *enum* mogą występować w postaci ustalonej i nieustalonej. Przyjmując *X* jako nazwę typu *enum* możliwe warianty typu dla zmiennych to: *X* lub *par X* dla parametrów oraz *var X* dla zmiennych decyzyjnych.

Typy *enum* są typami skończonymi. Ich domeną wartości jest zestaw przypadków wyliczeniowych. W przypadku porównania dwóch wartości typu *enum* jako większa uważana jest ta, która była zadeklarowana później. Przykład zmiennej typu *enum*:

```
enum kolor = {czerwony, zielony, niebieski};
var kolor: kol;
```

W tym wypadku zmienna *kol* jest typu wyliczeniowego *kolor* i może przyjąć jedną z trzech wartości: *czerwony*, *zielony* lub *niebieski*, a w przypadku porównania wartość *zielony* jest większa od *czerwony*.

Funkcje dedykowane dla typu *enum* to:

enum_next(X,x) – zwraca wartość następującą po wartości *x* w typie *enum* o nazwie *X*. Jeśli element *x* jest ostatni, wtedy funkcja zwraca wartość ”⊥”, co odpowiada wartości *false* typu *bool*.

enum_prev(X,x) – zwraca wartość poprzedzającą wartość *x* w typie *enum* o nazwie *X*. Jeśli element *x* jest pierwszy, wtedy funkcja zwraca wartość ”⊥”, co odpowiada wartości *false* typu *bool*.

to_enum(X,i) – zwraca wartość typu *enum* o nazwie *X*, która jest przypisana do wartości odpowiadającej wartości zmiennej *i* typu *integer*. W przypadku kiedy wartość *i* jest mniejsza lub równa 0 oraz kiedy przekracza liczbę elementów w typie *enum* o nazwie *X* funkcja zwraca ”⊥”, co odpowiada wartości *false* typu *bool*.

2.6. Set

Zestaw (ang. *set*) składa się ze zbioru niepowtarzających się wartości. Zmiennej złożonej *set* można używać do ograniczenia zakresu wartości typów, które nie są skończone (ang. *not finite*) co pozwala użyć ich w sytuacji, kiedy wymagane jest użycie typu skończonego.

W modelu typ *set* może występować w postaci ustalonej (*set of*). Kiedy typ *set* używany jest w postaci nieustalonej (*var set of*) dopuszczalne są jedynie elementy typu *int*, który musi być skończony (np. jako zakres lub zestaw wartości) oraz elementy typu *enum*.

```
enum dni_tygodnia = {pn, wt, sr, czw, pt, sob, ndz};
var set of dni_tygodnia: zestaw_dni;
```

Przedstawiony powyżej kod pokazuje sytuację, w której wartości, będące elementami zmiennej typu *set*, są typu danych *enum* o nazwie *dni_tygodnia*. Oznacza to, że zmienna *zestaw_dni* może zawierać jedną lub więcej wartości dostępnych

w zadeklarowanym typie danych *enum* o nazwie *dni_tygodnia*, jednak ze względu na to, że jest to typ *set* żadna z wartości nie może się powtórzyć.

Typ *set* jest typem skończonym tylko i wyłącznie wtedy, kiedy zawiera elementy typu skończonego, jeśli zawiera elementy, które nie są typu skończonego on sam również nie jest skończony. W przypadku kiedy typ *set* jest skończony jego domeną jest zbiór potęgowy domen jego elementów. Przykład: *set of 1..2* ma domenę w postaci: [{}, {1}, {1,2},{2}].

Podręcznik MiniZinc [21] wymienia następujące operatory dostępne dla typu *set*: '..' – zwraca zestaw wartości z określonego zakresu, np. *1..3* zwróci zestaw {1,2,3}, *diff* – zwraca różnicę pomiędzy zestawami, *in* – sprawdza, czy element znajduje się w zestawie, *intersect* – zwraca część wspólną zestawów, *subset* – sprawdza, czy zestaw jest podzbiorem innego zestawu, *superset* – sprawdza, czy zestaw jest nadzbiorem innego zestawu, *symdiff* – zwraca różnicę symetryczną zestawów, *union* – zwraca sumę zbiorów.

Literał dla zestawu w języku MiniZinc:

```
set of float: zestaw;
zestaw = {1.5, 3.0, 4.5};
```

2.7. Array

Tablica (ang. *array*) służy do przechowywania wielu elementów jednego typu. W języku MiniZinc tablice mogą zawierać elementy typów bazowych: *integer*, *float*, *bool*, *enum*, *string* oraz typ *set*. Specyfikacja języka MiniZinc [22] zawiera informację o tym, że tablice nie mogą się składać z innych tablic. Istnieją dwa sposoby deklaracji tablic. Pierwszy sposób, z jawnym indeksowaniem:

```
array[1..3] of int: tab;
```

W takim wypadku przy próbie przypisania wartości ich liczba musi się zgadzać z ilością dostępnych indeksów, w innym razie nastąpi błąd podczas kompilacji modelu. Drugi sposób deklaracji to indeksowanie niejawne:

```
array[int] of int: tab;
```

W tym wypadku nie będą sprawdzane indeksy przy próbie przypisania wartości.

Indeksami tablicy mogą być jedynie zestawy sąsiadujących ze sobą wartości typu *integer* lub *enum*. Jako przykład dla indeksowania tablicy typem *enum* można wykorzystać problem planowania zmian dla pracowników w ciągu tygodnia.

```
enum pracownik = {pracownik_1, pracownik_2, pracownik_3};
enum dzien = {pn, wt, sr, czw, pt, sob, ndz};
enum zmiana = {w, d, n};
array[pracownik,dzien] of var zmiana: plan_zmian;
```

W tym fragmencie kodu indeksami kolumn tablicy *plan_zmian* są wartości typu *enum* o nazwie *dzien*, natomiast poszczególne wiersze są indeksowane poprzez wartości typu *enum* o nazwie *pracownik*. Dodatkowo tablica będzie zawierała wartości typu *enum* o nazwie *zmiana*, gdzie wartość: *w* – odpowiada dniu wolnemu, *d* – zmianie dziennej, *n* – oznacza zmianę nocną.

Kiedy tablica jest indeksowana niejawnie przyjmowanym zestawem indeksów jest: *1..n* – gdzie *n* oznacza liczbę elementów w tablicy. W przypadku jawnego indeksowania tablicy nie jest wymagane aby pierwszym indeksem tablicy była wartość *1*. Jednak w przypadku użycia takiego indeksowania niemożliwe będzie wpisanie wartości do takiej tablicy bezpośrednio przy użyciu literału tablicowego. Aby przypisać wartości do tablicy indeksowanej w ten sposób należy użyć funkcji *array1d*, gdzie jako pierwszy argument podaje się zestaw indeksów, a drugim argumentem jest literał tablicowy zawierający wartości, które będą wpisane do tablicy zwróconej przez funkcję. Pokazuje to poniższy przykład:

```
set of int: zestaw = {-1,0,1,2};
array[zestaw] of int: tablica;

tablica = [-1,0,1,2]; %Błąd - Index set mismatch
tablica = array1d(zestaw, [-1,0,1,2]);
```

Próba przypisania wartości bezpośrednio poprzez literał tablicowy skończy się zwróceniem błędu kompilacji. Poprawnie zostaną wprowadzone do tablicy wartości przy użyciu funkcji *array1d*.

Niezależnie od użytego indeksowania w deklaracji tablicy rozmiary wszystkich tablic muszą być znane przed próbą rozwiązania modelu przez solver. MiniZinc nie posiada listy o zmiennej długości jako typu danych, jednak możliwa jest deklaracja

tablicy jednowymiarowej przy użyciu słowa kluczowego *list*. Dwie poniższe deklaracje są sobie równe:

```
list of int: tab;
array[int] of int: tab;
```

Tablica może składać się z parametrów lub zmiennych decyzyjnych. Tablice jednowymiarowe mogą być łączone przy użyciu operatora konkatenacji (++). Indeksy powstałej w ten sposób tablicy są zakresem o postaci *1..n* gdzie *n* oznacza liczbę elementów w obu łączonych tablicach, np. połączenie tablicy [1,2,3] z tablicą [3,4,5] zwróci tablicę o wartościach [1,2,3,3,4,5] z indeksami od 1 do 6.

Zestaw wbudowanych funkcji pozwala na zamianę listy (tablicy jednowymiarowej) na tablicę o określonej liczbie wymiarów. Funkcja ma postać: *arrayXd(zakres, lista)* – gdzie *X* odpowiada liczbie wymiarów (funkcje obsługują tablice od jednego do sześciu wymiarów), *zakres* określa indeksy kolejnych wymiarów, a *lista* jest tablicą jednowymiarową, której wartości będą umieszczone w tablicy zwróconej jako wynik funkcji.

Język MiniZinc nie posiada literałów tablicowych dla tablic mających więcej niż dwa wymiary. Aby wpisać wartości do tablic, które mają więcej niż dwa wymiary należy użyć zestawu funkcji *arrayXd* opisanych wcześniej. Literały tablicowe w języku MiniZinc dla tablic jednowymiarowych oraz dwuwymiarowych:

```
array[1..3] of int: tab_1d;
array[1..2, 1..3] of int: tab_2d;
tab_1d = [1, 3, 5];
tab_2d = [| 1, 2, 3
          | 4, 5, 6 |];
```

Dla tablic, których elementami są zmienne decyzyjne możliwe jest wykorzystanie zapisu zmiennej anonimowej (ang. *anonymous*). Zmienna anonimowa zapisywana jest jako ”_” w literale tablicowym, np.:

```
array[1..4] of var int: x = [1,_,_,_];
```

W przypadku użycia zmiennych anonimowych wymagane jest aby przynajmniej jeden element tablicy nie był anonimowy.

MiniZinc obsługuje wycinanie fragmentów tablic (ang. *array slicing*). Biorąc za przykład prostą tablicę dwuwymiarową:

```
array[1..4,1..4] of int: tablica =
  [| 1, 2, 3, 4
   | 5, 6, 7, 8
   | 9,10,11,12
   |13,14,15,16|];
```

Można wydobyć pojedynczy wiersz danej tablicy, pojedynczą kolumnę, lub wycinek tablicy przy pomocy wyrażeń przedstawionych poniżej:

```
array[int] of int: drugi_wiersz = tablica[2,..];
array[int] of int: druga_kolumna = tablica[..,2];
array[int,int] of int: wycinek_tablicy = tablica[2..3,2..3];
```

Tablica *drugi_wiersz* będzie zawierała wartości: *5,6,7,8,* analogicznie tablica *druga_kolumna* będzie zawierała wartości *2,6,10,14,* natomiast *wycinek_tablicy* będzie zawierał wartości *6,7,10,11*. Tablice powstałe w wyniku wycięcia zachowują swoje indeksy. Oznacza to, że *drugi_wiersz* oraz *druga_kolumna* mają indeksy w postaci zestawu *1..4*, natomiast *wycinek_tablicy* to tablica dwuwymiarowa o indeksach [*2..3,2..3*]

2.8. Option

Typ opcjonalny (ang. *option*) pozwala na deklarowanie zmiennych, które mogą być zawarte w modelu, ale nie muszą. Jest to użyteczne na przykład kiedy jedna ze zmiennych decyzyjnych nie musi być brana pod uwagę jeśli nie zostaną spełnione konkretne warunki. Typ *opt* pozwala zmiennej przyjąć wartość *absent* (nieobecna) zapisywaną przy użyciu ”<>”.

Typ *opt* może być użyty jedynie przy typach *bool, int, float* oraz ich odpowiednikach w postaci zestawów. Typ *opt* może być deklarowany zarówno jako parametr, jak i zmienna decyzyjna. Przykładowa deklaracja zmiennej:

```
var opt int: x = <>;
```

Podręcznik MiniZinc [21] wymienia trzy funkcje obsługujące zmienne typu *opt*: *occurs* zwraca prawdę jeśli zmienna ma wartość inną niż ”<>”, *absent* zawraca prawdę jeśli zmienna ma wartość ”<>”, *deopt* zwraca wartość argumentu funkcji, w przypadku wywołania funkcji z argumentem o wartości ”<>” zwracany jest błąd.

2.9. Annotation

Adnotacje (ang. *annotations*) są to wartości typu *ann*. Adnotacje w języku MiniZinc służą do przekazywania niedeklaratywnych informacji do solvera, który będzie rozwiązywał dany model. Mogą być używane przy wyrażeniach, deklaracjach zmiennych oraz przy elementach *solve*. Postać deklaracji nowej adnotacji:

annotation *nazwa*(*argumenty*);

nazwa – identyfikator tworzonej adnotacji

argumenty – definicje argumentów, które przyjmuje adnotacja

Specyfikacja języka MiniZinc [22] mówi, że adnotacje nie mogą być przeładowywane, oznacza to deklarowanie więcej niż jednej adnotacji o tej samej nazwie, ale z innymi argumentami. Aby dodać adnotację do elementu należy użyć operatora ”::” przykład:

```
var int: x ::adn1;
x=(1+2) ::adn2(0,10);
solve ::adn3(”p”)::adn4(”a”,2) maximize x;
```

Zagnieżdżenie adnotacji nie ma znaczenia, o czym wspomina specyfikacja języka MiniZinc [22].

W podręczniku MiniZinc [21] adnotacje są podzielone na trzy kategorie: adnotacje ogólne, adnotacje siły propagacji oraz adnotacje przeszukiwań, które zostały omówione w rozdziale trzecim. Adnotacje siły propagacji są przeznaczone dla elementów ograniczeń: *bounds* – solver powinien zastosować propagację granic do rozwiązania tego ograniczenia, *domain* – solver powinien użyć propagacji domeny do rozwiązania tego ograniczenia.

Przykłady adnotacji ogólnych: *add_to_output* – adnotacja przeznaczona dla zmiennych, deklaruje, że zmienna powinna być dodana do wyjścia modelu, jest brana pod uwagę kiedy model nie ma elementu wyjścia, *maybe_partial* – deklaruje, że wyrażenie może mieć niezdefiniowany wynik, co zapobiega wyświetlaniu ostrzeżeń (ang, *warnings*), *promise_total* – deklaruje funkcję jako całkowitą, co oznacza, że nie nakłada ona ograniczeń na swoje argumenty. Pełna lista adnotacji jest dostępna w podręczniku MiniZinc.

2.10. Typy ograniczone zakresem

MiniZinc pozwala na używanie typów *integer* oraz *float* z ograniczeniem dopuszczalnych wartości. Dla typu *integer* dostępne są trzy rodzaje deklaracji zmiennych: za pomocą operatora zakresu "..", poprzez wypisanie dopuszczalnych wartości lub za pomocą identyfikatora wskazującego na typ *set*:

```
var 1..3: x;
var {1,3,6}: x;
set of int: zestaw = {1,3,6};
var zestaw: x;
```

Tak zadeklarowane zmienne są traktowane jako skończony typ danych, ich domeną są wartości zawarte w zestawach.

Podobnie można deklarować zmienne typu *float*, tutaj również można użyć zakresu lub zestawu do ograniczenia możliwych wartości, jednak w przypadku typu *float* ograniczenie poprzez zakres nie wpływa na skończoność typu. Typ *float* może być typem skończonym tylko w przypadku ograniczenia możliwych wartości przez zestaw.

```
1.  var 1.0..3.0: x;
2.  var {1.5,3.0,3.5}: x;
3.  set of int: zestaw = {1.5,3.0,3.5};
4.  var zestaw: x;
```

W linii pierwszej zmienna *x* nie jest typu skończonego, jednak w linii drugiej oraz czwartej jest typu skończonego, ponieważ może przyjąć tylko trzy wartości.

2.11. Konwersje typów

Język MiniZinc używa typowania statycznego. Oznacza to, że w czasie kompilacji zmiennym przypisywane są typy, które określają rodzaj danych jakie będzie przechowywała dana zmienna. Przeciwieństwem dla typowania statycznego jest typowanie dynamiczne, gdzie zmiennym przypisywane są typy w czasie wykonywania programu, dzięki czemu w różnych chwilach wykonywania programu zmienna może przechowywać różne typy danych , przykładem takiego języka jest MATLAB.

Język MiniZinc jest do pewnego stopnia językiem o słabym typowaniu. Można go określić w ten sposób, ponieważ używa niejawnego rzutowania podczas wykonywania określonych operacji. Taką operacją może być na przykład dodanie

liczby typu *int* do liczby typu *float*, przed dodaniem liczba *int* jest niejawnie konwersowana na typ *float* i dopiero wtedy następuje operacja dodawania.

Jeden z rodzajów konwersji dotyczy typów, które mogą być zmiennymi decyzyjnymi, są to: *integer*, *float*, *boolean*, *enumerated* oraz *set*. W przypadku kiedy zostanie użyty parametr jednego z wymienionych typów, a oczekiwana będzie w tym miejscu zmienna decyzyjna, zostanie ten parametr odczytany jako zmienna decyzyjna.

MiniZinc niejawnie dokonuje konwersji typu *boolean* na typ *integer*, przyjmując, że fałszowi odpowiada wartość 0, a prawdzie wartość 1. Typ *integer* może być niejawnie konwersowany na typ *float*. Dodatkowo typ *boolean* może być konwersowany na typ *float* (najpierw konwersowany jest *bool* na *int*, a następnie *int* na *float*). Do jawnej konwersji tych typów można użyć funkcji: *bool2int*, *int2float*, *bool2float*. MiniZinc nie dopuszcza niejawnej konwersji typu *float* na typ *integer*, do tego celu służą funkcje: *floor* (zaokrąglenie w doł), *ceil* (zaokrąglenie w górę), *round* (zaokrąglenie do najbliższej liczby całkowitej). Funkcje te mogę być użyte jedynie do konwersji parametru typu *float* do parametru typu *int*.

Wśród typów złożonych dopuszczalna jest niejawna konwersja zestawu typu *X* do tablicy z elementami typu *X*. Do jawnej konwersji służy funkcja *set2array*. Dodatkowo możliwa jest konwersja niejawna tablicy z elementami typu *X* na tablicę z elementami typu *Y*, pod warunkiem, że możliwa jest konwersja typu *X* do typu *Y*. Można również jawnie dokonać konwersji przy użyciu funkcji: *bool2int*, *int2float*, *bool2float* podając jako argument tablicę z elementami odpowiedniego typu.

Widoczny poniżej przykład, pochodzący z podręcznika MiniZinc [21], dobrze pokazuje świadome wykorzystanie konwersji typów zmiennych przy tworzeniu ograniczeń:

```
% magiczna seria - problem polega na znalezieniu wartości tablicy tak
% aby odpowiadały one temu ile razy indeks występuje w tabeli jako
% wartość
% magiczna_seria.mzn - źródło The MiniZinc Handbook
int: n;
array[0..n-1] of var 0..n: s;
constraint forall(i in 0..n-1) (
s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));
solve satisfy;
output [ "s = \(s);\n" ];
```

Model przedstawia problem polegający na znalezieniu wartości dla tablicy *s* mającej *n* elementów tak, aby wartość każdego elementu tablicy odpowiadała temu ile razy indeks tego elementu występuje w tablicy jako wartość. Przykładem takiego zestawu wartości, dla *n=4*, może być *s* = [1,2,1,0], pamiętając, że indeksy tablicy stanowi zestaw *0..n-1*, wartości te należy rozumieć w następujący sposób: w tablicy występuje 1 raz wartość 0, 2 razy wartość 1, 1 raz wartość 2 oraz nie występuje ani raz wartość 3.

Ograniczenie pozwalające na osiągnięcie takiego wyniku jest zbudowane przy użyciu zagnieżdżonego wywołania generatora dla funkcji agregującej *sum*, który jest wykorzystany do stworzenia elementów tablicy, która następnie będzie wykorzystana jako argument dla funkcji *forall* (generatory oraz ich zastosowanie zostało omówione w rozdziale 3). Funkcja *forall* zwróci prawdę, kiedy wszystkie elementy tablicy podanej jako argument tej funkcji przyjmą wartość *true*.

Elementy tablicy dla funkcji *sum* są tworzone poprzez wykorzystanie konwersji typu danych. Za każdym razem kiedy rozważany indeks wystąpi jako wartość w tablicy funkcja *bool2int* zwróci wartość 1, w przypadku kiedy zależność *s[j]=i* zwróci fałsz funkcja zwróci wartość 0. Po przejrzeniu pod kątem danego indeksu całej tablicy funkcja *sum* zwróci wynik oznaczający ile razy dany indeks wystąpił jako wartość. Następnie ten wynik będzie porównany z *s[i]*, co oznacza wartość tablicy pod rozważanym indeksem. Kiedy wynik tej zależności będzie prawdą dla każdego elementu w tablicy *s* ograniczenie zostanie spełnione, a tablica *s* zostanie zwrócona jako wynik modelu.

```
constraint forall(i in 0..n-1) (
s[i] = (sum(j in 0..n-1)(s[j]=i)));
```

Warto dodać, że taki sam wynik zwróci model, w którym pominie się jawną konwersję typu *bool* na typ *int*, ponieważ MiniZinc dokonuje tej konwersji niejawnie, kiedy jest wymagana. Zamiana elementu ograniczenia w tym modelu na ten przedstawiony powyżej nie spowoduje, żadnych zmian w otrzymywanym wyniku, w obu przypadkach (przy ustawionym wyświetlaniu wszystkich wyników dla problemów satysfakcji ograniczeń) model, przy założeniu *n=4*, zwróci dwa wyniki:

```
s = [1, 2, 1, 0];
s = [2, 0, 2, 0];
```

3. Tworzenie złożonych modeli

W pierwszym rozdziale zostały omówione elementy składające się na model MiniZinc. Podczas modelowania skomplikowanych problemów bardzo często pojawia się potrzeba rozbudowania tych prostych elementów. W tym rozdziale zostaną przedstawione dodatkowe funkcjonalności, które mogą zostać użyte podczas tworzenia modeli w MiniZinc.

3.1. Wyrażenie warunkowe

Język MiniZinc udostępnia wyrażenie warunkowe w postaci:

if *warunek* **then** *wyrażenie_1* **else** *wyrażenie_2* **endif**

W tym wyrażeniu *warunek* oznacza wyrażenie typu *bool*. W przypadku kiedy warunek jest prawdziwy zostaje wykonane *wyrażenie_1*, kiedy warunek nie jest spełniony wykonywane jest *wyrażenie_2*. Oba wyrażenia muszą mieć ten sam typ, który jest jednocześnie typem danego wyrażenia warunkowego.

W języku MiniZinc instrukcja warunkowa nie jest instrukcją przepływu sterowania, stosowana jest jako wyrażenie warunkowe, co oznacza, że zawsze musi zwrócić jakąś wartość, przez co może być zawarta w innych wyrażeniach. Przykładowe użycie instrukcji warunkowej:

```
int: a = if x > y then x-y else y-x endif;
```

3.2. Zmienne lokalne, wyrażenie *let*

W modelach MiniZinc występuje globalna przestrzeń nazw, dlatego wszystkie zadeklarowane zmienne są widoczne w każdej części modelu. Wyjątkami są iteratory używane w generatorach, oraz argumenty predykatów, testów i funkcji. Innym sposobem wprowadzenia zmiennych oraz ograniczeń lokalnych do modelu jest wyrażenie *let*. Wyrażenia te są najczęściej używane wewnątrz predykatów oraz funkcji. Składnia wyrażenia *let* prezentuje się następująco:

let { *element*; (…) *elelment*; } **in** *wyrażenie*

Wewnątrz wyrażenia *let* można używać tylko elementów deklaracji zmiennych oraz elementów ograniczeń. *Wyrażenie* jest dowolnym wyrażeniem, w którym można

wykorzystać zmienne lokalne zawarte w wyrażeniu *let*. Ograniczeń w wyrażeniu *let* używa się z reguły do kontrolowania zmiennych lokalnych. W obrębie wyrażenia *let* każda zmienna może być zadeklarowana tylko jeden raz. Deklaracja zmiennej musi wystąpić zanim zostanie ona użyta. Lokalne parametry deklarowane wewnątrz wyrażenia *let* muszą zostać zainicjalizowane. Podobnie jest w przypadku lokalnych zmiennych decyzyjnych, jednak tutaj jest możliwość opuszczenia inicjalizacji, kiedy zmienna jest określona przez lokalne ograniczenia.

Przykład użycia wyrażenia *let* w modelu:

```
function var int: funkcja(int: a, int: b) =
let {
  var int: c;
  constraint c = a + b;
} in a + b + c;

int: c=1;

var int: x = funkcja(1, 2);
var int: y = 1 + 2 + c;
```

W tym przykładzie zadeklarowana jest funkcja, która zwraca wartość typu *var int*. Jako argument są podawane dwie zmienne typu *int*. Po znaku ”=” następuje wyrażenie, które będzie wykonane kiedy zostanie wywołana funkcja. W tym przypadku jest to wyrażenie *let*, w którym zadeklarowana jest zmienna lokalna *c* oraz ograniczenie dotyczące tej zmiennej. Po słowie kluczowym *in* następuje wyrażenie, które korzysta ze zmiennych zadeklarowanych wewnątrz wyrażenia *let*. Zmienne *x* oraz *y* w tym przykładzie przyjmą różne wartości, ponieważ w sytuacji, kiedy zmienna lokalna ma taką samą nazwę jak jedna ze zmiennych w globalnej przestrzeni nazw wyrażenie po słowie kluczowym *in* bierze pod uwagę wartość zmiennej lokalnej. W tym przypadku wartość *x* będzie równa 6, natomiast wartość *y* będzie równa 4.

3.3. Tablice, listy, zestawy

Najczęściej modele opisujące dany problem tworzy się z zamiarem sprawdzenia na nich zestawu różnych danych. Kiedy zestawy różnią się tylko wartościami zmiennych wystarczające są zmienne proste. Nierzadko jednak jest potrzeba zmiany nie tylko wartości zmiennych ale też ich liczebności, co przy prostych typach zmiennych

jest trudne lub wręcz niemożliwe do zrealizowania. W takich przypadkach używane są zmienne o typach złożonych: tablice, listy, zestawy.

Użycie tablic w programie można pokazać na przykładzie modelu z rozdziału pierwszego, który przedstawiał problem wydawania reszty przy użyciu najmniejszej ilości monet. Przerabiając ten program z użyciem tablic, deklaracja zmiennych może wyglądać następująco:

```
float: kwota;
array[int] of float: nominaly;
int: rozmiar = length(nominaly);
array[1..rozmiar] of var 0..100: ilosc;
```

Taki zapis pozwoli na modyfikację ilości oraz wartości nominałów w pliku danych dla tego modelu. Zmienna *kwota* również będzie inicjalizowana za pomocą pliku danych co pozwoli na szukanie rozwiązań dla różnych kwot. Użyty tutaj typ *float* pozwoli na podanie wymaganej kwoty, oraz wartości nominałów w bardziej naturalny sposób. Funkcja *length* zwraca liczbę elementów tablicy, dzięki czemu Tablica *ilosc* jest tworzona na podstawie rozmiaru tablicy nominałów, ponieważ ilość różnych monet zawsze będzie równa liczbie podanych nominałów. Tablica *nominaly* nie ma określonego rozmiaru, ponieważ ten będzie zależał od użytego pliku z danymi.

Przykładowy plik z danymi:

```
% wydawanie reszty - plik danych
% reszta_2.dzn
%żądana reszta
kwota = 88.88;
%tablica nominałów
nominaly = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100,
200, 500];
```

W przypadku użycia listy zamiast tablicy jedyną konieczną zmianą jest ingerencja w deklarację zmiennej, w przypadku listy deklaracja ma postać:

```
list of float: nominaly;
```

Jako, że listy są jedynie innym zapisem dla tablic jednowymiarowych obowiązują tutaj te same zasady co w przypadku tablic, tzn. nie można deklarować tablicy, której rozmiar nie będzie znany w czasie tworzenia modelu. W deklaracji zmiennej nie określa się rozmiaru listy co oznacza, że jej rozmiar jest podawany poprzez wpisanie do niej wartości. Dlatego też tablica *nominaly* może być

zadeklarowana jako lista, ponieważ jej wartości są inicjalizowane przy użyciu pliku danych. Inaczej sprawa wygląda z tablicą *ilosc*, nie posiada ona wartości przed rozwiązaniem modelu co oznacza, że w tym przypadku zastosowanie listy jest niemożliwe.

3.4. Generowanie tablic i zestawów

W języku MiniZinc jest możliwość generowania tablic oraz zestawów za pomocą odpowiednich wyrażeń. W zapisie, generatory dla tablic oraz zestawów, różnią się jedynie użytymi nawiasami, są to: ”[]” dla tablic oraz ”{}” dla zestawów.

Dla tablic: [*wyrażenie* | *generator,* (…), *generator* **where** *warunek*]
Dla zestawów: { *wyrażenie* | *generator,* (…), *generator* **where** *warunek* }

wyrażenie – przedstawia sposób w jaki mają być tworzone poszczególne elementy tablicy lub zestawu.

generator – ma postać *identyfikator*, (…), *identyfikator* **in** *wyrażenie_tablicowe*, gdzie identyfikatory oznaczają nazwy zmiennych, które będą iterowane po wartościach zawartych w wyniku wyrażenia tablicowego.

warunek – jest to wyrażenie typu *bool*, tylko elementy które spełniają dany warunek będą mogły być użyte do tworzenia elementów tablicy lub zestawu, każde *wyrażenie_tablicowe*, po którego wartościach iterowane są zmienne, może mieć dołączony warunek po słowie kluczowym *where*.

W omawianym wcześniej modelu przedstawiającym problem wydawania reszty można za pomocą generatora stworzyć tablicę nominałów waluty PLN. Generator może przyjąć następującą formę:

```
nominaly = [ i*j | i in [0.01, 0.1, 1, 10, 100], j in [1,2,5] ];
```

W tym przypadku każdy element tablicy powstaje poprzez pomnożenie wartości zmiennych *i* oraz *j*, Najszybciej zmienia się ostatnia zmienna z podanych. Oznacza to, że dla jednej wartości zmiennej *i* zmienna *j* przyjmie kolejno każdą ze swoich trzech wartości. Wynikiem powyższego zapisu będzie tablica z następującymi wartościami:

```
[0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 50.0,
100.0, 200.0, 500.0]
```

3.5. Złożone ograniczenia

W języku MiniZinc ograniczenia są zapisywane jako wyrażenia typu *bool*. Oznacza to, że w zapisie ograniczeń można używać tych samych operatorów co w wyrażeniach typu *bool*. Pozwala to na zapisywanie bardziej skomplikowanych ograniczeń. Dobrym przykładem może tutaj być omawiany program z wydawaniem reszty. Wcześniejsza wersja programu zawarta w pierwszym rozdziale przeprowadzała działania na liczbach całkowitych, więc w ograniczeniu wystarczyło przyrównać wymaganą kwotę do łącznej wartości monet przemnożonych przez ich wartości. Po zmianie typu *integer* na typ *float* konieczna jest zmiana tego ograniczenia, ponieważ używanie znaku równości do porównywania danych typu *float* bardzo często prowadzi do błędów w uzyskanych wynikach. W podstawowej wersji modelu ograniczenie wyglądało następująco:

```
constraint (g1*1+g2*2+g5*5+g10*10+g20*20+g50*50)=kwota;
```

W zmienionej wersji modelu wykorzystujemy tablice co oznacza, że aby przerobić to ograniczenie konieczne jest użycie wyrażenia wywołania generatora, które zwróci wymagany wynik. Wyrażenie wywołania generatora ma postać:

funkcja_agregująca ([*wyrażenie* | *generator*])

funkcja_agregująca – jest to identyfikator funkcji agregującej, która będzie użyta w danym wywołaniu generatora.

wyrażenie – przedstawia sposób w jaki tworzone będą poszczególne elementy dla funkcji agregującej.

generator – zawiera identyfikatory zmiennych oraz wyrażenie tablicowe, po którym będą te zmienne iterowane.

Używanie funkcji agregujących oraz wywołań generatorów są wręcz niezbędne przy pracy ze zmiennymi tablicowymi. Funkcje agregujące przyjmują jako argument tablicę jednowymiarową.

Dla tablic arytmetycznych dostępne są funkcje: *sum* – dodaje wszystkie elementy (dla pustej tablicy zwraca 0), *product* – mnoży wszystkie elementy (dla pustej

tablicy zwraca 1), *min* – znajduje najmniejszą wartość, *max* – znajduje największą wartość (dla pustej tablicy funkcje *min* oraz *max* zwracają błąd wykonania programu).

Funkcje przeznaczone dla tablic z wartościami *bool*: *forall* – zwraca ograniczenie, które jest logiczną koniunkcją zawartych ograniczeń (aby spełnić ograniczenie wszystkie ograniczenia muszą być spełnione), *exists* – zwraca ograniczenie będące logiczną alternatywą zawartych ograniczeń (aby spełnić ograniczenie wystarczy, że jedno z nich będzie spełnione), *xorall* – ograniczenie jest spełnione, kiedy nieparzysta liczba ograniczeń jest spełniona, *iffall* – ograniczenie jest spełnione, kiedy parzysta liczba ograniczeń jest spełniona.

W omawianym modelu przedstawiającym problem wydawania reszty policzenie łącznej kwoty użytych monet wymaga użycia wyrażenia wywołania generatora, które może wyglądać następująco:

```
sum([ ilosc[i] * nominally[i] | i in 1..rozmiar])
```

MiniZinc dopuszcza również inną formę zapisu dla wyrażeń wywołania generatora:

```
sum(i in 1..rozmiar) (ilosc[i] * nominaly[i])
```

Pamiętając o tym, że przy porównywaniu danych typu *float* nie należy stosować znaku równości należy przyjąć jakiś margines błędu. Dla omawianego przykładu może to być 1% wartości najmniejszego nominału:

```
float: blad = min(nominaly) / 100;
```

Następnie można zapisać finalną formę ograniczenia, które zapewni, że wyliczona kwota będzie należeć do przedziału otwartego (kwota-błąd ; kwota+błąd):

```
constraint sum(i in 1..rozmiar)(ilosc[i] * nominaly[i]) > (kwota-blad)
  /\
  sum(i in 1..rozmiar)(ilosc[i] * nominaly[i]) < (kwota + blad);
```

Zmienna decyzyjna przedstawiająca łączną sumę poszczególnych nominałów oraz element rozwiązania nie zawiera żadnych zmian w porównaniu z pierwotną wersją modelu:

```
var int: monety = sum(ilosc);
solve minimize monety;
```

Zmiana następuje w elemencie wyjścia, który należy zmodyfikować wykorzystując w nim używane tablice:

```
output ["Nominał - ilość\n"] ++
[if nominaly[i] < 1 then show_float(4, 2, nominaly[i])
else show_float(4, 0, nominaly[i]) endif ++
" = \(ilosc[i])\n" | i in 1..rozmiar]++
["Łącznie: \(monety)\n"];
```

Warto tutaj zwrócić uwagę na użycie generatora dla przedstawienia każdego nominału wraz z informacją ile sztuk danego nominału jest potrzebne aby wypłacić wymaganą kwotę. Zastosowanie wyrażenia warunkowego wewnątrz generatora pozwala na bardziej czytelne przedstawienie wyników poprzez modyfikację ilości wyświetlanych cyfr po przecinku.

Korzystając z przedstawionego wcześniej pliku danych otrzymany wynik będzie wyglądał następująco:

```
Nominał - ilość
0.01 = 1
0.02 = 1
0.05 = 1
0.10 = 1
0.20 = 1
0.50 = 1
   1 = 1
   2 = 1
   5 = 1
  10 = 1
  20 = 1
  50 = 1
 100 = 0
 200 = 0
 500 = 0
Łącznie: 12
```

Omawiając ograniczenia warto wspomnieć o funkcji *assert*, której można użyć np. do sprawdzania poprawności używanych plików danych. Funkcję wywołuje się następująco:

assert(*warunek*, *wiadomość*, *wyrażenie*);

warunek – wyrażenie typu *bool*

wiadomość – ciąg znaków, lub zmienna typu *string*

wyrażenie – wyrażenie, którego wynik zostanie zwrócony przez funkcję, jest to element opcjonalny

Funkcja działa w ten sposób, że sprawdza *warunek* i jeśli jest on spełniony zwraca wartość wynikającą z *wyrażenia*, w przeciwnym wypadku przerywa program zwracając błąd o treści *wiadomość*.

Wracając do omawianego programu z wydawaniem reszty, za pomocą funkcji *assert* można sprawdzić, czy podana kwota nie jest przypadkiem mniejsza lub równa 0, oraz, czy podane nominały mają wartości dodatnie.

```
constraint assert(kwota > 0, "Kwota musi być większa od 0");
constraint forall(i in 1..rozmiar) (assert(nominaly[i] > 0,
  "Nominał musi mieć wartość większą niż 0", true));
```

O ile w przypadku kwoty jest to proste ograniczenie, to w przypadku nominału jest potrzeba użycia funkcji agregującej, która sprawdzi, czy warunek jest spełniony dla każdego elementu tablicy, aby było to możliwe należy wpisać do listy argumentów funkcji *assert* wartość którą ma zwrócić, jeśli warunek jest spełniony, w tym wypadku jest to wartość *true*. Kiedy funkcja *assert* zwróci *true* dla każdego elementu tablicy, wtedy funkcja agregująca *forall* również zwróci wartość *true*, co będzie oznaczało, że ograniczenie zostało spełnione.

3.6. Ograniczenia globalne

Innym sposobem na tworzenie rozbudowanych ograniczeń jest użycie w modelu biblioteki zawierającej zestaw ograniczeń globalnych. Aby możliwe było użycie takich ograniczeń w modelu konieczne jest dołączenie pliku z danym ograniczeniem do tworzonego modelu, lub pliku ”globals.mzn”, który zawiera w sobie wszystkie ograniczenia globalne. W dokumentacji MiniZinc ograniczenia globalne są podzielone na 8 kategorii:

- All-Different oraz powiązane z nim ograniczenia

Najlepszym przykładem jest ograniczenie *all_different*, które zwraca *true* kiedy w tablicy podanej jako argument nie powtarza się żadna wartość.

- Ograniczenia leksykograficzne

Przykładem może być ograniczenie *lex_less*, które sprawdza czy pierwsza podana tablica ma mniejszą wartości leksykograficzną niż druga poprzez porównania kolejnych elementów tablic od pierwszego do ostatniego niezależnie od ich indeksów.

- Ograniczenia sortujące

Przykładowe ograniczenie to: *increasing*, które sprawdza, czy podana w argumencie tablica jest posortowana rosnąco (zezwala na zduplikowane wartości). W tej kategorii zawarta jest też funkcja, która zwraca permutację indeksów sortującą tablicę rosnąco.

- Ograniczenia łączące

Jednym z takich ograniczeń jest *int_set_channel*, jako argumenty przyjmuje tablicę zmiennych typu *int* oraz tablicę zawierającą zestawy zmiennych *int*. Ograniczenie wymaga, żeby indeks tablicy występował jako wartość w tablicy zestawów w zestawie o indeksie będącym wartością reprezentowaną przez rozważany indeks tablicy, matematycznie można to zapisać następująco: (**x**[*i*] = *j*) ↔ (*i* in **y**[*j*])

- Ograniczenia liczące

Za przykład może posłużyć ograniczenie *among*(*n, x, v*), gdzie *n* oznacza ilość powtórzeń, *x* jest tablicą wartości *int*, *v* jest zestawem wartości *int*. Ograniczenie jest spełnione kiedy wartości zawarte w zestawie *v* powtarzają się dokładnie *n* razy w tablicy *x*.

- Ograniczenia pakujące

Ciekawym ograniczeniem w tej kategorii jest *knapsack*(*w, p, x, W, P)*, gdzie *w* oznacza tablicę wag przedmiotów, *p* to tablica zysków, *x* ilość spakowanych przedmiotów z każdego typu, *W* suma wielkości wszystkich spakowanych przedmiotów, *P* suma zysków wszystkich spakowanych przedmiotów. Ograniczenie to wymaga aby przedmioty były spakowane w plecaku zgodnie z ich założonymi wagami oraz zyskami.

- Ograniczenia planujące

Przykładowym ograniczeniem jest *cumulative*(*s, d, r, b*), gdzie *s* to tablica czasów startowych dla zadań, *d* to tablica czasów trwania zadań, *r* to tablica wymaganych zasobów dla zadań, *b* jest górną granicą dla wykorzystanych zasobów. Ograniczenie jest spełnione kiedy zadania zaczynające się według czasów *s* trwających *d* i wymagających *r* zasobów nie przekraczają *b* wykorzystanych zasobów. Dodatkowo

tablica czasów startowych zawiera zmienne opcjonalne, dzięki czemu brakujące zadania nie są brane pod uwagę przy planowaniu.

- Ekstensjonalne ograniczenia

W tej kategorii za przykład może posłużyć ograniczenie *table*(*x*, *t*), gdzie *x* jest tablicą jednowymiarową, natomiast *t* to tablica dwuwymiarowa. Jest to przedstawienie ograniczenia *x* **in** *t*, gdzie *x* traktuje się jako krotkę, natomiast *t* jako zestaw krotek, ponieważ język MiniZinc nie posiada krotek są one kodowane za pomocą tablic.

3.7. Funkcje refleksji

Język MiniZinc posiada zestaw funkcji refleksyjnych przeznaczonych dla tablic oraz dla dziedzin wartości zmiennych. Tego typu funkcje przydają się podczas pisania bardziej ogólnych testów, predykatów oraz funkcji. Znajdują się w tej grupie funkcje zwracające indeksy tablic: *index_set* – zwraca indeksy tablicy jednowymiarowej w postaci zestawu, *index_set_XofY* – zwraca indeks konkretnego wymiaru z wielowymiarowej tablicy, *X* – odpowiada tutaj za konkretny wymiar, a *Y* – wskazuje ile wymiarów ma tablica (działa do maksymalnie sześciowymiarowych tablic).

Druga grupa to funkcje związane z domenami wartości zmiennych. Znajdują się tutaj między innymi takie funkcje jak: *dom* – która w postaci zestawu zwraca dziedzinę wartości zmiennej podanej jako argument, *dom_array* – która jako wynik zwraca zestaw będący sumą dziedzin wartości wszystkich elementów tablicy, wywołanie *dom* lub *dom_array*, kiedy jako argument poda się zmienną o typie, który nie jest skończony, spowoduje błąd wykonania funkcji. Funkcja *lb* zwraca wartość bliską lub równą dolnej granicy dziedziny wartości zmiennej, analogicznie działa funkcja *ub*, która zwraca wartość bliską lub równą górnej granicy dziedziny wartości zmiennej, odpowiednikami tych funkcji dla tablic są kolejno *lb_array* oraz *ub_array*.

3.8. Adnotacje przeszukiwania

Domyślnie język MiniZinc nie udostępnia możliwości zdefiniowania sposobu w jaki dany model ma zostać rozwiązany. Sposób na rozwiązanie modelu nie jest częścią tego modelu. MiniZinc pozwala jednak na przesyłanie informacji do solvera, który będzie rozwiązywał model, przy użyciu adnotacji. Podręcznik MiniZinc [21] przedstawia cztery adnotacje, które mówią solverowi w jaki sposób szukać rozwiązań

dla modelu zależnie od typu poszukiwanych zmiennych. MiniZinc nie jest w stanie zapewnić, że solver ma zaimplementowaną obsługę adnotacji wykorzystanych w rozwiązywanym modelu.

annotation anti_first_fail	Wybiera zmienną z największą domeną wartości
annotation dom_w_deg	Wybiera zmienną z największą domeną, podzieloną przez liczbę dołączonych ograniczeń ważonych na podstawie częstości powodowania niepowodzeń
annotation first_fail	Wybiera zmienną z najmniejszą domeną wartości
annotation impact	Wybiera zmienną, która ma największy wpływ na dotychczasowe przeszukiwanie
annotation input_order	Wybiera zmienne według kolejności ich wprowadzenia
annotation largest	Wybiera zmienną o największej wartości w jej domenie
annotation max_regret	Wybiera zmienną z największą różnicą pomiędzy dwiema najmniejszymi wartościami w swojej domenie
annotation most_constrained	Wybiera zmienną o najmniejszej domenie, w przypadku zmiennych o tej samej domenie decyduje ilość dołączonych do niej ograniczeń
annotation occurence	Wybiera zmienną z największą ilością dołączonych ograniczeń
annotation smallest	Wybiera zmienną o najmniejszej wartości w jej domenie

Tabela z adnotacjami strategii wyboru zmiennych. Źródło: podręcznik MiniZinc [21]

annotation indomain	Przypisuje wartości w kolejności rosnącej
annotation indomain_interval	Jeśli domena składa się z kilku ciągłych przedziałów wartości wybierany jest pierwszy przedział, w innym wypadku stosuje bisekcję na domenie
annotation indomain_max	Przypisuje największą wartość w domenie
annotation indomain_median	Przypisuje środkową wartość w domenie
annotation indomain_middle	Przypisuje wartość w domenie najbliższą średniej jej obecnych granic
annotation indomain_min	Przypisuje najmniejszą wartość w domenie
annotation indomain_random	Przypisuje losową wartość z domeny
annotation indomain_reverse_split	Stosuje bisekcję na domenie, wyłączając najpierw dolną połowę.
annotation indomain_split	Stosuje bisekcję na domenie, wyłączając najpierw górną połowę
annotation indomain_split_random	Stosuje bisekcję na domenie, losowo wybierając połowę do wyłączenia najpierw
annotation outdomain_max	Wyklucza największą wartość z domeny
annotation outdomain_median	Wyklucza środkową wartość z domeny
annotation outdomain_min	Wyklucza najmniejszą wartość z domeny
annotation outdomain_random	Wyklucza losową wartość z domeny

Tabela z adnotacjami strategii wyboru wartości. Źródło: podręcznik MiniZinc [21]

Jako argument adnotacji przeszukiwań używa się również adnotacji strategii przeszukiwań, tutaj dokumentacja przedstawia jedną adnotację: *annotation* complete – odpowiada za strategię pełnego przeszukiwania.

3.9. Adnotacje przeszukiwań dla typów *bool, int, float, set*

- Adnotacja dla typu *bool*

Deklaracja adnotacji:

bool_search(*array* [*int*] *of var bool*: **x**, *ann*: **select**, *ann*: **choice**, *ann*: **explore**)

x – tablica zmiennych typu *bool*, dla których określa się sposób poszukiwania rozwiązań modelu,
select – oznacza strategię jaką przyjmuje się podczas kolejności wyboru zmiennych,
choice – jest to sposób, który określa jak mają być wybierane wartości dla zmiennych,
explore – jest to sposób poszukiwań rozwiązania.

Adnotacja **choice** w adnotacji przeszukiwań dla typu *bool* może przyjąć jedynie wartości: *indomain_min*, *indomain_max*, *indomain_random*.

- Adnotacja dla typu *int*

Deklaracja adnotacji:

int_search(*array* [*int*] *of var int*: **x**, *ann*: **select**, *ann*: **choice**, *ann*: **explore**)

select, **choice**, **explore –** mają to samo znaczenie co w deklaracji adnotacji dal typu *bool*,
x – tablica zmiennych typu *int*, dla których określa się sposób przeszukiwań.

- Adnotacja dla typu *float*

Deklaracja adnotacji:

float_search(*array* [*int*] *of var float:* **x**, *float:* **prec**, *ann:* **select**, *ann:* **choice**, *ann*: **explore**)

select, **choice**, **explore** – jak wyżej,
prec – określa precyzję przeszukiwań dla zmiennych typu *float*,
x – tablica zmiennych typu *float*, dla których określa się strategię przeszukiwania.

Dla zmiennych typu *float* jako **choice** można wybrać jedynie wartości: *indomain_split*, *indomain_reverse_split* oraz *indomain_interval*.

- Adnotacja dla typu *set*

Deklaracja adnotacji:

set_search(*array*[*int*] *of var set of int*: **x**, *ann*: **select**, *ann*: **choice**, *ann*: **explore**)

select, **choice**, **explore** – tak samo, jak w poprzednich deklaracjach,
x – tablica zwierająca zmienne typu *set*, które zawierają elementy typu *int.*

Jako adnotację **choice** można tutaj używać tylko: *indomain_max*, *indomain_min*, *indomain_median*, *indomain_random*, oraz *oudomain_min*, *outdomain_max*, *outodmain_median*, *outdomain_random*.

- Adnotacja sekwencji przeszukiwań

Deklaracja adnotacji:

seq_search([*search_ann*, (…), *search_ann*])

search_ann – jest jedną z czterech wcześniej przedstawionych adnotacji przeszukiwań.

Adnotacje w modelu nie mają ustalonej kolejności. Dzięki adnotacji *seq_search* można wymusić kolejność dla sposobów przeszukiwania. Przykład użycia:

```
ann: search_strategy = seq_search([
    int_search(x, first_fail, indomain_min, complete),
    set_search(y, input_order, indmain_max, complete)
]);
solve ::search_strategy satisfy;
```

4. Zintegrowane środowisko programistyczne

Zintegrowane środowisko programistyczne, w skrócie IDE (ang. *Integrated Development Environment*), jest programem, który zawiera w sobie narzędzia potrzebne programiście do tworzenia oprogramowania. W skład zintegrowanego środowiska programistycznego mogą wchodzić takie komponenty jak np. edytor kodu źródłowego, kompilator, debugger, narzędzia do generowania interfejsu użytkownika, czy nawet system kontroli wersji.

MiniZinc IDE jest prostym zintegrowanym środowiskiem programistycznym przeznaczonym dla języka MiniZinc. Zawiera takie elementy jak prosty edytor tekstu z podświetlaniem składni języka MiniZinc, kompilator, który tłumaczy model MiniZinc do języka FlatZinc oraz pozwala na uruchamianie solverów dla skompilowanych modeli.

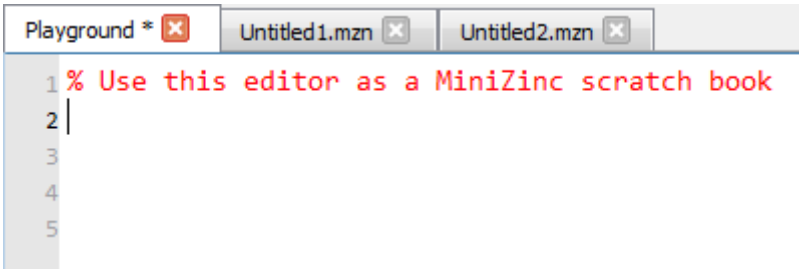
4.1. Instalacja MiniZinc w systemie Linux

Rekomendowany sposób instalacji systemu MiniZinc to użycie dołączonych pakietów binarnych, które można pobrać z oficjalnej strony języka MiniZinc (www.minizinc.org). Archiwum zawiera gotowy do użycia system MiniZinc. Po ściągnięciu pliku należy go rozpakować. Z poziomu folderu otrzymanego poprzez rozpakowanie archiwum można korzystać z narzędzi MiniZinc. W celu wygodniejszego dostępu do narzędzi MiniZinc można dodać ścieżkę folderu, który zawiera pliki wykonywalne, do zmiennej środowiskowej *PATH*. MiniZinc IDE należy uruchamiać używając skryptu *MiniZincIDE.sh*, który ustawia zestaw ścieżek, których MiniZinc IDE potrzebuje do poprawnego działania. Wśród zawartości folderu z systemem MiniZinc należy zwrócić uwagę na folder *bin*, który zawiera pliki wykonywalne oraz folder *share/minizinc*, w którym zawarte są implementacje ograniczeń globalnych dla poszczególnych solverów.

4.2. Edytowanie plików

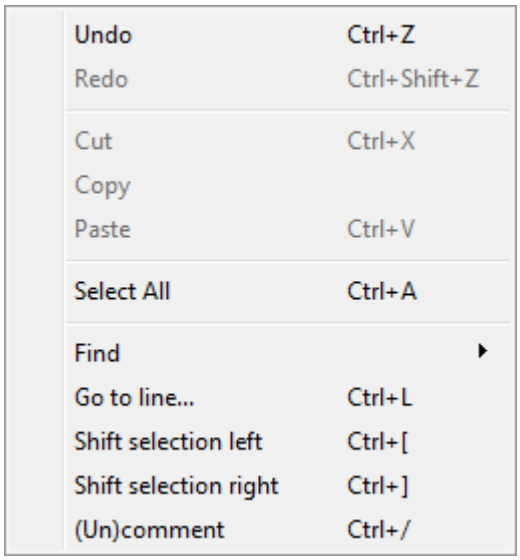
W MiniZinc IDE pliki można edytować za pomocą wbudowanego prostego edytora kodu. Edytor służy do tworzenia plików z dwoma rozszerzeniami: *.mzn* – przeznaczonym dla modeli MiniZinc oraz *.dzn* – który oznacza plik z danymi. Rozszerzenie z jakim zapisuje się plik ma znaczenie, ponieważ MiniZinc IDE

umożliwia uruchomienie jedynie plików z rozszerzeniem *.mzn*. Jeśli otwarte jest jednocześnie wiele plików można przełączać widok między nimi przy użyciu zakładek widocznych nad oknem edycji pliku, lub przy użyciu opcji *Previous tab/Next tab* widocznych w menu *View*. Po uruchomieniu MiniZinc IDE jest dostępna zakładka *Playground*, która pozwala na szybkie uruchamianie modeli bez konieczności zapisu pliku na dysku.



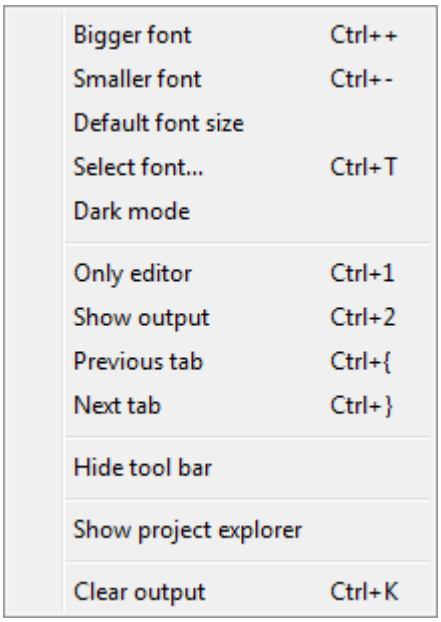
Przedstawienie zakładek widocznych, kiedy otwartych jest wiele plików.

Funkcje dostępne w edytorze można zobaczyć w menu *Edit*. Są tutaj podstawowe działania jak: wytnij (ang, *cut*), kopiuj (ang. *copy*), wklej (ang. *paste*), jednak dostępne są również bardziej zaawansowane funkcje.



Widok menu Edit w MiniZinc IDE

Za pomocą funkcji *Find* można wyszukać w kodzie określoną frazę. *Go to line* pozwala na przeniesienie kursora na podaną linię w otwartym pliku. *Shift selection left/Shift selection right* pozwala na zmniejszenie bądź zwiększenie wcięcia zaznaczonego tekstu (tabulacja w edytorze odpowiada dwóm znakom spacji). Ostatnia opcja w menu *Edit* to *(Un)comment*, która pozwala na zakomentowanie lub odkomentowanie zaznaczonego tekstu (znak komentarza w MiniZinc to ”%”).



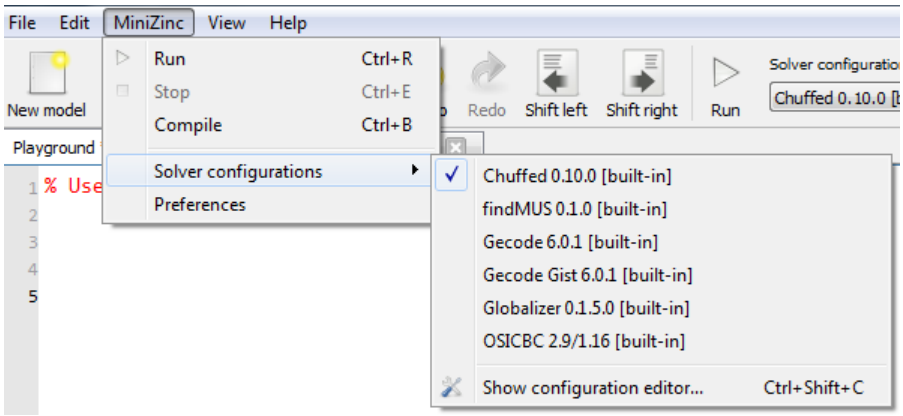
Widok menu View w MiniZinc IDE

Drugim menu, w którym można znaleźć opcje dotyczące edytora jest *View*. Za pomocą funkcji tego menu można zwiększyć, zmniejszyć lub przywrócić domyślny rozmiar używanej czcionki (*Bigger font/Smaller font/Default font size*).

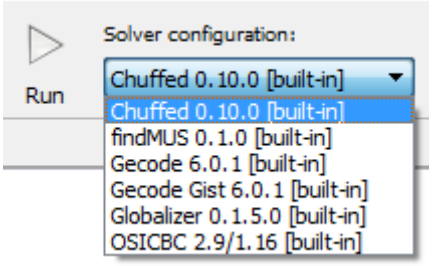
Zmiany używanej czcionki można dokonać w oknie dialogowym wywoływanym przez *Select font*. Edytor posiada również tryb ciemny (*Dark mode*), który zmienia schemat kolorów: dla tła ustawia czarny, natomiast czcionka jest koloru białego.

4.3. Wybór i zastosowania solverów

Solvery Dostępne dla MiniZinc IDE można zobaczyć w menu *MiniZinc*, po rozwinięciu *Solver configurations* widoczna jest lista dostępnych solverów. Solver zaznaczony na tej liście będzie używany do rozwiązania modelu po jego uruchomieniu.



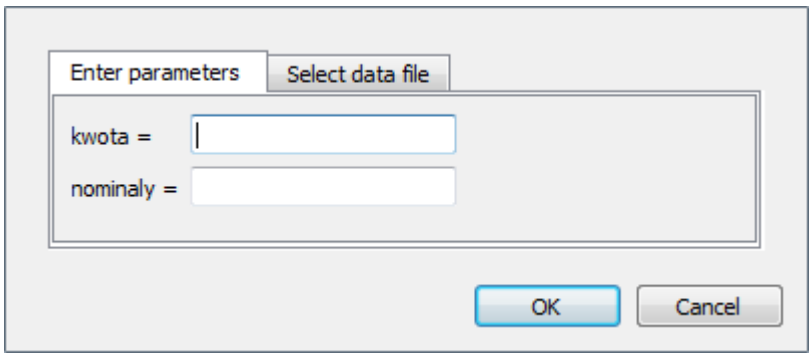
Lista dostępnych solverów, widziana z poziomu menu MiniZinc.



Lista solverów dostępna z poziomu paska narzędzi

Obecnie otwarty w edytorze model MiniZinc można uruchomić na trzy sposoby: poprzez naciśniecie ikony *Run* na pasku narzędzi MiniZinc IDE, wybranie *Run* w menu *MiniZinc* lub używając kombinacji klawiszy Ctrl+R. Podobnie jest z przerwaniem rozwiązywania modelu: przycisk *Stop* na pasku narzędzi, wybranie *Stop* w menu *MiniZinc* lub użycie skrótu klawiszowego Ctrl+E. Po uruchomieniu model najpierw jest kompilowany, a następnie rozwiązywany przez solver. Model jest rozwiązywany przy użyciu solvera wybranego z listy *Solver configuratins*. Uruchomienie modelu spowoduje wyświetlenie okna wyjścia jeśli jest ono wyłączone. Jest to okno dokowalne, które domyślnie jest umiejscowione na dole głównego okna MiniZinc IDE.

W przypadku kiedy uruchomiony zostanie model bez przypisanych wartości do zmiennych parametrycznych, a nie użyto żadnego pliku z danymi, który by przypisał wartości zmiennym, zostanie wyświetlone okno dialogowe widoczne poniżej.



Okno dialogowe pozwalające zainicjalizować wartości parametryczne modelu.

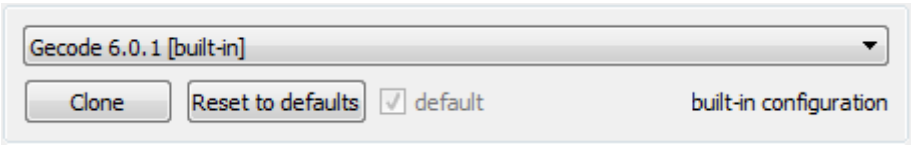
W oknie dialogowym widoczne są zmienne będące parametrami bez zainicjalizowanych wartości. Można te wartości uzupełnić ręcznie poprzez użycie odpowiednich literałów dla zmiennych lub wybrać jeden z otwartych plików z danymi.

Możliwe jest skompilowanie pliku modelu bez uruchamiania go. Do tej operacji służy *Compile* w menu *MiniZinc*. Utworzony w ten sposób kod FlatZinc zostanie

otworzony w edytorze jako nowa zakładka. Po zapisaniu takiego pliku z rozszerzeniem *.fzn* można go później uruchomić bez kompilacji.

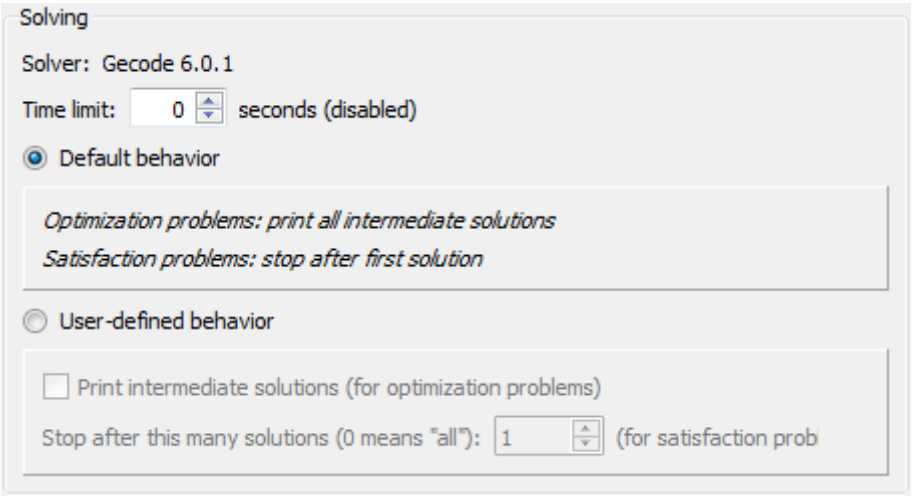
4.4. Konfiguracja solvera

Możliwa jest konfiguracja solvera, który jest wykorzystywany przez MiniZinc IDE. Ustawienia te są dostępne w oknie *Configuration*, które można wyświetlić za pomocą przycisku *Show configuration editor* na pasku narzędzi, lub z poziomu menu *MiniZinc* po rozwinięciu *Solver configurations*, gdzie pod listą dostępnych solverów znajduje się opcja *Show configuration editor*. Jest to okno dokowalne, domyślnie umieszczone przy prawej krawędzi MiniZinc IDE.



Pierwsza sekcja okna „Configuration”.

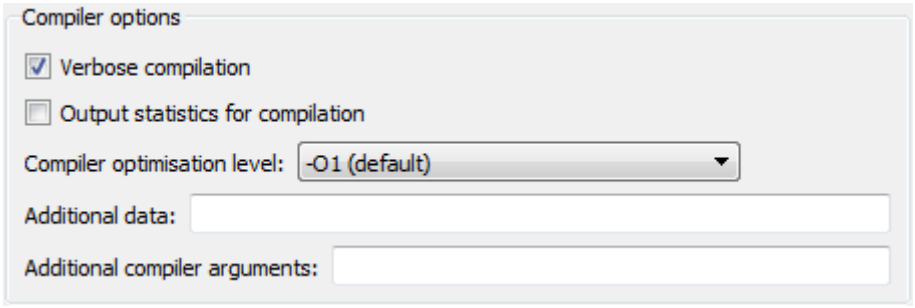
Okno *Configuration* jest podzielone na kilka sekcji. Pierwszą z nich stanowi rozwijana lista z dostępnymi dla MiniZinc IDE solverami. Z listy wybierany jest solver, dla którego będą zmieniane ustawienia. W przypadku solverów oznaczonych jako [*built-in*] zmiany ustawień solvera będą zapomniane po wyłączeniu MiniZinc IDE. Przycisk *Clone* pozwala na sklonowanie obecnej konfiguracji oraz zapisanie jej pod inną nazwą. Sklonowana konfiguracja jest zapamiętywana jako część projektu (projekty są omówione w sekcji 4 rozdziału 4).



Sekcja okna „Configuration” przeznaczona ustawieniom zachowania solvera podczas rozwiązywania modelu.

Druga sekcja dotyczy zachowania solvera podczas rozwiązywania modelu. *Solver*, oznacza solver, dla którego są zmieniane ustawienia. Pole opisane jako *Time limit* odnosi się do czasu, po upływie którego solver powinien przerwać działanie, jeśli nie zdążył wcześniej rozwiązać modelu, wpisanie liczby ”0” oznacza brak limitu czasowego dla solvera.

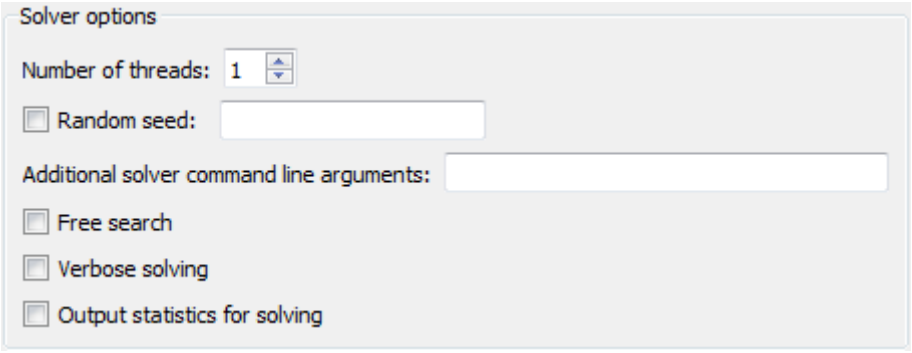
Poniżej do wyboru jest sposób zachowania się solvera podczas rozwiązywania modelu. Standardowo ustawione jest domyślne zachowanie dla solvera, którego nie można zmieniać. Polega ono na wyświetlaniu wszystkich kolejnych wyników dla problemów optymalizacyjnych oraz na zwróceniu tylko pierwszego wyniku dla problemów satysfakcji ograniczeń. Użytkownik może jednak zdefiniować jak powinien się zachowywać solver dzięki opcji *User-defined behavior*. W przypadku kiedy użytkownik definiuje zachowanie solvera może zaznaczyć, aby solver zwracał wszystkie kolejne rozwiązania kiedy rozwiązywany jest problem optymalizacyjny, w przeciwnym wypadku wyświetlony zostanie jedynie ostateczny wynik optymalizacji. Druga opcja dotyczy ilości rozwiązań, które solver ma zwrócić w przypadku problemu satysfakcji ograniczeń, domyślnie jest to jedno rozwiązanie, wpisanie ”0” spowoduje wydrukowanie wszystkich możliwych rozwiązań.



Ustawienia dla kompilatora w oknie „Configuration”

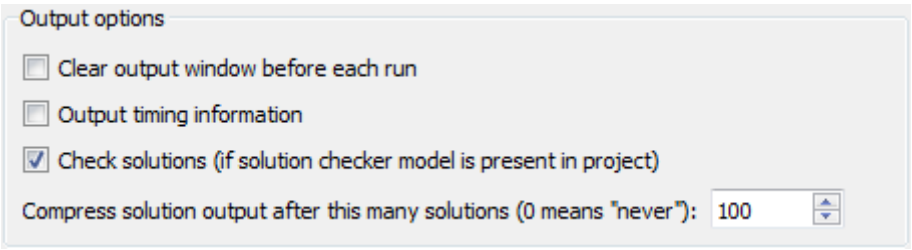
Trzecią sekcję okna *Configuration* stanowią ustawienia kompilatora. Pierwsza opcja *Verbose compilation* odpowiada za wyświetlenie w oknie wyjścia w jaki sposób została przeprowadzona kompilacja (obejmuje to np. użyte komendy oraz pliki wzięte pod uwagę podczas kompilacji). Opcja *Output statistics for compilation* powoduje zwrócenie na wyjście statystyk dotyczących przeprowadzonej kompilacji np. ilość oraz typy zmiennych, czy informacja o typie problemu (satysfakcja ograniczeń, lub optymalizacja). Z listy rozwijanej można wybrać poziom optymalizacji kodu FlatZinc przetłumaczonego przez kompilator, wyższy poziom optymalizacji powoduje wydłużenie procesu kompilacji. Pole *Additional data* pozwala na wprowadzenie danych

do modelu, pole jest traktowane jakby było treścią pliku danych i w taki sposób powinno być wypełniane. Pole *Additional compiler arguments* przeznaczone jest dla argumentów, które będą użyte przy wywołaniu kompilatora, należy je podać w takiej postaci, w jakiej byłyby wpisane w linii poleceń.



Opcje solvera w oknie „Configuration”

Sekcja czwarta okna *Configuration* odpowiada za opcje konfiguracyjne dla wybranego z listy solvera. Opcje, których solver nie obsługuje będą niedostępne. Opcja *Number of threads* oznacza ilość wątków, których solver ma użyć do rozwiązania modelu, *Random seed* – umożliwia ręczne ustawienie losowego ziarna dla generatorów liczb losowych. Pole *Additional solver command line arguments* pozwala na wprowadzenie argumentów, które zostaną użyte przy wywołaniu solvera. *Free search* pozwala solverowi na ignorowanie adnotacji dotyczących przeszukiwania, jednak nie wymaga ignorowania tych adnotacji. Opcja *Verbose solving* powoduje drukowanie logów z rozwiązywania modelu na domyślny strumień błędów. *Output statistics for solving* powoduje wyświetlenie w oknie wyjścia statystyk dotyczących rozwiązywania modelu przez solver, takie jak np. czas działania solvera, ilość znalezionych rozwiązań, liczba zmiennych.



Sekcja przeznaczona dla ustawień wyjścia w oknie „Configuration”

Ostatnia sekcja okna *Configuration* odpowiada za ustawienia dotyczące okna wyjścia. *Clear output window before each run* powoduje wyczyszczenie okna wyjścia przed każdym uruchomieniem modelu. Kiedy zaznaczona jest opcja *Output timing*

information po każdym wyniku zwróconym na wyjście zostaje dopisana informacja o czasie, który upłynął od uruchomienia modelu.

Domyślnie zaznaczona opcja *Check solutions* odpowiada za uruchomienie modelu, który ma za zadanie sprawdzić, czy wyniki otrzymane po rozwiązaniu modelu spełniają określone wymagania. Jeśli model jest zapisany jako *model.mzn,* wtedy model, który ma dokonać sprawdzenia wyników powinien mieć nazwę *model.mzc,* lub *model.mzc.mzn*. Może to być wykorzystane do sprawdzania, podczas rozwijania modelu, czy jego rozwiązania w dalszym ciągu spełniają założenia problemu.

Opcja *Compress solution output after this many solutions* odpowiada za skompresowanie wyświetlanych na wyjściu wyników w przypadku kiedy jest ich zbyt dużo. Liczba wpisana w to pole oznacza ile wyników ma zostać wyświetlonych przed rozpoczęciem kompresji kolejnych. Prosty model prezentujący tą opcję:

```
var 1..100: x;

solve satisfy;
constraint x > 0;

output["x = \(x)\n"];
```

Ten prosty model ma 100 możliwych rozwiązań, w przypadku kiedy opcja kompresji wyjścia zostanie ustawiona na 10 rozwiązań, a solver będzie miał zwrócić wszystkie możliwe rozwiązania dla problemów satysfakcji ograniczeń, końcowe wyniki będą widoczne w pokazanej poniżej postaci.

```
x = 10
----------
x = 11
----------
[ 9 more solutions ]
x = 21
----------
[ 19 more solutions ]
x = 41
----------
[ 39 more solutions ]
x = 81
----------
[ 18 more solutions ]
x = 100
----------
==========
```

Kiedy solver przekroczy liczbę 10 zwróconych rozwiązań kolejne 10 jest skompresowane, każda kolejna kompresja jest 2 razy większa od poprzedniej, z każdej kompresji wyświetlany jest ostatni wynik.

4.5. Korzystanie z projektów

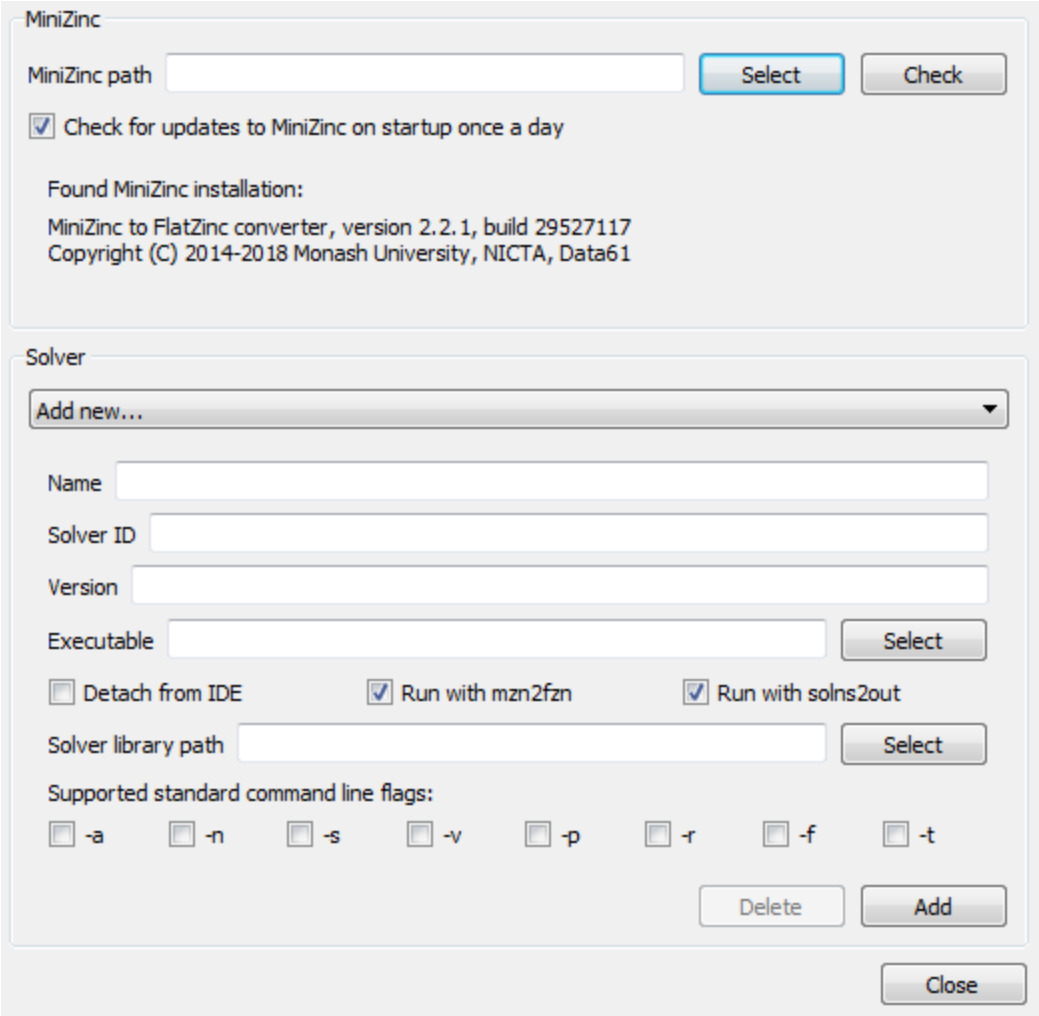
MiniZinc IDE pozwala na organizację pracy nad modelem poprzez wykorzystanie plików projektu. Projekt MiniZinc jest zapisywany z rozszerzeniem *.mzp*. Elementy projektu można podglądać w oknie *Project*, które włącza się przyciskiem *Show project explorer* z poziomu paska narzędzi lub przez menu *View*.

Obecnie otwarty w edytorze kodu model można uruchomić z użyciem pliku danych zawartych w projekcie poprzez kliknięcie na pożądanym pliku danych prawym przyciskiem myszy i wybranie *Run model with this data*.

Projekt zawiera informacje takie jak: ścieżki do każdego z dołączonych plików (w odniesieniu do pliku projektu), które oraz w jakiej kolejności pliki były otwarte w edytorze, jaki solver był ustawiony jako domyślny oraz sklonowane konfiguracje solverów. Do danych projektu nie należy zawartość okna wyjścia oraz konfiguracje solverów oznaczonych jako [*built-in*].

4.6. Dodanie nowego solvera do MiniZinc IDE

MiniZinc IDE posiada możliwość dodania nowego solvera. W tym celu należy wybrać *Preferences* w menu *MiniZinc*. Pierwszą część okna stanowią informację dotyczące zainstalowanej wersji MiniZinc IDE. Pole oznaczone jako *MiniZinc path* powinno zawierać bezwzględną ścieżkę do pliku wykonywalnego *minizinc*, w przypadku kiedy MiniZinc IDE było ściągnięte z oficjalnej strony języka MiniZinc z dołączonymi pakietami binarnymi ustawianie tej ścieżki nie jest konieczne. Druga część okna jest poświęcona dla konfiguracji solverów, które będą mogły być wykorzystane przez MiniZinc IDE. Wybranie opcji *Add new* z rozwijanej listy solverów pozwoli na uzupełnienie danych, których potrzebuje MiniZinc IDE, aby móc korzystać z solvera.



Okno dialogowe Preferences, dodawanie nowego solvera do MiniZinc IDE.

Pierwsze pole oznaczone jako *Name* odpowiada za nazwę, którą będzie identyfikowany dodawany solver. Pole *Solver ID* jest unikalnym identyfikatorem solvera, najczęściej zapisywany w odwróconej notacji nazwy domeny. *Version* odpowiada ciągowi znaków oznaczającemu wersję solvera. Jako *Executable* należy podać bezwzględną ścieżkę do pliku wykonywalnego solvera. Następne trzy opcje odpowiadają za obsługę solvera przez MiniZinc IDE: *Detach from IDE* – opcja ta jest wykorzystywana kiedy solver posiada własny interfejs użytkownika, uruchamiany jest wtedy w oddzielnym oknie, opcji *Run with mzn2fzn* należy użyć wtedy, kiedy solver wymaga kompilacji modelu do języka FlatZinc, *Run with solns2out* oznacza, że wyniki zwrócone przez solver zostaną sformatowane tak, aby pasowały do elementu wyjścia modelu, w przypadku kiedy solver używa natywnie modeli MiniZinc opcje *Run with mzn2fzn* oraz *Run with solns2out* powinny zostać wyłączone. Pole *Solver library path* pozwala na wskazanie ścieżki do implementacji bibliotek MiniZinc dla dodawanego

solvera, w przypadku kiedy to pole jest puste używane są standardowe biblioteki MiniZinc.

Supported standard command line flags pozwala na zaznaczenie, które ze standardowych argumentów są obsługiwane przez solver. Standardowe argumenty, które mogą być obsługiwane przez solvery:

-a – solver zwraca wszystkie wyniki niezależnie, czy rozwiązuje problem optymalizacyjny, czy satysfakcji ograniczeń,

-n <i> – solver powinien przerwać pracę po zwróceniu *i* wyników problemu satysfakcji ograniczeń,

-s – zwraca statystyki rozwiązywania modelu,

-v – drukuje logi dotyczące rozwiązywania modelu na standardowy strumień błędów, w przypadku kiedy solver chce wyświetlać te logi na standardowe wyjście wszystkie wiadomości muszą być komentarzami (muszą się zaczynać od znaku ”%”),

-p <i> – uruchamia *i* równoległych wątków solvera (jeśli solver obsługuje wielowątkowość),

-r <i> – używa *i* jako losowego ziarna dla generatorów liczb losowych używanych przez solver,

-f – pozwala solverowi zignorować wszystkie adnotacje przeszukiwań (nie musi ich ignorować),

-t <i> – określa po jakim czasie solver ma przerwać działanie, nawet jeśli nie znalazł wyniku.

Do przedstawienia sposobu dodania nowego solvera użyto JaCoP, przykład wykonywany jest na systemie Linux. JaCoP jest solverem dla programowania z ograniczeniami bazujący na języku Java. Można go użyć do rozwiązywania modeli MiniZinc, ponieważ posiada interfejs dla języka FlatZinc, do którego są kompilowane modele MiniZinc.

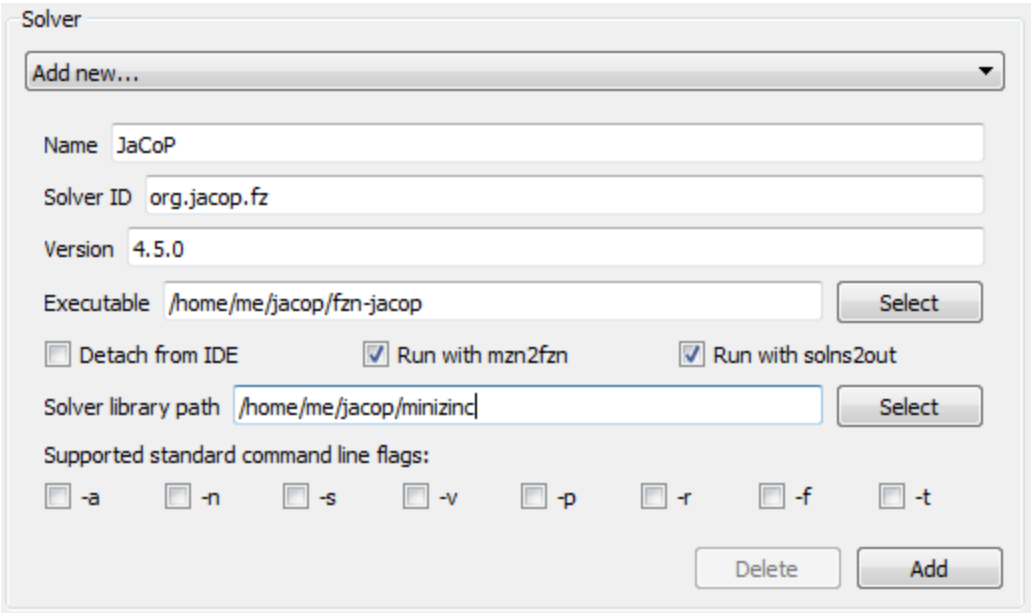
W polu *Name* jest nazwa, pod którą będzie widoczny solver na liście dostępnych solverów: *JaCoP*. W polu *Solver ID* wpisano: *org.jacop.fz*. W następnym polu należy podać ścieżkę do pliku wykonywalnego. Plik solvera jest plikiem o rozszerzeniu .jar,

aby umożliwić MiniZinc IDE uruchomienie solvera można stworzyć skrypt powłoki. Przykładowa zawartość pliku skryptu powłoki:

```
#!/bin/bash
exec java -cp /home/me/jacop.jar org.jacop.fz.Fz2jacop "$@"
```

Wykonanie tego skryptu spowoduje uruchomienie pliku *jacop.jar* przy użyciu *org.jacop.fz.Fz2jacop* jako klasy głównej. Ciąg znaków ”$@” na końcu oznacza listę argumentów użytych przy wywołaniu skryptu, w ten sposób kiedy MiniZinc IDE doda argumenty podczas wywołania skryptu powłoki zostaną one przekazane do komendy uruchamiającej solver. Plik należy zapisać np. pod nazwą *fzn-jacop*. Kiedy plik jest gotowy należy w polu *Executable* podać jego ścieżkę: */home/me/jacop/fzn-jacop*.

Solver JaCoP działa na modelach w języku FlatZinc, więc konieczne jest zaznaczenie opcji *Run with mzn2fzn* oraz *Run with solns2out*. Opcja *Detach from IDE* pozostaje w przypadku tego solvera wyłączona. Następnym polem jest *Solver library path*. Solver JaCoP posiada własne implementacje dla bibliotek MiniZinc. Są one dostępne do pobrania na platformie SourceForge [23] oraz w repozytorium na platformie GitHub pod ścieżką: src/minizinc/jacop/org/jacop/minizinc/ [24]. Folder zawierający implementacje bibliotek należy zapisać na dysku a następnie ścieżkę bezwzględną do tego pliku wpisać w pole *Solver library path*.



Dodanie nowego solvera do MiniZinc IDE.

Po uzupełnieniu koniecznych pól oraz kliknięciu *Add* nowy solver będzie widoczny w rozwijanej liście razem z pozostałymi solverami, w przypadku kiedy nie będą zaznaczone argumenty, które są obsługiwane przez solver część opcji konfiguracyjnych dotyczących solvera będzie niedostępna.

4.7. MiniZinc w linii poleceń

MiniZinc IDE nie jest jedynym sposobem na użycie narzędzia *minizinc*, można go również używać z poziomu linii poleceń. W przypadku kiedy folder zawierający narzędzie *minizinc* jest dodany do zmiennej środowiskowej w systemie można je wywołać za pomocą komendy w postaci:

```
minizinc [<options>] <model>.mzn [<data>.dzn …]
```

Do prezentacji sposobu użycia narzędzia *minizinc* z poziomu linii poleceń można wykorzystać prosty model MiniZinc:

```
include "alldifferent.mzn";
array[1..5] of var 1..10: tablica;
constraint alldifferent(tablica);
solve minimize sum(tablica);
output["Wynik minimalizacji: \(tablica)\n"];
```

Wywołanie narzędzia *minizinc* dla tego modelu może wyglądać następująco:

```
minizinc --solver JaCoP model.mzn
```

Przy takim wywołaniu *minizinc* wykorzysta solver o nazwie *JaCoP* do rozwiązania modelu, który wcześniej skompiluje do języka FlatZinc przy użyciu *mzn2fzn*. Wyniki rozwiązania modelu zostaną zwrócone przy użyciu *solns2out*. Kiedy narzędzie *minizinc* zakończy pracę zostanie zwrócony wynik w postaci:

```
Wynik minimalizacji: [1, 2, 3, 4, 5]
----------
==========
```

Istnieje również możliwość korzystania niezależnie spośród każdego z narzędzi wykorzystywanych przez *minizinc*. W pierwszej kolejności należy skompilować model napisany w języku MiniZinc. Aby narzędzie *minizinc* przeprowadziło tylko kompilację należy użyć argumentu ”-c”:

```
minizinc -c JaCoP model.mzn
```

Po wykonaniu tej operacji w folderze z modelem zostaną utworzone dwa nowe pliki: *model.fzn* – jest to model skompilowany do języka FlatZinc oraz *model.ozn* – który jest plikiem definiującym sposób formatowania wyników zwracanych przez model. Następnie należy użyć narzędzia *minizinc* podając jako model skompilowany plik *model.fzn*:

```
minizinc –-solver JaCoP model.fzn
```

Jednak takie wywołanie zwróci wyniki w standardowej formie bez formatowania użytego w modelu MiniZinc:

```
tablica = array1d(1..5,[1, 2, 3, 4, 5]);
----------
==========
```

W przypadku kiedy pożądane jest zwrócenie wyniku z uwzględnieniem formatowania wyjścia zawartego w modelu MiniZinc należy przekierować wyniki rozwiązania do narzędzia *minizinc* z odpowiednim argumentem:

```
minizinc –-solver JaCoP model.fzn | minizinc --ozn-file model.ozn
```

Przy wywołaniu narzędzia *minizinc* w ten sposób wyniki zostaną zwrócone przy wykorzystaniu pliku z formatowaniem wyjścia modelu (plik *.ozn*). Wyniki zostaną zwrócone w następujący sposób:

```
Wynik minimalizacji: [1, 2, 3, 4, 5]
----------
==========
```

5. Przykładowe modele w języku MiniZinc, ciekawe strony internetowe powiązane z MiniZinc

5.1. Problem mostu i pochodni

Pierwszym omawianym przykładem jest zamodelowany w języku MiniZinc problem mostu i pochodni. Problem przedstawiony jest następująco: Czworo ludzi idąc w nocy natrafiło na rzekę. Jedyną drogą na drugą stronę tejże rzeki jest cienki most, przez który mogą przejść jednocześnie tylko dwie osoby. Grupa posiada tylko jedną pochodnię, a ponieważ jest noc musi ona być używana podczas przechodzenia przez most. Każda z osób pokonuje most w określonym czasie: minutę, dwie minuty, pięć minut i osiem minut. Zwykle podawane są też ”role osób”, np. sportowiec, dziadek. Para, która przechodzi przez most dostosowuje prędkość do wolniejszej osoby. Pytanie brzmi, czy są wstanie przejść na drugą stronę mostu w ciągu piętnastu minut, nie przekraczając go więcej niż sześć razy. Przedstawiony poniżej model jest autorstwa Hakan Kjellerstrand (*www.hakank.org*):

```
% problem mostu i pochodni - problem polega na przekroczeniu mostu
% przez wszystkie osoby w możliwie najmniejszym czasie
% most_i_pochodnia.mzn - Hakan Kjellerstrand
int: num_persons;
int: max_time;
int: max_num_to_cross;
int: A = 1;
int: B = 2;
array[1..num_persons] of int: cross_time;

array[1..max_time, 1..num_persons] of var A..B: actions;
array[1..max_time] of var 0..sum(cross_time): times;
array[1..max_time] of var A..B: torch_place;
var 1..max_time: total_steps;
var int: total = sum(t in 1..max_time) ( times[t]*bool2int(t <=
total_steps));
array[1..max_time] of var set of 1..num_persons: transfered;

solve ::int_search(
        [actions[i,j] | i in 1..max_time, j in 1..num_persons] ++
        times ++
        torch_place ++
        [total_steps, total],
        occurrence,
        indomain_min,
        complete)
     minimize total_steps;
```

```
constraint

   forall(i in 1..num_persons) (
      actions[1,i] = A
   )
   /\
   torch_place[1] = A
   /\
   transfered[1] = 1..num_persons

   /\
   forall(t in 2..max_time) (
     exists(place in A..B) (
       torch_place[t-1] = place
       /\
       torch_place[t] != torch_place[t-1]
       /\
       let {
           var 1..max_num_to_cross: num_transfered =
card(transfered[t])
       }
       in
       times[t] = max(i in 1..num_persons) (
                     cross_time[i]*bool2int(
                              i in transfered[t]
                           )
                  )
       /\
       forall(i in 1..num_persons) (
          ((i in transfered[t]) <-> (
                                      actions[t-1,i] = place /\
                                      actions[t-1,i] != actions[t,i]
                                   ))
          /\
          (not(i in transfered[t]) <-> actions[t-1,i] = actions[t,i])
       )
     )

   )

   /\
   exists(t in 2..max_time) (
      forall(i in 1..num_persons) (
          actions[t, i] = B
      )
      /\
      torch_place[t] = B
      /\
      total_steps = t

   )
;
```

```
output
[
   "total_steps: " ++ show(total_steps) ++ "\n" ++
   "total: " ++ show(total)
]
++
[
  if p = 1 /\ t <= fix(total_steps) then "\n" else " " endif ++
    if t <= fix(total_steps) then
       show(actions[t,p]) ++
       if p = num_persons
            then " torch: " ++ show(torch_place[t]) ++ " transfered: "
++ show(transfered[t])
       else "" endif
    else "" endif
  | t in 1..max_time, p in 1..num_persons
]
;
```

Model MiniZinc autorstwa Hakan Kjellerstrand.
Link do modelu: www.hakank.org/minizinc/bridge_and_torch_problem.mzn

Model rozpoczyna się od zadeklarowania zmiennych, które będą używane podczas jego rozwiązywania. Pierwsza część zmiennych to parametry modelu, na ich podstawie tworzone są pozostałe zmienne. Widoczne są tutaj następujące zmienne parametryczne: *num_persons* – oznacza liczbę osób, która zamierza przekroczyć most, *max_time* – oznacza maksymalną ilość kroków, które można wykonać aby ukończyć zadanie, *max_num_to_cross* – oznacza liczbę osób, która może znajdować się na moście jednocześnie, *A* – oznacza miejsce startowe, *B* – oznacza miejsce docelowe, tablica *cross_time* – przechowuje czas potrzebny na przekroczenie mostu każdej z osób.

Druga część deklarowanych zmiennych to zmienne decyzyjne: tablica dwuwymiarowa *actions* – pokazuje, gdzie znajduje się każda osoba w kolejnych przekroczeniach mostu, tablica *times* – zawiera informacje o czasie potrzebnym na wykonanie kolejnych kroków, tablica *torch_place* – pokazuje, po której stronie mostu jest pochodnia w każdym kroku, *total_steps* – liczba oznaczająca liczbę kroków potrzebną do przekroczenia mostu przez wszystkie osoby, *total* – oznacza łączny czas, który jest potrzebny na przekroczenie mostu, tablica *transfered* – jest tablicą, której elementami są zbiory oznaczające osoby, które przekraczają most w danym kroku.

Następnie w modelu jest element rozwiązania, który ma dodaną adnotację przeszukiwania. Pierwszym argumentem adnotacji jest tablica zawierające zmienne decyzyjne, następnie *occurence* – oznacza, że przeszukiwanie zaczyna się od

zmiennych z największą ilością dołączonych ograniczeń, *indomain_min* – oznacza, że najpierw przypisywane są najmniejsze wartości w domenie zmiennych, ostatni argument, *complete*, oznacza, że ma zostać wykonane pełne przeszukiwanie. W tym przypadku zadaniem solvera jest minimalizacja zmiennej *total_steps* oznaczającej ilość kroków potrzebną do przekroczenia mostu przez wszystkie osoby, w przypadku jeśli pożądana jest minimalizacja czasu potrzebnego na przejście mostu należy zamienić *minimize total_steps* na *minimize total*.

Następną cześć modelu stanowią ograniczenia nakładane na zmienne, które gwarantują, że otrzymane rozwiązanie będzie spełniać założenia problemu. Na początku są nałożone ograniczenia opisujące początkowy stan: pierwszy wiersz tablicy *actions* wypełniony jest zmienną *A*, co oznacza, że wszyscy są na początku mostu, pierwsza wartość w tablicy *torch_place* również jest równa *A*, ponieważ pochodnia jest trzymana przez kogoś na początku mostu, natomiast pierwszy wiersz tablicy *transferred* oznacza, że wszyscy zostali „przesunięci" na początek mostu. Takie ustawienie wartości zmiennych jest pierwszym krokiem do rozwiązania problemu.

Następne ograniczenie jest bardziej złożone i ma zapewnić, że osoby będą przekraczały most w kolejności, która pozwoli na spełnienie wymagań rozwiązania. Ograniczenie jest przedstawione w postaci wyrażenia wywołania generatora dla funkcji agregującej *forall*, która wymaga, aby wszystkie elementy typu *bool* zawarte w tablicy będącej argumentem funkcji, miały wartość *true*. Generator ten zmienia wartość zmiennej *t* od *2* do *max_time*, zmienna *t* oznacza tutaj numer kroku, a startuje od *2* ponieważ *t* równe *1* oznacza sytuację początkową, która została określona przez poprzednie ograniczenia.

Elementy tablicy, która będzie argumentem dla funkcji *forall*, są tworzone poprzez zagnieżdżone wywołanie generatora dla funkcji agregującej *exists*. Funkcja *exists* sprawdza czy jakikolwiek element, wewnątrz tablicy będącej argumentem funkcji, ma wartość *true*. Generator zmienia wartość zmiennej *place* pomiędzy wartościami *A* oraz *B*. Element tablicy, będącej argumentem funkcji *exists*, jest koniunkcją kilku ograniczeń. Pierwsze z ograniczeń dotyczą tablicy *torch_place*, miejsce pochodni w kroku *t-1* musi być równe rozważanej wartości zmiennej *place*. Dodatkowo miejsce pochodni musi się zmieniać w każdym kroku, więc w kroku *t* musi być w innym miejscu niż w kroku poprzedzającym, czyli *t-1*.

Następne ograniczenie wykorzystuje wyrażenie *let* pozwalające na wprowadzenie do modelu zmiennych lokalnych. Zmienna lokalna *num_transfered* przechowuje informację o tym ile osób przekracza most w danym kroku. Informacja ta jest wykorzystywana w ograniczeniu dotyczącym tablicy *times*, której elementy określają ile czasu zajmuje przekroczenie mostu w kroku *t*. Czas przekroczenia mostu narzuca najwolniejsza osoba, spośród oznaczonych w tablicy *transfered* jako przechodzące most w danym kroku *t*.

Ostatnim ograniczeniem należącym do koniunkcji ograniczeń będącej argumentem funkcji *exists* jest zagnieżdżone wywołanie generatora dla funkcji agregującej *forall*. Indeksem tego wyrażenia generatorowego jest zmienna *i*, która przyjmuje wartości od 1 do *num_persons*. To ograniczenie definiuje związek pomiędzy tablicą *transfered* oraz tablicą *actions*. Każda osoba, która jest oznaczona w tablicy *transfered* jako przekraczająca most w kroku *t* musi zmienić wartość w tablicy *actions* na wartość przeciwną do wartości przyjętej w kroku *t-1*. Dodatkowo wartość wewnątrz tablicy *actions* w kroku *t-1* dla osoby przechodzącej przez most musi być równa wartości zmiennej *place*, rozważanej wewnątrz wyrażenia wywołania generatora dla funkcji agregującej *exists*. To ograniczenie występuje w koniunkcji z ograniczeniem, które sprawdza, czy ludzie, którzy nie przekraczają mostu w kroku *t* nie zmieniają wewnątrz tablicy *actions*, w stosunku do kroku poprzedniego, wartości oznaczającej, po której stronie mostu się znajdują.

Ostatnie ograniczenie sprawdza czy istnieje sytuacja, w której wszystkie osoby znajdują się już po drugiej stronie mostu razem z pochodnią. Efekt ten uzyskano poprzez użycie wyrażenia wywołania generatora dla funkcji agregującej *exists*, które zmienia wartości *t* od 2 do *max_time*. Element tablicy, która jest argumentem predykatu *exists* są zbudowane z koniunkcji trzech ograniczeń. Pierwsze ograniczenie to wyrażenie wywołania generatora dla funkcji *forall*, które sprawdza, czy każda osoba w tablicy *actions* przyjęła wartość odpowiadającą wartości zmiennej *B*, co oznaczałoby, że wszyscy przekroczyli most. Dodatkowo pochodnia musi być razem ze wszystkimi, więc zmienna w tablicy *torch_place* musi być równa *B* dla kroku *t*. Ostatnie ograniczenie określa wartość zmiennej *total_steps* będącej numerem kroku *t*, w którym poprzednie ograniczenia są spełnione.

W przypadku kiedy chcemy uzyskać odpowiedź na pytanie postawione w opisie problemu należy zmienić element *solve minimize* na *solve satisfy* oraz dodać do koniunkcji ograniczeń dwa dodatkowe: zmienna *total* nie może być większa od *15*, natomiast zmienna *total_steps* nie może przekroczyć *6*.

Ostatnim elementem modelu jest element wyjścia. Determinuje on w jaki sposób zostaną wyświetlone rozwiązania modelu. Najpierw wyświetlone zostają zmienne *total_steps* oraz *total* następnie z wykorzystaniem wyrażenia generatorowego tworzona jest jedna linia dla każdego wykonanego kroku, przedstawia ona, gdzie znajduje się każda z osób w danym korku (1 – przed przejściem przez most, 2 – po przejściu przez most), później do linii dopisywana jest pozycja pochodni oraz informacja, które osoby przekraczają most w danym kroku.

Omawiany model jest napisany w sposób, który pozwala na wprowadzenie różnych danych, dzięki czemu można sprawdzać rozwiązania dla odmiennych od oryginalnych założeń, jak np. większa liczba osób chcąca przekroczyć most. W przypadku, kiedy pożądane jest rozwiązanie modelu z oryginalnymi założeniami należy dołączyć plik z danymi, który może przyjąć formę przedstawioną poniżej.

```
% problem mostu i pochodni - plik danych
% most_i_pochodnia.dzn
max_time = 10;
max_num_to_cross = 2;
num_persons = 4;
cross_time = [1,2,5,8];
```

Dodając do modelu dodatkowo ograniczenia dotyczące maksymalnej liczby wykonanych kroków oraz ograniczenia czasowego zwrócony wynik będzie miał następującą postać:

```
total_steps: 6
total: 15
1 1 1 1 torch: 1 transfered: 1..4
2 2 1 1 torch: 2 transfered: 1..2
1 2 1 1 torch: 1 transfered: 1..1
1 2 2 2 torch: 2 transfered: 3..4
1 1 2 2 torch: 1 transfered: 2..2
2 2 2 2 torch: 2 transfered: 1..2
```

Pierwsze dwie linie zwróconego wyniku mówią, o tym, że most przekroczono *6* razy (biorąc pod uwagę pozycje startową jako pierwszy krok) i zajęło to łącznie 15

minut. Następnie na każde przekroczenie mostu przypada 1 linia. Pierwsze 4 cyfry mówią o tym, gdzie znajdują się poszczególne osoby: *1* – oznacza początek mostu, natomiast *2* – koniec mostu. Podobnie jest opisane miejsce pochodni: *torch: 1* – oznacza, że pochodnia znajduje się na początku mostu, *2* – oznacza koniec mostu. Na końcu każdej linii znajduje się zestaw liczb, odpowiadający poszczególnym osobom, oznaczający kto przekroczył w danym kroku most. Na przykładzie linii odpowiadającej kroku *t=4*: Osoba, której odpowiada pierwsze miejsce w tablicy *cross_time* znajduje się na początku mostu, pozostałe 3 osoby są na jego końcu, pochodnia znajduje się na końcu mostu, na końcu linii jest informacja, że w tym kroku most przekroczyły osoby, które są opisane poprzez 3 i 4 element tablicy *corss_time*.

5.2. Układanie domina

Problem domina polega na ułożeniu poszczególnych klocków w taki sposób aby zapełniły całą planszę przy zachowaniu początkowych założeń. Pola planszy wypełnione są liczbami od 0 do *high*, gdzie *high* oznacza największą liczbę na kawałku domina. Plansza zazwyczaj ma wysokość równą *high+1* oraz szerokość równą *wysokość+1*. Każdy klocek pokrywa dwa pola oraz posiada dwa numery. Numer na klocku musi odpowiadać numerowi na polu planszy, na którym jest położony. Do zapełnienia planszy należy użyć każdego dostępnego klocka, jednak żaden klocek nie może być użyty więcej niż jeden raz. Przedstawiony poniżej model jest autorstwa Hakan Kjellerstrand (*www.hakank.org*):

```
% układanie domina - problem polega na zapełnieniu planszy klockami
% domina, tak aby liczby na klocku odpowiadały tym na polach planszy
% domino.mzn – autor Hakan Kjellerstrand
include "globals.mzn";
int: r;
int: c;
int: m = (r*c) div 2;
int: high;

array[1..m, 1..2] of 0..high: pieces;
array[1..r, 1..c] of 0..high: board;
array[1..r, 1..c] of var 1..m: x;

solve :: int_search(
        [x[i,j] | i in 1..r, j in 1..c],
        largest,
        indomain_max,
        complete)
    satisfy;
```

```
constraint
  forall(p in 1..m) (
    count([x[i,j] | i in 1..r, j in 1..c], p, 2)
  )
;
constraint
  forall(i in 1..r, j in 1..c) (
      exists(a,b in {-1,0,1} where
            i+a >= 1 /\ j+b >= 1 /\
            i+a <= r /\ j+b <= c
            /\(abs(a) + abs(b) = 1)
           ) (
        (
          (pieces[x[i,j],1]     = board[i,j]     /\
           pieces[x[i+a,j+b],2] = board[i+a,j+b])
          \/
          (pieces[x[i+a,j+b],1] = board[i+a,j+b] /\
           pieces[x[i,j],2]     = board[i,j])
        )
        /\
        x[i,j] = x[i+a,j+b]
      )
  )
;
pieces = array2d(1..m, 1..2, [
         if k = 1 then
           i
         else
           j
         endif
         | i in 0..high, j in  0..high, k in 1..2
           where i <= j
         ]);

output
[
 "Board:"
]
++
[
  if j = 1 then "\n" else " " endif ++
    show(board[i,j])
  | i in 1..r, j in 1..c
] ++ ["\n\nPieces:"]
++
[
  if j = 1 then "\n" else " " endif ++
     if fix(x[i,j]) < 10 then " " else "" endif ++
     show(x[i,j])
  |i in 1..r, j in 1..c
] ++ ["\n"];
```

Model MiniZinc autorstwa Hakan Kjellerstrand.
Link do modelu: www.hakank.org/minizinc/domino.mzn

Na samym początku tego modelu pojawia się element dołączenia. Dołączony zostaje plik *globals.mzn*, który pozwala na wykorzystanie ograniczeń globalnych. Następnie są deklarowane zmienne, które będą używane w modelu. Zmienne *r* oraz *c* odpowiadają kolejno za ilość wierszy oraz kolumn planszy, na której będą układane klocki domina. Zmienna *m* odpowiada za ilość klocków domina, którą można umieścić na planszy, jest to ilość pól podzielona przez dwa, ponieważ każdy klocek pokrywa dwa pola. Kolejna zmienna *high* oznacza najwyższy numer znajdujący się na kawałku domina, dzięki tej zmiennej tworzony jest zestaw klocków od [*0, 0*] do [*high, high*].

Następnie w modelu deklarowane są trzy tablice: pierwsza z nich *pieces* zawiera zestaw klocków dostępnych do ułożenia na planszy, druga tablica *board* przechowuje informacje o tym, jakie numery są przypisane do poszczególnych pól planszy. Ostatnia tablica, o nazwie *x,* składa się z elementów będących zmiennymi decyzyjnymi, wypełnienie tej tablicy przez solver będzie oznaczało rozwiązanie modelu. Solver ma za zadanie wpisać do tej tablicy indeksy odpowiadające klockom, które będą ułożone na planszy, w odpowiadających im miejscach, na przykład klocek, którego indeks będzie zawarty w *x*[2,3] oraz w *x*[2,4] będzie znajdował się na planszy na polach *board*[2,3] oraz *board*[2,4]. Wartościami elementów tablic *pieces* oraz *board* mogą być wartości od *0* do *high*, ponieważ takie wartości widnieją na klockach domina. Tablica *x* natomiast zawiera wartości od *0* do *m*, co odpowiada indeksom klocków zawartych w tablicy *pieces*.

Kolejną częścią modelu jest element rozwiązania. Solver ma za zadanie znaleźć rozwiązanie spełniające ograniczenia. Dodatkowo użyto tutaj adnotacji przeszukiwań *int_search*, która ma wskazać solverowi w jaki sposób powinien szukać wartości dla zmiennych decyzyjnych zawartych w pierwszym argumencie adnotacji. Pozostałymi argumentami dla tej adnotacji są: *largest* – co oznacza, że najpierw wybierane są zmienne o największych wartościach w swojej domenie, *indomain_max* – nakazuje przypisywać zmiennym wartości od największy dostępnych w domenie, *complete* – oznacza, że ma zostać wykonywane pełne przeszukiwanie.

Kolejnym elementem modelu są ograniczenia, które solver będzie musiał spełnić aby rozwiązać model. Pierwsze ograniczenie jest wyrażeniem wywołania generatora dla funkcji *forall*, to ograniczenie gwarantuje, że indeks każdego klocka został użyty w trakcie pokrywania planszy tylko i wyłącznie dwukrotnie (ponieważ jeden indeks

odpowiada połowie klocka), w tym celu zawarto tutaj ograniczenie globalne *count*, którego argumentem są: tablica, wartość, która ma być liczona, oraz liczba, która określa ile razy powinna wystąpić w tablicy liczona wartość.

Następne ograniczenie definiuje sposób, w który mają być ułożone klocki na planszy. Dla każdego analizowanego pola planszy należy znaleźć sąsiada, z którym stworzy parę, na której będzie można położyć klocek domina. Jest to przykład sąsiedztwa von Neumanna, co oznacza, że brani pod uwagę są jedynie sąsiedzi położeni w poziomie lub pionie względem analizowanego pola planszy. Ograniczenie przyjmuje postać wyrażenia wywołania generatora dla funkcji *forall*. Wewnątrz tego wyrażenia zawarte są dwa generatory: dla zmiennej *i* od *1* do *r* oraz dla *j* od *1* do *c*, co sprawi, że warunek wewnątrz funkcji będzie sprawdzany dla każdego pola planszy. Elementy, które będą argumentem dla funkcji *forall*, są wynikiem zagnieżdżonego wyrażenia wywołania generatora dla funkcji agregującej *exists*.

Elementy tablicy, które będą zwrócone jako argument do funkcji *exists* są tworzone przy pomocy dwóch generatorów: dla zmiennej *a* oraz *b*, obie zmienne mają zestaw wartości w postaci {*-1, 0 1*}, jednak wartości, które mogą przyjąć są ograniczone warunkiem *where* wewnątrz wyrażenia generatora. Warunek przypisania wartości dla zmiennych *a* oraz *b* zapobiega przypadkowi, w którym podczas analizowania pola planszy na jej krawędzi są brani pod uwagę sąsiedzi spoza obszaru planszy. Dodatkowo koniunkcja z warunkiem, w którym suma wartości bezwzględnych *a* i *b* jest równa 1 zapobiega możliwości stworzenia pary z dwoma sąsiadami jednocześnie.

Elementy, które będą zwrócone jako tablica do funkcji *exists*, składają się z koniunkcji dwóch warunków. W pierwszym wartość w tablicy *board* musi odpowiadać wartości pierwszej na klocku z indeksem, który jest na odpowiadającej pozycji w tablicy *x* dla rozważanego pola planszy, dodatkowo druga wartość na klocku musi być równa z numerem na polu rozważanego sąsiada. Ograniczenie to jest złączone alternatywą z ograniczeniem, które rozważa te same pola na planszy jednak odwrotne ułożenie klocka domina. Wynik tej alternatywy występuje w koniunkcji z ograniczeniem, które wymaga aby w tablicy *x* występowała jednakowa wartość, będąca indeksem układanego klocka, w miejscach odpowiadających rozważanym polom planszy.

Ogólnie rzecz biorąc, wyrażenie wywołania generatora dla funkcji agregującej *forall* analizuje każde pole planszy szukając dla niego sąsiada, który będzie odpowiedni, aby postawić na tych polach klocek domina. Ograniczenie zostanie spełnione w chwili kiedy zostanie znaleziony sąsiad dla każdego pola planszy i tablica *x* zostanie wypełniona odpowiednimi indeksami.

Kolejnym elementem modelu jest element przypisania, w którym nadawane są wartości dla tablicy *pieces*. Wartości są wpisane do tablicy za pomocą funkcji *array2d*, która pozwala umieścić wartości z tablicy jednowymiarowej podanej jako argument do tablicy dwuwymiarowej, której indeksy są podane jako pierwszy oraz drugi argument funkcji. Pierwszy argument *1..m* oznacza pierwszy wymiar tablicy, natomiast *1..2* odnosi się do drugiego wymiaru tablicy, co oznacza, że ma ona *m* wierszy oraz *2* kolumny. W tej tablicy są umieszczone w kolejności leksykograficznej pary liczb widoczne na klockach domina.

Poszczególne wartości, które będą wpisane w dwuwymiarową tablicę *pieces* są tworzone przy pomocy wyrażenia generatora. Wyrażenie zawiera generatory dla trzech zmiennych: *i* w zakresie *0..high*, *j* również w zakresie *0..high*, oraz *k* w zakresie *1..2*. Zmienne *i* oraz *j* oznaczają numery znajdujące się na klockach domina, natomiast zmienna *k* decyduje o tym, czy dany numer jest wpisany do pierwszej, czy do drugiej kolumny w tablicy. Ostatni generator (dla zmiennej *k*) posiada warunek *where*, który dopuszcza stworzenie elementu jedynie w przypadku kiedy zmienna *i* jest mniejsza, lub równa zmiennej *j* co zapobiega pojawieniu się duplikatów klocków. Przykładowym wynikiem takiego przypisania, dla zmiennej *high* równej *3*, będzie tablica dwuwymiarowa zawierająca następujące zestawy numerów: [0,0], [0,1], [0,2], [0,3], [1,1], [1,2], [1,3], [2,2], [2,3], [3,3].

Ostatnim elementem modelu jest element wyjścia, który definiuje sposób wyświetlenia wyników. W pierwszej kolejności wyświetlane jest ułożenie numerów na polach planszy. W drugiej części w ten sam sposób jest wyświetlane, w jaki sposób są ułożone klocki domina poprzez podanie odpowiedniego indeksu klocka domina na polach planszy, na których leży.

W pliku danych dla tego modelu zawarte są informacje o wielkości planszy, posiada ona *7* rzędów, z których każdy zawiera *8* kolumn. Klocki zawierają numery od

1 do *6*. Ostatnim elementem pliku danych jest informacja o numerach znajdujących się na polach planszy wpisana do tablicy *board*.

```
% układanie domina - plik danych
% domino.dzn
r = 7;
c = 8;
high = 6;
board = array2d(1..r, 1..c,
     [
      3,1,2,6,6,1,2,2,
      3,4,1,5,3,0,3,6,
      5,6,6,1,2,4,5,0,
      5,6,4,1,3,3,0,0,
      6,1,0,6,3,2,4,0,
      4,1,5,2,4,3,5,5,
      4,1,0,2,4,5,2,0
    ])
;
```

Plik danych pochodzi ze strony Hakan Kjellerstrand: www.hakank.org/minizinc/domino_ecl.dzn

Używając tego pliku danych do rozwiązania modelu zwrócony zostanie następujący wynik (po lewej widoczna jest plansza oraz indeksy użytych klocków domina, po prawej są pokazane indeksy z odpowiadającymi im klockami domina):

```
Board:
3 1 2 6 6 1 2 2
3 4 1 5 3 0 3 6
5 6 6 1 2 4 5 0
5 6 4 1 3 3 0 0
6 1 0 6 3 2 4 0
4 1 5 2 4 3 5 5
4 1 0 2 4 5 2 0

Pieces:
10 10  9 27 22  2 14 14
20 20  9 27 22  2 21  7
26 28 28  8 16 16 21  7
26 13  5  8 19  4  4  1
25 13  5 18 19 15 24  1
25 12 12 18 23 15 24  6
11 11  3  3 23 17 17  6
```

ID	Wartości na klocku	ID	Wartości na klocku
1	0, 0	15	2, 3
2	0, 1	16	2, 4
3	0, 2	17	2, 5
4	0, 3	18	2, 6
5	0, 4	19	3, 3
6	0, 5	20	3, 4
7	0, 6	21	3, 5
8	1, 1	22	3, 6
9	1, 2	23	4, 4
10	1, 3	24	4, 5
11	1, 4	25	4, 6
12	1, 5	26	5, 5
13	1, 6	27	5, 6
14	2, 2	28	6, 6

5.3. Problem kolejnych cyfr

Problem kolejnych cyfr przedstawiony jest w formie działania:

```
  ABCD
  DCBA
 +****
------
 12300
```

W tym działaniu ciąg *ABCD* oznacza cztery kolejne cyfry. *DCBA* jest ciągiem tych samych cyfr w odwrotnej kolejności. Cztery kropki w trzeciej linii oznaczają ciąg tych samych czterech znaków jednak bez określonej kolejności. Znając wynik dodawania, czyli *12300*, należy podać cyfry, które mają zastąpić kropki w działaniu. Przedstawiony poniżej model jest autorstwa Hakan Kjellerstrand (*www.hakank.org*):

```
% problem kolejnych cyfry - problem polega na znalezieniu cyfr
% ukrytych pod znakami, tak aby działanie dało poprawny wynik
% kolejne_cyfry.mzn – autor Hakan Kjellerstrand

include "globals.mzn";
var 1..9: A;
var 1..9: B;
var 1..9: C;
var 1..9: D;
array[1..4] of var 1..9: fd = [A,B,C,D];

array[1..4] of var 1..9: dots;
var int: dots_num;

var int: ABCD = 1000*A + 100*B + 10*C + D;
var int: DCBA = 1000*D + 100*C + 10*B + A;

predicate toNum10(array[int] of var int: a, var int: n) =
          let { int: len = length(a) }
          in
          n = sum(i in 1..len) (
            ceil(pow(10.0, int2float(len-i))) * a[i]
          )
          /\ forall(i in 1..len) (a[i] >= 0)
;

predicate contains(var int: e, array[int] of var int: a) =
   exists(i in 1..length(a)) (
      a[i] = e
   )
;

solve :: int_search(fd ++ dots ++ [ABCD, DCBA, dots_num], first_fail,
indomain_min, complete) satisfy;
```

```
constraint
  all_different(fd)
  /\
  increasing(fd)
  /\
  all_different(dots)
  /\
  toNum10(dots, dots_num)
  /\
  12300 = ABCD + DCBA + dots_num

  /\
  contains(A, dots)
  /\
  contains(B, dots)
  /\
  contains(C, dots)
  /\
  contains(D, dots)
;

output [
   "  ", show(ABCD), "\n",
   "  ", show(DCBA), "\n",
   "+ ", show(dots_num), "\n",
   "------", "\n",
   " 12300","\n"
];
```

Model MiniZinc autorstwa Hakan Kjellerstrand.
Link do modelu: www.hakank.org/minizinc/consecutive_digits.mzn

Model ten zaczyna się od elementu dołączenia, który obejmuje plik *globalz.mzn* pozwalający na używanie w modelu ograniczeń globalnych dostępnych w MiniZinc. Po elemencie dołączenia następuje kilka elementów deklaracji zmiennych. Zmienne *A, B, C* oraz *D* definiowane są jako zmienne decyzyjne, których wartościami mogą być liczby całkowite z zakresu od 1 do 9.

Kolejną deklarowaną zmienną jest tablica *fd* zawierająca elementy typu *int* będące zmiennymi decyzyjnymi, elementami tej tablicy są wartości zmiennych *A, B, C* oraz *D*. Tablica *dots* podobnie jak tablica *fd* przechowuje cztery wartości typu *var int*, które oznaczają cyfry schowane pod kropkami w działaniu. Kolejne trzy zmienne, czyli: *ABCD*, *DCBA* oraz *dots_num* oznaczają wartości liczb ukrytych pod znakami w rozpatrywanym działaniu, będą one potrzebne do sprawdzenia poprawności znalezionych cyfr z wartością przedstawioną jako wynik działania.

Model zawiera dwa elementy predykatu, które są wykorzystywane później w tworzeniu ograniczeń. Pierwszym z nich jest predykat *toNum10,* który jako argumenty przyjmuje tablicę elementów typu *var int* oraz zmienną typu *var int*. Wykorzystano tutaj wyrażenie *let* pozwalające na wprowadzenie zmiennych lokalnych dla tego predykatu. Zadaniem tego predykatu jest wprowadzenie ograniczenia, które sprawdza, czy wartość liczbowa, przedstawiona za pomocą kolejnych cyfr będących elementami tablicy podanej jako argument predykatu, jest równa wartości zmiennej podanej jako drugi argument predykatu. Ograniczenie to występuje w koniunkcji z ograniczeniem, które przy użyciu wyrażenia wywołania generatora dla funkcji *forall* sprawdza, czy każdy element tablicy podanej jako argument jest większy, lub równy *0*.

Drugi predykat, o nazwie *contains*, przyjmuje jako argumenty zmienną typu *var int* oraz tablicę, której elementami są wartości typu *var int*. Ograniczenie zawarte w tym predykacie jest przedstawione za pomocą wyrażenia wywołania generatora dla funkcji *exists*, która ma sprawdzić, czy którykolwiek z elementów tablicy podanej jako drugi argument ma wartość równą wartości zmiennej podanej w pierwszym argumencie dla tego predykatu.

Następnym elementem w tym modelu jest element rozwiązania. Zadaniem solvera będzie satysfakcja ograniczeń. Do elementu dodana jest adnotacja przeszukiwań: *int_search*. Pierwszym argumentem jest konkatenacja tablic, które zawierają zmienne decyzyjne potrzebne do rozwiązania modelu, drugi argument, *first_fail,* oznacza, że najpierw wybierana jest zmienna z najmniejszą domeną, *indomain_min* oznacza, że solver powinien zacząć od przypisywania najmniejszych możliwych wartości w domenie zmiennych, ostatni argument *complete* oznacza, że powinno zostać wykonane kompletne przeszukiwanie w celu znalezienia rozwiązania modelu.

Następnie w modelu występuje element ograniczeń. Element obejmuje serię ograniczeń powiązanych koniunkcją. Pierwsze ograniczenia dotyczą tablicy *fd*, wartości elementów nie mogą się powtórzyć, oraz kolejne wartości wewnątrz tablicy powinny być ułożone rosnąco. Również tablica *dots*, której elementy przedstawiają cyfry ukryte pod kropkami w działaniu, powinna mieć elementy o różnych wartościach. Kolejne ograniczenie wykorzystuje predykat *toNum10,* który sprawdza, czy cyfry wewnątrz tablicy *dots* przedstawiają wartość równą wartości zmiennej *dots_num*. Następnie

sprawdzany jest wynik działania: wartości liczbowe przedstawione jako *ABCD* po dodaniu do wartości zmiennej *DCBA* zsumowane z wartością zmiennej *dots_num* powinny dać wynik przedstawiony w działaniu, czyli *12300*. Dodatkowo wykorzystując predykat *contains* sprawdzane jest, czy wszystkie cyfry ukryte pod zmiennymi *A, B, C,* oraz *D* występują w tablicy *dots*. Sytuacja, w której wszystkie powyższe ograniczenia będą spełnione będzie oznaczała, że znaleziono rozwiązanie modelu.

Ostatnim elementem modelu jest element wyjścia, który przedstawia wynik w formie dodawania pisemnego, w którym litery oraz kropki zostały zastąpione odpowiednimi cyframi, zwróconymi przez solver. Model ten nie wymaga, żadnego pliku danych, ze względu na to, że nie ma tutaj parametrów, które mogłyby się zmieniać pomiędzy kolejnymi rozwiązaniami modelu. Wynik zwrócony po rozwiązaniu tego modelu:

```
  2345
  5432
+ 4523
------
 12300
```

5.4. Inwestycje – problem plecakowy

Zaprezentowany poniżej model MiniZinc przedstawia sytuację, w której firma wybiera, które z możliwych inwestycji powinna zrealizować, tak aby łączna ich wartość była jak największa. Firma jest ograniczona poprzez zasoby, które może przeznaczyć na realizację projektów. Każda z dostępnych inwestycji ma przypisaną swoją wartość oraz zasoby konieczne do jej przeprowadzenia. Dodatkowo część projektów jest od siebie zależna w taki sposób, że mogą się wzajemnie wykluczać, lub dana inwestycja musi wystąpić w parze z inną. Przedstawiony poniżej model jest autorstwa Hakan Kjellerstrand (*www.hakank.org*):

```
% problem inwestycji - problem polega na wybraniu inwestycji do
% realizacji, tak aby ich łączna wartość była jak największa
% inwestycje.mzn – autor Hakan Kjellerstrand

int: num_projects;
int: max_budget;
int: max_persons;
int: max_projects;
```

```
array[1..num_projects] of int: values;
array[1..num_projects] of int: budgets;
array[1..num_projects] of int: personell;
int: num_not_with;
array[1..num_not_with, 1..2] of 1..num_projects: not_with;

int: num_requires;
array[1..num_requires, 1..2] of 1..num_projects: requires;

array[1..num_projects] of var 0..1: x;

var int: total_persons  = sum(i in 1..num_projects)
(x[i]*personell[i]);
var int: total_budget   = sum(i in 1..num_projects) (x[i]*budgets[i]);
var int: total_projects = sum(i in 1..num_projects) (x[i]);

var int: total_values   = sum(i in 1..num_projects) (x[i]*values[i]);

solve :: int_search(x, first_fail, indomain_min, complete) maximize
total_values;

constraint
   total_budget <= max_budget
   /\
   total_persons <= max_persons
   /\
   total_projects <= max_projects

   /\
   forall(i in 1..num_requires) (
      x[requires[i, 1]] - x[requires[i, 2]] <= 0
   )
   /\
   forall(i in 1..num_not_with) (
      x[not_with[i, 1]] + x[not_with[i, 2]] <= 1
   )
;

output
[
  "x: " ++ show(x) ++ "\n" ++
  "total_persons: " ++ show(total_persons) ++ "\n" ++
  "total_budget: " ++ show(total_budget) ++ "\n" ++
  "total_projects: " ++ show(total_projects) ++ "\n" ++
  "total_values: " ++ show(total_values) ++ "\n"
];
```

Model MiniZinc autorstwa Hakan Kjellerstrand.
Link do modelu: www.hakank.org/minizinc/knapsack_investments.mzn

Model rozpoczyna się od deklaracji zmiennych opisujących zasoby, które firma może przeznaczyć na realizację inwestycji. Zmienna *num_projects* oznacza liczbę

dostępnych do realizacji projektów, *max_budget* oznacza budżet, którego nie mogą przekroczyć łączne koszty inwestycji, *max_persons* określa liczebność personelu, który może wziąć udział w realizacji projektów, *max_projects* definiuje ile najwięcej projektów można zrealizować.

Później deklarowane są tablice, które przechowują informacje dotyczące możliwych inwestycji. Tablica *values* zawiera informację o wartości każdego z projektów, *budgets* opisuje jakie koszty poniesie firma w celu realizacji inwestycji, natomiast *personell* określa ile pracowników będzie trzeba przeznaczyć na każdą z inwestycji.

Następnie deklarowane są tablice, które opisują zależności występujące pomiędzy inwestycjami. Tablica *not_with* zawiera informacje o tym, które z inwestycji się wzajemnie wykluczają, zmienna *num_not_with* określa rozmiar tej tablicy poprzez informację ile jest przypadków wykluczenia jednej inwestycji przez inną. Tablica *requires* przechowuje informacje, o tym, które inwestycje, w celu ich realizacji, wymagają jednoczesnego wykonywania innej inwestycji. Rozmiar tablicy jest definiowany przez zmienną *num_requires*, która określa ile jest przypadków takiej zależności. Obie tablice posiadają dwie kolumny, wartość pierwszej z nich określa, której inwestycji dotyczy dany wiersz tablicy, a druga kolumna określa inwestycję, przy której występuje zależność względem inwestycji w kolumnie pierwszej.

Kolejnym elementem modelu jest zestaw zmiennych decyzyjnych, dla których wartości będzie musiał znaleźć solver. Zmienna tablicowa *x* po rozwiązaniu modelu będzie zawierać informacje, o tym, które z inwestycji powinny być zrealizowane, odpowiadające im elementy tablicy *x* przyjmą wartość *1*, natomiast odrzucone inwestycje będą określone elementami o wartości *0*.

Zmienna *total_persons* określa ile osób jest wymagane do realizacji wybranych inwestycji, liczba ta jest określana przez sumę poszczególnych elementów zmiennej tablicowej *x* przemnożonych przez odpowiadające im elementy w tablicy *personell,* jest to możliwe dzięki temu, że elementy tablicy *x* przyjmują wartość *1* dla inwestycji do realizacji oraz wartość *0* dla odrzuconych inwestycji. Podobnie jest określana wartość dla zmiennej *total_budget,* która mówi o łączonym koszcie inwestycji wybranych do realizacji, w tym wypadku elementy tablicy *x* są przemnażane z odpowiednimi

elementami tablicy *budgets*. Zmienna *total_projects* jest sumą wartości elementów tablicy *x*, oznacza ona liczbę inwestycji wybranych do realizacji. Zmienna decyzyjna *total_values* określa łączną wartość inwestycji wybranych do realizacji, definiowana jest podobnie jak w przypadku zmiennych *total_persons* i *total_budget*, jej wartość odpowiada sumie elementów tablicy *x* przemnożonych przez odpowiadające im elementy tablicy *values*.

Element rozwiązania nakazuje solverowi dokonanie optymalizacji wartości zmiennych decyzyjnych w celu maksymalizacji wartości zmiennej *total_values*. Zastosowano tutaj adnotację przeszukiwań *int_search* o następujących argumentach: *x* – oznacza tablicę, dla elementów której stosowana jest adnotacja, *first_fail* – najpierw wybiera zmienne z najmniejszą domeną wartości, *indomain_min* – przypisuje wartości zaczynając od najmniejszych w domenie, *complete* – oznacza przeszukiwanie pełne.

Kolejnym elementem modelu są ograniczenia, których solver musi przestrzegać podczas poszukiwań wartości dla zmiennych decyzyjnych. Pierwsza grupa ograniczeń zapewnia, że wybrane inwestycje do realizacji nie przekraczają zasobów, którymi dysponuje firma, jest to koniunkcja warunków ograniczająca wartości dla *total_budget*, *total_persons* oraz *total_projects*.

Druga grupa ograniczeń odpowiada za przestrzeganie zależności występujących pomiędzy poszczególnymi inwestycjami. Pierwsze z tych ograniczeń odpowiada za inwestycje, które wymagają jednoczesnej realizacji innych inwestycji. Efekt ten osiągnięto poprzez zastosowanie wywołania wyrażenia generatora dla funkcji agregującej *forall*. Funkcja zwróci wartość *true* kiedy warunek zapisany w wyrażeniu generatora będzie prawdziwy dla wartości *i* w zakresie od 1 do *num_requires*. Wynik odejmowania zawartego w warunku przyjmie wartość większą niż *0* tylko w przypadku kiedy będzie przeznaczona do realizacji inwestycja, która znajduje się w pierwszej kolumnie tablicy *requires*, jednak element w drugiej kolumnie danego wiersza będzie odpowiadał inwestycji, która nie została wybrana do realizacji. Jeśli wystąpi taka sytuacja, to ograniczenie nie zostanie spełnione.

Drugie ograniczenie, które określa czy są przeznaczone do realizacji inwestycje, które się wykluczają, jest definiowane przez wyrażenie wywołania generatora dla funkcji *forall*, podobnie jak w poprzednim ograniczeniu. Różnica polega na tym, że

brane pod uwagę są elementy tablicy *not_with*. Wynik dodawania zawartego w warunku zwróci wartość większą od *1* w przypadku kiedy do realizacji będą przeznaczone inwestycje oznaczone w tablicy *not_with* jako wykluczające się wzajemnie, co będzie oznaczało, że ograniczenie nie jest spełnione.

Poniżej przedstawiono plik danych użyty przy rozwiązywaniu modelu.

```
% problem inwestycji - plik danych
% inwestycje.dzn
num_projects = 15;
max_budget = 225;
max_projects = 9;
max_persons = 28;
values = [600,400,100,150, 80,120,200,220,90,380,290,130,80,270,280];
budgets = [35,34,26,12,10,18,32,11,10,22,27,18,16,29,22];

num_not_with = 6;
not_with = array2d(1..num_not_with, 1..2, [
  1, 10,
  5, 6,
  6,5,
  10, 1,
  11, 15,
  15, 11
]);

num_requires = 5;
requires = array2d(1..num_requires, 1..2, [
  3, 15,
  4, 15,
  8, 7,
  13, 2,
  14, 2
]);

personell = [5,3,4,2,2,2,4,1,1,5,3,2,2,4,3];
```

Plik danych pochodzi ze strony Hakan Kjellerstrand:
www.hakank.org/minizinc/knapsack_investments.mzn

Uruchomienie modelu z zastosowaniem powyższego pliku danych zwraca wynik:

```
x: [1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1]
total_persons: 26
total_budget: 211
total_projects: 9
total_values: 2370
```

Otrzymany wynik oznacza, że do realizacji przeznaczono projekty oznaczone numerami: 1, 2, 4, 6, 7, 8, 12, 14, 15. Do realizacji wybranych inwestycji potrzeba *26* osób, łączny koszt wynosi *211*. Do realizacji wybrano *9* projektów, których łączna wartość wynosi *2370*.

5.5. Planowanie produkcji

Problem planowania produkcji polega na zoptymalizowaniu harmonogramu opisującego proces tworzenia finalnego produktu pod względem wybranych aspektów, np. minimalizacja czasu potrzebnego na przetworzenie produktu, lub zoptymalizowanie wykorzystania maszyn używanych w produkcji. Przedstawiony poniżej model MiniZinc odpowiada sytuacji, w której celem jest zminimalizowanie czasu potrzebnego na przeprowadzenie założonych zadań na zestawie maszyn. Problem zakłada, że na maszynie może być wykonywane tylko jedno zadanie jednocześnie, dodatkowo każde zadanie musi przechodzić przez maszyny w odpowiedniej kolejności. Przedstawiony poniżej model jest autorstwa Hakan Kjellerstrand (*www.hakank.org*):

```
% planowanie produkcji - problem polega na wykonaniu przez
% maszyny założonych zadań w jak najmniejszym czasie
% planowanie_produkcji.mzn - autor Hakan Kjellerstrand

int: n;
int: m;

set of int: J = 1..n;
set of int: M = 1..m;

array[J, M] of M: sigma;

array[J, M] of int: p;

array[J,M] of var int: x;

array[J,J,M] of var 0..1: Y;

var int: K = sum(j in J, a in M) (p[j,a]);

var int: z;

solve :: int_search(
        [x[i,j] | i in J, j in M] ++
        [Y[i,j,a] | i,j in J, a in M] ++ [K, z],
        first_fail,
        indomain_min,
        complete
        ) minimize z;
```

```
constraint
  z >= 0
  /\
  z <= K
  /\
  forall(i in J, a in M) (
       x[i,a] >= 0
       /\
       x[i,a] <= K
  )
  /\
  forall(j in J, t1 in 1..m, t2 in 1..m where t1 != t2) (
        sigma[j,t1] != sigma[j,t2]
  )

  /\
  forall(i in J, j in J, a in M where i != j) (
      x[i,a] >= x[j,a] + p[j,a] - K * Y[i,j,a]
  )

  /\
  forall(i in J, j in J, a in M where i != j) (
      x[j,a] >= x[i,a] + p[i,a] - K * (1 - Y[i,j,a])
  )

  /\
  forall(j in J, t in 2..m) (
      x[j, sigma[j,t]] >= x[j, sigma[j,t-1]] + p[j, sigma[j,t-1]]
  )

  /\
  forall(j in J) ( z >= x[j, sigma[j,m]] + p[j, sigma[j,m]])
;

output [
  "\nz: ", show(z), "\n",
  "x: "
] ++
[
  if a = 1 then "\n" else " " endif ++
    show(x[i,a])
  | i in J, a in M
] ++ [ "\nY:"] ++
[
  if a = 1 /\ j = 1 then "\n" else "" endif ++
  if a = 1 then "\n" else " " endif ++
    show(Y[i,j,a])
  | i,j in J, a in M
] ++ ["\n"]
;
```

Model MiniZinc autorstwa Hakan Kjellerstrand.
Link do modelu: www.hakank.org/minizinc/jssp.mzn

Pierwszą część modelu stanowią deklaracje zmiennych parametrycznych, które służą do opisu założeń dla harmonogramu procesu produkcji. Zmienna *n* odpowiada ilości zadań do wykonania, *m* określa ilość dostępnych maszyn, zestaw wartości *J* określa zadania, które trzeba wykonać na maszynach, natomiast zestaw *M* określa zbiór dostępnych maszyn. Tablica *sigma* zawiera permutacje maszyn, która oznacza kolejność maszyn, na których ma być wykonane dane zadanie, każdemu zadaniu odpowiada jeden wiersz tej tablicy. Tablica *p* określa ile czasu zajmuje wykonanie konkretnego zadania na danej maszynie.

Kolejna cześć modelu to deklaracja zmiennych decyzyjnych, których wartości będzie ustalał solver w czasie rozwiązywania modelu. Tablica *x* przechowuje informacje, o tym kiedy rozpoczynają się wykonywać kolejne etapy zadań na konkretnych maszynach. Trójwymiarowa tablica *Y* mówi o tym, które z dwóch etapów zadania (oznaczonych przez indeksy dwóch pierwszych wymiarów) jest pierwsze w kolejności do wykonania na maszynie, na którą wskazuje indeks trzeciego wymiaru. Zmienna decyzyjna *K* została dodana w celu ograniczenia maksymalnych wartości dla zmiennych *z* oraz wartości w tablicy *x*. Zmienna *z* określa łączny czas wykonania wszystkich założonych zadań.

Następnie w modelu występuje element rozwiązania. Zadaniem solvera będzie tak dobrać wartości zmiennych decyzyjnych aby zminimalizować wartość zmiennej *z*. Do elementu dołączona jest adnotacja przeszukiwań *int_search*: pierwszy argument stanowi konkatenacja tablic, w których zawarte są zmienne decyzyjne występujące w modelu, drugi argument, *first_fail*, oznacza, że najpierw wybierane są wartości dla zmiennej z najniższą domeną wartości, *indomain_min* oznacza, że najpierw przypisywane są najmniejsze wartości z domeny wartości zmiennej, ostatni argument *complete* określa pełne przeszukiwanie.

Kolejnym elementem występującym w omawianym modelu jest element ograniczeń. Pierwsza grupa ograniczeń odpowiada za zredukowanie zbioru możliwych wartości dla zmiennej *z* oraz wartości elementów tablicy *x* do zestawu: *0..K*. Kolejne ograniczenie odpowiada za sprawdzenie, czy każde zadanie zostanie wykonane tylko raz na każdej maszynie, osiągnięto to poprzez wykluczenie możliwości powtórzenia się wartości w każdym z wierszy tablicy *sigma*.

Konkatenacja dwóch kolejnych ograniczeń ustala wartości dla tablicy *Y* wraz z określeniem kolejności wykonywanych etapów zadań na maszynach. Do zbudowania tego ograniczenia wykorzystano wyrażenie wywołania generatora dla funkcji agregującej *forall.* Funkcja *forall* zwraca wartość *true*, kiedy każdy z elementów tablicy będącej argumentem funkcji ma wartość *true*.

Ograniczenia będące elementami tablicy, która będzie argumentem funkcji *forall*, sprawdzają, które z rozważanych zadań, oznaczanych przez *i* oraz *j* będzie wykonywane jako pierwsze na maszynie definiowanej przez *a*. W pierwszym ograniczeniu kiedy czas rozpoczęcia zadania *i* na maszynie *a* będzie większy niż czas skończenia zadania *j* na maszynie *a* element tablicy *Y* przyjmie wartość *0* dla odpowiedniego elementu. W przeciwnym wypadku element tablicy *Y* powinien przyjąć wartość *1*.

Dla drugiego ograniczenia sytuacja jest odwrotna. W przypadku kiedy zadanie *j* na maszynie *a* rozpocznie się po skończeniu zadania *i* na tej maszynie wartość odpowiedniego elementu tablicy *Y* przyjmie wartość *1*, w przeciwnym wypadku będzie to wartość *0*.

Kolejne ograniczenie zapobiega sytuacji, w której maszyna zaczęłaby wykonywanie kolejnego etapu zadania przed skończeniem wcześniejszego etapu na innej maszynie. Ograniczenie tworzy wyrażenie wywołania generatora dla funkcji *forall* poprzez odpowiednie powiązanie wartości tablicy *x* z tablicą *sigma*, która odpowiada za kolejność maszyn dla danego zadania oraz tablicą *p*, która oznacza czasy wykonywania etapów zadań na odpowiednich maszynach.

Ostatnie ograniczenie poprzez użycie wyrażenia wywołania generatora dla funkcji *forall* definiuje wartość zmiennej *z* jako większą lub równą czasowi potrzebnemu na wykonanie wszystkich zadań. Ponieważ problem jest optymalizacyjny i ma za zadanie znaleźć minimum dla zmiennej *z* powinna ona przyjąć wartość równą czasowi wykonania wszystkich zadań według zoptymalizowanego harmonogramu zawartego w tablicy *x*.

Ostatnim elementem modelu jest element wyjścia, określa on w jaki sposób ma zostać wyświetlony wynik zwrócony przez solver po rozwiązaniu modelu. Najpierw wyświetlana jest wartość zmiennej *z*, następnie tablica *x*, która przechowuje czasy

rozpoczęcia kolejnych etapów zadań na określonych maszynach. Następnie dla każdej maszyny zostają wyświetlone odpowiadające jej wartości w tablicy *Y*.

Plik danych użyty do uruchomienia omawianego modelu:

```
% planowanie produkcji - plik danych
% planowanie_produkcji.dzn
n = 6;
m = 6;
sigma = array2d(J, M,  [
 3, 1, 2, 4, 6, 5,
 2, 3, 5, 6, 1, 4,
 3, 4, 6, 1, 2, 5,
 2, 1, 3, 4, 5, 6,
 3, 2, 5, 6, 1, 4,
 2, 4, 6, 1, 5, 3]);

p = array2d(J, M,  [
 3, 6, 1, 7, 6, 3,
10, 8, 5, 4, 10, 10,
 9, 1, 5, 4, 7, 8,
 5, 5, 5, 3, 8, 9,
 3, 3, 9, 1, 5, 4,
10, 3, 1, 3, 4, 9]);
```

Plik danych pochodzi ze strony Hakan Kjellerstrand: www.hakank.org/minizinc/jssp.mzn

Uruchomienie tego modelu zwróci wynik przedstawiony poniżej:

```
z: 55
x:
27 30 22 36 49 43
40  0  8 50 13 28
18 27  0  5 30  9
13  8 23 28 37 46
50 22 13 54 25 38
30 13 49 16 45 19
```

```
Y:

0 0 0 0 0 0  0 1 1 0 1 1  1 1 1 1 1 1
1 0 0 1 0 0  0 0 0 0 0 0  1 0 1 1 0 1
0 0 0 0 0 0  0 1 0 0 1 0  0 0 0 0 0 0
0 0 1 0 0 1  0 1 1 0 1 1  0 0 1 1 1 1
1 0 0 1 0 0  1 1 1 1 1 1  1 0 1 1 0 1
1 0 1 0 0 0  0 1 1 0 1 0  1 0 1 1 1 1

1 1 0 1 1 0  0 1 1 0 1 1  0 1 0 1 1 1
1 0 0 1 0 0  0 0 0 0 0 0  1 0 0 1 0 1
1 1 0 0 0 0  0 1 0 0 1 0  0 1 0 0 0 0
0 0 0 0 0 0  0 0 1 0 1 1  0 0 0 1 0 1
1 1 0 1 0 0  0 0 0 0 0 0  1 1 0 1 0 1
1 1 1 0 1 0  0 0 1 0 1 0  0 0 0 0 0 0
```

Znaleziony optymalny czas ukończenia produkcji przedstawia zmienna $z = 55$. Wartości tablicy x odpowiadają czasom rozpoczęcia kolejnych etapów zadań na poszczególnych maszynach. Dla zwiększenia przejrzystości wartości tablicy Y dla poszczególnych maszyn są podzielone na dwa rzędy po trzy kolumny (element wyjścia definiuje wyświetlanie ich kolejno w jednej kolumnie), poprawna kolejność odczytywania (maszyny od 1 do 6): od lewej do prawej oraz od góry do dołu.

5.6. Zagadka Einsteina

Zagadka Einsteina znana jest w kilku różnych wariantach. Poniższy model prezentuje wariant, w którym w jednej linii znajduje się 5 różnych domów o różnych kolorach. Każdy dom jest zamieszkiwany przez osobę innej narodowości. Na każdą osobę przypada jeden gatunek hodowanego zwierzęcia, które również się nie powtarzają. Dodatkowo każda osoba pija 5 różnych typów napojów oraz pali papierosy 5 różnych marek. Problem polega na tym, żeby na podstawie założeń do zagadki odpowiedzieć na pytanie: kto hoduje rybki. Przedstawiony poniżej model jest autorstwa Hakan Kjellerstrand (*www.hakank.org*):

```
% zagadka Einsteina - problem polega na wskazaniu hodowcy rybek
% wykorzystując otrzymane informacje
% einstein.mzn - autor Hakan Kjellerstrand
include "globals.mzn";
set of 1..5: a = 1..5;
set of 1..5: b = 1..5;
set of 1..5: c = 1..5;
set of 1..5: d = 1..5;
set of 1..5: e = 1..5;

array[a] of var a: A;
array[b] of var b: B;
array[c] of var c: C;
array[d] of var d: D;
array[e] of var e: E;

array[a] of string: nationalities = ["Norwegian", "Dane", "Briton",
"Swede", "German"];

solve satisfy;

constraint
    all_different(A) /\
    all_different(B) /\
    all_different(C) /\
    all_different(D) /\
    all_different(E) /\
```

```
    A[3] = B[3] /\
    A[4] = C[3] /\
    A[2] = D[2] /\
    B[4] + 1 = B[5] /\
    B[4] = D[1] /\
    E[3] = C[2] /\
    D[3] = 3 /\
    B[1] = E[1] /\
    A[1] = 1 /\
    (E[2] = C[1]+1 \/ E[2] = C[1]-1) /\
    (C[4] = E[1]+1 \/ C[4] = E[1]-1) /\
    E[4] = D[4] /\
    B[2] = 2 /\
    A[5] = E[5] /\
    (E[2] = D[5] + 1 \/ E[2] = D[5] - 1)
;

output [
   "A: ", show(A), "\n",
   "B: ", show(B), "\n",
   "C: ", show(C), "\n",
   "D: ", show(D), "\n",
   "E: ", show(E), "\n"
] ++
[
  "The " ++ show(nationalities[fix(A[C[5]])]) ++ " owns the fish\n"
]++["\n"];
```

Model MiniZinc autorstwa Hakan Kjellerstrand.
Link do modelu: www.hakank.org/minizinc/einstein_hurlimann.mzn

Model rozpoczyna się elementem dołączenia pliku *globals.mzn*, który pozwala na korzystanie z ograniczeń globalnych w modelu. Następnie deklarowane są zmienne, które pozwolą opisać przedstawioną zagadkę. Zestawy typu *int* o wartościach od *1* do *5* odpowiadają konkretnym danym na temat poszczególnych domów. Zestaw *a* oznacza narodowości osób mieszkających w domach. Cyfry od *1* do *5* oznaczają kolejno: Norwega, Duńczyka, Brytyjczyka, Szweda, Niemca. Zestaw *b* odpowiada za kolory domów, dla wartości od *1* do *5* są to kolory: żółty, niebieski, czerwony, zielony, biały. Zestaw *c* odpowiada za gatunki zwierząt: kot, ptak, pies, koń, rybka. Zestaw *d* określa typy spożywanych napojów: kawa, herbata, mleko, sok, woda. Zestaw *e* odpowiada za marki palonych papierosów: *Dunhill, Marlboro, Pall-Mall, Bluemaster, Prince.*

Kolejnym elementem modelu jest deklaracja tablic, które będą przyporządkowywać odpowiednie właściwości do numerów domów. Każda tablica odpowiada za konkretną właściwość, podobnie jak w przypadku zestawów, są to: *A* – dla narodowości, *B* – dla kolorów, *C* – dla gatunku zwierząt, *D* – dla typu napojów, *E* –

dla marek papierosów. Wartość elementu tablicy oznacza numer domu, którego dotyczy dana informacja. Indeks, na którym występuje numer domu określa wartość tej właściwości dla tego domu, przykładowo jeśli *B[2]* zawiera wartość *4* oznacza to, że czwarty dom jest koloru niebieskiego.

Tablica *nationalities* została stworzona w celu bardziej przejrzystego przedstawienia wyniku modelu. Element rozwiązania w tym modelu przekazuje informację dla solvera, że jest to problem satysfakcji ograniczeń.

Kolejną część modelu stanowią ograniczenia, które solver będzie musiał spełnić podczas szukania właściwych wartości dla zmiennych decyzyjnych. Na początku zaznaczono, że żadna z cech nie powtarza się w dwóch różnych domach, zastosowano w tym celu ograniczenie globalne *alldifferent*, które sprawdza, czy elementy tablicy podanej jako argument nie mają powtarzających się wartości.

Druga część ograniczeń to zapis informacji zawartych w treści zagadki (każdy punkt odpowiada jednej linii ograniczeń, w nawiasach podano informację, do której wartości w tablicy odnosi się dane ograniczenie):

- Brytyjczyk (A[3]) mieszka w czerwonym (B[3]) domu, znak równości oznacza, że obie dane dotyczą tego samego domu.
- Szwed (A[4]) posiada psa (C[3]).
- Duńczyk (A[2]) pija herbatę (D[2]).
- Zielony (B[4]) dom znajduje się bezpośrednio po lewej stronie domu białego (B[5]), jeśli do numeru domu doda się *1* oznacza to dom po jego prawej stronie, odjęcie *1* oznacza dom po jego lewej stronie.
- Mieszkaniec zielonego (B[4]) domu pija kawę (D[1]).
- Palący papierosy *Pall-Mall* (E[3]) hoduje ptaka (C[2]).
- Osoba mieszkająca w środkowym (3 – numer domu) domu pija mleko (D[3]).
- W żółtym (B[1]) domu pali się papierosy marki *Dunhill* (E[1]).
- Pierwszy (1 – numer domu) dom jest zamieszkany przez Norwega (A[1]).
- Palacz papierosów *Marlboro* (E[2]) jest sąsiadem hodowcy kota (C[1]), co oznacza, że numer domu hodowcy kota jest o *1* większy lub mniejszy niż numer domu, w którym pali się papierosy *Marlboro*.

- Hodowca koni (C[4]) mieszka obok osoby palącej papierosy marki *Dunhill* (E[1]).
- Palący papierosy *Bluemaster* (E[4]) pija sok (D[4]).
- Dom o numerze 2 jest koloru niebieskiego (B[2]).
- Niemiec (A[5]) pali papierosy *Prince* (E[5]).
- Sąsiad osoby palącej *Marlboro* (E[2]) pije wodę (D[5]).

Ostatnim elementem modelu jest element wyjścia. Wynik zwrócony po rozwiązaniu modelu składa się z Tablic od *A* do *E*, których wartości elementów wskazują numer domu, którego dotyczy dana informacja (na informację wskazuje pozycja numeru w tablicy). Użyta w elemencie wyjścia funkcja *fix* sprawdza, czy zmienna decyzyjna ma ustaloną wartość, jeśli tak to zwraca jej wartość, w przeciwnym wypadku przerywa działanie funkcji.

Wynik zwrócony po rozwiązaniu modelu:

```
A: [1, 2, 3, 5, 4]
B: [1, 2, 3, 4, 5]
C: [1, 3, 5, 2, 4]
D: [4, 2, 3, 5, 1]
E: [1, 2, 3, 5, 4]
The "German" owns the fish
```

Na końcu widnieje odpowiedź na pytanie zadane w opisie problemu, osoba hodująca rybki to Niemiec. Sposób odczytania tabeli można przedstawić na przykładzie domu o numerze 4: dom jest zamieszkiwany przez Niemca (*4* na ostatnim miejscu w tablicy oznaczającej narodowości), kolor domu to zielony (*4* na czwartym miejscu w tablicy kolorów), mieszkaniec tego domu hoduje rybki (*4* na ostatniej pozycji tablicy określającej hodowane zwierzę), mieszkaniec domu pija kawę (*4* na pierwszym miejscu tablicy z napojami), Niemiec pali papierosy marki *Prince* (*4* na ostatnim miejscu tablicy z markami papierosów).

5.7. Ciekawe strony internetowe powiązane z MiniZinc

Oficjalna strona języka *MiniZinc*

Link: *www.minizinc.org*

Na stronie można znaleźć informacje dotyczące dotychczasowego rozwoju języka w postaci zmian wprowadzanych w kolejnych wersjach języka oraz MiniZinc

IDE. Możliwe jest pobranie narzędzi niezbędnych do pracy z tym językiem dla systemów: Windows, Linux, Mac OS.

Repozytorium MiniZinc w serwisie *GitHub*

Link: *www.github.com/MiniZinc*

Poza dostępnymi bibliotekami języka MiniZinc oraz kodu źródłowego MiniZinc IDE można tutaj znaleźć takie rzeczy jak przykładowe modele prezentujące różne aspekty języka oraz modele stworzone w celu testowania wydajności różnych solverów.

Kursy w serwisie *Coursera*

Link: *www.coursera.org/learn/basic-modeling*
www.coursera.org/learn/advanced-modeling

Kursy są prowadzone przez Prof. Peter James Stuckey (The University of Melbourne) oraz Prof. Jimmy Ho Man Lee (The Chinese University of Hong Kong). Kursy mają na celu nauczyć tworzenia modeli dla optymalizacji dyskretnej przy użyciu języka MiniZinc.

Strona Hakan Kjellerstrand poświęcona językowi MiniZinc

Link: *www.hakank.org/minizinc/*

Można tam znaleźć wiele modeli autorstwa Hakan Kjellerstrand podzielonych na kategorie, np. zagadki, programowanie liniowe, problemy nieliniowe (zmienne typu *float*).

***The MiniZinc Handbook* w wersji online**

Link: *www.minizinc.org/doc-2.2.1/en/index.html*

Podręcznik MiniZinc zawiera *The MiniZinc Tutorial*, który przedstawia krok po kroku tworzenie modeli w języku MiniZinc, *The MiniZinc User Manual* opisujący MiniZinc IDE i korzystanie z narzędzia *minizinc* w linii poleceń oraz *The MiniZinc Reference Manual* zawierające specyfikacje języka MiniZinc i języka FlatZinc oraz bibliotekę ograniczeń globalnych i wbudowanych funkcji języka MiniZinc.

Zakończenie

Istnieje wiele solverów, które zostały stworzone w celu rozwiązywania problemów programowania z ograniczeniami, jednak z reguły są one dedykowane dla konkretnego języka programowania. Oznacza to, że kiedy pojawi się potrzeba wypróbowania stworzonego modelu na innym solverze konieczne jest przepisanie modelu na język odpowiadający danemu solverowi. Jest to bardzo czasochłonne, kiedy chciałoby się sprawdzić rozwiązania zwrócone przez różne solvery.

Celem przyświecającym podczas tworzenia języka MiniZinc było stworzenie języka, który mógłby być standardem dla pisania modeli, a jednocześnie zminimalizowanie pracy twórców solverów koniecznej do przystosowania solveru dla języka MiniZinc. Rozwiązaniem tego problemu był język FlatZinc, który jest prostym językiem, niezależnym od solvera, do tworzenia modeli problemów programowania z ograniczeniami. FlatZinc nie wymaga dużego nakładu pracy w celu implementacji w solverze wsparcia dla tego języka. Biorąc pod uwagę liczną grupę solverów, w których zaimplementowano interfejs dla języka FlatZinc można wysnuć wniosek, że język MiniZinc, którego modele są kompilowane do języka FlatZinc, ma szansę stać się standardem dla programowania z ograniczeniami.

Język MiniZinc jest na wystarczającym poziomie abstrakcji, aby pozwolić na modelowanie większości problemów z zakresu programowania z ograniczeniami, jednak nie jest na tyle wysokopoziomowym językiem, aby uniemożliwić jego zmapowanie na języki różnych solverów. Zważając na główne cechy języka MiniZinc, tzn. jego prostotę, ekspresyjność wynikającą z poziomu abstrakcji oraz łatwość w implementacji wsparcia dla tego języka u solverów, jest on dobrą próbą stworzenia języka, który mógłby być przyjęty jako standard.

Bibliografia

[1] *Constraint Programming -- the Paradigm to Watch*
Mark Wallace; 1:7--13, 2007.
[online] http://constraintprogramming.com/letters/Papers/v1/wallace.pdf
[03.12.2018]

[2] www.wikipedia.org/wiki/Declarative_programming

[3] www.wikipedia.org/wiki/Sketchpad

[4] www.wikipedia.org/wiki/Constraint_programming

[5] *6 Building Industrial Applications with Constraint Programming* (2001)
Simonis Helmut
[online]https://pdfs.semanticscholar.org/e151/28cc77daecf3332156f4f534120f1fc2f8b7.pdf [03.12.2018]

[6] *MiniZinc: Towards A Standard CP Modelling Language*
Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand,
Gregory J. Duck and Guido Tack
[online] https://www.comp.nus.edu.sg/~gregory/papers/cp07.pdf [03.12.2018]

[7] http://www.gecode.org/flatzinc.html

[8] https://github.com/chuffed/chuffed

[9] http://www.choco-solver.org/

[10] http://eclipseclp.org/

[11] http://ie.technion.ac.il/~ofers/HCSP/index.html

[12] http://jacop.osolpro.com/

[13] https://dtai.cs.kuleuven.be/software/minisatid

[14] https://github.com/ehebrard/Mistral-2.0

[15] http://code.google.com/p/or-tools

[16] http://www.it.uu.se/research/group/astra/software

[17] http://picat-lang.org/

[18] http://www.sics.se/isl/sicstuswww/site/index.html

[19] http://scip.zib.de/

[20] https://github.com/informarte/yuck

[21] The MiniZinc Handbook
Peter J. Stuckey, Kim Marriott, Guido Tack
[online] https://www.minizinc.org/doc-2.2.1/en/index.html [03.12.2018]

[22] *Specification of MiniZinc*
Nicholas Nethercote, Kim Marriott, Reza Rafeh, Mark Wallace,
Maria Garcia de la Banda
[online] https://www.minizinc.org/doc-2.2.1/en/spec.html [03.12.2018]

[23] www.sourceforge.net/projects/jacop-solver/

[24] www.github.com/radsz/jacop