

AUTOMATED CHECKING AND GRADING OF CP MODELS

Carleton Coffrin, Jip Dekker, Jimmy H.M. Lee, Jason Nguyen, [Peter J. Stuckey](#), Guido Tack,
Allen Zhong

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
- MiniZinc Project Files
- Non-MiniZinc Checking/Grading



WHY CHECK MODELS?

- Learning Modelling is **HARD!**
- The **more feedback** we give learners the **better**
- Projects with automated feedback allow
 - Modellers to understand **how** their model went wrong
 - Help modellers find **where** their model went wrong

WHY CHECK MODELS (MOOC)?

- Massive Online Open Coursewares (MOOCs)
 - Have many thousands of students
 - Need to grade either:
 - by peer (very challenging for complex technical subject); or
 - Automatically
- Our MOOCs have >60000 enrollees
- Automatic feedback is vital for students to progress in these challenging courses

OUTLINE

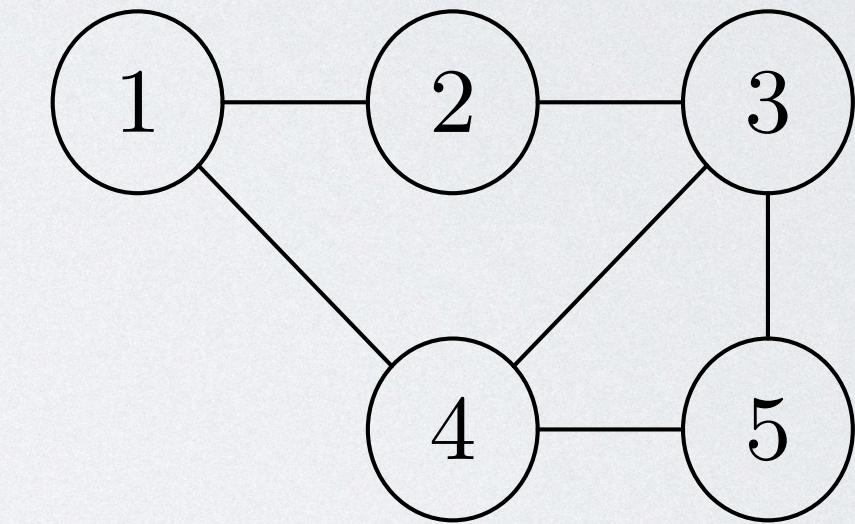
- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



GRAPH COLORING EXAMPLE

- Simple colouring model

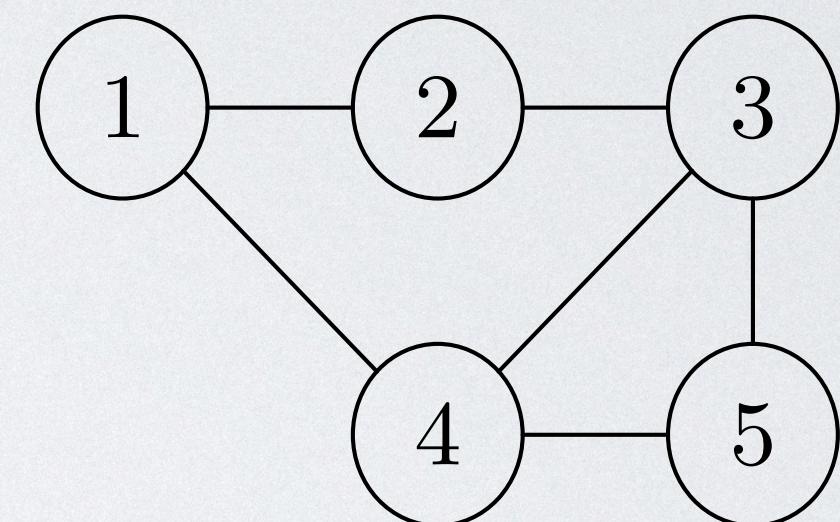
```
int: n;                                % number of nodes
set of int: NODE = 1..n;
array[int] of tuple (NODE,NODE): e;      % (undirected) edges
set of int: COLOR = 1..n;                % colors
array[NODE] of var COLOR: c;            % decision: node color
constraint forall( p in e )
    ( c[p.1] != c[p.2] ); % coloring constraint
var COLOR: nc = max(c);                % minimize used colors
solve minimize nc;
```



- Data file: `d.dzn` holds `n = 5; e = [(1,2),(1,4),(2,3),(3,4),(3,5),(4,5)];`

BASIC CHECKING

- Given data: `(d.dzn) n = 5; e = [(1,2),(1,4),(2,3),(3,4),(3,5),(4,5)];`
- Is `c = [1,2,3,3,2]; nc = 3;` a solution?
- We can check **simply** by running the (correct) model
 - `minizinc color.mzn d.dzn -D"c = [1,2,3,3,2];"`
 - MiniZinc responds: `Warning: model inconsistency detected\n in call
'forall'\n in array comprehension expression\n with p = (3,4)\n in
binary '!='` operator expression



BASIC CHECKING

- For this example, quite a good error message from compiler
- Usually just =====UNSATISFIABLE=====
- A correct model can check answers
- Beware: symmetry breaking! `constraint seq_precede_chain(c);`
- Can reject correct answers: e.g. `c = [2,1,2,1,3];`

ASIDE MULTIPLE ASSIGNMENTS

- Why not
 - `minizinc color.mzn d.dzn -D"c = [1,2,3,3,2]; nc = 3;"`
 - Response: Error: type error: multiple assignment to the same variable
- Fix
 - `minizinc color.mzn d.dzn -D"c = [1,2,3,3,2]; nc = 3;" --allow-multiple-assignments`

MINIZINC BASIC CHECKING

- Flag: `-output-mode dzn`
 - Outputs all declared variables without RHS definitions
 - Perfect as input to be checked
- Flag: `-output-objective`
 - Outputs the objective value: `_objective = ?`
- Tool: `mzn-test.py` automates basic checking

BASIC CHECKING

- Correct/incorrect but usually no useful feedback
- Fine for e.g. scoring solutions in a competition 😜
- Not valuable as a teaching tool

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



USEFUL FEEDBACK

- CP languages are perfect for expressing [constraints](#)
- They are also good for writing [error detection](#)
- MiniZinc is very good for providing feedback on errors in solutions

CHECKER MODELS

- A weakness of basic checking is if the input format it not correct
 - e.g. `minizinc color.mzn d.dzn -D"c = [1,2,3,2];"`
 - Output: Error: evaluation error: Index set mismatch. Declared index set of `c' is [1..5], but is assigned to array with index set [1..4]. You may need to coerce the index sets using the `array1d` function.
- Checkers need to handle incorrect input as gracefully as possible

EXAMPLE CHECKER MODEL

- Takes the same data declarations, treats decisions as fixed

```
int: n;                                     % number of nodes
set of int: NODE = 1..n;
array[int] of tuple (NODE,NODE): e;          % (unidirected) edges
set of int: COLOR = 1..n;                     % colors
%%% parameter declarations indentical to model
array[int] of int: c;                        % decision: node color
int: nc;                                      % decision: no colors
%%% decision declarations relaxed and not var
.
```

EXAMPLE CHECKER MODEL

- Check each constraint *in the output* of the check model!

```
output [ if check(length(c) = n, "Color array \c does not have length \n") /\  
        forall(i in NODE)  
          (check(c[i] in COLOR,  
                "node \i is colored \c[i] outside range 1..\n") /\  
                 forall(p in e)  
                   (check(c[p.1] != c[p.2],  
                         "adjacent nodes \p.1 and \p.2 are both colored \c[p.1]\n") /\  
                          let { int: colors_used = card({ co | co in c}); } in  
                            check(nc = colors_used,  
                                  "Declared objective \nc not equal to number of colors used \ncolors_used\n")  
                        then "ALL CONSTRAINTS HOLD\n"  
                        else "ERROR in solution"  
                      endif );  
.
```

Check input length

Check valid values

Check colouring constraint

Check objective is correct

Note that use of $nc = \max(c)$ INCORRECTLY fails correct solutions

FEEDBACK

- The checker gives feedback, e.g.
 - `minizinc d.dzn color.mzc.mzn -D"c = [1,2,3,3,2]; nc = 3;"`

ERROR: adjacent nodes 3 and 4 are both colored 3
ERROR in solution

- Checker models can be run with the original model, e.g.
 - `minizinc color.dzn d.dzn color.mzc.mzn -a`
 - Will check every solution created by `color.mzn`

CHECKING LIBRARY

- The `check` function is just MiniZinc

```
test check(bool: b, string: s) =  
    if b then true else trace( "ERROR: " ++ s, false) endif;
```

- We have a set of standard checking functions, although we don't distribute them with Minizinc currently

- `check_int`: check an integer is in domain
- `check_array_int`: check each integer in an array in domain
- `check_alldifferent`: check alldifferent holds
- ...

OBJECTIVE FEEDBACK

- It is tempting to ignore the objective
- **LESSON LEARNT:**
 - Many students make **mistakes** in defining the objective
 - Checkers should give **feedback** about this too

CHECKERS AND PROJECTS

- Default name for a checker for `model.mzn` is `model.mzc.mzn`
- The `.mzc` tells MiniZinc its a checker file
- By default a checker will be run if available in the project

CHECKERS AND PROJECTS

Zn color.mzn — Untitled Project

New model Open Save Copy Cut Paste Undo Redo Shift left Shift right Run + check Solver configuration Gecode 6.3.0 Show configuration editor

color.mzn d.dzn color.mzc.mzn

```
1 int: n; % number of nodes
2 set of int: NODE = 1..n;
3 array[int] of tuple (NODE,NODE): e; % (unidirected) edges
4 set of int: COLOR = 1..n; % colors
5 array[NODE] of var COLOR: c; % decision: node color
6 constraint forall( p in e )
7     ( c[p.1] != c[p.2] ); % coloring constraint
8 var COLOR: nc = max(c);
9 solve minimize nc; % minimized used colors
10
11
12
```

Output

Hide all dzn default Errors

```
c = [2, 1, 2, 1, 3];
=====
=====

Finished in 296msec.
Running color.mzn, d.dzn, color.mzc.mzn
% Solution checker report:
% ALL CONSTRAINT HOLD
c = [2, 1, 2, 1, 3];
=====

Finished in 283msec.
```

Line: 5, Col: 62 283msec

Run and check

Checker output

CHECKER AND PROJECTS

- Visible checkers give too much info on the project, e.g.

```
forall(p in e)
  (check(c[p.1] != c[p.2],
    "adjacent nodes \p{1} and \p{2} are both colored \c[p.1]\n"))
```

- So checkers can be compiled/encrypted
 - `minizinc --compile-solution-checker color.mzc.mzn`
 - Or using compile button in the IDE
- Encrypted form usually included in the project
- Note: **not seriously encrypted** a truly dedicated student could eventually decrypt.

CHECKERS AND PROJECTS

Zinc IDE screenshot showing a project named "color.mzp" with files "color.mzn", "d.dzn", and "color.mzc". The "color.mzc" file is selected. The "Output" tab shows the solver's log.

Project:

- color.mzp *
- Models
 - color.mzn
- Data (right click to run)
 - d.dzn
- Checkers (right click to run)
 - color.mzc
- Solver configurations
- Other files

Output:

```
=====  
Finished in 873msec.  
Running color.mzn, d.dzn, color.mzc  
ERROR: Declared objective 4 not equal to number of colors used 3  
ERROR: Declared objective 4 not equal to number of colors used 3  
% Solution checker report:  
% ERROR in solution  
c = [3, 2, 3, 2, 4];  
=====  
=====  
Finished in 279msec.
```

Line: 11, Col: 33

Project file

Project contents

Encrypted checker

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



HIDDEN VARIABLES

- Sometimes the variables we want to check a constraint
 - Aren't part of the solution
- How do we handle this?
- Now we find out why checking is done in output!

PHOTO LINEUP EXAMPLE

- Consider lining up a list of people in a photo such that:
 - No more than two of a gender in a row
 - Minimising total distance between people adjacent in the list
- Starting model

```
enum PERSON;  
int: n = card(PERSON);  
enum GENDER = { M, F, 0 };  
array[PERSON] of GENDER: g;  
set of int: POSN = 1..n;  
array[PERSON] of var POSN: pos;
```

% set of people
% number of people
% set of genders
% the gender of each person
% set of positions
% decs: a position for each person

HIDDEN VARIABLES

- In order to enforce the gender constraint

- We want the inverse viewpoint

array[POSN] of var PERSON: who; % view: a person for each position

- Adding the viewpoint to the initial model gives the game away
- We want to compute the viewpoint during checking

PHOTO LINEUP SOLUTION

- Full model

```
enum PERSON;
int: n = card(PERSON);                                % set of people
% number of people
enum GENDER = { M, F, O };
array[PERSON] of GENDER: g;                            % set of genders
% the gender of each person
set of int: POSN = 1..n;                             % set of positions
array[PERSON] of var POSN: pos;                        % decs: a position for each person

array[POSN] of var PERSON: who;                         % view: a person for each position
include "inverse.mzn";
constraint inverse(pos,who);                           % channel from decisions to view
constraint forall(i in 1..n-2)
  (g[who[i]] != g[who[i+1]] \/
   g[who[i+1]] != g[who[i+2]]);
solve minimize sum(p in PERSON where p < max(PERSON))
  (abs(pos[p] - pos[enum_next(PERSON,p)]));
```

CHECKING WITH HIDDEN VARIABLES

- The checker computes the values of hidden variables

- BUT make sure they can take a value

```
array[PERSON] of int: pos;  
array[POSN] of var PERSON: who;  
constraint if forall(i in PERSON)(pos[i] in POSN) /\  
    alldifferent(pos)  
    then inverse(pos,who)  
    else forall(i in 1..n)(who[i] = min(PERSON)) endif;
```

(Hidden) decision variables

Validity check

- Hidden variables are decision variables for the checker model
- Usually best that they are fixed by constraints

Default value constraints

CHECKING WITH HIDDEN VARIABLES

- We can make use of hidden variables values in output

```
output [if check_array_int(pos, n, POSN, "pos") /\  
       check_alldifferent(pos, "pos") /\  
       forall(i in 1..n-2)  
         (check(g[fix(who[i])] != g[fix(who[i+1])] \/  
          g[fix(who[i+1])] != g[fix(who[i+2])],  
          "three people of the same gender " ++  
          "\g[fix(who[i])] " ++  
          " in positions \i.. \i+2 \n")) /\  
let { int: obj = sum(p in PERSON where p < max(PERSON))  
      (abs(pos[p] - pos[enum_next(PERSON,p)])); } in  
check(obj = _objective, "calculated objective \obj " ++  
      "does not agree with computed value \(_objective)\n")  
then "CORRECT: All constraints hold"  
else "INCORRECT" endif];
```

Checking ordering constraint
using hidden variables

Short circuit computation:
checking won't reach here if
inverse view not defined

- The **fix** function converts a var to a par (available in output only)

VISUALISING SOLUTIONS

- Another kind of feedback that checkers can provide is visualisation of solutions
 - We can just use output statements (ASCII visualisation)
 - Or provide arbitrary graphics (D3 javascript)
- We can show the “hidden viewpoint” without mentioning it **explicitly!**

VISUALISING SOLUTIONS

- Simple visualisation for the photo lineup problem

```
output [ "\(\who[i]) (\(g[who[i]])), " | i in 1..n ] ++ ["\n"];
```

- Shows the lineup with gender

- Easy to check if order constraint is violated

% Solution checker report:

% CORRECT: All constraints hold

HEL (F), LIAM (0), KARA (0), ED (M), JIM (M), ANN (F), BOB (M),
pos = [6, 7, 5, 4, 3, 2, 1];

CHECKERS SUMMARY

- MiniZinc model taking decision vars and objective as **fixed** arguments
- **Weaken** the type of decision variables to be as broad as possible
- Add variable declarations for **hidden** variables
- Constrain the hidden variables to compute the hidden viewpoint
 - Ensure the constraints cannot **fail**
- Build an output statement that **checks**
 - Type/domains of decision variables
 - Checks constraint and points out **exactly** where a constraint fails
 - Checks constraints on hidden variables
 - Recalculates the true objective and compares to input value

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
 - Non-MiniZinc Checking/Grading



GRADING CP SOLUTIONS

- Optimisation solutions can be automatically graded
 - First they must be correct
 - But then we can grade them on the objective value reached
- Two choices
 - Known data: the student just submits solutions (unlimited runtime)
 - Unknown data: the student submits the model

GRADING CP SOLUTIONS

- Known Data:
 - Advantage: student sees for which data their model works well/badly
 - Disadvantage: copying solutions is easy, does not check modelling
- Unknown Data:
 - Advantage: can test weird side cases/completeness of model
 - Disadvantage: students find it frustrating to improve on unseen data

AUTO GRADING

- The autograder system supports both
 - Known data
 - By default run on the students machine with fixed runtime
 - Model submission/Unknown Data
 - Run on many data instances on the server
 - Usually a short runtime

CHECKING + AUTO GRADING

- For assignments we usually provide a **very basic checker**
 - checks that the output from the model is the correct format
- Detailed checkers:
 - great for **self directed learning**
 - not so great for assessing students skills and knowledge

BUILDING A GRADER

- Similar to a checker:
 - Takes the input data
 - Also a set of objective value thresholds for each instance
- array[int] of float: thresholds;
- **LESSON LEARNT:** Build a [complete error checker](#) with detailed messages
- If the solution is valid compute score using thresholds otherwise 0

BUILDING A GRADER

- We build a detailed error string (not using output statement)

```
function string: check(bool: b, string: s) =
    if b then "" else "ERROR: " ++ s endif;

string: errors = check(length(c) = n, "Color array \c does not have length \n") ++
concat(i in NODE)
    (check(c[i] in COLOR,
        "node \i is colored \c[i] outside range 1..\n") ++
concat(p in e)
    (check(c[p.1] != c[p.2],
        "adjacent nodes \p.1 and \p.2 are both colored \c[p.1]\n"));
```

- The detailed output available to marker but not to student

BUILDING A GRADER

- We usually assign a grade depending on the proportion of thresholds passed

```
float: grade = if errors != "" then 0.0  
            else mgrade(_objective,thresholds) endif;
```

```
function float: mgrade(int: v, array[int] of float: t) =  
    let { int: l = length(t);  
          int: p = arg_max([ v < t[i] | i in index_set(t)] ++ [true]); }  
    in (p-1) / l;
```

- e.g Maximising with thresholds [0,20,25,29,30] and obj 26 gives 0.6

- This is all programmed in the grader *as you want it*

- Write a grading function using the thresholds in any way you choose!

GRADING MODELS

- For MOOCs grading of submissions must be automatic
- For Monash subjects we use
 - Auto grading **only** for the first assignment
 - Auto grading **plus** grading a written report for later assignments

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
- [MiniZinc Project Files](#)
- Non-MiniZinc Checking/Grading



MINIZINC PROJECT FILES

- MiniZinc allows the creation of projects including:
 - Models: usually a starting model with correct data defines
 - Data: a directory of data files
 - Checker: encrypted
 - Submission links: so submission can be made from the IDE
- Loading a project file brings the IDE to a fixed state

PROJECT FILES

The screenshot shows the Zn IDE interface with the following components:

- Toolbar:** Includes New model, Open, Save, Copy, Cut, Paste, Undo, Redo, Shift left, Shift right, Run + check (set to Chuffed 0.12.1), and Submit to FIT5216 S1 2023.
- Project Explorer:** Shows the project structure:
 - Assignment 1 Airdefence Planning
 - Models
 - Data (right click to run)
 - Checkers (right click to run)
 - Solver configurations
 - Other files
- Code Editor:** Displays the file `airdefence.mzn` with the following content:

```
1 % Building an airdefence plan
2 int: W; % width of area
3 set of int: COL = 1..W;
4 int: H; % height of area
5 set of int: ROW = 1..H;
6
7 array[ROW,COL] of int: value; % value of position. 0 means unavailable
8 % different units available
9 enum EQUIPMENT;
10
11 array[EQUIPMENT] of int: cost; % cost of unit
12 array[EQUIPMENT] of int: avail; % number available
13 array[EQUIPMENT] of int: radius; % max defense radius
14
15 int: budget; % budget for equipment;
16 int: limit; % max number of equipment;
```
- Output Panel:** Shows a "Hide all" button.

Starting model:
Note no decision variables

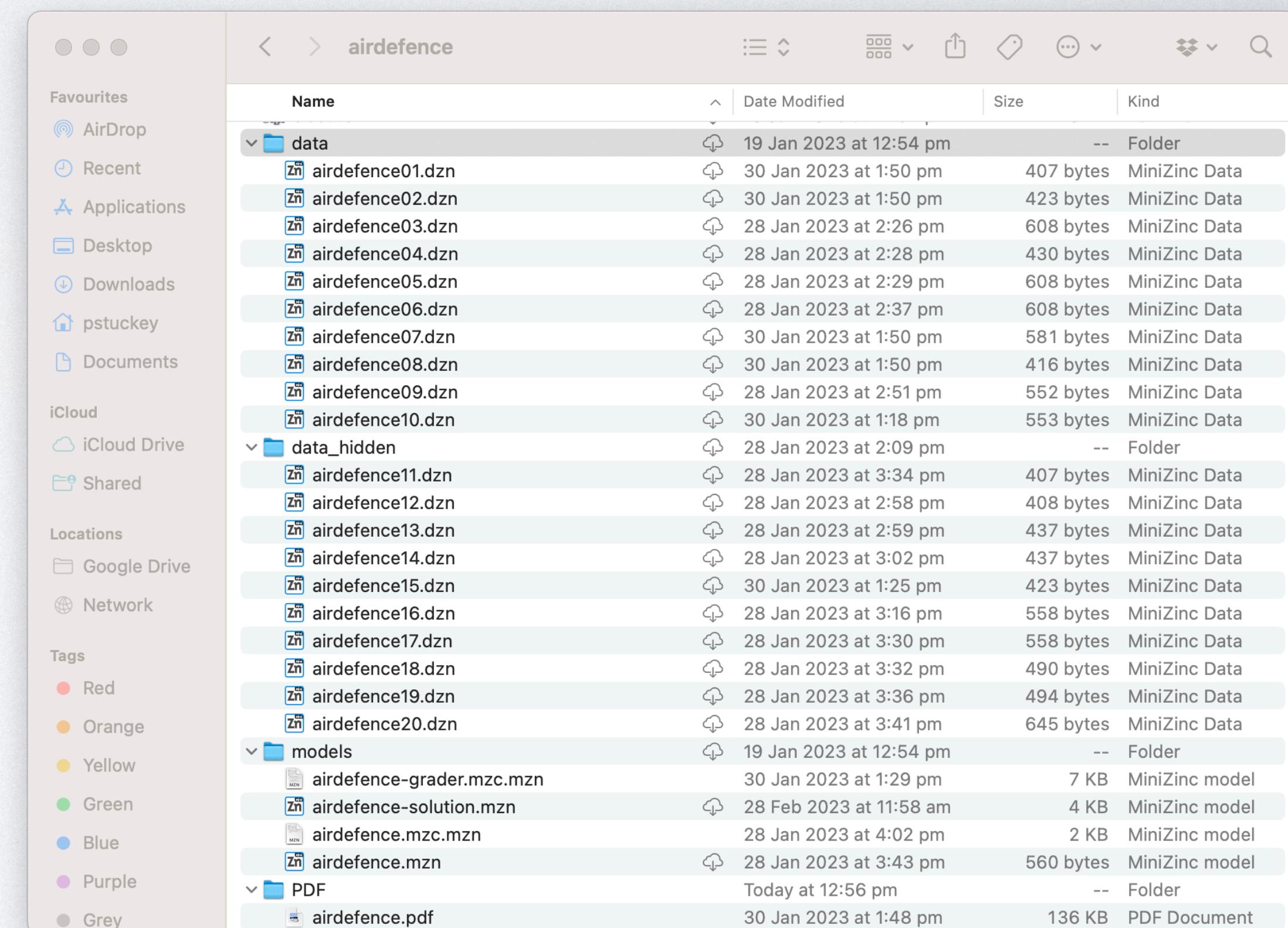
Data file directory

(Basic) Checker included

Submit button

BUILDING PROJECTS

- We have infrastructure for constructing projects
- Components:
 - **data**: visible data instances
 - **data_hidden**: hidden instances (model checking)
 - **models**: starting model, full solution, checker, grader
 - **PDF**: document describing the project
- We submit a zip file to the project builder page

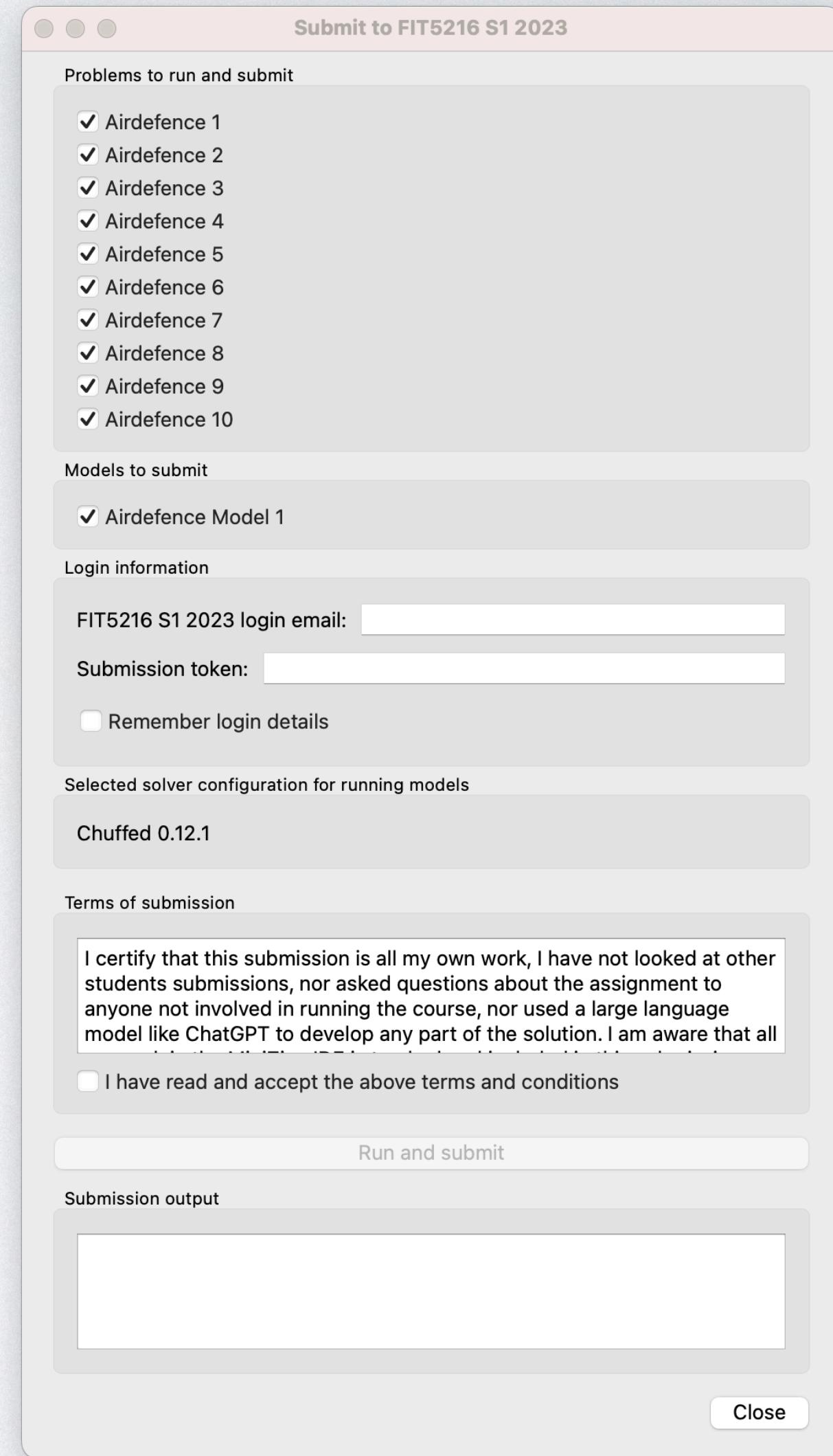


BUILDING PROJECTS

- **LESSON LEARNT:** Build a full solution to the assignment yourself
 - Useful for testing checker, grader, particularly error messages
 - Used for setting thresholds for each instance
- **Test grader well**
 - When the grader is **wrong** you will suffer
 - Beware of “**correct**” solutions that your solution would never generate
 - The infrastructure allows it to be changed (and automatically regrades)
 - Build a **visualiser** if its easy enough

SUBMISSION OF PROJECTS

- The submit button open a submission window
 - Student ID
 - Submission token (id verification)
 - Choice of which known data/whether unknown data is run
 - Perhaps some statement to acknowledge
 - Submits via web interface



STUDENT INTERFACE

- Students can examine
 - all feedback from all their submissions
 - all text of all their submissions
 - Leaderboard if enabled
- By default mark is maximum of all submissions
- Submission numbers can be limited

The screenshot shows a Moodle-based student interface. At the top, there's a navigation bar with tabs for Home Page - my.monash, FIT5216_S1_2023: Assignment 1, and other course links. The main content area displays the results for 'Assignment 1 MiniZinc submission'. It includes a sidebar with 'Quick Links' to Unit Previews, FIT5216 Handbook, My Grades, and IT Student Portal. A 'Teaching Excellence Awards' section is also visible. The assignment results table shows four submissions with scores of 1/1, 10/10, 20/20, and 20/20 respectively. Below the table, there's a 'Your Submissions' section listing five entries with dates ranging from 27 February to 09 March 2023, each with a 'Regrade' and 'Download' link. At the bottom, there are links to 'Assignment 1 Specification' and 'Assignment 1 Moodle submission (Weight: 5%)'.

Date	Score	Action	Action
09 March 2023 at 11:01 AM AEDT	20/20	Regrade	Download
28 February 2023 at 1:27 PM AEDT	13/20	Regrade	Download
28 February 2023 at 1:02 PM AEDT	20/20	Regrade	Download
27 February 2023 at 10:17 AM AEDT	20/20	Regrade	Download

INSTRUCTOR INTERFACE

- Instructors can
 - Examine all submissions, and all (detailed) feedback
 - View detailed log of submission
 - Impersonate an individual student
 - Modify grader and regrade some or all solutions
 - Modify project (but students need to re-download)
 - Examine grader queue

The screenshot shows a web browser window for the Monash LMS. The URL is lms.monash.edu/mod/lti/view.php?id=11405614. The page title is "FIT5216_S1_2023: Assignment". The header includes the Monash University logo and several icons. The main content area displays the title "FIT5216 Modelling discrete optimisation problems -" and the subtitle "Assignment 1 MiniZinc submission". Below this, there is a section for "Airdefence 5" with a "Feedback" box stating "CORRECT: no errors found". A "Logs" section contains a large block of text representing the command-line output of the grading process, including logs for the grader starting, submission parsing, exercise initialisation, and solution grading.

```
INFO:root:Grader started: ['/worker.py']
INFO:root:Submission partId: Kio9Sago9P
INFO:root:Initialising exercise library from /shared/assignment/meta.yml
INFO:root:Exercise Kio9Sago9P parsed as: SolutionExercise(name='Airdefence
5', checker=PosixPath('/shared/assignment/models/airdefence-
grader.mzc.mzn'), timeout=datetime.timedelta(seconds=15), solver='gecode',
param_file=None, UNSAT=False,
data=PosixPath('/shared/assignment/data/airdefence05.dzn'), thresholds=[0.0,
60.0, 67.0, 68.0])
INFO:root:Grading solution exercise `Airdefence 5'
INFO:root:Submission contained the OPTIMAL SOLUTION status
INFO:root:Run /shared/assignment/models/airdefence_grader.mzc.mzn with
```

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



NON-MINIZINC CHECKING/GRADING

- What if my projects aren't in MiniZinc?
 - WHY? 
- Most of the infrastructure can still be used
 - HOW?

NON-MINIZINC CHECKING/GRAADING

- Obviously we don't support Essence/OPL/Gecode MiniModel/MyFavoriteSolver/ [model submissions](#)
- But the infrastructure can be used for [known data](#) checking/grading
- Define MiniZinc versions of the decision variables
- For each instance build a MiniZinc data file with
 - Instance number, sizes of each of decision variables
- Give a template MiniZinc model for students to fill in the solutions they find

NON-MINIZINC CHECKING/GRAADING

- Solution file (.mzn)
- Checker works as usual
- Submission and grader work as usual

The screenshot shows the Minizinc IDE interface. The top bar displays the title "nonmzn.mzn — Untitled Project". The toolbar includes standard file operations (New model, Open, Save, Copy, Cut, Paste, Undo, Redo, Shift left, Shift right) and a "Run" button set to "Gecode 6.3.0". A "Solver configuration" dropdown is also visible.

The main code editor window contains the following Minizinc code:

```
1 int: instance_no; % instance number (in data file)
2 int: size1; % size of solution_vars_1
3 int: size2; % size of solution_vars_2
4 array[1..size1] of var int: solution_vars_1;
5 array[1..size2] of var int: solution_vars_2;
6
7 constraint if instance_no = 1 then
8     solution_vars_1 = [1,5,2,3,2] /\ % values added by student
9     solution_vars_2 = [123,452,367,146,241,8,5,4,2,5]
10    elseif instance_no = 2 then
11        solution_vars_1 = [4,5,9] /\
12        solution_vars_2 = [1018,231,146,909,562,673]
13    else
14        solution_vars_1 = [4,5,9,8] /\
15        solution_vars_2 = [1018,231,146,12,1005,345,123,13]
16    endif
```

The bottom panel shows the "Output" window with the following log:

```
Running nonmzn.mzn
solution_vars_1 = [1, 5, 2, 3, 2];
solution_vars_2 = [123, 452, 367, 146, 241, 8, 5, 4, 2, 5];
-----
Finished in 92msec.
```

At the very bottom, the status bar indicates "Line: 8, Col: 49" and "92msec".

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



EXPERIENCE

- We have used some form of auto grading since 2016
 - First Coursera Course
 - Didn't use the "output trick" had two graders
 - One to check the hidden variables were defined and compute them
 - One to check the solution with hidden variables
 - Used Python-based submission script rather than projects
 - On Coursera more than 60000 students probably > 500K assignments marked

EXPERIENCE

- We use the same infrastructure for Monash modelling course
- 3 assignments: grader + format checker
 - Make up assignment marks
- 20 workshop questions: detailed feedback checker
 - Participation marks only
- In the 2023 version: 80 students
 - a total of 8043 assignment submissions: 33 per person per assignment!
 - A total of 2143 workshop submissions (remember this is not number of checks)
 - Any submission gets the full participation marks, so students did work to get full marks

EXPERIENCE

- We have other infrastructure built, used in our online Monash course
- Peer feedback
 - After submission date closes
 - Each student is asked to give feedback on X other students models
 - The feedback is made available to the original student
 - The feedback given by a student is used in computing their grade.
- Peer feedback is a useful learning tool, we plan to use it for workshop questions

OUTLINE

- Checking Models
 - Basic checking
 - Error messages
 - Hidden variables
- Grading Models
 - Grading by objective
 - MiniZinc Project Files
- Non-MiniZinc Checking/Grading



CONCLUSION

- Providing **detailed feedback** to modellers about errors in their solution is:
 - Not too difficult for CP problems
 - Very useful for student learning
- Providing **automatic grading** for assignments is
 - Required for MOOCs
 - Useful for any course (allows multiple submissions/learning/improvement)
- We hope you can take some of these ideas/tools and **make use of them**

QUESTIONS

- Find MiniZinc at minizinc.org

