Automated grading of student constraint modelling projects

Hendrik Bierlee[1]
KU Leuven, Belgium

Tias Guns
KU Leuven, Belgium

Abstract

Auto-grading student constraint models requires a checker that validates solutions against the problem specification. In practice, for example in the Coursera Massive Open Online Course on MiniZinc [5], the checker is developed separately from the teacher's own constraint model [1]. This duplicates effort and risks inconsistencies that produce confusing feedback and unfair grades.

We present a framework for CPMpy [3] in which the checker is automatically derived from the teacher's constraint model. By adding a feedback template to each constraint, granular feedback is returned when a student solution violates a specified requirement constraint, domain, or objective. Decision variables in the teacher's model that are intentionally hidden from the student are assigned consistently via a *Maximum Satisfiable Subset* (MSS) computation. The same model also provides a performance baseline for grading, generates instances of various difficulty, and can be obfuscated and distributed for local checking. We illustrate the approach on an example *Flexible Jobshop Scheduling* (FJSS) project, demonstrating violation detection on a toy instance, and checker performance on larger instances.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Applied computing → Interactive learning environments

Keywords and phrases constraint modelling, auto-grading, education

Supplementary Material *Software (Source Code)*: `link`

Funding This research is partly funded by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt)

1 Introduction

Automated grading is a common approach to assessing student programming projects. For constraint modelling specifically, the prime example is the popular MiniZinc Coursera Massive Open Online Course (MOOC)[2], where the students submit their models to a server. The server solves the model under a time limit, then runs a *checker* on the generated candidate solution to return feedback on the correctness and performance of the submission. The feedback may include which constraints are violated and how. This feedback is a vital part of graded assessments and ungraded learning exercises.

However, implementing a checker for a project is a time-consuming and error-prone task. The main non-trivial aspect is to keep the checker consistent with the specification. Otherwise, the checker may give incorrect feedback on a student model solution, which can be costly. For the student, incorrect feedback can be confusing and demotivating. Worse, an update to the checker may require a complete re-check, which entails running all submitted models of all students over all instances for the entire time limit, in order to determine the

[1] Corresponding author
[2] `https://www.coursera.org/learn/solving-algorithms-discrete-optimization`

new best grade. If this happens during the project submission window, extensions may be needed to allow for the full time. The updated grade may be higher or lower for different students. A final concern is that if a checker is to be distributed to the student to allow local checking, then it should not reveal any knowledge outside the specification to the student. However, a procedural checker running asserts may look similar to a constraint model.

At ModRef'17, Coffrin *et al.* [1] described how solutions are checked for the MiniZinc MOOC. A constraint model solving the specified problem, which we will call a *solving model*, can be used to check a candidate solution by fixing the decision variables. An unsatisfiable solve result indicates a violation of the specification. This check is solver-free if all variables are fixed by the candidate solution. If there are *hidden variables* in the solving model, which are unspecified and thus not part of the candidate solution, a solver is used to assign them. However, this approach lacks granular feedback beyond correct and incorrect. Additionally, the solving model should not include *unspecified constraints* (e.g. symmetry breaking constraints to improve performance).

These obstacles motivate a separate *checking model*, in which constraints are annotated with feedback messages if violated, there are no unspecified constraints, and the non-hidden variables have become fixed parameters. The checking and solving models will look quite similar, to the point that checking models are obfuscated using e.g. a substitution cipher. However, the checking model can check but not solve the problem. When developing a project, a solving model is practically still required, given that the teacher needs to dry-run the specification, test the checking model, get performance baselines on instances, potentially distribute it as a sample model, and so on. Thus, two slightly different models need to be kept in sync with the specification at all times, which can cause the aforementioned issues.

In this paper, we build on this prior work and propose a CPMpy auto-grading framework which merges the functionality of both the checking and (efficient) solving model into a single *hidden model*. As needed, the hidden model can either be formulated as a solving model (for obtaining baseline objectives, generating instances), or as a checking model (for giving granular feedback, safely obfuscated for distribution). Furthermore, we propose a somewhat more robust (less reversible) method of checker model obfuscation, and handle another hidden variable issue using *Maximum Satisfiable Subset* (MSS) extraction.

2 Background

2.1 Constraint Programming (CP)

We cover the basics of *Constraint Programming* (CP) following Rossi *et al.* [7]. We define a *Constraint Satisfaction Problem* (CSP) $P \equiv (\mathcal{X}, D, \mathcal{C})$ as a tuple of integer variables $\mathcal{X}$, with domains defined by D, and $\mathcal{C}$ a set of constraints. A constraint c with scope $scope(c) = [x_1, \ldots, x_n], x_i \in \mathcal{X}$ is defined (implicitly) as a set of solutions of the form $\{\hat{v}_1, \ldots, \hat{v}_n\}$ for its variable sequence. A *solution* of P is an assignment θ over variables $\mathcal{X}$ such that $\theta(x) \in D(x), x \in \mathcal{X}$ and $[\theta(x_1), \ldots, \theta(x_n)] \in c$ for all $c \in \mathcal{C}$ where $scope(c) = [x_1, \ldots, x_n]$. A *Constrained Optimisation Problem* (COP) extends a CSP with an objective function f over $\mathcal{X}$, yielding a tuple $\langle \mathcal{X}, D, \mathcal{C}, f \rangle$. A solution θ is *optimal* if there exists no other solution θ' achieving a better objective value, i.e. $f(\theta) \leq f(\theta')$ for minimization (or $\geq$ for maximization).

2.2 Maximum Satisfiable Subset (MSS)

Given hard constraints $\mathcal{C}_H$ and soft constraints $\mathcal{C}_S$ where $\mathcal{C}_H \cup \mathcal{C}_S$ is unsatisfiable, a *Maximum Satisfiable Subset* (MSS) is a maximal subset $S \subseteq \mathcal{C}_S$ satisfiable together with $\mathcal{C}_H$ [4].

3 Method

We use a *Flexible Jobshop Scheduling* (FJSS) modelling project as running example, with the following specification:

> Given an instance of n jobs, m tasks per job, k machines, and an $n \times m$ duration matrix (e.g. `{"n":3, "m":2, "k":2, "duration":[[3,2],[2,4],[1,3]]}`), the model contains variables X where $X[i, j]$ denotes the start time for job i, task j. Constraints: tasks within a job must be executed in order; start times are non-negative; each task must run on exactly one machine; tasks on the same machine cannot overlap. Objective: minimize total flowtime (i.e. sum of end times).

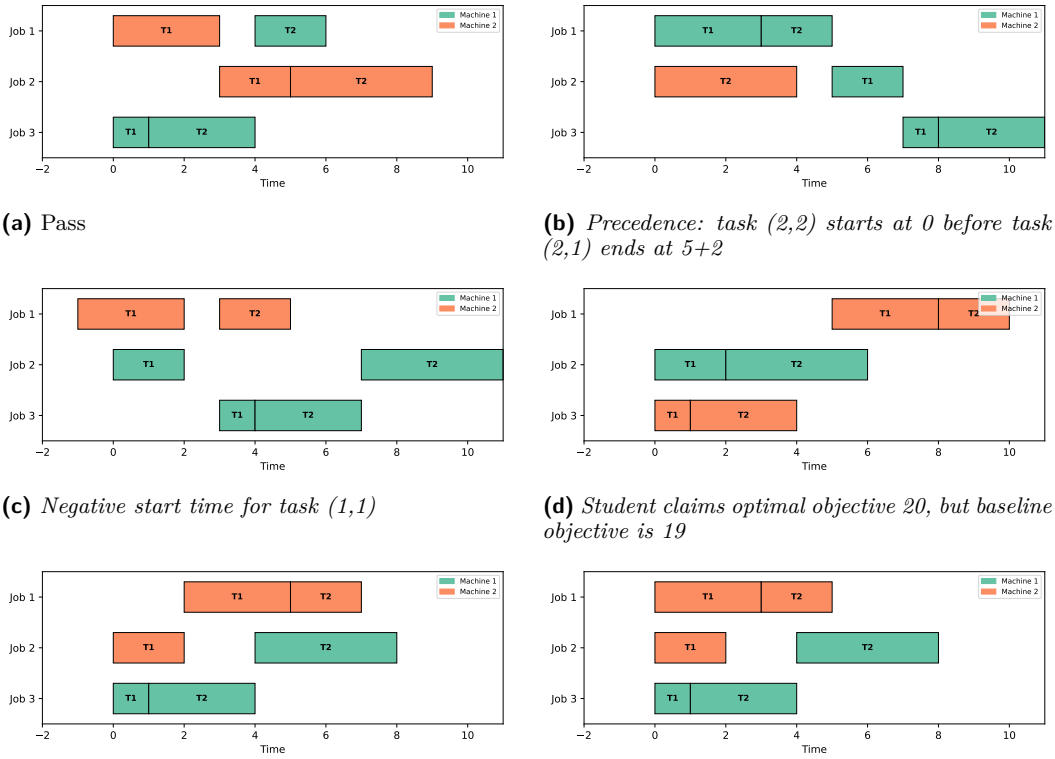
3.1 Checking via hidden model

Formally, the hidden model constructed for a given instance is a constraint model, either a CSP or COP (if an objective is specified). An example of a hidden model constructor for FJSS is shown in Listing 1. We assume the hidden model correctly implements the specification, and that in a preparation phase it has been solved to assert feasibility, and possibly find a baseline objective value o (not necessarily optimal). Note that most constraints are added with a *description*: a template string that describes the requirement in natural language and may include placeholders for variable values. A subset of the variables, $\mathcal{X}' \subseteq \mathcal{X}$ is communicated to the student in the spec. For FJSS, these are the start times, `X`.

```python
class FJSSModel(HiddenModel, FJSSPublic):
    def __init__(self, instance, checking=True):
        n, m, k = instance["n"], instance["m"], instance["k"]
        dur = np.array(instance["duration"])

        # Shared decision variables: start time for each task
        X = cp.intvar(0, dur.sum(), shape=(n, m), name="X")
        super().__init__(instance, shared_variables=X)

        for i in range(n):
            for j in range(m - 1):
                self.add(X[i, j] + dur[i, j] <= X[i, j + 1], descr=f"Precedence:
                ↪  task ({i + 1},{j + 2}) starts at {{{X[i, j + 1].name}}} before
                ↪  task ({i + 1},{j + 1}) ends at {{{X[i, j].name}}}+{dur[i,
                ↪  j]}")

        # Non-negativity / domains
        if checking:
            for i in range(n):
                for j in range(m):
                    self.add(X[i, j] >= 0, descr=f"Negative start time for task
                    ↪  ({i + 1},{j + 1})")

        self.minimize(sum(X[i, m - 1] + dur[i, m - 1] for i in range(n)))
```

Listing 1 Partial hidden model for the FJSS project

Now, the student submits a constraint model from which a candidate solution θ over $\mathcal{X}'$ is generated with objective value o', possibly with a claim of optimality. An optimal and



(a) Pass

(b) *Precedence: task (2,2) starts at 0 before task (2,1) ends at 5+2*

(c) *Negative start time for task (1,1)*

(d) *Student claims optimal objective 20, but baseline objective is 19*

(e) *Student objective value is 8, but hidden model computes a different value for the same assignment*

(f) *Tasks (1,1) and (2,1) overlap on machine 2*

Figure 1 Checker output for different candidate submissions on the example toy FJSS instance.

correct solution for the toy instance is shown in Figure 1a. First, θ is checked to ensure it assigns all shared variables. We first consider the simple case where all variables are shared, i.e. $\mathcal{X}' = \mathcal{X}$. Since θ is a total assignment, each constraint can be evaluated directly without a solver. If a constraint is violated, its description is rendered with the concrete values from θ, producing a human-readable feedback message. Figure 1b visualizes a student solution violating the precedence constraint, with the resulting feedback included in the caption. The domains in the hidden model are not explicitly checked, so that we can provide tight bounds when solving. For example, this is the case for the upper bounds of the start times, but the lower bounds are specified to be non-negative, so we explicitly add $x \geq 0$ for each $x \in \mathcal{X}$ in the hidden model (Figure 1c). Note that the upper bounds in the hidden model may be tighter than the student's to improve the performance of the solving model without restricting the student model.

We can verify that the student's objective value o' is correctly calculated by adding the constraint $f = o'$ (e.g. to catch the case where the student minimized makespan rather than total flowtime, see Figure 1e). If the solver reported optimality for (correct) objective value o', we can also add the constraint $f(\theta) \leq o$ (if minimizing) to check against sub-optimality (Figure 1d). Super-optimal candidate solutions need no explicit checks as they should already violate an existing constraint.

When constructing the hidden model, a `checking` argument indicates whether we will solve the problem or check a candidate solution: essentially whether to generate a solving or a checking model. For instance, the lower bound constraints are redundant for solving, and

are added only if `checking` is true. More importantly, we omit unspecified constraints (e.g. symmetry breaking constraints) if `checking` is true. If `checking` is true, we can intentionally replace a constraint by its decomposition to give more granular feedback.

3.2 Hidden variables

As discussed by Coffrin *et al.* [1], hidden variables, i.e. $\mathcal{X} \setminus \mathcal{X}'$, can be the key challenge to a modelling project. The PHOTO problem is given as example, which has shared `pos` and hidden `who` variables, connected by an `Inverse(pos,who)` constraint. A MiniZinc solving and checking model are shown in Listings 4 and 5. A hidden model version for it is shown in Listing 2.

```python
class PhotoModel(HiddenModel, PhotoPublic):
    def __init__(self, instance, checking=True):
        n, gender = instance["n"], cp.cpm_array(instance["gender"])

        # Shared decision variables: position of each person (0-indexed)
        pos = cp.intvar(0, n - 1, shape=n, name="pos")
        super().__init__(instance, shared_variables=pos)

        self.add(cp.AllDifferent(pos), descr="Positions are not unique")

        if checking:
            for i in range(n):
                self.add(  # Position domain bounds are specified
                    (pos[i] >= 0) & (pos[i] <= n - 1),
                    descr=f"Person {i + 1} was not in range 0..{n - 1}",
                )

        who = cp.intvar(0, n - 1, shape=n, name="who")  # Hidden variables
        self.add(cp.Inverse(pos, who))  # who[j] = person at position j

        for j in range(n - 2):
            self.add(
                (gender[who[j]] != gender[who[j + 1]])
                | (gender[who[j + 1]] != gender[who[j + 2]]),
                descr=f"3 consecutive same gender at ({j + 1},{j + 2},{j + 3})",
            )

        # Minimize total distance between consecutively numbered people
        self.minimize(sum(abs(pos[i] - pos[i + 1]) for i in range(n - 1)))
```

Listing 2 Photo Lineup (after [1]): arrange n people in a line minimizing total distance between consecutively numbered people, with no more than 2 consecutive people of the same gender. The `who` variables (inverse of `pos`) are hidden.

The presence of hidden variables makes checking harder in two ways. First, to assign the hidden variables, we need to invoke a solver. Note that this effectively splits up the checker model into *soft* constraints (which can be violated, returning messages), and *hard* constraints, which connect hidden variables to shared variables and avoid trivial assignments. This leads to the second problem, namely that the hard constraints may make the checking model unsatisfiable for certain incorrect candidate solutions (e.g. one which has `pos` not

all different), which is an uninformative result. In the MiniZinc PHOTO checking model (Listing 5), this is resolved by adding *guards* to the hard constraints so that they remain satisfiable for any candidate solution.

In the hidden model version of the PHOTO problem (see Listing 2), hard constraints are added by omitting the description. To add a guard, we use constraints (e.g. by `(AllDifferent(pos)).implies(Inverse(pos, who))`), which are resolved by the solver, rather than using the output phase as is done in the MiniZinc checking model. To fix the hidden variables to their candidate solution, we also use explicit hard constraints, i.e. $C_\theta \equiv \forall_{x \in \mathcal{X}'} x = \theta(x)$.

However, by employing a *sequential checking strategy*, we can avoid the guard altogether in a natural way. Note that in Listing 2, the `AllDifferent(pos)` is naturally placed before the `Inverse`. If `pos` is not all different, the infeasibility will signal a natural end to the check. In general, we alternate between incrementally posting and solving hard constraints (initially C_θ), checking soft constraints, then incrementally solving the next hard constraint (to assign additional hidden variables, e.g. `Inverse(pos,who)`), and so on. If the hidden model is carefully ordered, soft constraints will always have all their variables assigned, and we will not get unsatisfiable without violating at least one soft constraint. If no ordering works, guards can be used.

3.3 Improving hidden variable assignments

Yet, a third problem, which cannot be prevented with a guard, is posed by the hidden variables in the FJSS model, namely the task-machine assignment variables, `Y` (Listing 3). Hard constraints (linear equalities) enforce a task-machine assignment to avoid a trivial assignment setting all `Y` to false. No guard is needed as clearly no shared variables are in scope, so there is no risk of infeasibility. The shared and hidden variables connect via the `NonOverlapOptional` constraints, which should be soft, as we want to relay how the candidate solution violates them. In fact, a decomposition is added when `checking` is enabled, in order to show on which machine which tasks overlap. However, solving the task-machine constraint yields an arbitrary task-machine assignment (e.g. all tasks assigned to the same machine) without regard for how many task-machine overlap soft constraints are violated.

Instead of an arbitrary assignment of hidden variables, we get a favourable one by extracting a MSS from the hidden model. The complement of the MSS are the soft constraints violated by the associated assignment of the shared **and** hidden variables. In Figure 1f, three tasks are scheduled at time 0, which cannot be covered by the two machines, leading to an unavoidable violation. However, everywhere else, the particular task-machine MSS assignment carefully avoids the overlap. While a MSS can be costly to compute or even construct (because every soft constraint requires reification), note that we only execute it on a subset of the soft and hard constraints due to the sequential checking strategy.

3.4 Obfuscation

As mentioned, due to the similarity between a checking model and a solving model, obfuscation is required if we are to provide local checking for students of graded projects (e.g. outside the context of self-directed learning). A cipher method will suffice in most cases, however, in open-source software (or in interpreted languages such as Python), even encryption is perfectly reversible in principle, since both key and method are exposed. A less thorough, but also less reversible method of obfuscation is to transform the constraints using equivalence-preserving mutations, similar to those used in fuzz testing of constraint modelling systems [8].

```python
        # Hidden variables: which machine does which job-task
        Y = cp.boolvar(shape=(n, m, k), name="Y")

        # Hard constraint: each task runs on exactly one machine
        for i in range(n):
            for j in range(m):
                self.add(sum(Y[i, j, :]) == 1)

        if not checking:
            for r in range(k):
                self.add(  # No two tasks on the same machine may overlap
                    cp.NoOverlapOptional(X.flatten(), dur.flatten(),
                    ↪  is_present=Y[:, :, r].flatten())
                )
        else:
            all_tasks = [(i, j) for i in range(n) for j in range(m)]
            for r in range(k):
                for (i1, j1), (i2, j2) in combinations(all_tasks, 2):
                    self.add(
                        (Y[i1, j1, r] & Y[i2, j2, r]).implies(
                            (X[i1, j1] + dur[i1, j1] <= X[i2, j2])
                            | (X[i2, j2] + dur[i2, j2] <= X[i1, j1])
                        ),
                        descr=f"Tasks ({i1 + 1},{j1 + 1}) and ({i2 + 1},{j2 + 1})
                        ↪  overlap on machine {r + 1}",
                    )
```

Listing 3 Hidden model for FJSS (continued): hidden variables and overlap decomposition

Concretely, we randomly shuffle constraint order (within soft/hard blocks, to preserve the sequential checking strategy) and commutative arguments; flip comparison sides ($a \leq b$ to $b \geq a$); swap strict and non-strict comparators ($a \leq b$ to $a < b{+}1$); and shift both sides by a random constant ($a \leq b$ to $a{+}k \leq b{+}k$). Note that we have more freedom to obfuscate if the feedback is less granular, which is more usually the case for graded assignments. The description templates attached to each constraint cannot be obfuscated further, which does limit the effect. We can go further by *transforming* the hidden model into a low-level representation such as *Conjunctive Normal Form* (CNF). We can even apply obfuscation before and after this process. However, this introduces new hidden variables with a mix of soft and hard constraints, so this idea is currently not implemented. An obfuscated hidden model for FJSS for the toy instance is shown in Listing 6.

3.5 Instance generation

The hidden model also serves as the basis for an instance generator. Each project defines a list of *instance configurations* (e.g. for FJSS: number of jobs, tasks, machines, and duration range). Instances are generated from each configuration to obtain the desired performance profile with regards to the hidden model within the set timeout. A more systematic approach to generating instances from a constraint model, e.g. AutoIG [2], can be incorporated.

Note that, unlike the MiniZinc checking model, a separate hidden model is created for each instance, as a CPMpy model is not generic over instances (there is no split such as between `.mzn` and `.dzn` files). This does mean that large instances can create large checkers,

especially if they are obfuscated as CNF. It also means that students cannot make and check new instances, which could or could not be a helpful limitation depending on the project context. A large instance may also become hard to check (especially if the MSS is used). The flow objective was intentionally chosen because even small instances are hard to solve, whereas a makespan objective (without further changes, such as machine-dependent durations) requires large and slow checker binaries.

4 Evaluation

Table 1 shows the checker performance on five FJSS instances of increasing size, using Google's CP-SAT solver [6] via CPMpy on an Apple M1 (8 cores, 16 GB RAM), solved single-threaded with a 60-second timeout. For each instance, we check three candidate solutions: the correct (optimal or best-found) solution, a random assignment, and a perturbed solution in which variables from the correct solution are changed one at a time (by ± 1–3) until a violation is detected. The solve time shows that the basic instance generation is a useful tool, as the third instance can be re-used with multiple seeds to find a range of instances. All checks complete in under 0.5 seconds, fast enough for interactive use.

Table 1 Checker performance on FJSS instances. Config: $(n, m, k, (dur_{\min}, dur_{\max}))$. Solve and check times in seconds.

Config	Solve	Check time		
		Correct	Random	Perturbed
$(3, 2, 2, (1, 5))$	0.007	0.03	0.04	0.04
$(4, 3, 3, (2, 6))$	0.061	0.05	0.11	0.11
$(5, 3, 3, (3, 8))$	32	0.07	0.17	0.16
$(5, 4, 3, (3, 8))$	TO	0.09	0.29	0.33
$(6, 4, 3, (3, 8))$	TO	0.12	0.44	0.38

We also report qualitative feedback from MSc. students enrolled at a constraint solving course which included two constraint modelling projects using initial prototypes of the checker. When asked which aspects of the course as a whole were good (not limited to the projects), out of 29 responses, the project was mentioned 20 times ('practical', 'challenging', 'realistic', 'fun'), and the auto-grader was mentioned specifically as well. When asked which aspects in the course could be improved, the project was mentioned 9 times out of 28 responses. The main issue was that the project was too difficult and time-consuming. This is primarily a matter of tuning the project workload to the number of course credits, but it also shows that an auto-grader with unlimited attempts sets a rigid bar.

5 Conclusion

We presented a CPMpy framework for automated grading of constraint modelling projects in which the checker is derived directly from a single hidden model. By annotating constraints with feedback templates, the framework provides granular violation messages; hidden variables are handled via a sequential checking strategy and, if needed, MSS computation; and the same model generates instances, produces baselines, and can be obfuscated for local distribution. The approach eliminates the need for a separate checker, reducing development effort and the risk of inconsistencies.

References

1 Peter J Stuckey Carleton Coffrin, Siqi Liu, and Guido Tack. Solution checking with minizinc. In *ModeRef2017, The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.

2 Nguyen Dang, Özgür Akgün, Joan Espasa, Ian Miguel, and Peter Nightingale. A framework for generating informative benchmark instances. In *International Conference on Principles and Practice of Constraint Programming*, 2022.

3 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

4 Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016. URL: `https://doi.org/10.1007/s10601-015-9183-0`, `doi:10.1007/S10601-015-9183-0`.

5 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, Lecture Notes in Computer Science, pages 529–543. Springer, 2007. `doi:10.1007/978-3-540-74970-7\_38`.

6 Laurent Perron and Frédéric Didier. Cp-sat. URL: `https://developers.google.com/optimization/cp/cp_solver/`.

7 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: `https://www.sciencedirect.com/science/bookseries/15746526/2`.

8 Wout Vanroose, Ignace Bleukx, Jo Devriendt, Dimos Tsouros, Hélène Verhaeghe, and Tias Guns. Mutational fuzz testing for constraint modeling systems. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, Girona, Spain, September 2-6, 2024*, LIPIcs, pages 29:1–29:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: `https://doi.org/10.4230/LIPIcs.CP.2024.29`, `doi:10.4230/LIPICS.CP.2024.29`.

A MiniZinc Photo solving and checking model

```
int: n;                           % number of people
set of int: PERSON = 1..n;        % set of people
enum GENDER = { M, F, O };        % set of genders
array[PERSON] of GENDER: g;       % the gender of each person
set of int: POSN = 1..n;          % set of positions

array[PERSON] of var POSN: pos;  % decisions: 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(i in 1..n-1)(abs(pos[i] - pos[i+1]));
```

4 MiniZinc solving model for the PHOTO problem.

```
int: n;
set of int: PERSON = 1..n;
```

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

array[POSN] of var PERSON: who;
int: _objective;

test alldifferent(array[int] of int: x) =
                 forall(i,j in index_set(x) where i < j)(x[i] != x[j]);

include "inverse.mzn";
constraint if alldifferent(pos) then inverse(pos,who)
           else forall(i in 1..n)(who[i] = 1) endif;

output [if
  check_array(pos, n, POSN, "pos") /\
  check_alldifferent(pos,"pos") /\
  check_array(fix(pos), n, PERSON, "who") /\
  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(i in 1..n-1)(abs(pos[i] - pos[i+1])); } in
    check(obj = _objective, "calculated objective \(obj) does not agree with
    ↪ objective from the model (\(_objective))\n")
  then
    "CORRECT: All constraints hold"
  else
    "INCORRECT"
  endif];

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

test check_alldifferent(array[int] of int: x, string: name) =
  forall(i, j in index_set(x) where i < j)
        (check(x[i] != x[j], name ++ "[\(i)] = \(x[i]) = " ++
         name ++ "[\(j)] " ++
         "when they should be different\n"));

test check_int(int: x, set of int: vals, string: name) =
  check(x in vals, "integer \(name) is not in values \(vals)\n");

function bool: check_array(array[int] of int: x, int: length, set of int: vals,
↪ string: name) =
  check(length(x) = length, "array \(name) is not of length \(length)\n") /\
  forall(i in index_set(x))
        (check(x[i] in vals, "element \(i) of array \(name), \(x[i]) is not in
        ↪ values \(vals)\n"));
```

5 MiniZinc checking model for the PHOTO problem (after [1]). Note the guard on `inverse`.

B Obfuscated FJSS hidden model

```
60 <= 60 + (X[0,0])
  description: Negative start time for task (1,1)
0 < (X[2,0]) + 1
  description: Negative start time for task (3,1)
-89 <= (X[1,0]) + -89
  description: Negative start time for task (2,1)
X[2,1] > -1
  description: Negative start time for task (3,2)
((X[2,0]) + 1) < (1 + (X[2,1]))
  description: Precedence: task (3,2) starts at {X[2,1]} before task (3,1) ends at {X[2,0]}+1
(sum(X[0,0], np.int64(3), -58)) < (sum(X[0,1], 1, -58))
  description: Precedence: task (1,2) starts at {X[0,1]} before task (1,1) ends at {X[0,0]}+3
53 < sum(1, 53, X[0,1])
  description: Negative start time for task (1,2)
82 < sum(X[1,1], 82, 1)
  description: Negative start time for task (2,2)
((X[1,1]) + 68) > (sum(68, X[1,0], np.int64(2), -1))
  description: Precedence: task (2,2) starts at {X[1,1]} before task (2,1) ends at {X[1,0]}+2
58 == sum(57, Y[2,0,0], Y[2,0,1])
1 == (Y[1,1,1]) + (Y[1,1,0])
(Y[0,0,0]) + (Y[0,0,1]) == 1
1 == (Y[1,0,0]) + (Y[1,0,1])
1 == (Y[0,1,1]) + (Y[0,1,0])
1 == (Y[2,1,0]) + (Y[2,1,1])
((Y[0,1,1]) and (Y[0,0,1])) -> ((((X[0,0]) + 3) <= (X[0,1])) or (((X[0,1]) + 2) < (1 + (X[0,0]))))
  description: Tasks (1,1) and (1,2) overlap on machine 2
((Y[1,1,1]) and (Y[0,1,1])) -> (((sum(np.int64(2), X[0,1], -71)) <= (-71 + (X[1,1]))) or (((X[1,1]) + 4)
↪  <= (X[0,1])))
  description: Tasks (1,2) and (2,2) overlap on machine 2
((Y[0,1,0]) and (Y[0,0,0])) -> ((((X[0,1]) + 2) <= (X[0,0])) or ((sum(X[0,0], np.int64(3), -12)) <
↪  (sum(X[0,1], -12, 1))))
  description: Tasks (1,1) and (1,2) overlap on machine 1
((Y[0,1,1]) and (Y[1,0,1])) -> ((((X[0,1]) + 2) < ((X[1,0]) + 1)) or (((X[0,1]) + -58) >
↪  (sum(np.int64(2), -58, -1, X[1,0]))))
  description: Tasks (1,2) and (2,1) overlap on machine 2
((Y[2,1,0]) and (Y[2,0,0])) -> (((3 + (X[2,1])) <= (X[2,0])) or (((X[2,1]) + -62) > (sum(X[2,0], -1, -62,
↪  np.int64(1)))))
  description: Tasks (3,1) and (3,2) overlap on machine 1
((Y[2,0,1]) and (Y[0,0,1])) -> (((-63 + (X[0,0])) > (sum(-1, -63, X[2,0], np.int64(1)))) or ((X[2,0]) >
↪  (sum(X[0,0], -1, np.int64(3)))))
  description: Tasks (1,1) and (3,1) overlap on machine 2
((Y[1,0,1]) and (Y[2,1,1])) -> (((sum(31, np.int64(2), X[1,0])) < (sum(1, 31, X[2,1]))) or (((X[2,1]) +
↪  3) <= (X[1,0])))
  description: Tasks (2,1) and (3,2) overlap on machine 2
((Y[1,0,0]) and (Y[0,1,0])) -> ((((X[0,1]) + 39) >= (sum(np.int64(2), 39, X[1,0]))) or ((X[1,0]) >=
↪  ((X[0,1]) + 2)))
  description: Tasks (1,2) and (2,1) overlap on machine 1
((Y[1,1,0]) and (Y[1,0,0])) -> ((((X[1,1]) + 4) <= (X[1,0])) or ((X[1,1]) > (sum(-1, np.int64(2),
↪  X[1,0]))))
  description: Tasks (2,1) and (2,2) overlap on machine 1
((Y[1,1,0]) and (Y[0,0,0])) -> ((((X[0,0]) + -34) > (sum(-34, -1, np.int64(4), X[1,1]))) or ((sum(33,
↪  X[0,0], np.int64(3))) < (sum(X[1,1], 33, 1))))
  description: Tasks (1,1) and (2,2) overlap on machine 1
((Y[0,0,1]) and (Y[1,0,1])) -> (((X[1,0]) >= (3 + (X[0,0]))) or ((X[0,0]) >= (2 + (X[1,0]))))
  description: Tasks (1,1) and (2,1) overlap on machine 2
((Y[2,1,1]) and (Y[1,1,1])) -> (((sum(X[2,1], 77, np.int64(3))) <= (77 + (X[1,1]))) or (((X[1,1]) + 4) <
↪  ((X[2,1]) + 1)))
  description: Tasks (2,2) and (3,2) overlap on machine 2
((Y[0,0,1]) and (Y[1,1,1])) -> (((sum(np.int64(4), 8, X[1,1])) <= (8 + (X[0,0]))) or ((sum(-51, X[0,0],
↪  np.int64(3))) < (sum(-51, 1, X[1,1]))))
  description: Tasks (1,1) and (2,2) overlap on machine 2
((Y[0,1,1]) and (Y[2,0,1])) -> (((X[0,1]) >= (1 + (X[2,0]))) or ((X[2,0]) >= ((X[0,1]) + 2)))
  description: Tasks (1,2) and (3,1) overlap on machine 2
((Y[1,0,0]) and (Y[2,1,0])) -> (((sum(29, np.int64(2), X[1,0])) <= (29 + (X[2,1]))) or ((44 + (X[1,0])) >
↪  (sum(-1, X[2,1], 44, np.int64(3)))))
  description: Tasks (2,1) and (3,2) overlap on machine 1
((Y[2,1,0]) and (Y[0,0,0])) -> ((((X[0,0]) + 3) <= (X[2,1])) or (((X[2,1]) + 3) < ((X[0,0]) + 1)))
  description: Tasks (1,1) and (3,2) overlap on machine 1
((Y[0,1,0]) and (Y[2,0,0])) -> ((((X[0,1]) + 2) < (1 + (X[2,0]))) or ((85 + (X[0,1])) > (sum(85, X[2,0],
↪  np.int64(1), -1))))
  description: Tasks (1,2) and (3,1) overlap on machine 1
((Y[1,1,1]) and (Y[1,0,1])) -> (((-31 + (X[1,0])) > (sum(np.int64(4), -1, X[1,1], -31))) or
↪  ((sum(np.int64(2), -5, X[1,0])) < (sum(X[1,1], 1, -5))))
  description: Tasks (2,1) and (2,2) overlap on machine 2
((Y[1,0,0]) and (Y[0,0,0])) -> (((7 + (X[0,0])) >= (sum(7, np.int64(2), X[1,0]))) or ((sum(85,
↪  np.int64(3), X[0,0])) < (sum(X[1,0], 85, 1))))
  description: Tasks (1,1) and (2,1) overlap on machine 1
```

```
((Y[0,1,1]) and (Y[2,1,1])) -> (((sum(-44, np.int64(3), X[2,1])) <= ((X[0,1]) + -44)) or ((18 + (X[2,1])) > (sum(18, X[0,1], np.int64(2), -1))))
  description: Tasks (1,2) and (3,2) overlap on machine 2
((Y[0,0,1]) and (Y[2,1,1])) -> ((((X[0,0]) + 94) >= (sum(np.int64(3), 94, X[2,1]))) or (((X[0,0]) + 3) < ((X[2,1]) + 1)))
  description: Tasks (1,1) and (3,2) overlap on machine 2
((Y[2,0,1]) and (Y[2,1,1])) -> (((sum(85, X[2,1], np.int64(3))) <= (85 + (X[2,0]))) or ((40 + (X[2,1])) > (sum(40, X[2,0], -1, np.int64(1)))))
  description: Tasks (3,1) and (3,2) overlap on machine 2
((Y[0,1,0]) and (Y[1,1,0])) -> (((X[0,1]) >= (4 + (X[1,1]))) or ((sum(-66, X[0,1], np.int64(2))) <= ((X[1,1]) + -66)))
  description: Tasks (1,2) and (2,2) overlap on machine 1
((Y[2,0,1]) and (Y[1,0,1])) -> (((X[1,0]) > (sum(X[2,0], -1, np.int64(1)))) or ((71 + (X[2,0])) > (sum(np.int64(2), X[1,0], 71, -1))))
  description: Tasks (2,1) and (3,1) overlap on machine 2
((Y[0,0,0]) and (Y[2,0,0])) -> (((X[0,0]) > (sum(np.int64(1), -1, X[2,0]))) or ((3 + (X[0,0])) < (1 + (X[2,0]))))
  description: Tasks (1,1) and (3,1) overlap on machine 1
((Y[2,0,1]) and (Y[1,1,1])) -> ((((X[2,0]) + 23) > (sum(X[1,1], 23, -1, np.int64(4)))) or ((67 + (X[1,1])) >= (sum(67, X[2,0], np.int64(1)))))
  description: Tasks (2,2) and (3,1) overlap on machine 2
((Y[2,0,0]) and (Y[1,0,0])) -> (((sum(-87, X[1,0], np.int64(2))) < (sum(-87, 1, X[2,0]))) or (((X[1,0]) + -56) > (sum(-1, X[2,0], -56, np.int64(1)))))
  description: Tasks (2,1) and (3,1) overlap on machine 1
((Y[2,0,0]) and (Y[1,1,0])) -> ((((X[2,0]) + 10) >= (sum(X[1,1], 10, np.int64(4)))) or ((sum(46, np.int64(1), X[2,0])) <= (46 + (X[1,1]))))
  description: Tasks (2,2) and (3,1) overlap on machine 1
((Y[0,1,0]) and (Y[2,1,0])) -> ((((X[2,1]) + -92) > (sum(-1, np.int64(2), -92, X[0,1]))) or ((17 + (X[0,1])) > (sum(17, X[2,1], -1, np.int64(3)))))
  description: Tasks (1,2) and (3,2) overlap on machine 1
((Y[1,1,0]) and (Y[2,1,0])) -> (((sum(X[1,1], 33, np.int64(4))) < (sum(X[2,1], 1, 33))) or (((X[2,1]) + 3) <= (X[1,1])))
  description: Tasks (2,2) and (3,2) overlap on machine 1

jective: min sum(X[0,1], np.int64(2), (X[1,1]) + 4, (X[2,1]) + 3)
seline: 19 (optimal: True)
```

6 Obfuscated hidden model for the toy FJSS instance.