UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI
MATEMATICA E INFORMATICA

Corso di Laurea in Informatica

Tesi di Laurea

Implementazione di specifiche MiniZinc tramite il linguaggio Java esteso con la libreria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Davide Allevi

Anno Accademico 2014/2015

Ai miei genitori,
Rita e Ignazio

Indice

Capitolo 1

Il linguaggio MiniZinc

MiniZinc è un linguaggio di modellazione sviluppato da NICTA in collaborazione con Univ. di Melbourne/Monash.[2, 6, 7]
In questo capitolo verrà introdotta una breve panoramica su questo linguaggio di modellazione.

1.1 Tipi di dato

I tipi permessi per le variabili sono:

- **Integer** (*int* o range *1...n* o *set of int*):
  - Gli Integer rappresentano i numeri interi.
  - Possono essere fixed o unfixed.
  - Gli interi *fixed* sono scritti come **int** o **par int** invece quelli *unfixed* sono scritti come **var int**.
  - Una variabile intera fixed deve essere inizializzata a *instance-time*; mentre non è così per una variabile unfixed.
- **Float** (*float* o range *1.0..f* o *set of float*):
  - I Float rappresentano i numeri reali.

  - Possono essere fixed o unfixed.

  - I float *fixed* sono scritti come **float** o **par float** invece quelli *unfixed* sono scritti come **var float**.

  - Una variabile float fixed deve essere inizializzata a *instance-time*; mentre non è così per una variabile unfixed.

- **Boolean** (*bool*):

  - I Boolean rappresentano il true o il false.

  - Possono essere fixed o unfixed.

  - I Boolean *fixed* sono scritti come **bool** o **par bool** invece quelli *unfixed* sono scritti come **var bool**.

  - Una variabile Boolean fixed deve essere inizializzata a *instance-time*; mentre non è così per una variabile unfixed.

- **String**(*string*):

  - Non possono essere delle decision variable.

  - Le string sono primitive.

  - Le espressioni di stringa vengono usate nelle asserzioni, output e annotazioni.

  - Le string devono essere fixed.

  - Le string *fixed* sono scritte come **string** o **par string**.

  - Una variabile string (che può essere solo fixed) deve essere inizializzata a *instance-time*.

- **Array** (*array[range] of type*):

  - Gli *array* in MiniZinc sono mappe da interi fixed a valori.
  - I *valori* possono essere di qualsiasi tipo.
  - *Array-di-array* non sono permessi.
  - Tutti gli array sono *mono-dimensionali*, i multi-dimensionali possono essere simulati usando una tupla come indice.
  - La dimensione di un array deve essere *fixed* mentre i suoi elementi possono essere fixed o unfixed.
  - Una variabile array esplicitamente indicizzata deve essere inizializzata a instance-time solo se i suoi elementi vengono inizializzati a instance-time.
  - Una variabile array implicitamente indicizzata deve essere inizializzata a instance-time, così che siano noti la sua lunghezza e i suoi indici.

- **Set** (*set of type*):
  - Un *set* è una collezione senza duplicati.
  - I set possono contenere elementi di qualsiasi tipo e possono essere *fixed* o *unfixed*.
  - Se un set è unfixed, i suoi elementi devono essere finiti.
  - Una variabile set fixed deve essere inizializzata a *instance-time*; mentre non è così per una variabile unfixed.[5]

1.2 Variabili

In MiniZinc ci sono due tipi di variabili: i *Parameter* che sono **fixed** e le *Decision variable* che sono **unfixed**.

1.2.1 Parameter

I *parameter* sono come le variabili nei normali linguaggi di programmazione, i cui valori sono fissati a instance-time (non durante il solving process).
A differenza delle variabili in molti linguaggi di programmazione, un parameter può solo essere assegnato ad un singolo valore non modificabile ovvero non può avere più assegnamenti.
I tipi di queste variabili sono: integer(*int*), numeri in floating point(*float*), Boolean(*bool*) e stringhe(*string*), ma abbiamo anche gli *array* e i *set*. Quindi queste variabili sono dichiarate con un tipo (o a un range/set) e possono anche essere introdotte da **par** come parola chiave.
La sintassi di una dichiarazione di parameter è la seguente:

$$\langle type - inst - expr \rangle \text{: } \langle variable \rangle = \langle expression \rangle;$$

dove:

- *type−inst−expr*: rappresenta il modo di istanziazione che in questo caso è **par** e il tipo della variabile. Se l'istanziazione non è esplicita, essendo opzionale, allora la variabile sarà un parameter
- *variable*: è il nome della variabile
- *expression*: è il valore assegnato alla variabile

Esempio di una dichiarazione di variabile parameter:

int: a = 3;

In modo equivalente anche:

par int: a = 3;

1.2.2 Decision variable

Le *decision variable* sono come le variabili in matematica. A differenza dei parameter e delle variabili in un linguaggio di programmazione standard, le decision variable non necessitano di essere dichiarate per forza con un certo valore. La differenza principale tra questi due tipi di variabili è data dall'*istanziazione*.

Per ogni decision variable si cerca di dare un insieme di possibili valori che le variabili possono assumere, che viene definito *dominio*.
Quest'ultimo può essere dato come parte della dichiarazione della variabile e il tipo delle decision variable è dedotto dal tipo di valori nel dominio.
I tipi di queste variabili sono: integer (*int*), numeri in floating point (*float*), Boolean (*bool*) e *set*, ma sono supportati anche gli *array* i cui elementi sono decision variable.

Quindi le decision variable sono dichiarate con un tipo e vengono introdotte da **var** come parola chiave.

Tipicamente queste decision variable sono dichiarate usando un range o un set che definiscono il dominio della variabile.
La sintassi di una *decision variable* è la seguente:

$$\langle type - inst - expr \rangle\text{: } \langle variable \rangle \text{ [= } \langle expression \rangle \text{]};$$

dove:

- *type−inst−expr*: rappresenta il modo di istanziazione che in questo caso è **var** e il tipo della variabile.
- *variable*: è il nome della variabile
- *expression*: è un eventuale valore assegnato alla variabile

Esempio di una decision variable con dominio:

```
var 1..3: a;
```

dove:

- *1..3*: è il dominio della variabile

- *a*: è il nome della variabile

  Il tipo della variabile in questo esempio sarà integer poichè nel dominio sono presenti valori interi, quindi le decision variable presenti nel modello saranno integer.

1.3 Espressioni aritmetiche

- Operatori aritmetici standard:
  - Integer: * div mod + -
  - Float: * / + -

MiniZinc non fa il casting automatico da integer a float. Il casting deve essere fatto esplicitamente con la funzione **int2float(intexp)**.

- Operatori aritmetici relazionali:
  ==(oppure =) != > < >= <=

1.4 Output

Uno statement di output ha la seguente forma:

output [⟨*string expression*⟩, ... , ⟨*string expression*⟩];

Esempio:

output [″wa=″, show(wa),...];

dove:
show(wa) stampa il valore di wa dopo aver eseguito il modello.

Quindi uno statement di output è seguito da una lista di string.
Quest'ultime sono tipicamente:

- String literal, le quali sono scritte all'interno di doppie virgolette (″ ″)

  oppure

- Un espressione della forma *show*(X), dove X è il nome di una decision variable o di un parameter.

I caratteri speciali sono introdotti da backslash come ad esempio $\backslash n$ e $\backslash t$ che rappresentano rispettivamente la newline e un tab.
Le string literal devono entrare in una singola riga, mentre le stringhe più lunghe possono essere divise tra linee multiple usando l'operatore di concatenazione delle stringhe.

Per esempio la string literal:

Invalid datafile : Amount of flour is non-negative

è equivalente all'espressione:

Invalid datafile: ++
Amount of flour is non-negative.

Un modello può contenere al massimo uno statement di output.

1.5 Constraint

I *constraint* sono il nucleo del modello MiniZinc.
Un constraint può essere una qualunque espressione booleana costruita utilizzando i letterali Booleani *true*,*false* e gli operatori logici: $\bigwedge$,$\bigvee$,$\leftarrow$, $\rightarrow$,$\leftrightarrow$.
Un constraint ha la seguente forma:

constraint ⟨*Boolean expression*⟩;

Esempio:

```
int: nc = 3;
var 1..nc: wa;
var 1..nc: nt;
constraint wa != nt;
```

dove:

- nc: è una variabile parameter
- wa e nt: sono due decision variable
- wa != nt: è il constraint che risulta vero se le due decision variable *wa* e *nt* sono diverse tra loro.

Quindi un constraint è una relazione che deve essere valida per la risoluzione del problema.

1.5.1 Constraint di confronto

Relazioni:

- Uguaglianza:

  X==Y oppure (X=Y)

  dove:

  - X e Y sono decision variable oppure espressioni numeriche.
  - Il constraint risulta vero se e solo se il valore di X è uguale a quello di Y.

- Disuguaglianza:

  X!=Y

  dove:

  - X e Y sono decision variable oppure espressioni numeriche.
  - Il constraint risulta vero se e solo se il valore di X è diverso da quello di Y.

Operatori di confronto:

- $<$

  $$X<Y$$

  dove:

  - X e Y sono decision variable oppure espressioni numeriche.
  - Il constraint risulta vero se e solo se il valore di X è minore di quello di Y.

- $\leq$

  $$X\leq Y$$

  dove:

  - X e Y sono decision variable oppure espressioni numeriche.
  - Il constraint risulta vero se e solo se il valore di X è minore o uguale a quello di Y.

- $>$

  $$X>Y$$

  dove:

  - X e Y sono decision variable oppure espressioni numeriche.
  - Il constraint risulta vero se e solo se il valore di X è maggiore di quello di Y.

- $\geq$

  $$X\geq Y$$

  dove:

  - X e Y sono decision variable oppure espressioni numeriche.
  - Il constraint risulta vero se e solo se il valore di X è maggiore o uguale a quello di Y.

1.5.2 Constraint assert

MiniZinc fornisce i constraint assert che permettono di controllare la validità dei valori delle variabili.
La forma è **assert(Boolexp, Stringexp)** e restituisce true se *Boolexp* è vera, altrimenti stampa *Stringexp* come messaggio di errore.
Esempio:

```
constraint assert(x > 0, "Errore: x =< 0")
```

Si noti che l'espressione assert è un'espressione booleana e per questo motivo è considerato come un tipo di constraint.

1.5.3 Global Constraint

MiniZinc include una libreria di *global constraint*.
Per utilizzare questa libreria è sufficiente includere all'inizio del modello la voce con il global constraint che si vuole usare. Ad esempio se si vuole utilizzare il global constraint *alldifferent* verrà incluso nel modello nel seguente modo:

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

Alcuni esempi di global constraints sono:

- **Alldifferent**: è un constraint che prende un array di variabili e impone che i valori fra loro siano diversi.
  La sintassi del constraint *alldifferent* ha la seguente forma:

  ```
  alldifferent(array[int] of var int: x)
  ```

- **Cumulative**:
  La sintassi del constraint *cumulative* ha la seguente forma:

  ```
  cumulative(array[int] of var int: s, array[int] of var int: d,
  array[int] of var int: r, var int: b)
  ```

  dove:

  - s sono gli istanti di inizio di attività
  - d sono le durate

  - r sono le richieste di risorse
  - b il limite di capacità delle risorse

  Dato l'intervallo [min, max] dove min = $\min_i$ { s_i },
  max = max{ s_i+d_i } -1, il vincolo *cumulative* assicura che
  max{ $\sum_j$: $s_j \leq i \leq s_j$ r_i } $\leq$ b

- **Table**: è un constraint il quale impone che una tupla di variabili prenda un valore da un insieme di tuple. Poiché non ci sono tuple in MiniZinc questo è codificato utilizzando gli array.
  La sintassi del constraint *table* ha la seguente forma:

table(**array[int] of var bool: x, array[int, int] of bool: t**) *table*(**array[int] of var int: x, array[int, int] of int: t**)

  a seconda che le tuple siano Boolean o integer.

1.6 Solve

Ogni modello deve avere esattamente un *solve*, il quale specifica che tipo di soluzione viene cercata.
I solve hanno la seguente sintassi:

- **solve satisfy**: indica che si tratta di un problema di soddisfacimento: si cerca di trovare un valore per le decision variable che soddisfi i constraint.
- **solve maximize** ⟨*expr*⟩: specifica che si vuole trovare una soluzione che massimizzi l'espressione ⟨*expr*⟩ nello statement solve chiamato *goal(objective)*. L'objective può essere un qualsiasi tipo di espressione aritmetica.
- **solve minimize** ⟨*expr*⟩: specifica che si vuole trovare una soluzione che minimizzi l'espressione ⟨*expr*⟩ nello statement solve chiamato sempre *goal(objective)*.
  Anche in questo caso l'objective può essere un qualsiasi tipo di espressione aritmetica.

Esempi:

```
solve satisfy;
solve maximize a*x + y - 3*z;
```

I solve determinano se il modello rappresenta un problema di soddisfacimento di vincoli o un problema di ottimizzazione.
Nei solve minimize/maximize le espressioni possono avere tipo *integer* o *float*.

1.7 Datafile

I dati di input per un modello possono essere caricati da console oppure da un file di dati (*datafile*).
Un problema può nascere quando si vogliono modificare i dati dei constraint riusando lo stesso modello usato fino a quel momento, questo problema viene risolto grazie ai *datafile*.
MiniZinc, come la maggior parte degli altri linguaggi di modellazione, consente l'uso dei datafile per impostare il valore dei parametri dichiarati nel modello originale e questo consente al modello stesso di essere più facilmente usato con differenti dati, eseguendolo con differenti datafile.
I datafile devono avere **.dzn** come estensione.
Qui sotto ora vedremo la differenza fra un modello senza e con datafile.

Nel seguente esempio si vogliono cucinare dei dolci per una festa, cercando in base alla quantità e al costo degli ingredienti, di massimizzare il profitto di dolci da infornare.

Esempio di modello senza datafile con i dati caricati da console:

```
% Baking cakes for the school fete (senza datafile)

% dichiaro due decision variable b e c

var 0..100: b;      % no. of banana cakes
var 0..100: c;      % no. of chocolate cakes

% vincoli per ogni ingrediente

% flour
constraint 250*b + 200*c ≤ 4000;

    % banana
constraint 2*b ≤ 6;

    % sugar
constraint 75*b + 150*c ≤ 2000;

    % butter
constraint 100*b + 150*c ≤ 500;

    % cocoa
constraint 75*c ≤ 500;

% maximaze our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = ", show(b),
"no. of chocolate cakes = ", show(c), ];
```

Usando il seguente comando verrà eseguito il modello:

```
$ mzn-g12fd programma.mzn
```

che stampa il seguente risultato:

```
no. of banana cakes = 2, no. of chocolate cakes = 2
```

Esempio di modello con datafile:

% Baking cakes for the school fete (con datafile)

(Fuori dal modello)

datafile.dzn ≡

flour = 4000;
banana = 6;
sugar = 2000;
butter = 500;
cocoa = 500;

(Dentro il modello)

% dichiaro le variabili parameter alla quale verrà assegnato il valore corrispondente nel datafile
int : flour; % no. grams of flour available
int : banana; % no. grams of bananas available
int : sugar; % no. grams of sugar available
int : butter; % no. grams of butter available
int : cocoa; % no. grams of cocoa available

N.B: stessi nomi usati nel datafile

% controllo che i valori della variabili siano non negativi

constraint assert(flour ≥ 0,″Invalid datafile: ″ ++ ″Amount of flour is non-negative″);
constraint assert(banana ≥ 0,″Invalid datafile: ″ ++ ″Amount of banana is non-negative″);
constraint assert(sugar ≥ 0,″Invalid datafile: ″ ++ ″Amount of sugar is non-negative″);
constraint assert(butter ≥ 0,″Invalid datafile: ″ ++ ″Amount of butter is non-negative″);
constraint assert(cocoa ≥ 0,″Invalid datafile: ″ ++ ″Amount of cocoa is non-negative″);

% dichiaro due decision variable *b* e *c*

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% vincoli per ogni ingrediente

% flour
constraint 250*b + 200*c ≤ flour;

% bananas
constraint 2*b ≤ banana;

% sugar
constraint 75*b + 150*c ≤ sugar;

% butter
constraint 100*b + 150*c ≤ sugar;

% cocoa
constraint 75*c ≤ cocoa;

% **maximaze** our profit
solve maximize 400*b + 450*c;

output [″no. of banana cakes = ″, show(b),
″no. of chocolate cakes = ″, show(c),];

Usando il seguente comando verrà eseguito il modello usando il datafile:

```
$ mzn-g12fd programma.mzn datafile.dzn
```

che stampa lo stesso risultato del modello senza datafile, ma con la possibilità di andare a modificare solo il datafile senza andare a toccare il modello stesso, essendo cosi più facilmente riutilizzabile.

1.8 Strutture dati

In questa sezione verranno introdotte due strutture dati fornite da MiniZinc: gli *array* e i *set*.

1.8.1 Array

MiniZinc fornisce array mono o multi-dimensionali che vengono dichiarati nel seguente modo:

array[$\langle index - set_1 \rangle$, ..., $\langle index - set_n \rangle$] **of** $\langle type - inst \rangle$

L'*index-set* di un array può essere o un range di interi o il nome di una variabile che rappresenta un set di interi.
Esempio:

- (*array con un range di interi*)

```
array[0..5] of int: profit;
```

- (*array con il nome di una variabile che rappresenta un set di interi*)

```
int: nproducts;
set of int: Products = 1..nproducts;
array[Products] of int: profit;
```

Gli array possono avere qualsiasi tipo base: integer, Boolean, float o string; inoltre possono anche contenere set ma non possono contenere array. Gli array literal mono-dimensionali hanno la seguente forma:

[$\langle expr_1 \rangle$, ... , $\langle expr_n \rangle$]

mentre gli array letterali bi-dimensionali hanno la seguente forma:

[| $\langle expr_{1,1} \rangle$, ... , $\langle expr_{1,n} \rangle$, ..., $\langle expr_{m,1} \rangle$, ... , $\langle expr_{m,n} \rangle$ |]

dove m sono le righe e n le colonne.
Inoltre ci sono delle funzioni come *array1d*, *array2d*, etc, che possono essere usate per inizializzare un array di qualsiasi dimensione da una lista.

La seguente sintassi è per chiamare queste funzioni:

array*nd*($\langle index - set_1 \rangle$, ... , $\langle index - set_n \rangle$, $\langle list \rangle$)

ritorna un array n dimensionale, con gli index-set dati dai primi n argomenti e con l'ultimo argomento che contiene gli elementi dell'array.
Gli elementi degli array sono accessibili nel solito modo: *a*[*i*,*j*] prendendo in questo caso l'elemento nell'i-esima riga e nella j-esima colonna.
L'operatore concatenazione $'++'$ può essere usato per concatenare due array mono-dimensionali.
La funzione *length* ritorna la lunghezza di un array mono-dimensionale.

Esempi:

- (*array1d possono essere inizializzati tramite una lista*)

  profit = [400, 450];
  capacity = [4000, 6, 2000, 500, 500];

- (*array2d possono essere inizializzati con una lista contenente* $''$ | $''$ *per separare le righe*)

  consuption = [| 250, 2, 75, 100, 0, | 200, 0, 150, 150, 75 |];

  equivalente:

  consuption = **array2d**(1..2, 1..5, [250,2,75,100,0,200,0,150,150,75])

- (*operatore di concatenazione*)

  [4000, 6] ++ [2000, 500, 500] = [4000, 6, 2000, 500, 500]

1.8.2 Set

I *set* sono dichiarati con la seguente forma:

set of $\langle type - inst \rangle$

dove sono ammessi solo set di *integer*, *float* or *Boolean*.

I set literal hanno la seguente forma:

{ $\langle expr_1 \rangle$,..., $\langle expr_n \rangle$ }

o sono un range di espressioni su integer o float che hanno la forma:

$\langle expr_1 \rangle$ .. $\langle expr_2 \rangle$

Le operazioni standard sui set sono: **in** (appartenenza elemento), **subset, superset** (contenimento su set), **union** (unione), **intersect** (intersezione), **diff** (differenza su set), **symdiff** (differenza simmetrica su set), **card** (cardinalità di un set).

Esempi:

- (unione A ∪ B)

```
% insieme S
set of int: S;

% insiemi A = {1,2} e B = {7,4}
set of int: A = 1..2;
set of int: B = {7,4};

% unione A U B
S = A union B;

solve satisfy;

output["S=",show(S)];
```

Output

S={1, 2, 4, 7}

- (intersezione A ∩ B)

```
% insieme S
set of int: S;

% insiemi A = {1,2} e B = {7,4}
set of int: A = 1..2;
set of int: B = {7,4};

% intersezione
S = A intersect B;


solve satisfy;

output["S=",show(S)];
```

Output

S={}

- (unione A \ B)

```
% insieme S
set of int: S;

% insiemi A = {1,2} e B = {7,4}
set of int: A = 1..2;
set of int: B = {7,4};

% differenza A \ B
S = A diff B;

solve satisfy;

output["S=",show(S)];
```

Output

S={1..2}

- (appartenenza A $\in$ B)

```
% boolean x e una variabile y inizializzata a 1
bool: x;
int: y = 1;

% insiemi A = {1,2} e B = {7,4}
set of int: A = 1..2;
set of int: B = {7,4};

% appartenenza
x = y in A;

solve satisfy;

output["x=",show(x)];
```

Output

x=true;

Se fosse stato $x = y \; in \; B$ allora nell'output avrebbe dato x=false poichè 1 non appartiene all'insieme B contenente solo 7 e 4.

1.9 Comprehension

MiniZinc implementa le comprehension per specificare insiemi di valori ammissibili. Qui sotto verranno introdotte le *list comprehension* e le *set comprehension*.

La generica forma di una *list comprehension* è la seguente:

$$[\ \langle expr \rangle\ |\ \langle generator - exp \rangle\ ]$$

dove:

- $\langle expr \rangle$ specifica come costruire gli elementi della lista di output dagli elementi generati da $\langle generator - exp \rangle$.
- $\langle generator - exp \rangle$ consiste in una sequenza di espressioni di generatori, separati da una virgola e seguiti se si vuole da una espressione Booleana.

$\langle generator \rangle$, ..., $\langle generator \rangle$
$\langle generator \rangle$, ..., $\langle generator \rangle$ **where** $\langle bool - exp \rangle$

dove $\langle bool - exp \rangle$ nella seconda forma, funge da filtro sull'espressione di generatore, quindi solo gli elementi che soddisfano l'espressione Booleana sono usati per costruire gli elementi nella lista di output.
Un generatore $\langle generator \rangle$ ha la seguente forma:

$$\langle identifier \rangle, ..., \langle identifier \rangle\ \textbf{in}\ \langle array - exp \rangle$$

Ogni *identifier* è un iteratore che assume i valori dell'espressione array a sua volta, con l'ultimo identificatore che varia più rapidamente.

I *set comprehension* sono quasi identici alle list comprehension, la sola differenza è l'uso delle ′ { ′ e ′ } ′ per includere le espressioni piuttosto che ′ [′ e ′] ′.

Esempi:

- (*esempio di list comprehension*)

  [i + j | i, j in 1..3 *where* j < i]

  restituisce:

  [1 + 2, 1 + 3, 2 + 3] che è [3, 4, 5]

- (*esempio di set comprehension*)

  {i + j | i, j in 1..3 *where* j < i}

  restituisce:

  {1 + 2, 1 + 3, 2 + 3} che è {3, 4, 5}

1.10 Aggregation function

MiniZinc fornisce diverse funzioni per iterare su un array o un set, che si chiamano *aggregation function*.
Le aggregation function per gli array aritmetici sono:

- **sum**: ritorna la somma degli elementi dell'array.
  Sintassi:

  (*con array*)

function int: **sum**(*array [$T] of int: X*)

  (*con array di variabili logiche*)

function var int: **sum**(*array [$T] of var int: X*)

- **product**: ritorna il prodotto degli elementi dell'array.
  Sintassi:

  (*con array*)

function int: **product**(*array [$T] of int: X*)

  (*con array di variabili logiche*)

function var int: **product**(*array [$T] of var int: X*)

- **min**: ritorna l'elemento minimo dell'array.
  Sintassi:

  (*con array*)

function $T: **min**(*array [$U] of $T: X*)

  (*con array di variabili logiche*)

function var int: **min**(*array [$U] of var int: X*)

- **max**: ritorna il più grande elemento dell'array.
  Sintassi:

  (*con array*)

function $T: **max**(*array [$U] of $T: X*)

  (*con array di variabili logiche*)

function var int: **max**(*array [$U] of var int: X*)

Esempio:

- (esempio di uso funzione *max*)

```
int nresources;
set of int: Resorces = 1..nresources
array[Resources] of int: capacity:
array[Resources] of var 0..max(capacity): used;
```

Quando vengono applicate su un array vuoto, *min* e *max* danno un errore run-time mentre *sum* ritorna 0 e *product* ritorna 1.

MiniZinc fornisce anche alcune aggregation function per gli array contenenti espressioni Booleane:

- **forall**: ritorna un singolo constraint che rappresenta la congiunzione logica dei constraint presenti all'interno del forall quindi garantisce che tutti i vincoli siano soddisfatti;
- **exists**: ritorna la disgiunzione logica dei constraint quindi exists garantisce che almeno uno dei vincoli sia soddisfatto;

MiniZinc ha una speciale sintassi per certi tipi di chiamate di espressioni, che rendono il modello molto più leggibile chiamate: **generator call expression** con la seguente sintassi:

$$\langle agg - func \rangle \ (\ \langle generator - exp \rangle\ )\ (\ \langle exp \rangle\ )$$

dove le parentesi tonde intorno a $\langle generator - exp \rangle$ $\langle exp \rangle$ non sono opzionali. Questa sintassi può anche essere riscritta in modo equivalente nel seguente modo:

$$\langle agg - func \rangle \ (\ [\ \langle expr \rangle \mid \langle generator - exp \rangle\ ]\ )$$

con $\langle agg - func \rangle$ che può essere una qualsiasi *aggregation function* descritta precedentemente.

Esempio:

- (esempio di uso *forall* con la sintassi generator call expression)

  forall([a[i] != a[j] | i,j **in** 1..3 **where** i < j])

  equivalente a:

  forall(i,j **in** 1..3 **where** i < j) (a[i] != a[j])

  dove a è un array aritmetico con l'index set 1..3.
  La list comprehension restituisce:

  [a[1] != a[2], a[1] != a[3], a[2] != a[3]]

  e la funzione forall ritorna la congiunzione logica:

  [a[1] != a[2] $\wedge$ a[1] != a[3] $\wedge$ a[2] != a[3]].

In questo modo si verifica che gli elementi dell'array a siano differenti.

1.11 If-then-else-endif

MiniZinc fornisce un' espressione condizionale: *if-then-else-endif*.

if $\langle boolexp \rangle$ **then** $\langle exp_1 \rangle$ **else** $\langle exp_2 \rangle$ **endif**

viene valutata $\langle exp_1 \rangle$ se l'espressione Booleana è vera, $\langle exp_2 \rangle$ altrimenti. Il tipo dell'espressione condizionale è quello di $\langle exp_1 \rangle$ e $\langle exp_2 \rangle$ che devono avere lo stesso tipo. L'espressione Booleana non ammette decision variable, ma solo parameter.

Esempi:

- *(if-then-else-endif)*:

```
int: r = if y != 0 then x div y else 0 endif;
```

assegna alla variabile r, x diviso y se y è diverso da zero, altrimenti restituisce zero.

1.11.1 Fix

La funzione **fix** consente, solo nell'output, di controllare che il valore di una decision variable sia fisso e converte quest'ultima in un parameter.

- (esempio d'uso funzione *fix*):

```
set of int: Colors = 1..3;

% dichiaro tre variabili parameters
int: red = 1;
int: yellow = 2;
int: blue = 3;

% creo un array inizializzato di string chiamato name
array[Colors] of string: name = ["red","yellow","blue"];

% dichiaro una decision variable di nome x e di dominio Colors
var Colors: x;

% vincolo che x deve essere diverso da red
constraint x != red;

output [ name[fix(x)] ];
```

come mostrato qui sopra, nello statement di output possiamo stampare il colore usando l'array *name* e utilizzando la funzione *fix*, che ritorna il valore della decision variable.

Capitolo 2

JSetL

JSetl è una libreria Java che combina il paradigma di programmazione orientato agli oggetti con i concetti dei linguaggi CLP (*Constraint Logic Programming*), come variabili logiche, unificazione, risoluzione di vincoli, non determinismo.
Come caratteristiche più interessanti troviamo le variabili logiche (**LVar, IntLVar, LSet, LList**), i vincoli (**Constraint**) e il risolutore di vincoli (**SolverClass**) che permette la risoluzione del problema. [1, 3, 4]

2.1 Variabili logiche

In JSetl, una variabile logica è un'istanza della classe *LVar*.
Questa classe fornisce costruttori per creare variabili logiche e un numero di semplici metodi per testare e utilizzare variabili logiche.
Una variabile logica è creata dalla seguente chiamata:

LVar NomeVar = **new** *LVar*(extName, VarValue);

dove:

- *NomeVar* è il nome della variabile;
- *extName* è il nome opzionale esterno;
- *VarValue* è un parametro opzionale che ne rappresenta il valore.

Quando il dominio di una variabile è ristretto ad un singolo valore, si dice che la variabile è *bound*; in tutti gli altri casi viene definita *unbound*.

2.2 IntLVar

Le *IntLVar* sono variabili logiche alle quali viene assegnato un dominio di valori interi che rappresenta l'insieme di valori che la singola variabile può assumere.
Le IntLVar possono essere create in due modi principali:

IntLVar NomeVar = **new** *IntLVar*();

dove:
NomeVar è il nome della variabile alla quale viene associato un dominio contenente tutti gli interi.

IntLVar NomeVar = **new** *IntLVar*(a,b);

dove:

- *a* è un numero intero;
- *b* è un numero intero;
- a $\leq$ b;

In questo caso viene creata una variabile IntLVar *NomeVar* con associato il dominio:

$$\{x \in Z \mid a \leq x \leq b\}$$

Quando il dominio è ristretto ad un singolo elemento la variabile diventa *bound*; in tutti gli altri casi è *unbound*.

2.3 LList

Una *lista logica l* è un tipo speciale di variabile logica il cui valore è una coppia $\langle elems, rest \rangle$ dove:

- *elems*: è una lista $[e_0,...,e_n]$, n ≥ 0, di oggetti di tipi arbitrari.
- *rest*: è una l-list vuota oppure una l-list unbound che rappresenta il resto di l.

In JSetL, una lista logica è definita come un'istanza della classe *LList*, che estende la classe *LCollection*.
Una lista logica può essere creata in due modi principali:

LList NomeLista = **new** *LList*();

dove:
NomeLista è il nome della lista unbound.

LList NomeLista = **new** *LList*(extName, ListValue);

dove:

- *NomeLista* è il nome della lista;
- *extName* è il nome opzionale esterno;
- *ListValue* è un parametro opzionale che ne rappresenta il valore e deve essere di tipo *List*.

2.4 LSet

Un *insieme logico s* (più semplicemente un *l-set s*) è un tipo speciale di variabile logica il cui valore è una coppia $\langle elems, rest \rangle$ dove:

- *elems*: è un insieme e_0,...,e_n, n $\geq$ 0, di oggetti di tipi arbitrari.
- *rest*: è un l-set vuoto oppure un l-set unbound che rappresenta il resto di s.

In JSetL, un insieme logico è definito come un'istanza della classe *LSet*, che estende la classe *LCollection*.
I metodi forniti dalla classe *LSet* sono fondamentalmente gli stessi di quelli della classe *LList* ma applicati su oggetti LSet.
Un insieme logico può essere creato in due modi principali:

LSet NomeInsieme = **new** *LSet*();

dove:
NomeInsieme è il nome dell'insieme unbound.

LSet NomeInsieme = **new** *LSet*(extName, SetValue);

dove:

- *NomeInsieme* è il nome dell'insieme;
- *extName* è il nome opzionale esterno;
- *SetValue* è un parametro opzionale che ne rappresenta il valore e deve essere di tipo *Set*.

2.5 MultiInterval

La classe *MultiInterval* è un insieme di interi M $\subset$ Z definito da n $\geq$ 0 intervalli I_1,I_2,...,$I_n \in I_\alpha \setminus \emptyset$ tale che:

- M = $I_1 \cup I_2 \cup$.....$\cup I_n$

- M = $I_1 < I_2 <.....< I_n$

dove:

$$[a...b] < [c...d] \overset{def}{\iff} b < c - 1$$

L'insieme del MultiInterval M $\subset$ Z sarà chiamato M_α.

2.6 Constraint

Un *constraint* rappresenta un'operazione che può essere applicata a variabili logiche o a collezioni logiche.
Questa operazione può essere eseguita anche se gli oggetti logici coinvolti non hanno alcun valore preciso ad essi associati.
Un constraint in JSetL è un'espressione che può avere una delle seguenti forme:

- (constraints atomici)
  - il constraint vuoto, denotato [ ]
  - $e_0.op(e_1,...,e_n)$ or $op(e_0,e_1,...,e_n)$ con n=0,..,3
    dove *op* è il nome del constraint e e_i ($0 \leq i \leq 3$) sono espressioni i cui tipi dipendono da *op*.

- (constraint composti)
  - $c_1.and(c_2)$ (congiunzione)
  - $c_1.or(c_2)$ (disgiunzione)
  - $c_1.orTest(c_2)$ (disgiunzione)
  - $c_1.impliesTest(c_2)$ (implicazione)
    dove c_1 e c_2 sono constraints JSetL e *and*,*or*,*orTest*,*impliesTest* rappresentano le congiunzioni logiche ($c_1 \wedge c_2$), disgiunzioni ($c_1 \vee c_2$) e implicazioni ($c_1 \rightarrow c_2$) rispettivamente tra c_1 e c_2.
- (constraint negati)
  - $c_1.notTest()$ (negazione)
    dove c_1 è un JSetL constraint e *notTest()* rappresenta la negazione di c_1 ($\neg\ c_1$).

I Constraints in JSetL sono definiti come un'istanza della classe *Constraint*. Gli oggetti Constraint sono creati utilizzando costruttori e altri metodi di questa classe (es. and, or).

2.7 La classe SolverClass

I Constraint sono risolti usando un *constraint solver*. In JSetl un constraint solver può essere creato come un'istanza della classe **SolverClass**.
I vincoli creati grazie alla classe **Constraint** o attraverso i metodi propri delle variabili logiche vengono passati ad un solver (un'istanza della classe **SolverClass**) che li aggiunge al proprio *constraint store*, che contiene tutti i vincoli attivi, tramite il metodo add:

void add (Constraint c)

La risoluzione dei vincoli avviene utilizzando due metodi della classe **SolverClass**:

void solve()

che fa partire il meccanismo di risoluzione di un vincolo memorizzato nel solver. Se non trova una soluzione il metodo genera un'eccezione del tipo *failure exception*.
Troviamo anche il vincolo *check*:

void check(Constraint c)

che controlla se i constraint C presenti già nel constraint store con il constraint c (C $\wedge$ c) è ancora soddisfaciente o no ritornando true o false.

Capitolo 3

Funzionalità aggiunte in JSetl per MiniZinc

In questo capitolo sono presentate e spiegate le funzionalità aggiunte a JSetL necessarie per poter emulare alcuni meccanismi presenti nel linguaggio di MiniZinc.
In particolare sono presenti operazioni avanzate su array che in Java e JSetL non sono presenti.
Quindi per realizzare queste operazioni verranno usate delle classi dedicate alla manipolazione di array di IntLVar a 1 e 2 dimensioni.

3.1 La classe LArray1D

Questa classe manipola, emulando il comportamento di MiniZinc, gli array di IntLVar monodimensionali.

public static IntLVar[ ] generateArray(int n)

dove:

- n sarà la lunghezza dell'array generato

Questo metodo genera un array monodimensionale di lunghezza n e di elementi di tipo IntLVar a cui non è assegnato un dominio quindi, per definizione il loro dominio è [minint, Maxint].

public static IntLVar[ ] generateArray(int n, int a, int b)

dove:

- n sarà la lunghezza dell'array generato
- a e b rappresentano gli estremi del dominio

Questo metodo genera un array monodimensionale di lunghezza n e con elementi di tipo IntLVar con dominio d di estremi a e b.

public static Vector<IntLVar> arrayToVector(IntLVar[ ] v)

Il metodo restituisce un oggetto della classe Vector contenente una copia di ogni variabile dell'array v.

public static Constraint allIn(IntLVar[ ] a, MultiInterval m)

dove:

- a è un array di variabili IntLVar
- m è un MultiInterval rappresentante un dominio

Questo metodo restituisce un Constraint rappresentante il vincolo:

$\forall$a $\in$ array, a $\in$ m

public static Constraint allDiff(IntLVar[ ] a)

Il metodo restituisce un Constraint rappresentante il vincolo: $\forall$i,j $\in$ [0, array.length() - 1], i $\neq$ j $\Rightarrow$ array[i] $\neq$ array[j]

public static Constraint labelArray(IntLVar[ ] a)

Il metodo restituisce un Constraint che, quando viene analizzato, attiva il meccanismo di labeling su ogni variabile di a.

public static Constraint domArray(IntLVar[] a, MultiInterval r)

È un metodo equivalente a allIn.

public static Constraint sumArray(IntLVar[] a, IntLVar v)

Il metodo restituisce un Constraint contenente il vincolo:

$$v = \sum_{i=0}^{array.length-1} a_i$$

dove:

$\forall$i $\in$ [0, *a.length()* -1], $a_{\text{i}} \in$ array

public static Constraint productArray(IntLVar[] a, IntLVar v)

Il metodo restituisce un Constraint contenente il vincolo:

$$v = \prod_{i=0}^{array.length-1} a_i$$

dove:

$\forall$i $\in$ [0, *a.length()* -1], $a_{\text{i}} \in$ a

public static Constraint maxArray(IntLVar[] a, IntLVar v)

Crea e ritorna il constraint, il quale garantisce che la variabile v contenga il massimo tra tutti gli elementi dell'array a.

public static Constraint minArray(IntLVar[] a, IntLVar v)

Crea e ritorna il constraint, il quale garantisce che la variabile v contenga il minimo tra tutti gli elementi dell'array a.

3.2 La classe LArray2D

Questa classe manipola, emulando il comportamento di MiniZinc, gli array a due dimensione. Metodi della classe:

public static IntLVar[ ][ ] generateArray(int n, int m)

dove:

- n rappresenta il numero di righe della matrice
- m rappresenta il numero di colonne della matrice.

Questo metodo genera una matrice a 2 dimensioni di n righe e m colonne e di tipo IntLVar alla quale non è assegnato un dominio, quindi per definizione il loro dominio è [minint, Maxint].

public static IntLVar[ ][ ] generateArray(int n, int m, MultiInterval d, SolverClass sol)

Questo metodo genera una matrice a 2 dimensioni di n righe e m colonne e di tipo IntLVar, e a sol sono stati aggiunti i constraint che forzano il dominio degli oggetti ad essere uguale a d.

public static IntLVar[ ] extractArray(IntLVar[ ][ ] Mat, int x, int y)

dove:

- *Mat* è una matrice n x m
- x $\in$ [-1,n] $\wedge$ y $\in$ [-1,m]
- x = -1 $\Leftrightarrow$ y $\neq$ -1 $\wedge$ y = -1 $\Leftrightarrow$ x $\neq$ -1

Il metodo restituisce un array monodimensionale contenente:

- se x = -1 $\wedge$ y $\neq$ -1

l'array contiene tutti gli elementi della colonna y

public static IntLVar[] matrixToArray(IntLVar[][] matrix)

Il metodo restituisce un array monodimensionale contenente tutti gli elementi della matrice matrix ordinati per righe.

public static Constraint allDiff(IntLVar[][] matrix)

matrix matrice n x m
Il metodo restituisce un Constraint rappresentante il vincolo:

$$\forall\ x_1,\ x_2 \in [0,n]$$
$$\forall\ y_1,\ y_2 \in [0,m]$$

$x_1 \neq x_2 \vee y_1 \neq y_2 \Rightarrow matrix[x_1][y_1] \neq matrix[x_2][y_1]$.

public static Constraint labelMatrix(IntLVar[][] matrix)

Il metodo restituisce un Constraint che, quando viene analizzato, attiva il meccanismo di labeling su ogni variabile della matrice.

Capitolo 4

Traduzione da MiniZinc a JSetL

Dopo aver descritto nei due capitoli precedenti rispettivamente il linguaggio MiniZinc e la libreria JSetL di Java, in questo capitolo verrà mostrato come tradurre un modello MiniZinc in un programma Java utilizzando la libreria JSetL.

4.1 Struttura del programma Java

Il programma Java ha la seguente struttura:

```
package <Nome Package>

import JSetL.*;
import JSetL.lib.*;
import JSetL.preprocessor.*;
import JSetL.test.*;

public class <NomeProgramma> {

public static SolverClass solver = new SolverClass();

public static void esempio() throws Failure{


/*
creo e inizializzo le variabili
*/
```

```
/*
vengono aggiunti i constraint che verranno risolti dal solver
*/

/*
resto del codice necessario per gestire la risoluzione del
    problema e per stampare o salvare la soluzione.
*/
  }
}

public static void main(String[] args) throws Failure, IOException{

NomeProgramma es = new NomeProgramma();
try {
  es.esempio();
  } catch (Failure e) {

  e.printStackTrace();
  }
```

4.2 Traduzione delle variabili

In questa sezione verrà usata la seguente notazione:

- V per indicare il nome di una variabile in MiniZinc
- $\overline{V}$ per indicare il nome di una variabile tradotta in Java
- E per indicare un'espressione in MiniZinc
- $\overline{E}$ per indicare un'espressione tradotta in Java

4.2.1 Parameter

⟨*type*⟩**:** V = E; *oppure* **par** ⟨*type*⟩**:** V = E;

Traduzione in Java+JSetL:

⟨*type*⟩ $\overline{V}$ = $\overline{E}$;

4.2.2 Decision variable

var *minD..maxD*: V;

Traduzione in Java+JSetL:

IntLVar $\overline{V}$ = **new** IntLVar(″V″,*minD*,*maxD*);

dove:

- V è il nome della variabile
- $''V''$ è il nome esterno della variabile
- *minD*,*maxD* sono rispettivamente il valore minimo e massimo della variabile

Commento:
realizzate soltanto decision variable intere.

4.2.3 Array

Array monodimensionali

array[1..i] **of** ⟨*type*⟩**:** A;

Traduzione in Java+JSetL:

⟨*type*⟩[ ] A = **new** ⟨*type*⟩[i];

Array di decision variable monodimensionali

array[1..i] **of var** minD..maxD: A;

Traduzione in Java+JSetL:

IntLVar[ ] A = **LArray1D**.generateArray(i, minD, maxD);

dove:

- *A* è un nome dell'array
- *i* rappresenta la dimensione dell'array
- *generateArray* è una funzione della classe LArray1D che genera un array monodimensionale di IntLVar
- *minD, maxD* sono rispettivamente il dominio minimo e massimo dell'array

Commento:
realizzate soltanto array di decision variable monodimensionali interi.

Array bidimensionali

array[1..i,1..j] **of** ⟨*type*⟩**:** A;

Traduzione in Java+JSetL:

⟨*type*⟩[ ][ ] A = **new** ⟨*type*⟩[i][j];

Array di decision variable bidimensionali

array[1..i][1..j] **of var** minD..maxD: A;

Traduzione in Java+JSetL:

IntLVar[ ][ ] A = **LArray2D**.generateArray(i, j, minD, maxD);

dove:

- *A* è il nome dell'array
- *i,j* rappresentano la dimensione dell'array, dove *i* sono le righe e *j* sono le colonne
- *generateArray* è una funzione della classe LArray2D che genera un array bidimensionale di IntLVar
- *minD,maxD* sono rispettivamente il dominio minimo e massimo dell'array

Commento:
realizzate soltanto array di decision variable bidimensionali interi.

4.2.4 Set

set of int: S = 1..n;

Traduzione in Java+JSetL:

IntLSet S = **new** IntLSet(1,n);

Commento:
realizzate soltanto set di interi.

4.2.5 Esempio

Modello MiniZinc:

```
% decision variable di nome v
var 1..5: v;

% parameter di nome p
par int: p = 5;

% set di tipo int e di nome S
set of int: S = 1..5;
```

```
% array di tipo int e di nome A inizializzato con [4,3,5,2,1]
array[1..p] of var 1..5: A;

% vincoli
constraint A[1]= 4;
constraint A[2]= 3;
constraint A[3]= 5;
constraint A[4]= 2;
constraint A[5]= 1;

% funzione max che trova il max elemento all'interno dell'array
v = max (i in S) (A[i]);

solve satisfy;

% stampa
output["massimo=", show(v)];
```

Output

massimo = 5

Traduzione in Java+JSetL:

```
class LArrayAggregation {

    public static SolverClass solver = new SolverClass();

    public static void main (String[] args)
    throws IOException, Failure {

  int p = 5;

  IntLVar[] a = LArray1D.generateArray(5);
  a[0].setValue(4);
  a[1].setValue(3);
  a[2].setValue(5);
  a[3].setValue(2);
  a[4].setValue(1);
  IntLVar v = new IntLVar("massimo");
  solver.solve(LArray1D.maxArray(a, v));
  v.output();
      }
}
```

Output

massimo = 5

4.3 Datafile

In MiniZinc i datafile vengono salvati come file con estensione *.dzn* e per essere utilizzati vengono chiamati direttamente nella fase di esecuzione come mostrato nel cap. 1:

(Fuori dal modello)

datafile.dzn ≡

```
x = 4;
y = 6;
z = 2;
```

(Dentro il modello)

```
/*

Inizio programma

*/



% Dichiaro le variabili
int: x;
int: y;
int: z;
```

```
/*

Corpo del programma

*/

% solve
solve satisfy;

% stampa i valori delle variabili presente nel datafile

output [" x = ", show(x),
        " y = ", show(y),
        " z = ", show(z),
];
```

Usando il seguente comando verrà eseguito il modello assegnando ad ogni variabile il valore corrispondente del datafile aggiungendo il comando datafile.dzn:

$ mzn-g12fd programma.mzn **datafile.dzn**

Traduzione in Java+JSetL:

Per creare un datafile in Java+JSetL viene usato un file.txt, dove ad ogni variabile viene associato il proprio valore, dopodichè viene caricato il file.txt nel programma Java con il seguente codice:

(Fuori dal programma)

Esempio di file.txt:

datafile.txt ≡

```
x = 4; // prima riga
y = 6; // seconda riga
z = 2; // terza riga
```

(Dentro il programma)

```
public class <NomeProgramma> {

public static SolverClass solver = new SolverClass();

  // funzione che preso come parametro una stringa ritorna 0
  // se la stringa e' un intero e -1 altrimenti
  public static int isInt(String num){
    try{
      Integer.parseInt(num);
      return 0;
    }catch(Exception e){
      return -1;
    }
  }

  public static void main(String[] args) throws Failure,
      IOException{

    FileReader f;
    f=new FileReader("prova.txt");

    BufferedReader b;
    b=new BufferedReader(f);

    String s;

    // inizializzo un array di tipo int dove verranno salvati i
    // valori interi delle variabili
    int[] val_int = new int[3];

    // inizializzo un array di tipo string dove verranno salvati i
    // nomi delle variabili
    String[] val_str = new String[3];

    // inizializzo array di tipo string dove verranno salvati i
    // nomi delle variabili e i loro valori che verranno poi
    // trasformati in interi
```

```
  String[] stringhe = new String[3];

  int i=0;
  int j=0;

  while(true){

    s=b.readLine(); //legge una riga alla volta

    if(s == null)
      break;

    // assegno all'array parts la stringa divisa grazie alla
    // funzione split che salva le stringhe presenti nel file
    // eliminando = e ;.
    String[] parts = s.split("=|;");
    stringhe[i] = parts[1];

    if(isInt(stringhe[j])== 0)
      val_int[j]=Integer.parseInt(stringhe[j]);
    else if (isInt(stringhe[j])== -1) {
      val_str[j]=stringhe[j];
    }

    i++;
    j++;

  }

  b.close(); // chiude il file

  // assegno ad x,y e z i valori corrispondenti specificati in
     prova.txt
  int x = new Integer(val_int[0]);
  int y = new Integer(val_int[1]);
  int z = new Integer(val_int[2]);

  // stampa
  System.out.println("x = " + x);
  System.out.println("y = " + y);
  System.out.println("z = " + z);
}
```

```
}
```

Output

x = 4
y = 6
z = 2

4.4 Espressioni

In questa sezione si userà la seguente notazione:

- V per indicare una variabile semplice (con valore numerico) o una costante intera in Java
- I per indicare una variabile IntLVar
- P per indicare un parameter
- D per indicare una decision variable
- E per indicare un'espressione in MiniZinc (che può contenere interi o decision variable)
- $\bar{E}$ per indicare un'espressione tradotta in Java

Operazioni:

- *addizione*
  - $P_1 + P_2$

    Traduzione in Java+JSetL:

    $V_1 + V_2$
  - $D_1 + D_2$

    Traduzione in Java+JSetL:

    $I_1.sum(I_2)$

- D + P

  Traduzione in Java+JSetL:

I.*sum*(**new** IntLVar(V))

- P + D

  Traduzione in Java+JSetL:

I.*sum*(**new** IntLVar(V))

- D + E

  Traduzione in Java+JSetL:

I.*sum*($\bar{E}$)

- E + D

  Traduzione in Java+JSetL:

I.*sum*($\bar{E}$)

- E + P

  Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}$ + V

  Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}$.*sum*(V)

- E_1 + E_2

  Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}_1$ + $\bar{E}_2$

Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}_1.sum(\bar{E}_2)$

- *sottrazione*
  - P_1 - P_2

    Traduzione in Java+JSetL:

V_1 - V_2

  - D_1 - D_2

    Traduzione in Java+JSetL:

$I_1.sub(I_2)$

  - D - P

    Traduzione in Java+JSetL:

I.sub(**new** IntLVar(V))

  - P - D

    Traduzione in Java+JSetL:

(I.mul(-1)).sub(**new** IntLVar(V))

  - D - E

    Traduzione in Java+JSetL:

I.$sub(\bar{E})$

  - E - D

    Traduzione in Java+JSetL:

(I.mul(-1)).$sub(\bar{E})$

  - E - P

    Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}$ - V

    Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}.sub$(V)

  - E_1 - E_2

    Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}_1$ - $\bar{E}_2$

    Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}_1.sub(\bar{E}_2)$

- *prodotto*
  - P_1 * P_2

    Traduzione in Java+JSetL:

V_1 * V_2

  - D_1 * D_2

    Traduzione in Java+JSetL:

$I_1.mul(I_2)$

  - D * P

    Traduzione in Java+JSetL:

I.*mul*(**new** IntLVar(V))

  - P * D

Traduzione in Java+JSetL:

(I.mul(**new** IntLVar(V))

– D * E

Traduzione in Java+JSetL:

I.$mul(\bar{E})$

– E * D

Traduzione in Java+JSetL:

(I.$mul(\bar{E})$

– E * P

Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}$ * V

Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}.mul$(V)

– E_1 * E_2

Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}_1$ * $\bar{E}_2$

Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}_1.mul(\bar{E}_2)$

- *divisione*
  - P_1 / P_2 Traduzione in Java+JSetL:

V_1 / V_2

  - D_1 / D_2

    Traduzione in Java+JSetL:

$I_1.div(I_2)$

  - D / P

    Traduzione in Java+JSetL:

I.div(**new** IntLVar(V))

  - P / D

    Traduzione in Java+JSetL:

(I.div(**new** IntLVar(V))

  - D / E

    Traduzione in Java+JSetL:

I.$div(\bar{E})$

  - E / D

    Traduzione in Java+JSetL:

(I.$div(\bar{E})$

  - E / P

    Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}$ / V

Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}.div(\mathrm{V})$

- E_1 / E_2

Traduzione in Java+JSetL (con E espressione di interi):

$\bar{E}_1$ / $\bar{E}_2$

Traduzione in Java+JSetL (con E espressione con decision variable):

$\bar{E}_1.div(\bar{E}_2)$

4.5 Constraint

Vediamo le differenze fra i constraint in MiniZinc e in Java+JSetL.

Notazione:

- X e Y per indicare decision variable oppure espressioni numeriche (in MiniZinc)
- A e B per indicare le corrispondenti variabili logiche oppure espressioni numeriche che restituiscono un IntLVar (in JSetL)

Relazioni:

- *uguaglianza*
  - | **constraint** X = Y; | → | A.eq(B) |

    Esempio MiniZinc:

```
% decision variable X e Y
var 1..2: X;
var 2..3: Y;

constraint X = Y;

solve satisfy;

% stampa
output["X=",show(X), "\t Y=", show(Y),];
```

    Output

    X = 2 Y = 2

Traduzione in Java+JSetL:

```
// A e B due variabili logiche rispettivamente di dominio
    (1,2) e (2,3)
IntLVar A = new IntLVar("A",1,2);
IntLVar B = new IntLVar("B",2,3);

// forza le variabili IntLVar a prendere in modo non
    deterministico uno dei valori del dominio
  solver.add(A.label());
  solver.add(B.label());

// constraint
solver.add(A.eq(B));

solver.solve();

// stampa
A.output();
B.output();
```

Output

A = 2
B = 2

- *disuguaglianza*
  - | **constraint** X != Y; | → | A.neq(B) |

Esempio MiniZinc:

```
% decision variable X e Y
var 1..2: X;
var 2..3: Y;

constraint X != Y;

solve satisfy;

% stampa
output["X=",show(X), "\t Y=", show(Y),];
```

Output

X = 1 Y = 2

Traduzione in Java+JSetL:

```
IntLVar A = new IntLVar("A",1,2);
IntLVar B = new IntLVar("B",2,3);

// forza le variabili IntLVar a prendere in modo non
    deterministico uno dei valori del dominio
  solver.add(A.label());
  solver.add(B.label());

// constraint
solver.add(A.neq(B));

solver.solve();

// stampa
A.output();
B.output();
```

Output

A = 1
B = 2

- *minore*
  - | **constraint** X < Y; | → | A.lt(B) |

Esempio Minizinc:

```
% decision variable X e Y
var 1..2: X;
var 2..3: Y;

constraint X < Y;

solve satisfy;

% stampa
output["X=",show(X), "\t Y=", show(Y),];
```

Output

X = 1 Y = 2

Se invece si prova a mettere Y < X l'output sarà "UNSATISFIABLE" perchè il dominio di Y sarà sempre maggiore o uguale a quello di X.

Traduzione in Java+JSetL:

```
IntLVar A = new IntLVar("A",1,2);
IntLVar B = new IntLVar("B",2,3);

// forza le variabili IntLVar a prendere in modo non
    deterministico uno dei valori del dominio
  solver.add(A.label());
  solver.add(B.label());

// constraint
solver.add(A.lt(B));

solver.solve();

// stampa
A.output();
B.output();
```

Output

A = 1
B = 2

Se invece si prova a mettere B.lt(A) il metodo solve genererà un'eccezione del tipo *failure exception* perchè il dominio di B sarà sempre maggiore o uguale a quello di A mentre se viene usato il metodo check ritornerà *false*.

- *minore o uguale*
  - **constraint** X $\leq$ Y; $\rightarrow$ A.le(B)
- *maggiore*
  - **constraint** X $>$ Y; $\rightarrow$ A.gt(B)
- *maggiore o uguale*
  - **constraint** X $>=$ Y; $\rightarrow$ A.ge(B)

4.6 List e Set comprehension

In MiniZinc le list comprehension vengono scritte nel seguente modo:

[i + j | i, j in 1..3 *where* j < i]

Esempio:

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

A = [i + j | i, j in 1..3 where j < i];

solve satisfy;

output["A=",show(A)];
```

Output

A=[3, 4, 5]

mentre le set comprehension:

{i + j | i, j in 1..3 *where* j < i}

Esempio:

```
set of int: S;

S = {i + j | i, j in 1..3 where j < i};

solve satisfy;

output[show(S)];
```

Output

S=3..5

Traduzione in Java+JSetL:

setof → è un semplice metodo che va a creare un set che rispetti certe proprietà date.

```
// list/set comprehension che restituiscono list/set di interi

   //[i + j | i,j in 1..3 where i < j]

     IntLVar r = new IntLVar(); // creo una variabile logica r
     IntLVar i = new IntLVar(); // creo una variabile logica i
     IntLVar j = new IntLVar(); // creo una variabile logica j
     solver.add(r.eq(i.sum(j))); // r = i + j
     solver.add(i.dom(1,3).and(j.dom(1,3))); // i,j in 1..3

 // forza le variabili IntLVar a prendere in modo non
    deterministico uno dei valori del  dominio
    solver.add(i.label().and(j.label()));
    solver.add(j.lt(i)); // j < i

 // colleziona in un set tutti i valori della variabile logica r
    che soddisfa i constraint
    LSet s = solver.setof(r);
    s.setName("s").output();
```

Output

s = {3,4,5}

4.7 Aggregation function

Le aggregation functions **sum**, **product**, **min**, **max**, di MiniZinc possono essere realizzate in Java + JSetL utilizzando le funzioni della libreria LArray1D che operano su array di IntLVar, in particolare: *sumArray*, *productArray*, *maxArray*, *minArray*.

Esempio MiniZinc:

- *sum*

```
% decision variable di nome v
var 1..3: v;
% set di tipo int e di nome S
set of int: S = 1..3;
```

```
% array di decision variable e di nome A
array[S] of var 1..3: A;

%vincoli
constraint A[1]=1;
constraint A[2]=1;
constraint A[3]=1;

% funzione sum
v = sum (i in S) (A[i]);

solve satisfy;

% stampa
output["v=", show(v)];
```

Output

v = 3

Traduzione in Java+JSetL:

```
// crea un array di IntLVar di dimensione 3 con dominio (1,3)
  IntLVar[] a = LArray1D.generateArray(3,1,3);

  solver.add(a[0].eq(1));
  solver.add(a[1].eq(1));
  solver.add(a[2].eq(1));

  IntLVar s = new IntLVar("v");
  solver.solve(LArray1D.sumArray(a, s));
  s.output();
  }
}
```

Output

v = 3

Esempio MiniZinc

- *min*

```
% decision variable di nome v
var 1..3: v;
% set di tipo int e di nome S
set of int: S = 1..3;

% array di decision variable e di nome A
array[S] of var 1..3: A;

%vincoli
constraint A[1]=1;
constraint A[2]=2;
constraint A[3]=3;

% funzione min
v = min (i in S) (A[i]);

solve satisfy;

% stampa
output["v=", show(v)];
```

Output

v = 1

Traduzione in Java+JSetL:

```
  IntLVar[] a = LArray1D.generateArray(3,1,3);

  solver.add(a[0].eq(1));
  solver.add(a[1].eq(2));
  solver.add(a[2].eq(3));

  IntLVar m = new IntLVar("v");
  solver.solve(LArray1D.minArray(a, m));
  m.output();
      }
}
```

Output

v = 1

MiniZinc fornisce anche 4 aggregation function per gli array contenenti espressioni Booleane che sono: **forall**, **exists**, **xorall**, **iffall**.
Per certi tipi di chiamate di espressioni viene usata la sintassi del *generator call expression* che rende il modello più leggibile:

$$\langle agg - func \rangle \ (\ \langle generator - exp \rangle \ ) \ (\ \langle exp \rangle \ )$$

in modo equivalente:

$$\langle agg - func \rangle \ (\ [\ \langle expr \rangle \mid \langle generator - exp \rangle \ ] \ )$$

Esempio:

```
array[1..3] of int: A = [1,2,3];

constraint forall(i,j in 1..3 where i < j) (A[i] != A[j]);

solve satisfy;

output[show(A)];
```

Output

A =[1, 2, 3]

Se invece si prova a inizializzare l'array A con [2,2,3] l'output darà "UNSATISFIABLE" perchè abbiamo il primo e il secondo valore che sono uguali e per il forall abbiamo che A[i] != A[j] con i minore di j.

Traduzione in Java+JSetL:

```
    //forall([a[i] != a[j] | i,j in 1..3 where i < j])
    // o (equivalente)
    //forall (i,j in 1..3 where i < j) (a[i] != a[j])

int[] a = new int[3];

a[0]= 1;
a[1]= 2;
a[2]= 3;
```

```
int i,j;

Boolean c;
c = true;

for(i=0; i<3 && c; i++)
  for(j=0; j<3 && c; j++)
    if(i < j) c=(a[i] != a[j]);

    System.out.println(c);
```

Output

true

Se invece si prova a inizializzare l'array A con [2,2,3] l'output darà *"false"* perchè abbiamo il primo e il secondo valore che sono uguali e per il forall abbiamo che A[i] != A[j] con i minore di j.

4.8 Solve

In MiniZinc abbiamo tre tipi di solve: **satisfy**, **maximize**, **minimize**.

4.8.1 solve satisfy

Esempio:

```
int: n = 3;

var 1..n: a; var 1..n: b;
var 1..n: c; var 1..n: d;

constraint a != b;
constraint a != c;
constraint b != c;

solve satisfy;

output [ "a =", show(a), "\t b =", show(b),
         "\t c =", show(c), "\t d =", show(d),  ];
```

Output

a = 3 b = 2 c = 1 d = 1

Traduzione in Java+JSetL:

```
int n = 3;

IntLVar a = new IntLVar("a",1,n);
IntLVar b = new IntLVar("b",1,n);
IntLVar c = new IntLVar("c",1,n);
IntLVar d = new IntLVar("d",1,n);
```

```
// constraint
solver.add(a.neq(b));
solver.add(a.neq(c));
solver.add(b.neq(c));

// forza le variabili IntLVar a prendere in modo non determistico
   uno dei valori del    dominio
solver.add(a.label());
solver.add(b.label());
solver.add(c.label());
solver.add(d.label());

// solve satisfy
solver.solve();

// output
a.output();
b.output();
c.output();
d.output();
```

Output

a = 1
b = 2
c = 3
d = 1

4.8.2 solve maximize

Esempio:

```
int: n = 100;
var 1..n: a; % torte al cioccolato
var 1..n: b;     % torte alla crema

% ingredienti per i due tipi di torte

% farina
```

```
constraint 250*a + 200*b <= 4000;
% crema
constraint 2*a <= 6;
% zucchero
constraint 75*a + 150*b <= 2000;
% burro
constraint 100*a + 150*b <= 500;
% cacao
constraint 75*a <= 500;

% maximize il profitto
solve maximize 400*a + 450*b;

output [ "a =", show(a), "\t b =", show(b),];
```

Output

a = 1 b = 1
a = 2 b = 1
a = 3 b = 1
a = 2 b = 2

Traduzione in Java+JSetL:

```
IntLVar a = new IntLVar("a",0,100);// torte al cioccolato
IntLVar b = new IntLVar("b",0,100);// torte alla crema

solver.add(a.mul(250).sum(b.mul(200)).le(4000)); //farina
solver.add(a.mul(2).le(6)); //crema
solver.add(a.mul(75).sum(b.mul(150)).le(2000)); //zucchero
solver.add(a.mul(100).sum(b.mul(150)).le(500)); //burro
solver.add(b.mul(75).le(500)); //cacao

//maximize il profitto
IntLVar profit = new IntLVar("profit");
solver.add( profit.eq(a.mul(400).sum(b.mul(450))) ); // profit =
    a*400 + b*450

solver.add(a.label()); // attribuisci valori alle variabili
solver.add(b.label());
```

```
  // massimo profitto
  Integer m = solver.maximize(profit);
  System.out.println("\nIl massimo profitto e': " + m);

  // valori di a e b in corrispondenza al massimo profitto
  solver.add(profit.eq(m));
  solver.solve();
  System.out.println("\nValori massimi (a = no. torte di
      cioccolato; b = no. torte alla crema):");
  a.output();
  b.output();

  return;
    }

}
```

Output

Il massimo profitto e': 1700

Valori massimi (a = no. torte di cioccolato;b = no. torte alla crema):
a = 2
b = 2

4.8.3 solve minimize

Il **solve minimize** è uguale al maximize per usarlo ma invece di trovare una soluzione che massimizza l'espressione nello statement solve, la minimizza.

Capitolo 5

Esempio di programma tradotto

5.1 Planning

5.1.1 MiniZinc

In questo tipo di problema si vuole determinare la quantità di ciascun tipo di prodotto per massimizzare il profitto.

Planning.mzn

```
int: nproducts;
set of int: Products = 1..nproducts;

% profit per unit for each product
array[Products] of int: profit;
array[Products] of string: pname;

% Number of resources
int: nresources;
set of int: Resources = 1..nresources;

% amount of each resource available
array[Resources] of int: capacity;
array[Resources] of string: rname;

% units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;
```

```
constraint assert (forall (r in Resources, p in Products)
        (consumption[p,r] >= 0),"Error: negative consumption");

% bound on number of Products
int: mproducts = max (p in Products)
        (min(r in Resources where consumption[p,r] > 0)
                (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Products] of var 0..max(capacity): used;

% Production cannot use more than the available Resources:
constraint forall (r in Resources) (
        used[r] = sum (p in Products) (consumption[p, r] * produce[p])
        /\ used[r] <= capacity[r]
        );

%Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);

output [ show(pname[p]) ++ " = " ++ show(produce[p]) ++ ";\n" |
                p in Products ] ++
           [ show(rname[r]) ++ " = " ++ show(used[r]) ++ ";\n" |
                r in Resources ];
```

Planning-datafile.dzn

```
nproducts = 2;
profit = [400, 450];
pname = ["banana-cake", "chocolate-cake"];

nresources = 5;
capacity = [4000, 6, 2000, 500, 500];
rname = ["flour", "banana", "sugar", "butter", "cocoa"];

consumption = [|250, 2, 75, 100, 0,
                                | 200, 0, 150, 150, 75 |];
```

Output

banana-cake = 2;
chocolate-cake = 2;
flour = 900;
banana = 4;
sugar = 450;
butter = 500;
cocoa = 150;
.....

5.1.2 Java+JSetL

Planning.java

```
public class Planning {

  public static SolverClass solver = new SolverClass();

  //Ritorna un boolean true se l'array bidimensionale a[][] di P
     righe e R colonne e' maggiore o uguale di zero, altrimenti
     ritorna false
  public static Boolean forall_int(int[][] a, int P, int R){

       ....
  }

  //Funzione che ha come parametri un array monodimensionale ca[],
     un array bidimensionale co[][] e con P ed R che sono
     rispettivamente le righe e le colonne. Questa funzione
     ritorna una lista l1 di minimi presi dalla lista l2
  public static List<Integer> ListComprehension_using_int(int ca[],
     int co[][],int P, int R){

    ....
  }

  //Funzione che ha come paramentri un intero Ml1 e una lista l1.
     Ritorna il massimo della lista l1 assegnadolo alla variabile
     Integer Ml1
```

```
public static Integer Max_ListComprehension_using_int(Integer
    Ml1, List<Integer> l1){

  ....
}

//Funzione che prende come parametri un intero M un array
    monodimensionale a[], un intero R come dimensione dell'array.
    Ritorna il massimo dell' array a di dimensione R
public  static Integer Max_array(Integer M, int a[], int R){

  ....
}

//Funzione che prende come parametro una stringa di nome num e
    controlla facendo parseInt se e' intero, se lo e' ritorna 0
    altrimenti ritorna -1
public static int isInt(String num){

  ....
}


public static void main(String[] args) throws Failure,
    IOException {

....

  // array di interi dove sono salvati i valori
  int[] val_int = new int[26];

  // array di stringhe dove sono salvati i nomi delle variabili
  String[] val_str = new String[26];

....


  int nproducts = new Integer(val_int[0]);
  int Products = nproducts;


  // profit per unit for each product
```

```
int[] profit = new int[Products];
String[] pname = new String[Products];
for(i = 0; i < Products; ++i){
  profit[i] = new Integer(val_int[i+1]);
  pname[i] = val_str[i+3];
}

// number of resources

int nresources = new Integer(val_int[5]);
int Resources = nresources;

// amount of each resource available

int[] capacity = new int[Resources];
String[] rname = new String[Resources];
for(i = 0; i < Resources; ++i){
  capacity[i] = new Integer(val_int[i+6]);
  rname[i] = new String(val_str[i+11]);
}

// units of each resource required to produce 1 unit of product

int[][] consumption = new int[Products][Resources];
int k=16;
for(i=0; i < Products; ++i){
  for(j=0; j < Resources; ++j){
    consumption[i][j] = new Integer(val_int[j+k]);
  }
  k = k+5;
}

//forall (r in Resources, p in Products) (consumption[p,r] >=
   0))
// or (equivalent)
//forall([consumption[p,r] >= 0 | r in Resources, p in
   Products])

int r,p;
Boolean cond = true;
cond = forall_int(consumption, Products, Resources);

// constraint assert
```

```
LVar assertC = new LVar(true);
solver.solve(assertC.eq(cond));



// bound on number of Products

int mproducts = new Integer(Integer.MIN_VALUE);
List<Integer> l1 = new ArrayList<Integer>();
List<Integer> l2 = new ArrayList<Integer>();

l1 = ListComprehension_using_int(capacity, consumption,
    Products, Resources);
mproducts = Max_ListComprehension_using_int(mproducts, l1);

// Variables: how much should we make of each product

IntLVar[] produce =
    LArray1D.generateArray(Products,0,mproducts);
Integer max_capacity = 0;

max_capacity = Max_array(max_capacity, capacity, Resources);

IntLVar[] used =
    LArray1D.generateArray(Resources,0,max_capacity);


// Production cannot use more than the available Resources


//forall([used[r] = sl3 /\ used[r] <= capacity[r] | r in
    Resources])
// or (equivalent)
//forall (r in Resources) (used[r] = sl3 /\ used[r] <=
    capacity[r])
Constraint c = new Constraint();

for(r=0; r<Resources; r++) {
  //sl3=sum(consumption[p,r] * produce[p] | p in Products)
  // or (equivalent)
  //sl3=sum(p in Products) (consumption[p,r] * produce[p])
  List<IntLVar> l3 = new ArrayList<IntLVar>();
  for(p=0; p<Products; p++) {
```

```
        IntLVar elem = new IntLVar();
        //
            elem.setConstraint(elem.eq(produce[p].mul(consumption[p][r]+1)));
        elem.setConstraint(elem.eq(produce[p].mul(new
            IntLVar(consumption[p][r]))));
        l3.add(elem);
      }

      IntLVar[] a = LArray1D.generateArray(l3);
      IntLVar sl3 = new IntLVar("sl3");
      c.add(LArray1D.sumArray(a,sl3));
      c.add(used[r].eq(sl3));
      c.add(used[r].le(capacity[r]));

    }

    solver.add(c);
    solver.add(LArray1D.labelArray(produce));
    solver.add(LArray1D.labelArray(used));


    solver.solve();

    do {
      System.out.println(pname[0]+" = "+produce[0]);
      System.out.println(pname[1]+" = "+produce[1]);
      System.out.println(rname[0]+" = "+used[0]);
      System.out.println(rname[1]+" = "+used[1]);
      System.out.println(rname[2]+" = "+used[2]);
      System.out.println(rname[3]+" = "+used[3]);
      System.out.println(rname[4]+" = "+used[4]);
    } while(solver.nextSolution());



    return;

  }

}
```

Planning-datafile.txt

```
nproducts=2;
profit=400;
profit(1)=450;
pname=banana-cake;
pname(1)=chocolate-cake;
nresources=5;
capacity=4000;
capacity(1)=6;
capacity(2)=2000;
capacity(3)=500;
capacity(4)=500;
rname=flour;
rname(1)=banana;
rname(2)=sugar;
rname(3)=butter;
rname(4)=cocoa;
consumption=250;
consumption(1)=2;
consumption(2)=75;
consumption(3)=100;
consumption(4)=0;
consumption(5)=200;
consumption(6)=0;
consumption(7)=150;
consumption(8)=150;
consumption(9)=75;
```

Output

```
banana-cake = 2
chocolate-cake = 2
flour = 900
banana = 4
sugar = 450
butter = 500
cocoa = 150
.....
```

Funzioni implementate:

```
public static Boolean forall_int(int[][] a, int P, int R){

   Boolean c;

   int i;
   int j;

   c = true;

   for(i=0; i<P && c; i++)
     for(j=0; j<R && c; j++)
       c = (a[i][j] >= 0);

   return c;
 }
```

```
 public static List<Integer> ListComprehension_using_int(int ca[],
    int co[][],int P, int R){

 List<Integer> l1 = new ArrayList<Integer>();
 List<Integer> l2 = new ArrayList<Integer>();

   int p;
   int r;

   //ml2=min([capacity[r] div consumption[p,r] | r in Resources
      where consumption[p,r]>0])
   // or (equivalent)
   //ml2=min(r in Resources where consumption[p,r]>0) (capacity[r]
      div consumption[p,r])
   for(p=0; p<P; p++){
     for(r=0; r<R; r++)
       if(co[p][r] > 0) l2.add(ca[r]/co[p][r]);

     //min(l2)
     Integer ml2 = new Integer(Integer.MAX_VALUE);
     Iterator<Integer> Il2 = l2.iterator();
```

```
    do{
      Integer elem = Il2.next();
      if(elem < ml2) ml2 = elem;

    }while(Il2.hasNext());

    l1.add(ml2);
  }
  return l1;
}
```

```
public static Integer Max_ListComprehension_using_int(Integer
    Ml1, List<Integer> l1){

  //max( ml2 | p in Products])
  // or (equivalent)
  //max(p in Products) (ml2)

  //max(l1)
  Ml1 = new Integer(Integer.MIN_VALUE);
  Iterator<Integer> Il1 = l1.iterator();

  do{
    Integer elem = Il1.next();
    if(elem > Ml1) Ml1 = elem;

  }while(Il1.hasNext());

  return Ml1;
}
```

```
  public  static Integer Max_array(Integer M, int a[], int R){

  int i;

  M = new Integer(Integer.MIN_VALUE);
  for(i=0; i<R; i++)
    if(a[i] > M) M = a[i];
```

```
    return M;
  }
```

```
    public static int isInt(String num){
    try{
      Integer.parseInt(num);
      return 0;
    }catch(Exception e){
      return -1;
    }
  }
```

Lettura da Planning-data.txt

```
// per accedere al contenuto di un file
    FileReader f;
    f=new FileReader("planningdata.txt");

// per poter leggere una riga alla volta
    BufferedReader b;
    b=new BufferedReader(f);

    String s;

    String[] parts = new String[100];
    int[] val_int = new int[26];
    String[] val_str = new String[26];
    String[] stringhe = new String[26];

    int i=0;
    int j=0;

    while(true){

      s=b.readLine(); //legge una riga alla volta
      if(s == null)
        break;

      parts = s.split("=|;");
      stringhe[i] = parts[1];
```

```
    if(isInt(stringhe[j])== 0)
      val_int[j]=Integer.parseInt(stringhe[j]);
    else if (isInt(stringhe[j])== -1) {
      val_str[j]=stringhe[j];
    }

    i++;
    j++;

  }

  b.close(); // chiude il file
```

Capitolo 6

Conclusioni

In questo lavoro di tesi è stata studiata la possibilità di creare delle regole standard per poter tradurre direttamente il codice di MiniZinc in codice Java con l'aiuto di JSetl.
Per verificare tutto ciò abbiamo definito la traduzione di vari esempi di programmi MiniZinc ed abbiamo controllato che il comportamento del codice Java rispecchi quello definito negli esempi MiniZinc.
Il risultato ottenuto è stato vedere che un traduzione da MiniZinc in Java + JSetl è fattibile, senza riscontrare troppi problemi, inoltre sono state aggiunte nuove funzionalità su JSetl che lo hanno reso molto più versatile.

Come sviluppo futuro, visto che già tutti i compilatori effettuano la traduzione automatica di MiniZinc in FlatZinc, un linguaggio più a basso livello rispretto a MiniZinc, si potrebbe partire proprio da quest'ultimo per fare un traduttore più preciso per Java + JSetl. Visto che per MiniZinc è stato provato essere fattibile si può pensare che lo sia anche per FlatZinc.

Bibliografia

[1] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[2] Kim Marriott and Peter J.Stuckey with contributions from Leslie De Koninck and Horst Samulowitz
A MiniZinc Tutorial
http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf

[3] Gianfranco Rossi e Roberto Amadini
JSetL User's Manual
http://cmt.math.unipr.it/jsetl/JSetLUserManual-v.2.3.pdf

[4] JSetL Home Page
http://cmt.math.unipr.it/jsetl.html

[5] Specification of MiniZinc
https://www.minizinc.org/2.0/doc-lib/minizinc-spec.pdf

[6] Alan M. Frisch
Introduction to the MiniZinc Modelling Language
http://www-module.cs.york.ac.uk/copr/minizinc1.pdf
Artificial Intelligence Group, Department of Computer Science University of York.

[7] Vittorio Maniezzo
Tutorial MiniZinc
http://www3.csr.unibo.it/m̃aniezzo/didattica/IntArt/Tutorial_MiniZinc.pdf

Ringraziamenti

Prima di tutto vorrei ringraziare la mia famiglia per avermi dato la possibilità di intraprendere questo percorso e per avermi sempre incoraggiato, inoltre vorrei ringraziare anche tutti i familiari partendo dai miei cugini, gli zii fino ad arrivare ai nonni.
Insieme a loro vorrei ringraziare tutti gli amici del dipartimento che ho conosciuto in questi anni passati qui a Parma, in particolare gli amici del corso di informatica partendo da Roberto, Filippo, Gianluca, Juan e Danilo fino ad arrivare ai più ″Big″: Francesco, Thomas, Nicola e Marco, inoltre vorrei anche ringraziare le amiche matematiche Francesca, Antea e Marta che mi hanno sempre aiutato per qualsiasi problema matematico avessi ma sopratutto per avermi dato consigli utili nei momenti difficili e con loro vorrei anche ringraziare tutti i compagni di corso che non sono stati nominati ma che sono stati altrettanto importanti.
Vorrei poi ringraziare anche il professore Gianfranco Rossi per avermi aiutato durante il periodo di tesi con molta disponibilità e pazienza.
Infine vorrei ringraziare, non per importanza, tutti gli amici di San Benedetto del Tronto e dintorni che da una vita mi aiutano sempre nei momenti difficili e che mi sono sempre stati vicini venendomi anche a trovare spesso qui a Parma.

Appendice A

Esempi di traduzione completi

A.1 Colouring Australia

A.1.1 MiniZinc

```
int: nc = 3;

var 1..nc: wa; var 1..nc: nt; var 1..nc: sa; var 1..nc: q;
var 1..nc: nsw; var 1..nc: v; var 1..nc: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;
solve satisfy;

output ["wa=", show(wa), "\t nt=", show(nt),
"\t sa=", show(sa), "\n", "q  =", show(q),
"\t nsw=", show(nsw), "\t v=", show(v), "\n",
"t=", show(t), "\n"];
```

Output

wa = 3 nt = 2 sa = 1
q = 3 nsw = 2 v = 3
t = 1

A.1.2 Java+JSetl

```
package esercizioColouringAustralia;

import JSetL.*;

public class ColouringAustralia {

  public static SolverClass solver = new SolverClass();

  public static void main(String[] args) throws Failure{

    int nc = 3;

// creo variabile con Setl simile al decision variable di Minizinc
    con domino [1..nc]
  IntLVar wa = new IntLVar("wa",1,nc);
  IntLVar nt = new IntLVar("nt",1,nc);
  IntLVar sa = new IntLVar("sa",1,nc);
  IntLVar q = new IntLVar("q",1,nc);
  IntLVar nsw = new IntLVar("nsw",1,nc);
  IntLVar v = new IntLVar("v",1,nc);
  IntLVar t = new IntLVar("t",1,nc);
  IntLVar[] allVars = {wa,nt,sa,q,nsw,v,t};



  solver.add(wa.neq(nt));
  solver.add(wa.neq(sa));
  solver.add(nt.neq(sa));
  solver.add(nt.neq(q));
  solver.add(sa.neq(q));
  solver.add(sa.neq(nsw));
  solver.add(sa.neq(v));
```

```
  solver.add(q.neq(nsw));
  solver.add(nsw.neq(v));

// forza le variabili IntLVar a prendere in modo non
    deterministico uno dei valori del dominio
  solver.add(IntLVar.label(allVars));


    solver.solve();

  wa.output();
  nt.output();
  sa.output();
  q.output();
  nsw.output();
  v.output();
  t.output();

  }

}
```

Output

wa = 1
nt = 2
sa = 3
q = 1
nsw = 2
v = 1
t = 1

A.2 SendMore

A.2.1 MiniZinc

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
```

```
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint      1000 * S + 100 * E + 10 * N + D
                + 1000 * M + 100 * O + 10 * R + E
= 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

output ["    ",show(S),show(E),show(N),show(D),"\n",
        "+ ",show(M),show(O),show(R),show(E),"\n",
        "= ",show(M),show(O),show(N),show(E),show(Y),"\n"];
```

Output

```
9567
+ 1085
= 10652
```

A.2.2 Java+JSetl

```
package esercizioSendMore;

import JSetL.Failure;

import JSetL.IntLVar;
import JSetL.SolverClass;

public class SendMore {

  public static SolverClass solver = new SolverClass();

  public static void main(String[] args) throws Failure{

  IntLVar s = new IntLVar("S",1,9);
```

```
  IntLVar e = new IntLVar("E",0,9);
  IntLVar n = new IntLVar("N",0,9);
  IntLVar d = new IntLVar("D",0,9);
  IntLVar m = new IntLVar("M",1,9);
  IntLVar o = new IntLVar("O",0,9);
  IntLVar r = new IntLVar("R",0,9);
  IntLVar y = new IntLVar("Y",0,9);

  IntLVar send = new IntLVar("SEND");
  IntLVar more = new IntLVar("MORE");
  IntLVar money = new IntLVar("MONEY");

  IntLVar caratteri [] = {s,e,n,d,m,o,r,y};

  solver.add(IntLVar.allDifferent(caratteri));

// SEND = S*1000 + E*100 + N*10 + D
  solver.add(send.eq(s.mul(1000).sum(e.mul(100)).sum(n.mul(10)).sum(d)));

//MORE = M*1000 + O*100 + R*10 + E
  solver.add(more.eq(m.mul(1000).sum(o.mul(100)).sum(r.mul(10)).sum(e)));

//MONEY = M*10000 + O*1000 + N*100 + E*10 + Y
solver.add(money.eq(m.mul(10000).sum(o.mul(1000)).sum(n.mul(100)).sum(e.mul(10)).sum(y

//SEND + MORE = MONEY
  solver.add(money.eq(send.sum(more)));

  solver.add(s.label());
  solver.add(e.label());

  try {
      solver.solve();
    } catch (Failure e1) {
      e1.printStackTrace();
    }

  System.out.print(s);
  System.out.print(e);
  System.out.print(n);
  System.out.print(d);
  System.out.print("<===> SEND + \n ");
  System.out.print(m);
```

```
  System.out.print(o);
  System.out.print(r);
  System.out.print(e);
  System.out.print("<===> MORE + \n ");
  System.out.print(m);
  System.out.print(o);
  System.out.print(n);
  System.out.print(e);
  System.out.print(y);
  System.out.print(" <===> MONEY ");

  }

}
```

Output

9567<===> SEND +
1085<===> MORE +
10652 <===> MONEY