Initial Specification of Zinc 0.1

Kim Marriott Reza Rafeh Mark Wallace
Maria Garcia de la Banda

February 15, 2006

1 Introduction

Zinc is a new modelling language which provides:

- mathematical notation like syntax (coercion, overloading, iteration, sets, arrays)
- expressive constraints (FD, set, linear arithmetic, integer)
- different kinds of problems (satisfaction, explicit optimisation, preference (soft constraints))
- separation of data from model
- high-level data structures and data encapsulation (lists,sets, arrays, records, constrained types)
- extensibility (user defined functions, constraints)
- reliability (type checking, assertions)
- solver-independent modelling
- simple, declarative semantics

Zinc extends OPL and moves closer to CLP language such as ECLiPSe

2 Overview of a Zinc model

Zinc is a typed (mostly) first-order functional language. The main components of a Zinc model are now described. Statements of the following form comprise a Zinc model. They can occur in any order subject to the restriction that all identifiers (type, function, variable and parameters, record fields and enum constants) must be declared before they are used. More precisely a model is

⟨program⟩ ::= ⟨statement⟩ **;** ... **;** ⟨statement⟩ [**;**]
⟨statement⟩ ::= ⟨include-stmt⟩ | ⟨enum-type⟩ | ⟨record-type⟩ | ⟨type-stmt⟩
| ⟨var-decl⟩ | ⟨assign⟩ | ⟨constraint-stmt⟩ | ⟨assertion-stmt⟩
| ⟨output-stmt⟩ | ⟨optimise-stmt⟩ | ⟨predicate-stmt⟩ | ⟨function-stmt⟩

Comments are between `/*` and `*/`. Comments nest. A `%` indicates that the rest of the line is a comment.

The following is a simple Zinc model for the N-queens problem:

```
datafile "file.dat";
int: n;
type Domain=1..n;
array[Domain] of var Domain :queens;
predicate noattack(int: i,j; var int:Qi,Qj)=
      Qi        != Qj     /\
      Qi + i    != Qj + j /\
      Qi - i    != Qj - j;

constraint   forallv(i,j in Domain where i<j)
                noattack(i,j,qeens[i],queens[j])
```

Zinc is case sensitive and a free format modelling language. In the above model *queens* is an array which contains n variables, each storing the column number for a queen. The *noattack* predicate ensures that two queens cannot attack each other. The *forallv* expression imposes the *noattack* constraint on each pair of queens. We will explain these features in more detail in next sections.

2.1 Include statements

Include statements :

⟨include-stmt⟩ ::= **include** ⟨string-expr⟩

These include other Zinc models and take file names. For instance

```
include "schedule"
```

includes "schedule" the library of types and constraints for resource scheduling. If the name is not a complete path expression then the file is searched for in the current directory then the standard library. The search directories will be able to be set with a command line option.

The program can also include data files. These are not loaded until the model is executed: and so a model can be semantic checked in the absence of these. Currently data files are only allowed to contain assignment statements which can initialise enumerated types and parameter/variables. These can contain list, set and array comprehensions. These are effectively inserted into the model by adding the initialisation to the declaration in the model.

⟨include-stmt⟩ ::= **datafile** ⟨string-expr⟩

In our N-queens example we included a data file by using the statement:

```
datafile "file.dat"
```

This could contain the statement

```
n = 9;
```

2.2 Type definitions

Type expressions in Zinc can be:

⟨type-expr⟩ ::= ⟨base-type⟩ [**:** ⟨ident⟩ **where** ⟨bool-exp⟩] | ⟨set-expr⟩
⟨base-type⟩ ::= **float** | **int** | **bool** | **string** | ⟨tuple-defn⟩ | ⟨set-defn⟩
| ⟨list-defn⟩ | ⟨array-defn⟩ | ⟨type-ident⟩
⟨tuple-defn⟩ ::= **tuple(** [**var**] ⟨type-expr⟩**,**. . . **,** [**var**] ⟨type-expr⟩**)**
⟨set-defn⟩ ::= **set of** ⟨type-expr⟩
⟨list-defn⟩ ::= **list**[[⟨int-exp⟩]] **of** [**var**] ⟨type-expr⟩
⟨array-defn⟩ ::= **array**[⟨type-expr⟩**,**. . . **,**⟨type-expr⟩] **of** [**var**] ⟨type-expr⟩

Example type expressions are

```
tuple(int; var float)
set of list of int
```

Type identifiers are either identifiers bound to previously defined types or polymorphic types which are identifiers prefixed with "$". Currently polymorphic identifiers are only allowed in functions and predicate declarations.

Note that lists are allowed an optional integer argument which is their size. This is primarily for use with list containing decision variables.

Zinc provides type equivalence declarations:

⟨type-stmt⟩ ::= **type** ⟨type-ident⟩ **=** ⟨type-expr⟩

One powerful feature of Zinc is that it allows the user to specify "constrained types." For instance

```
Domain = 1..n;
Domain =  int: j  where  j in 1..n;
type FloatPlus = float: x  where x >= 0;
```

Note that the first two statements are equivalent and correspond to an integer constrained to be in the set $\{1, .., n\}$. Note that in the last two statements i and x are locally scoped identifier used in the Boolean expression after the *where* keyword. The Boolean expression associated with a variable declared to have a constrained type is either tested after initialization if the variable is a parameter or else generates a constraint if it is a decision variable.

The programmer can declare new enumerated types

⟨enum-type⟩ ::= **enum** ⟨type-ident⟩ [= **{** ⟨enum-elements⟩[,**...**] **}**]
⟨enum-elements⟩ ::=⟨ident⟩,. . . , ⟨ident⟩

For example:

```
enum country={Australia,USA,China,Canada,England}
```

The ordering on the elements in an enumerated type is given by the order in the definition.

New record types can also be declared

⟨record-type⟩ ::= **record** ⟨type-ident⟩ [= ⟨record-decl⟩]
⟨record-decl⟩ ::= ⟨simple-rec-decl⟩
⟨record-decl⟩ ::= **{** ⟨disc-union-decl⟩**;**... **;** ⟨disc-union-decl⟩ **}**
⟨simple-rec-decl⟩ ::= **(**⟨record-field⟩**;** . . . **;** ⟨record-field⟩**)** [**where** ⟨bool-exp⟩]
⟨disc-union-defn⟩ ::= ⟨ident⟩⟨simple-rec-decl⟩
⟨record-field⟩ ::= [**var**] ⟨type-expr⟩**:**⟨ident⟩,. . . ,⟨ident⟩

Note that all fields in a record must have a name. They can also have associated constraints:

```
record Task = (
       int: Duration;
       var int: Start, Finish
       ) where Finish == Start+Duration;
```

And here is an example of a discriminated union type:

```
 record multi_point ={
                     intp(int: x,y);
                     floatp(float: x,y)
                     }
```

Currently all user-defined types are monotypes. In the future we may allow types which are polymorphic in other types and also parameters for the associated constraints.

We distinguish types which are *finite sets*. Finite set types are bool, enum types, set expressions, and sets of finite set types.

2.3 Parameter and variable declarations and assignments

Parameter/variable declarations have the form

⟨var-decl⟩ ::= [⟨annotations⟩] [**var**] ⟨type-expr⟩**:** ⟨ident⟩[=⟨expr⟩],. . . ,⟨ident⟩[=⟨expr⟩]

The `var` keyword distinguishes parameters from decision variables. It can only be used with type expressions that are finite sets or with floats. Note that lists, records and arrays can contain decision variables in which case this restriction also applies. In our N-queens example, *queens* is an array which contains n

integer decision variables eher n is a paramter. We will explain annotations in Section 2.7.

A parameter/variable is *ground* if it is not a decision variable and contains no decision variables. A parameter can be assigned a value (once). All components of the parameter must be assigned a value. However decision variables can be assigned the special constant "_" which means "don't know."

A parameter /variable can be either be initialised using the optional assignment statement in its declaration or in a subsequent assignment statement. This is the syntax of initializing a parameter:

⟨assign⟩ ::= ⟨ident⟩ =⟨expr⟩

For instance, the following statement:

```
int : A = 10
```

can be separated in two statements as follows:

```
int : A ;
...
A = 10
```

Parameter /variable initialisation and assignment statements in the model are performed in textual order after assignments in any included data files have been inserted as initialisation statements in the model. When an assignment is executed all identifiers in the statement must have been initialised.

2.4 Constraints

Constraint statements:

⟨constraint-stmt⟩ ::= [⟨annotations⟩] **constraint** ⟨bool-expr⟩

Here is a very simple example of a constraint:

```
constraint A*X < B
```

By using the *holds* function, you can take advantage of reification in Zinc. This function takes a constraint and returns 1 if the constraint holds and 0 otherwise.

2.5 Optimisation statements

Optimisation statements:

⟨optimise-stmt⟩ ::= [⟨annotations⟩] **minimize** ⟨arith-expr⟩
⟨optimise-stmt⟩ ::= [⟨annotations⟩] **maximize** ⟨arith-expr⟩

For example:

```
maximize A*X+Y-3*Z
```

2.6 Assertion and output statements

Assertion statements are used to check data integrity. The boolean expression must be ground. If the assertion fails the string is written out to standard output and evaluation stops.

⟨assertion-stmt⟩ ::=**assert**[(⟨string-expr⟩)] ⟨bool-expr⟩

For example:

```
set of int: S1 , S2;
Assert("S1 and S2 must have the same cardinality") | S1 | == | S2 |
```

(where $|A|$ means cardinality of A) ensures the modeller that after initialization, S1 and S2 would be two sets with the same cardinality.

After the problem is solved output statements are evaluated and written to standard output. It is likely that the format may change

⟨output-stmt⟩ ::=**output** ⟨string-expr⟩

such as:

```
output   write(a);
```

2.7 Annotations

The programmer can annotate variable declarations, constraints and optimisation statements:

⟨annotations⟩ ::= ⟨annotation⟩ . . . ⟨annotation⟩ **::**
⟨annotation⟩ ::= ⟨ident⟩[⟨tuple-exp⟩]

Currently Zinc knows about three kinds of annotations: *class*(⟨string-exp⟩) which indicates the class of variable and is intended to guide translation to the underlying solver, and *strength*(⟨float-exp⟩) and *level*(⟨int-exp⟩) which indicate that a constraint/objective is soft and its strength and level.

For example:

```
int : strong =1;
var float  : x,y;
level(strong) strength(2.0):: constraint
x < 20 /\
y < 20 ;

class("hard"):: constraint  x == y;
```

Zinc will try to satisfy constraints in lower numbered levels in preference to higher numbered levels. Within each level, the strength gives the relative priority.

In the future more annotations may be allowed.

3 Expressions and Constraints

Here we detail the built in functions. Note that the majority of useful functions will be defined in libraries. Many of the Zinc built-in functions are defined in a library named stdlib which is included in a model automatically.

Currently Zinc does not support pattern matching. This is intended to be supported in the future.

All types have built in comparison functions:

```
=<  >=  > <  ==  !=
```

These are generated automatically for user-defined types using standaard lexicographic ordering in records etc.

A unary function or predicate P which takes a list can be called with an expression of form $P(G)E$ which is syntactic sugar for the expression $P([E \mid G])$ where G is a standard generator expression with an optional filter. Any ambiguity can be resolved by using parentheses.

For example, the expression

```
forallv(i,j in Domain where i<j) noattack(i,j,qeens[i],queens[j])
```

in the N-queens example is equivalent to

```
forallv([ noattack(i,j,qeens[i],queens[j]) | i,j in Domain where i<j])
```

forallv is a built-in function defined in stdlib. Note that the first version of Zinc does not support overloading. So for each function which can be used with different types, you must have multiple declarations with different function names. For example, *forallv* takes a list of *var bool* and returns *var bool*, but *forall* is another version of this function which takes list of *bool* and returns *bool*.

3.1 Conditional Statements

Zinc provides if-then-else statements:

if ⟨bool-exp⟩ **then** ⟨exp⟩ **else** ⟨exp⟩

For example:

```
if x=<y then x else y
```

Currently the Boolean expression must be ground but in the near future we will drop this restriction.

Zinc also provides a case statement for accessing the different cases in a discriminated union:

case ⟨disc-union-exp⟩ **{** ⟨ident⟩**:**⟨exp⟩ **,** ..., ⟨ident⟩**:**⟨exp⟩ **}**

For example, we can use the *multi_point* record in Section 2.2 with the following code:

```
multi_point: r;
int: M=case r {intp: r.1;floatp: 0}
```

3.2 Arithmetic expressions

Integer and float constants have the usual form: i.e. 123, -123, 1.005, 1E05.

Arithmetic operations are standard. Float functions are:

```
+ - * / ceil floor round
```

Integer functions are

```
+  - * div mod
```

Note that some arithmetic functions are overloaded. There is automatic coercion from integers to floats.

3.3 Enumerated types

Enumerated type constants have the same syntax as indentifiers. Constants in different enumerated types must be distinct.

An enumerated type can be partially initialised by including the special constant `...` in the set. The remainder of the type is then initialised using another declation with the same name:

```
enum Shifts = { idle, ...};
...
Shifts = { idle, day, night};
```

3.4 Tuples and Records

A tuple expression has form

⟨tuple⟩ ::= **(** ⟨expr⟩, . . . , ⟨expr⟩ **)**

The fields in the tuple are accessed as if they were a record with field names 1, 2, 3, etc. For example $(3, 4).1$ returns 3.

A record expression has form

⟨record-exp⟩ ::= [⟨ident⟩]**(** ⟨ident⟩:⟨expr⟩, . . . , ⟨ident⟩:⟨expr⟩ **)**

Record fields are accessed with the expression ⟨rec-exp⟩.⟨field-name⟩.

There is automatic coercion from an n-ary tuple to a record with n-fields if the types of the corresponding fields are compatible. This allows records to be initialised with tuples. For example, we can initialise a record of type *Task* using

```
Task : T =(10,_,_);
```

which initializes *Duration* field with 10 and two variable fields *Start* and *Finish* remains unchanged. If we use field names for initializing records, we do not need to initialise fields thaat aare deecision variable. Thus the following initialization is equivalent to our previous one:

```
Task : T =(Duration:10)
```

Here is an example of initializing parameters of discriminated union types (using the *multi_point* record in Section 2.2):

```
multi_point : P1 = intp(2,3), P2=floatp(2.3,5.6)
```

3.5 Sets

A set expression has form

⟨setexp⟩ ::= ⟨ident⟩ | ⟨type-exp⟩
⟨setexp⟩ ::= ⟨int-exp⟩ **..**⟨int-exp⟩
⟨setexp⟩ ::= **{** ⟨exp⟩ **,** ... **,** ⟨exp⟩ **}**
⟨setexp⟩ ::= **{** ⟨exp⟩ | ⟨generators⟩ [**where** ⟨bool-exp⟩] **}**
⟨setexp⟩ ::= ⟨setexp⟩ ⟨set-op⟩ ⟨setexp⟩
⟨set-op⟩ ::= **union** | **inter** | **diff** | **symdiff**
⟨generators⟩ ::= ⟨generator⟩ **;** ... **;** ⟨generator⟩
⟨generator⟩ ::= ⟨ident⟩ **,** ... **,** ⟨indent⟩ **in** ⟨list-exp⟩

Currently the "where" condition in a generator must be ground. This restriction will be removed in the future.

A type expression is allowed as a set expression if it is a finite set type. This is particularly useful with enumerated types since they can be treated as sets.

Ranges can be used to specify a set. Currently ranges can only be used for integers but in the future will be allowed for enumerated types.

Containment operations on sets are **subset**, **supset** etc. Note that the predefined ordering on sets is a lexicographic ordering and so not equivalent to containment.

$| S |$ gives the cardinality of set S while X *in* S holds if X is a member of S.

There is automatic coercion from ground sets to lists where the resulting list is ordered from the smallest to largest element in the set. Thus $S[1]$ is the smallest element in a set while $S[\ |\ S\ |\ ]$ is the largest.

However there is not automatic coercion from a var set to a list since this is not well-defined. This means that var sets cannot be used as generators.

3.6 Arrays

We allow the user to declare *static* arrays whose index types in the declaration are finite set types and *dynamic* arrays whose index sets are computed from the array initialization. When a static array is initialised Zinc checks that the actual index set equals the finite set type in the declaration of the array.

An array expression has form

⟨arrayexp⟩ :: ⟨ident⟩
⟨arrayexp⟩ ::= [⟨tuple-exp⟩:⟨exp⟩ **,** ... **,** ⟨tuple-exp⟩:⟨exp⟩] [**default**

⟨exp⟩]
⟨arrayexp⟩ ::= [⟨tuple-exp⟩:⟨exp⟩ | ⟨generators⟩ [**where** ⟨bool-exp⟩]
] [**default** ⟨exp⟩]

An array of type *array[S1] of array[S2] of T* can be automatically coerced to one of type *array[S1,S2] of T* as long as the index sets are in accord.

Default values are allowed when initialising a static array. Furthermore a list can be can be automatically coerced to a static one-dimensional array. This is handy for initialisation but seems a bit dangerous.

For example:

```
array[1..100] of int: a =[1:5,6:12,9:1,24:6] default 0;
```

declares an array with 100 elements, of which 4 of them are initialized explicitly and the remainder set to 0. A sparse representation is used with defaults.

Other examples of array initialisation include

```
array[int] of int: b =[i:2*i | i in 1..4];
array[Countries] of int: demand = [1,4,3,10,5]
```

The expression $A[\langle \text{exp} \rangle, \ldots, \langle \text{exp} \rangle]$ returns the appropriate element in an array. The expression $indexSet(A, i)$ returns the i^{th} index set of the array. For instance with the above initialisations $indexSet(b, 1)$ is $\{1, 2, 3, 4\}$.

3.7 Lists

A list expression has form

⟨listexp⟩ :: ⟨ident⟩
⟨listexp⟩ ::= [⟨exp⟩ **,** ..., ⟨exp⟩]
⟨listexp⟩ ::= [[⟨exp⟩:]⟨exp⟩ | ⟨generators⟩ [**where** ⟨bool-exp⟩]] [**default** ⟨exp⟩]
⟨listexp⟩ ::= ⟨listexp⟩ ⟨list-op⟩ ⟨listexp⟩
⟨list-op⟩ ::= **@**

$length(L)$ gives the length of a list L while X *in* L holds if X is a member of L.

A list L is regarded as a one-dimensional array with the index set $1 \ldots length(L)$. Thus $L[i]$ gives the i^{th} element of list L.

When declaring a list an optional length for the list is allowed. This allows an n-ary list of variable to be simply created.

Zinc provides the higher-order functions

foldl(⟨listexp⟩**,** ⟨binary-function⟩**,** ⟨exp⟩ **)**
foldr(⟨listexp⟩**,** ⟨binary-function⟩**,** ⟨exp⟩ **)**

for computing over lists. Thus if we define the following functions:

```
function plus(int:x,y)=x + y;
function and(bool:x,y)=x /\ y;
```

then $foldl(L, plus(), 0)$ is sum, $foldl(L, and(), true)$ is forall.

3.8 Booleans

The usual Boolean operators and constants are provided

```
true false and or xor ~ => <=  <=>
```

3.9 Strings

String constants follow C convention but allow unicode.

We need functions to print data structures to a string.

We are yet to decide the operations allowed on strings and whether a string is a primitive type or a list of characters. Concatenation will probably be one primitive.

4 User Defined Functions and Constraints

⟨predicate-stmt⟩ ::= **predicate**⟨param-list⟩ [= ⟨body⟩] **;**
⟨function-stmt⟩ ::= **function**⟨params⟩**:** ⟨type-expr⟩ [= ⟨body⟩] **;**
⟨params⟩ ::= **(**⟨record-field⟩**; . . . ;** ⟨record-field⟩**)**
⟨body⟩ ::= ⟨exp⟩ | **{** ⟨local-vars⟩ **;** ⟨exp⟩ **}**
⟨local-vars⟩ ::= ⟨var-decl⟩ **; . . . ;** ⟨lvar-decl⟩

Note that the type expressions can include polymorphic type identifiers in the function and predicate declaration.

The return type of a predicate is implicitly a Boolean.

Functions and constraints are currently not allowed to be recursive.

```
predicate even(int:x) =
    x mod 2 == 0;

predicate between($T: x,y,z) =
   (x =< y /\ y =< z) \/ (z=<y /\ y=<x);

function min($T: x, y):$T =
   if x =< y then x else y;

predicate CD(int:A,B)=
{
    var 2..min(A,B):C;
```

```
    A mod C == 0 /\
    B mod C ==0

};
predicate serial(Resistor : x,y,z) =
     z.r == x.r + y.r    /\
     z.i ==x.i      /\
     z.i == y.i
```

Predicate *even* checks that its argument is an even number while polymorphic predicate *between* takes three variables and checks that value of y lies between values of x and z. Function *min* takes two parameters and gives their minimum. Note that currently this will give an error if called with non-ground arguments but in the future this will be allowed. Predicate *CD* takes two integer values checks whether they have a common divisor greater than 1 or not. It declares a new variable C which its domain is $2..min(A, B)$ because a common divisor for two numbers lies between 1 and the minimum of those numbers (of course, here we are not interested in 1). Predicate *serial* constrains the resistor z to be equivalent to connecting the two resistors x and y in series.

Currently a user defined constraint or function with local decision variables is not allowed to occur in a negative context. In the future this will be relaxed to allow local decision variables if they are functionally dependent upon the parameters or global variables.

A user defined binary function or predicate can be used as an infix operator by quoting it, such as:

```
A `min` B
```

A Production Problem

```
enum Products = { kluski, capellini, fettucine };
enum Resources={flour, eggs };
record ProductData=(
            float: demand,insideCost,outsideCost;
            array[Resources] of float:consumption
                    );
array[Products] of ProductData :product;
array[Resources] of float: capacity ;
array[Products] of var float:inside,outside;

constraint
    forall(r in Resources)
(capacity[r] >=
sum(p in Products) product[p].consumption[r]*inside[p]);
constraint
    forall(p in Products)
        (inside[p]+outside[p]>= product[p].demand);
minimize
    sum(p in Products)
product[p].insideCost*inside[p]+product[p].outsideCost*outside[p];
```

An example data file is

```
product = [ kluski     :( 100, 0.6, 0.8, [flour: 0.5,eggs: 0.2 ] ),
          capellini  :( 200, 0.8, 0.9, [flour: 0.4, eggs: 0.4 ]),
          fettucine  :( 300, 0.3, 0.4, [flour: 0.3, eggs: 0.6 ])];
  capacity = [ flour:20, eggs:40 ]
```

B Stable Marriage

```
enum Women = {Helen,Tracy,Linda,Sally,Wanda};
enum Men= {Richard,James,John,Hugh,Greg};
array[Women,Men] of int :RankWomen;
array[Men,Women] of int :RankMen   ;
array[Men] of var Women : Wife;
array[Women] of  var Men : Husband;
constraint
    forall(M in Men)
        (Husband[Wife[M]] == M);
constraint
    forall(W in Women)
        (Wife[Husband[W]] == W);
constraint
    forall(M in Men, O in Women)
        (RankMen[M,O]<RankMen[M,Wife[M]]=>RankWomen[O,Husband[O]]<RankWomen[O,M]);
constraint
    forall(W in Women, O in Men)
        (RankWomen[W,O]<RankWomen[W,Husband[W]]=>RankMen[O,Wife[O]]<RankMen[O,W]);
```

An example data file is

```
RankWomen=[
   Helen:   [Richard:1,James:2,John:4,Hugh:3,Greg:5],
   Tracy:   [Richard:3,James:5,John:1,Hugh:2,Greg:4],
   Linda:   [Richard:5,James:4,John:2,Hugh:1,Greg:3],
   Sally:   [Richard:1,James:3,John:5,Hugh:4,Greg:2],
   Wanda:   [Richard:4,James:2,John:3,Hugh:5,Greg:1]
           ];
RankMen = [
   Richard  : [Helen:5,Tracy:1,Linda:2,Sally:4,Wanda:3],
   James    : [Helen:4,Tracy:1,Linda:3,Sally:2,Wanda:5],
   John     : [Helen:5,Tracy:3,Linda:2,Sally:4,Wanda:1],
   Hugh     : [Helen:1,Tracy:5,Linda:4,Sally:3,Wanda:2],
   Greg     : [Helen:4,Tracy:3,Linda:2,Sally:1,Wanda:5]
         ]
```

C Social Golfers

```
int: Weeks;
int: GroupSize;
enum Players = {...};
int: Groups =  |Players| div GroupSize;
assert Groups * GroupSize == |Players|;
array[1..Weeks,1..Groups] of var set of Players: group;

predicate maxOverlap(list of var set of $E: sets; int n) =
    forall(i,j in 1..length(sets) where i < j)
       (| sets[i] intersect sets[j] | =< n);

% symmetry breaking constraints
constraint
      forall(i in 1..Weeks-1) (group[i,1] < group[i+1,1]);
constraint
     forall(i in 1..Weeks; j in 1..Groups-1 ) (group[i,j] < group[i,j+1]);
% right size groups
constraint
     forall (i in 1..Weeks; j in 1..Groups)  (| group[i,j] | == GroupSize);
% no-one plays in more than one group each week
constraint
    forall (i in 1..Weeks) (maxOverlap([group[i,j] | j in Groups],0));
% groups change each week
constraint
    maxOverlap([group[i,j] | i in 1..Weeks; j in Groups],1);
```

An example data file is

```
Weeks=3;
GroupSize=3;
Players={a,b,c,d,e,f,g,h,i}
```