MONASH University

Model without Numbers (!)

Type-safe optimisation in MiniZinc

Guido Tack, 19 June 2026

(joint work with Jip Dekker, Peter Stuckey, Jason Nguyen)

Goals for this seminar

I'll try to convince you that…

- You don't need to think about 0/1 variables, big-M constraints, how to linearise complex constraints
- Many "things" we model are not numbers but abstract values of enumerated types
- The remaining "things" may be numbers, but they have units
- MiniZinc can help you write cleaner, more self-documenting, "more correct" models

Overview

- Intro to MiniZinc modelling
- Logic and element constraints
- Enumerated types
- Unit types
- Object types

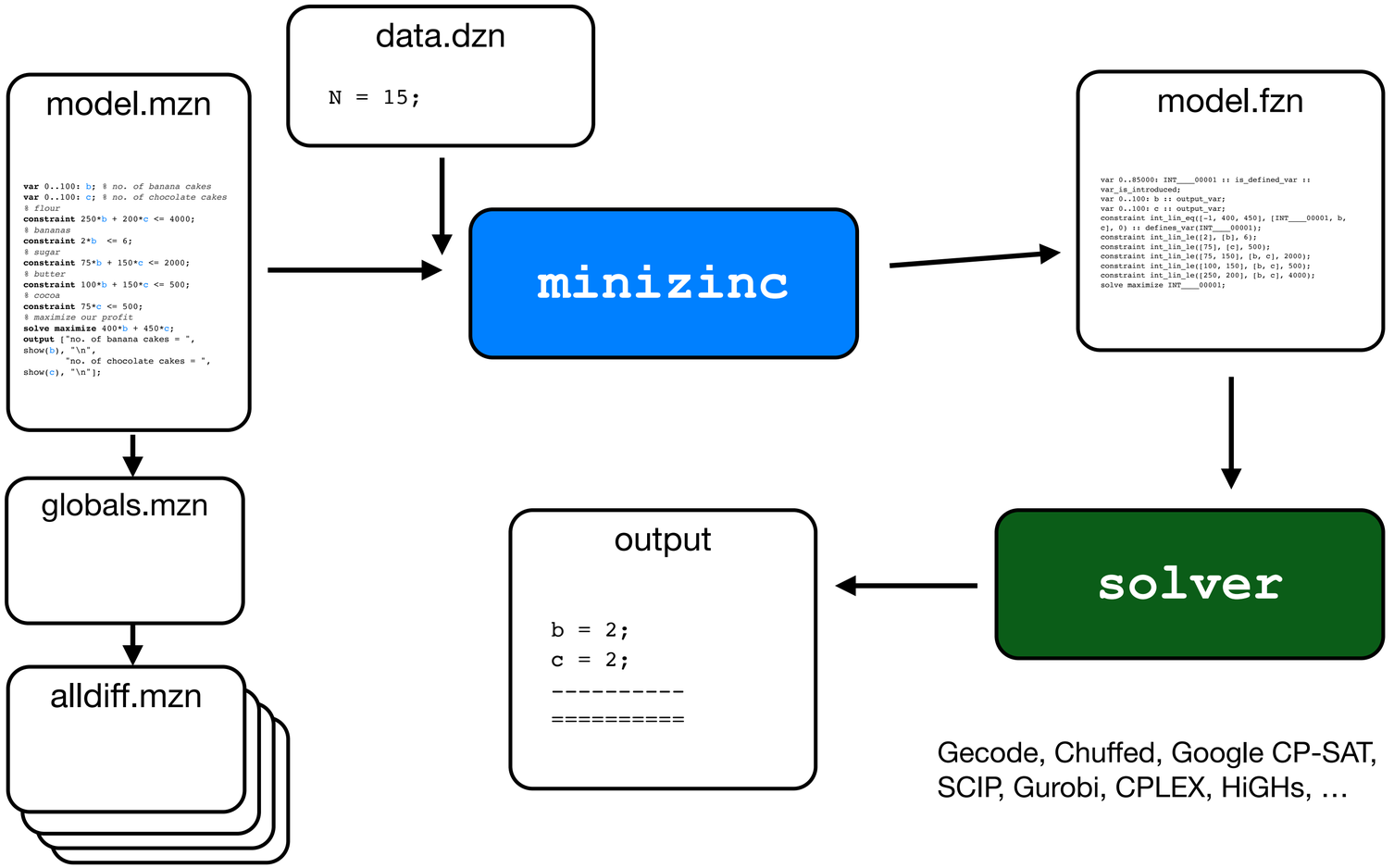
What is MiniZinc?

High-level, solver independent optimisation modelling

- Similar to AMPL, GAMS, OPL, …, a bit less similar to JuMP, Pyomo, …
- A model describes a class of problems
- Combine it with instance data
- MiniZinc translates into solver input for a range of very different solvers
- Developed at Monash University and OPTIMA

MiniZinc Overview

Architecture



Using MiniZinc

- Directly from your browser:

  https://play.minizinc.dev/

- Or download a self-contained package:

  https://www.minizinc.org/

- **Examples for this seminar:** https://github.com/MiniZinc/model-without-numbers

- The browser version runs directly in the browser (no server involved) and comes with four open-source solvers

- The download version comes with more solvers and supports Gurobi, CPLEX and others

First example: cryptarithms

(just a warm-up)

- A cryptarithm is a mathematical puzzle which requires determining the digit for each letter in an equation. The most famous cryptarithm is

```
    S E N D
+   M O R E
= M O N E Y
```

- Each letter is a different digit
- The first letter in each word can’t be 0
- The equation must hold

Part 1: don’t use 0/1s, big M etc.

Example: Assignment Problem

First, as a MIP model

- Assign *n* workers to *n* tasks - one task per worker, one worker per task
- Each task/worker pair has a cost, minimise total cost of the assignment

Example: Assignment Problem

First, as a MIP model

- Assign *n* workers to *n* tasks - one task per worker, one worker per task
- Each task/worker pair has a cost, minimise total cost of the assignment

```
int: n;
set of int: WORKER = 1..n;
set of int: TASK   = 1..n;
array[WORKER, TASK] of int: cost;

% x[w,t] = 1  iff  worker w is assigned task t
array[WORKER, TASK] of var 0..1: x;

constraint forall(w in WORKER)( sum(t in TASK)(x[w,t]) = 1 );  % one task per worker
constraint forall(t in TASK)( sum(w in WORKER)(x[w,t]) = 1 );  % one worker per task

solve minimize sum(w in WORKER, t in TASK)( cost[w,t] * x[w,t] );
```

Example: Assignment Problem

Now, using “idiomatic” MiniZinc

- First change:
  - Model assignment directly as integer variables rather than 0/1s
  - Use **all_different** global constraint to enforce permutation
- Second change:
  - Model cost function using **array element access**

Example: Assignment Problem

Now, using “idiomatic” MiniZinc

```
include “all_different.mzn";

int: n;
set of int: WORKER = 1..n;
set of int: TASK   = 1..n;
array[WORKER, TASK] of int: cost;

% assign[w] = the task that worker w performs
array[WORKER] of var TASK: assign;

constraint all_different(assign);

solve minimize sum(w in WORKER)( cost[w, assign[w]] );
```

assign[w] is a variable!

https://docs.minizinc.dev/en/stable/lib-globals.html

Global constraints

all_different and others

- “global constraints” are just “constraints over arrays of variables”
- Think of them as a library of **common combinatorial substructures**
- MiniZinc provides a lot of them in its standard library
- Several advantages:
  - cleaner, more self-documenting models
  - dedicated algorithms in CP (Constraint Programming) solvers
  - efficient linearisations (for solvers that don’t have native algorithms)

Element constraints

A real modelling workhorse

- Very simple syntax: `x[y]` where `x` is any array, and `y` is an integer **variable**
- Extremely versatile:
  - `y` stands for some “object”, `x` stands for “attributes” of that object, `x[y]` selects the attribute
  - alternatively, `x` represents a **finite function** *f*, and `x[y]` is *f(y)*
- Translated efficiently

Logic constraints

Think “if-then-else”, not “big-M”

- Let’s say worker 1 and worker 2 share a piece of heavy equipment, and they can’t both be allocated a demanding (long running) task at once.
- MIP model:
  - introduce 0/1 variables and big-M to indicate if allocated task is demanding, then constrain to be <= 1
- High-level model:
  - use element constraint plus logic constraint

Logic constraints

Think “if-then-else”, not “big-M”

```
int: n;
enum WORKER = W(1..n);
enum TASK = T(1..n);
array[WORKER, TASK] of int: cost;

array[TASK] of int: dur;              % task duration
int: H;                               % "heavy" threshold
int: M = max(t in TASK)(dur[t]);

% x[w,t] = 1  iff  worker w is assigned task t
array[WORKER, TASK] of var 0..1: x;

constraint forall(w in WORKER)( sum(t in TASK)(x[w,t]) = 1 );  % one task per worker
constraint forall(t in TASK)( sum(w in WORKER)(x[w,t]) = 1 );  % one worker per task

array[WORKER] of var 0..1: heavy;     % heavy[w] = 1  if worker w's task exceeds H
constraint forall(w in WORKER)(
  sum(t in TASK)(dur[t] * x[t,w]) <= H + M * heavy[w]  % duration > H  => heavy[w] = 1
);
constraint heavy[1] + heavy[2] <= 1;      % the shared pair: at most one heavy

solve minimize sum(w in WORKER, t in TASK)( cost[w,t] * x[w,t] );
```

Logic constraints

Think “if-then-else”, not “big-M”

```
include “all_different.mzn";

int: n;
set of int: WORKER = 1..n;
set of int: TASK   = 1..n;
array[WORKER, TASK] of int: cost;

array[TASK] of int: dur;              % task duration
int: H;                               % "heavy" threshold

% assign[w] = the task that worker w performs
array[WORKER] of var TASK: assign;

constraint dur[assign[1]] < H \/ dur[assign[2]] < H;

% alternatively, write as if-then-else:
%constraint if dur[assign[1]] > H then dur[assign[2

constraint all_different(assign);

solve minimize sum(w in WORKER)( cost[w, assign[w]] );
```

Logic “or”

element constraint

Part 2: type-safe enums

When “things” are not integers

Enumerated types

- Workers and tasks are not numbers!
- Let’s revisit the MIP model we just looked at:

```
array[WORKER, TASK] of var 0..1: x;
array[WORKER] of var 0..1: heavy;

constraint forall(w in WORKER)(
  sum(t in TASK)(dur[t] * x[t,w]) <= H + M * heavy[w]
);
constraint heavy[1] + heavy[2] <= 1;
```

this doesn’t look right!

- Can you spot the mistake?
- But it compiled and solved!

Another example: production planning

Choose k products within a weight limit, maximise profit

```
int: k;    % number of products to choose
int: limit;  % available weight limit
int: n;       % number of products

include "all_different.mzn";

set of int: PRODUCT = 1..n;
array[PRODUCT] of int: wght;
array[PRODUCT] of int: profit;
array[1..k] of var PRODUCT: chosen;
constraint all_different(chosen);
constraint sum(i in 1..k)(wght[chosen[i]]) <= limit;
solve maximize sum(i in 1..k)(profit[i]);
```

this doesn’t look right!

The problem: “raw” integers

Ubiquitous in discrete models, but “considered harmful”

Integers are used to model

- sets of objects
- position (packing)
- time (scheduling)
- order (TSP, VRP)

How can we "remove" integers from a discrete optimisation modelling language?

Enumerated types

Basically integers, but type-safe

- A named set of entities:

```
enum Worker = { Maria, Peter, Guido, Jip, Jason };
```

- Or an anonymous set:

```
enum Task = T(1..10);
```

- MiniZinc treats different enums as different types.
  Mixing them is a **static type error**.

- Let’s write the two models with enums.

MIP style with enums
(detecting the bug)

```
int: n;
enum WORKER = W(1..n);
enum TASK = T(1..n);
array[WORKER, TASK] of int: cost;

array[TASK] of int: dur;              % task duration
int: H;                               % "heavy" threshold
int: M = max(t in TASK)(dur[t]);

% x[w,t] = 1  iff  worker w is assigned task t
array[WORKER, TASK] of var 0..1: x;

constraint forall(w in WORKER)( sum(t in TASK)(x[w,t]) = 1 );  % one task per worker
constraint forall(t in TASK)( sum(w in WORKER)(x[w,t]) = 1 );  % one worker per task

array[WORKER] of var 0..1: heavy;     % heavy[w] = 1  if worker w's task exceeds H
constraint forall(w in WORKER)(
  sum(t in TASK)(dur[t] * x[t,w]) <= H + M * heavy[w]  % duration > H  => heavy[w] = 1
);
constraint heavy[1] + heavy[2] <= 1;     % the shared pair: at most one heavy

solve minimize sum(w in WORKER, t
```

Type error: array index 1 must be `WORKER', but is `TASK'

Production planning with enums

```
int: k;    % number of products to choose
int: limit;  % available weight limit
int: n;       % number of products

include "all_different.mzn";

enum PRODUCT = P(1..n);
array[PRODUCT] of int: wght;
array[PRODUCT] of int: profit;
array[1..k] of var PRODUCT: chosen;
constraint all_different(chosen);
constraint sum(i in 1..k)(wght[chosen[i]]) <= limit;
solve maximize sum(i in 1..k)(profit[i]);
```

Type error: array index must be `PRODUCT', but is `int'

Part 3: units

But what about actual numbers?

They’re (almost) never just numbers

```
int: k;     % number of products to choose
int@kg: limit;  % available weight limit

enum PRODUCT; % set of products available
array[PRODUCT] of int@kg: wght;
array[PRODUCT] of int@c: profit;
array[1..k] of var PRODUCT: chosen;
constraint alldifferent(chosen);
constraint sum(i in 1..k)(profit[chosen[i]])
           <= limit;
solve maximize sum(i in 1..k)
                   (wght[chosen[i]]);
```

Choose k products from a set PRODUCT within a weight limit to maximize profit

The linear equation involves two incompatible units

Unit types
(coming soon to MiniZinc)

Dimensions: are types of unit which are compatible

- `unit type weight;`
- `unit type worth;`

Units: are the units for a dimension. Each can be converted to another

- `unit weight: g;`
- `unit weight: kg = 1000@g;`
- `unit weight: tonne = 1000@kg;`
- `unit weight: uston = 907.185@kg;`
- `unit worth: c;`
- `unit worth: dollar = 100@c;`

Derived unit types

Dimensions: can be derived from others

- `unit type distance;`
- `unit distance: m;`
- `unit distance: km = 1000@m;`
- `unit type area = distance*distance;`
- `unit area: sqm = m * m;`
- `unit type time;`
- `unit time: s;`
- `unit time: minute = 60@s;`
- `unit time: hour = 60@minute;`
- `unit type velocity = distance/time;`
- `unit velocity: kph = km / hour;`

Derived unit types

Automatic coercion:

```
var int@kg: people_weight = sum(p in People) (person_weight[p]);
var int@g: food_weight = sum(p in People) (food_weight[dinner[p]]);
var int@g: total_weight = people_weight + food_weight;
```

Not allowed (no automatic rounding):

```
var int@kg: total_weight = people_weight + food_weight;
```

More units

(see the paper for details)

- Abstract unit types: when all that matters is that it’s a unit (we don’t care which one)
- Coordinate types:
  - Distinguish between e.g. points and distances
  - Example start times versus durations in a scheduling problem
- Counting/per element unit types

https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2025.10

Part 4: objects

A typical model structure
(let’s look at the assignment problem again)

- We use an enum to identify “things”, entities, objects
- We use arrays to express attributes of objects (cost, duration)
- Variables range over the entities, element constraints allow us to access attributes
- But many systems are modelled using Object Oriented techniques
  - UML / class diagrams
  - ERDs / data bases

MiniZinc with Objects
(coming soon-ish)

- Extend MiniZinc syntax with classes
- Variables (and class attributes) can refer to other objects
- We can “create” new objects on demand
- Example: configuration problems

Example: clinic scheduling

(slightly contrived example…)

- Schedule a small network of clinics, each serving a group of patients.
- Every patient is seen on a day and cared for by a nurse, and higher-risk patients also need a follow-up visit.
- Each clinic has to keep a balanced caseload

Example: clinic scheduling

Traditional model

```
enum Clinic = { C1, C2 };
enum Patient = { P1, P2, P3, P4 };
enum Nurse = { N1, N2, N3, N4 };

array[Clinic] of set of Patient: clinic_patients = [
  { P1, P2 },
  { P3, P4 }
];
array[Patient] of set of Nurse: patient_nurses = [
  { N1 },
  { N2 },
  { N3 },
  { N4 }
];

array[Clinic] of var 1..2: clinic_zone;
array[Patient] of var 1..2: patient_risk;
array[Patient] of var 1..2: patient_day;
array[Patient] of var bool: followup_occurs;
array[Patient] of var 1..2: followup_token;
array[Nurse] of var 1..1: nurse_hours;
```

Example: clinic scheduling

Object-oriented model

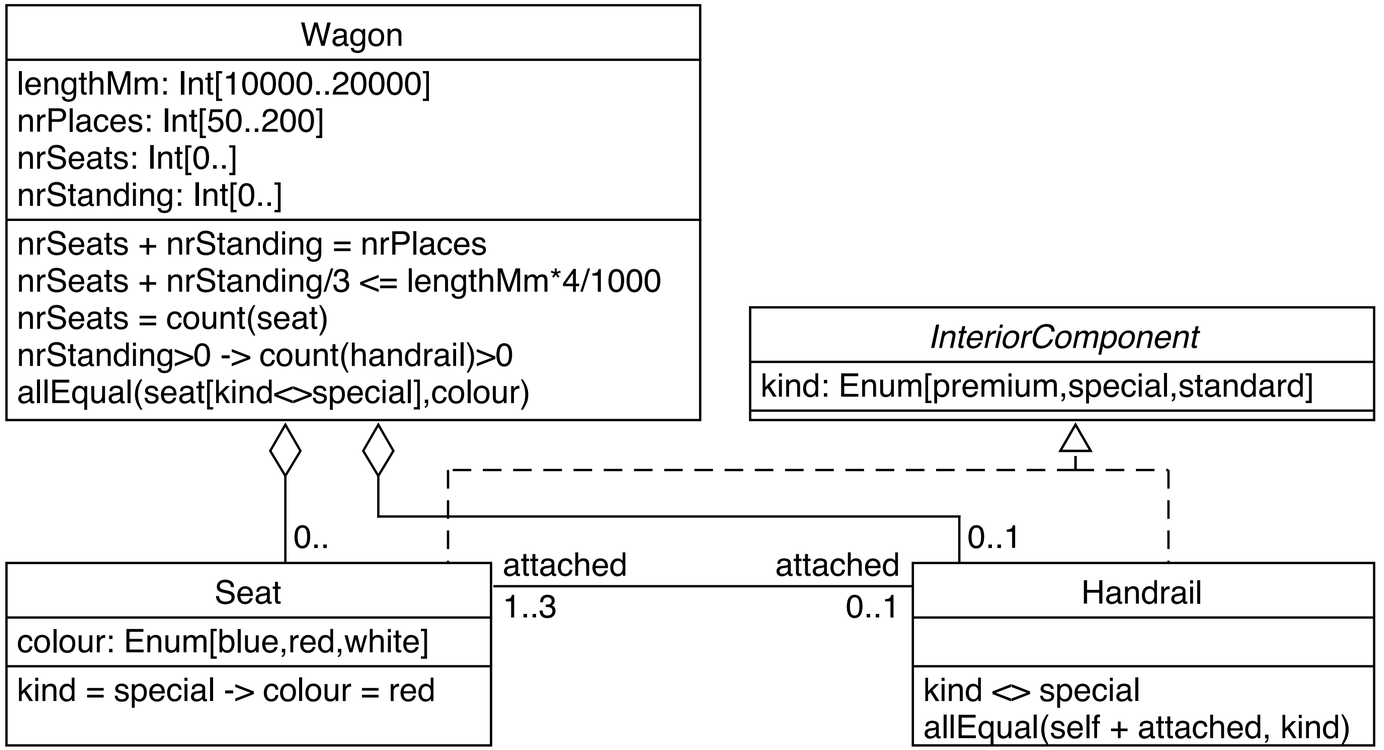
```
class Nurse (
  var 1..maxHours: hours;
);

class Patient (
  var 1..2: risk;
  var DAY: day;
  var opt new Followup: followup;
  var set(1..maxNurses) of new Nurse: nurses;
  constraint occurs(followup) <-> risk = 2;
  constraint not occurs(followup) \/ followup.token = 2;
  constraint sum(n in nurses)(n.hours) = totalHours;
);

class Clinic (
  var 1..2: zone;
  var set(1..k) of new Patient: patients;
  constraint sum(p in patients)(p.risk) <= 8;
  constraint sum(p in patients)(p.day) = 3;
);
```

Example: configuring a railway wagon

(see paper for details)



https://link.springer.com/article/10.1007/s10601-025-09380-3

Example: configuring a railway wagon
(see paper for details)

```
class Wagon (
  LengthMm: lengthMm;
  Places: nrPlaces;
  0..max(Places): nrStanding;
  set(0..maxSeats) of new Seat: seats;
  int: nrSeats = card(seats);
  opt new Handrail: handrail;

  constraint nrSeats + nrStanding = nrPlaces;
  constraint nrSeats + nrStanding div 3 <= (lengthMm div 1000)*4;
  constraint nrStanding > 0 -> handrail != <>;
);

class InteriorComponent ( Kind: kind; );

class Handrail extends InteriorComponent (
  set(1..3) of Seat: attached;
  Wagon: wagon;
  constraint kind != special;
  constraint forall(s in attached)(kind = s.kind);
);

class Seat extends InteriorComponent (
  Colour: colour;
  opt Handrail: handrail;
  Wagon: wagon;
  constraint if kind = special then colour = red endif;
);
```

Thanks!
(summary)

- MiniZinc allows you to model problems at a high level of abstraction
- Logic constraints and element constraints replace 0/1 and big-M
- Enumerated types make models more type safe
- Unit types build on this concept
- Object types simplify integration with other systems, and modelling of complex problems (such as configuration problems)