# Modeling and Solving the Multi-Agent Pathfinding Problem in Picat

Roman Barták
Charles University
Praha, Czech Republic
bartak@ktiml.mff.cuni.cz

Neng-Fa Zhou
CUNY Brooklyn College
and Graduate Center
New York, USA
zhou@sci.brooklyn.cuny.edu

Roni Stern and Eli Boyarski
Ben Gurion University of the Negev
Beer Sheva, Israel
{roni.stern,eli.boyarski}@gmail.com

Pavel Surynek
AIST, Tokyo, Japan
Charles University
Praha, Czech Republic
pavel.surynek@aist.go.jp

***Abstract*—The multi-agent pathfinding (MAPF) problem has attracted considerable attention because of its relation to practical applications. In this paper, we present a constraint-based declarative model for MAPF, together with its implementation in Picat, a logic-based programming language. We show experimentally that our Picat-based implementation is highly competitive and sometimes outperforms previous approaches. Importantly, the proposed Picat implementation is very versatile. We demonstrate this by showing how it can be easily adapted to optimize different MAPF objectives, such as minimizing makespan or minimizing the sum of costs, and for a range of MAPF variants. Moreover, a Picat-based model can be automatically compiled to several general-purpose solvers such as SAT solvers and Mixed Integer Programming solvers (MIP). This is particularly important for MAPF because some MAPF variants are solved more efficiently when compiled to SAT while other variants are solved more efficiently when compiled to MIP. We analyze these differences and the impact of different declarative models and encodings on empirical performance.**

## I. Introduction

The multi-agent pathfinding (MAPF) problem amounts to finding a plan for agents to move within a graph from their starting locations to their destinations, such that no agents collide with each other at any time. MAPF can be solved suboptimally in polynomial time [14], but finding an optimal solution is NP-hard for common optimization criteria [20], [27]. MAPF has been intensively studied because the problem occurs in various forms in practical applications, such as robotics and games [7], [17], and the problem also provides a platform for studying search algorithms [16], [19], [26].

Recently, MAPF solvers were proposed using declarative models, relying on off-the-shelf solvers to find solutions. These solvers include CSP (Constraint Satisfaction Problems) [15], SAT (Boolean Satisfiability) [23], [24], ASP (Answer Set Programming) [8], and MIP (Mixed Integer Programming) [28]. Declarative models are easy to implement and maintain, can easily be altered for other variants, and are amenable to new domain-specific constraints. SAT-based MAPF solutions are especially promising; they have been shown to be competitive with some well-designed heuristic search algorithms [24].

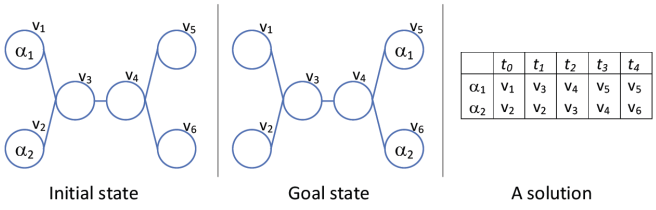
All of the declarative models follow the planning-as-satisfiability approach [11], which finds a bounded-length sequence of states, where the first state corresponds to the initial state, the last state satisfies the goal condition, and each pair of successive states constitutes a valid action. An efficient declarative solution requires a good model of variables and constraints, a fast solver, and a decent model encoding.

In this paper, we give a constraint-based declarative model for MAPF that is very natural, versatile, and efficient. It follows prior SAT-based modeling of the problem [21], [22], using a Boolean variable to indicate whether an agent occupies a vertex of the graph in a state, and constraints to ensure the validity of all states and state transitions. Our declarative model is implemented in Picat, a general-purpose language that provides several tools for modeling and solving combinatorial problems [31]. Implementing the problem in Picat and defining it in a declarative way provides great flexibility. Important variants of the MAPF problem can be implemented by performing minimal changes in our Picat program. In particular, we show how to implement a MAPF solver that optimizes different objective functions, or a MAPF solver that generates plans that are robust to unexpected delays, or even a MAPF solver that supports agents with different priorities.

Modeling MAPF in Picat also enables easy comparison of different underlying solvers and encodings. Indeed, we performed a set of experiments with our Picat models that compare several SAT-based and MIP-based solutions. In the regular MAPF setting, the SAT-based solvers were more competitive than MIP. However, when including agents with different priorities, the MIP solver performs better, highlighting the benefit of our Picat program, in which changing to a different solver is extremely easy.

In addition, we compare our declarative model empirically to a several state-of-the-art MAPF solvers, including a MAPF solver based on Answer Set Programming (ASP) [8], a sophisticated SAT encoding called MDD-SAT [24], and the best MAPF solver from the Conflict-Based Search family of MAPF solvers [4]. The results show that our Picat-based solution is significantly better than the ASP solution and comparable to MDD-SAT for the standard makespan MAPF variant.

The contributions of the paper include the following: (1) a high-level declarative solution in Picat for the MAPF problem that is easily accessible and more efficient than other declarative solutions; (2) adaptations of this declarative solution to accommodate important MAPF variants with different constraints and different objectives, including an encoding of




| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ |
|---|---|---|---|---|---|
| $\alpha_1$ | $v_1$ | $v_3$ | $v_4$ | $v_5$ | $v_5$ |
| $\alpha_2$ | $v_2$ | $v_2$ | $v_3$ | $v_4$ | $v_6$ |

Fig. 1. An instance of the MAPF problem and a solution.

different agent priorities, which have never been addressed before; and (3) a comprehensive empirical comparison with different declarative and non-declarative solutions, encodings, and underlying solvers.

## II. The MAPF Problem

The input for the MAPF Problem is a graph $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges, and a set of agents $A = \{a_1, a_2, \ldots, a_k\}$, where each agent $a_i$ has a starting vertex $start(a_i) \in V$ and a goal vertex $goal(a_i) \in V$. At each time step, an agent can stay at its current vertex or move to an adjacent vertex. To represent the option of an agent waiting, we assume that the relation represented by the graph $G$ is reflexive, meaning that $(v, v) \in E$, for $v \in V$. The MAPF problem amounts to finding a path $P_i = <v_{i0}, v_{i1}, \ldots, v_{im}>$ for each agent $a_i (i = 1, 2, \ldots, k)$, such that $v_{i0} = start(a_i)$, $v_{im} = goal(a_i)$, $(v_{it}, v_{i(t+1)}) \in E$ (for $t = 0, 1, \ldots, m-1$), and no two agents collide at any time: $v_{it} \neq v_{jt}$, for $i, j = 1, 2, \ldots, k$, $i \neq j$, and $t = 0, 1, \ldots, m$. The set of satisfying paths for the agents constitutes a *plan* for the MAPF problem.

*Definition 1 (End time):* The *end time* of a path $P_i = <v_{i0}, v_{i1}, \ldots, v_{im}>$ for an agent $a_i$ is the time point $e$ that satisfies the following conditions: (1) $v_{i(e-1)} \neq v_{ie}$; (2) $v_{it} = goal(a_i)$ for $t = e, e+1, \ldots, m$.
In other words, the end time is the time at which the agent reaches its goal, and will stay at the goal afterwards.

*Definition 2 (Makespan):* The *makespan* of a plan is the maximum end time of the paths in the plan.

One of the most common MAPF objective functions is to find a plan that has the minimum makespan. The basic declarative model we present in this paper is designed to optimize this objective function, and we show later how to adapt it to other objective functions.

Figure 1 gives a sample instance of the MAPF problem. In the initial state, agent $a_1$ occupies vertex $v_1$, and agent $a_2$ occupies vertex $v_2$. The goal of the problem is to move agent $a_1$ to vertex $v_5$ and agent $a_2$ to vertex $v_6$. Figure 1 also shows a solution plan for the problem. At time step $t_0 \rightarrow t_1$, agent $a_1$ moves from vertex $v_1$ to vertex $v_3$, and agent $a_2$ waits at vertex $v_2$. At the next step $t_1 \rightarrow t_2$, agent $a_1$ moves from vertex $v_3$ to vertex $v_4$, and agent $a_2$ moves from vertex $v_2$ to vertex $v_3$. In the solution plan, the end time of agent $a_1$ is 3, the end time of agent $a_2$ is 4, and so the makespan is 4.

## III. The Picat Language

Picat is a logic-based multi-paradigm language that integrates logic programming, functional programming, constraint programming, and scripting. Picat takes many features from other languages, including logic variables, unification, backtracking, pattern-matching rules, functions, list and array comprehensions, loops, assignments, tabling for dynamic programming and planning, and constraint solving with CP, SAT, and MIP. The reader is referred to [31] for the details of the language and the constraint modules.

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: the *non-backtrackable* rule: $Head, Cond$ => $Body$ and the *backtrackable* rule: $Head, Cond$ ?=> $Body$. In a predicate definition, the $Head$ takes the form $p(t_1, \ldots, t_n)$, where $p$ is a predicate name, and $n$ is the arity. The condition $Cond$, which is an optional goal, specifies a condition under which the rule is applicable. For a call $C$, if $C$ matches $Head$ and $Cond$ succeeds, then the rule is said to be *applicable* to $C$. When applying a rule to call $C$, Picat rewrites $C$ into $Body$. If the used rule is backtrackable, then the program will backtrack to $C$ if $Body$ fails. A *function* is a special kind of a predicate that is defined by non-backtrackable rules. In a function definition, the $Head$ takes the form $f(t_1, \ldots, t_n) = Term$, where $f$ is a function name and $Term$ is a result to be returned. If $Cond$ and $Body$ are both true, then they can be omitted together with the => arrow.

Picat supports *tabling*, which caches previously calculated solutions, and reuses them in subsequent computations. In Picat, both predicates and functions can be tabled. In order to have all calls and answers of a predicate or function tabled, users just need to add the keyword `table` before the first rule. For a predicate definition, the keyword `table` can be followed by a tuple of table modes, including `+` (input), `-` (output), `min`, `max`, and `nt` (not tabled). For a predicate with a table mode declaration that contains `min` or `max`, Picat tables one optimal answer for each tuple of the input arguments. For example, the following predicate `shortest_dist` computes the shortest-path cost of a given pair of vertices in a unit-cost graph that is represented by the predicate `edge/2`:

```
table (+,min)
shortest_dist((V,V),Cost) =>
    Cost = 0.
shortest_dist((V,FV),Cost) =>
    edge(V,NextV),
    shortest_dist((NextV,FV),Cost1),
    Cost = Cost1+1.
```

We use this tabled predicate later in this paper.[1]

Picat provides three solver modules, `cp`, `sat`, and `mip`, for modeling and solving CSPs. As a constraint programming language, Picat resembles CLP(FD) : the operator `::` is used

[1]When applied to finding single-source shortest-path costs, this tabled predicate implements Dijkstra's algorithm, except that it tables shortest-path costs from the encountered vertices to the destination vertex.

for domain constraints, the operators `#=`, `#!=`, `#>`, `#>=`, `#<`, `#<=`, and `#=<` are used for arithmetic constraints, and the operators `#/\` (and), `#\/` (or), `#^` (xor), `#~` (not), `#=>` (if), and `#<=>` (iff) are used for Boolean constraints. Picat supports several global constraints, such as `all_different/1`, `element/3`, and `cumulative/4`. In addition to intensional constraints, Picat also supports expressing extensional, or table, constraints. The common interface that Picat provides for the solver modules allows seamless switching from one solver to another, and the basic language constructs, such as arrays, loops, and list comprehensions, make Picat a convenient modeling language for CSPs.

For the `sat` module, the Picat SAT compiler [29], [30] translates constraints into a logic formula in the conjunctive normal form (CNF) for the underlying SAT solver. Picat employs the *log-encoding* for compiling domain variables and constraints. For a domain with the maximum absolute value $n$, $log_2 n$ Boolean variables are used. If the domain contains both negative and positive values, then another Boolean variable is used to encode the sign. Each combination of values of these Boolean variables represents a valuation for the domain variable. PicatSAT flattens constraints into primitive ones, and performs numerous optimizations, both in the phase of breaking constraints into primitive ones, and in the phase of compiling primitive constraints into adders and comparators [30]. For optimization problems, Picat uses branch-and-bound to optimize the objective. It first posts the problem as a constraint satisfaction problem, ignoring the objective. Once a solution is found, Picat uses binary search to find a solution with the optimum value.

For the `mip` module, constraints are compiled into inequality ($\leq$) constraints. The compilation follows the standard textbook recipe [2]. For example, the reification constraint `B #<=> (X #=< Y)` is translated into `X-Y-M1*(1-B) #=< 0` and `Y-X+1-M2*B #=< 0`, where `M1` and `M2` are constants:

```
M1 = ubd(X)-lbd(Y)+1
M2 = ubd(Y)-lbd(X)+2
```

where `ubd(X)` is the upper bound of the domain of `X`, and `lbd(X)` is the lower bound. The compiler avoids introducing big constants when linearizing constraints if possible. For example, the constraint `B #=> (sum(L) #>= C)`, where `L` is a list of non-negative integer-domain variables and `C` is a positive constant, is translated into `sum(L) #>= C*B`.

## IV. A MAPF Model and its Implementation in Picat

This section gives a CSP model for MAPF, together with its implementation in Picat. The first presented model is designed to optimizie makespan. Then, we propose a technique for processing the state variables before running the model, so as to help the underlying solver find solutions faster. Then, an extended program is presented for optimizing the sum-of-costs for a given makespan.

### A. The MAPF Model

We can model the MAPF problem as a CSP, following the planning-as-satisfiability framework [11]. We use a Boolean variable $B_{tav}$ to indicate if agent $a$ ($a = 1, 2, \ldots, k$) occupies vertex $v$ ($v = 1, 2, \ldots, n$) at time $t$ ($t = 0, 1, \ldots, m$). The following constraints ensure the validity of every state and every transition:

(1) Each agent occupies exactly one vertex at each time.
$\Sigma_{v=1}^{n} B_{tav} = 1$ for $t = 0, \ldots, m$, and $a = 1, \ldots, k$.

(2) No two agents occupy the same vertex at any time.
$\Sigma_{a=1}^{k} B_{tav} \leq 1$ for $t = 0, \ldots, m$, and $v = 1, \ldots, n$.

(3) If agent $a$ occupies vertex $v$ at time $t$, then $a$ occupies a neighboring vertex at time $t + 1$. [2]
$B_{tav} = 1 \Rightarrow \Sigma_{u \in neibs(v)}(B_{(t+1)au}) \geq 1$
for $t = 0, \ldots, m-1$, $a = 1, \ldots, k$, and $v = 1, \ldots, n$.

Note that constraints (1) and (3) entail that

$$B_{tav} = 1 \Rightarrow \Sigma_{u \in neibs(v)}(B_{(t+1)au}) = 1,$$

for $t = 0, \ldots, m-1$, $a = 1, \ldots, k$, and $v = 1, \ldots, n$. It is more efficient to use inequality ($\geq$) in constraint (3) than equality, because the inequality constraint can be compiled into one CNF clause.

The model consists of $k \times (m+1) \times n$ Boolean variables, where $k$ is the number of agents, $m$ is the makespan, and $n$ is the number of vertices in the graph. The *exactly-one* constraint $\Sigma_{v=1}^{n} B_{tav} = 1$ in (1) is encoded as a conjunction of the *at-least-one* constraint $\Sigma_{v=1}^{n} B_{tav} \geq 1$ and the *at-most-one* (AMO) constraint $\Sigma_{v=1}^{n} B_{tav} \leq 1$. The at-least-one constraint is encoded as one clause. The number of clauses generated from the constraints in the model depends on how the AMO constraint is encoded. The PicatSAT compiler employs the *2-product algorithm* [5], which encodes the summation $\Sigma_{i=1}^{l} B_i$ in the AMO constraint as the Cartesian product of two subsets of Boolean variables. Based on this encoding, the model requires $O(m \times k \times \sqrt{n} + m \times n \times \sqrt{k})$ auxiliary variables and $O(m \times k \times n)$ clauses to encode.

### B. The Implementation in Picat

Figure 2 shows an implementation of the CSP model in Picat. The program uses the `sat` module. We can switch to `mip` or `cp` by changing the import declaration.

For a given MAPF problem, the graph is represented by the predicate `neibs(V,Neibs)`, which binds `Neibs` to the list of the neighboring vertices of a given vertex `V`. Let `N` be the number of vertices in the graph, and `As` be a list of agents, where each agent is a pair that indicates the starting and ending vertices of the agent. The predicate `path(N,As)` finds a path for each of the agents.

The function call `len(As)` returns the number of agents (the length of the list). The predicate

```
lower_upper_bounds(As,LB,UB)
```

computes the lower and upper bounds on an optimal makespan. The lower bound, `LB`, is the maximum of the shortest-path costs, which is the number of steps required to

[2] Recall that we assume the graph is reflexive.

```
import sat.

path(N,As) =>
    K = len(As),
    lower_upper_bounds(As,LB,UB),
    between(LB,UB,M),
    B = new_array(M+1,K,N),
    B :: 0..1,

    % Initialize the first and last states
    foreach (A in 1..K)
        (V,FV) = As[A],
        B[1,A,V] = 1,
        B[M+1,A,FV] = 1
    end,

    % Each agent occupies exactly one vertex
    foreach (T in 1..M+1, A in 1..K)
        sum([B[T,A,V] : V in 1..N]) #= 1
    end,

    % No two agents occupy the same vertex
    foreach (T in 1..M+1, V in 1..N)
        sum([B[T,A,V] : A in 1..K]) #=< 1
    end,

    % Every transition is valid
    foreach (T in 1..M, A in 1..K, V in 1..N)
        neibs(V,Neibs),
        B[T,A,V] #=>
        sum([B[T+1,A,U] : U in Neibs]) #>= 1
    end,

    solve(B),
    output_plan(B).
```

Fig. 2. A program in Picat for MAPF.

move the most difficult agent from its initial vertex to its destination vertex. The upper bound, `UB`, is the sum of the shortest-path costs. If only one agent is allowed to move at each step, then `UB` is the number of steps required to move all of the agents to their destinations.[3]

The predicate call `between(LB,UB,M)` generates a number `M` between `LB` and `UB`; it generates the next number on backtracking. If a plan is found for a value of `M`, then the plan is guaranteed to be optimal with the makespan `M`. Otherwise, if no plan is found for the value, then the program backtracks to `between` to generate the next number. This *generate-and-test* step is repeated until a plan is found for some value `M`, or until no plan is found after all of the numbers in the range `LB..UB` have been tried. Note, that `M` determines the size of the model and hence it is better to start with smaller `M` (makespan).

The function call `new_array(M+1,K,N)` creates a three dimensional array, where the first dimension indicates the time points from 1 through `M+1`, the second dimension refers to the agents `1..K`, and the third dimension represents the vertices `1..N`. The call `B :: 0..1` changes the entries of the array `B` to Boolean variables, which are 0/1 integer domain variables.

There are four `foreach` loops in the encoding. The first `foreach` loop initializes the first and last states. For an agent `A` whose initial vertex is `V` and whose goal vertex is `FV`, the entries `B[1,A,V]` and `B[M+1,A,FV]` are set to 1. Note that, since array indices in Picat are 1-based, the initial state has index 1, and the goal state has index `M+1`. The remaining three `foreach` loops encode the constraints (1), (2), and (3) in the model. The call `solve(B)` performs three operations: (1) it compiles all the accumulated constraints into CNF; (2) it calls the SAT solver to solve the CNF formula; (3) it retrieves the solution from the SAT solver and produces bindings for the variables in B. If a different solver module is imported, then constraints are translated into different encodings.

[3]Note that it is assumed that there is a path connecting each agent's starting vertex to its destination vertex; otherwise, the predicate `lower_upper_bounds` fails.

### C. Preprocessing

Consider an agent $a$ whose starting vertex is $start(a)$ and whose goal vertex is $goal(a)$. For a vertex $v$, let $d(s, v)$ be the shortest distance from $start(a)$ to $v$, and $d(v, g)$ be the shortest distance from $v$ to $goal(a)$. Agent $a$ cannot occupy vertex $v$ at times $t = 0, 1, \ldots, d(s, v) - 1$:

$$B_{tav} = 0 \text{ for } t = 0, 1, \ldots, d(s, v) - 1.$$

Similarly, agent $a$ cannot occupy vertex $v$ at times $t = m - d(v, g) + 1, \ldots, m$:

$$B_{tav} = 0 \text{ for } t = m - d(v, g) + 1, \ldots, m.$$

Setting these $B_{tav}$ values is done in a pre-processing phase, i.e., before giving the model to the underlying solver. Note that this preprocessing phase requires finding shortest-path costs and setting the variables, which takes $O(k \times n^2 + k \times n \times m)$ overall running time, where $k$ is the number of agents, $m$ is the makespan, and $n$ is the number of vertices. For large graphs, this complexity is prohibitive. For grid maps, Manhattan distances can be substituted as estimates of shortest-path costs.

## V. Modeling MAPF Variants

The MAPF problem has many variants that correspond to the broad range of MAPF applications. One of the main strengths of using constraint programming in general and Picat in particular is the ease with which one can support different variants of the problem. In this section, we demonstrate how important variants of MAPF can be encoded using our Picat modeling by making only minimal changes to the encoding.

### A. The Sum-of-Costs Objective

The declarative model described above is designed to optimize for makespan. However, different objective functions have been studied for MAPF, such as finding a plan with minimum sum of costs.

*Definition 3 (Sum of costs):* The *sum-of-costs* (SOC) of a plan is the sum of the end times of the paths in the plan.
In the MAPF example depicted in Figure 1, the SOC is 7. Note that makespan and SOC cannot generally be simultaneously optimized, i.e., there are cases where the makespan-optimal plan is different from the SOC-optimal plan.

We can easily extend the core model to make it find a plan with the minimum sum-of-costs instead of minimum makespan. To this end, we introduce a variable $E_a$ for each agent $a$ ($a = 1, 2, ..., k$) that represents the end time of the agent. The domain of $E_a$ has the cost of the shortest path from $a$'s initial vertex to its goal vertex as its lower bound, and the maximum makespan as its upper bound.

```
end_time(B,As,K,M,E) =>
    E = new_array(K),
    E :: 1..M+1,
    foreach (A in 1..K, T in 1..M+1)
        (V,FV) = As[A],
        if T > 1 then
            E[A] #= T #=> B[T-1,A,FV] #= 0
        end,
        foreach (T1 in T..M+1)
            E[A] #= T #=> B[T1,A,FV] #= 1
        end
    end.
```

Fig. 3. The Picat code for constraints (4) and (5) in the SOC model.

Let $goal(a)$ be $g$, $a$'s goal vertex. The following constraints enforce that the agent reaches its goal at time $E_a$ and stays at the goal afterwards.

(4) Agent $a$ reaches its goal vertex $g$ at time $E_a$, i.e., agent $a$ does not occupy vertex $g$ at time $E_a - 1$:
For $t = 2, 3, \ldots, m$, $E_a = t \rightarrow B_{(t-1)ag} = 0$.

(5) Agent $a$ occupies vertex $g$ from time $E_t$ through $m$.
For $t = E_a, E_a + 1, \ldots, m$, $B_{tag} = 1$.

The objective is to minimize the sum-of-costs: `min(`$\Sigma_{a=1}^{k} E_a$`)`. The predicate given in Figure 3 encodes constraints (4) and (5).

The constraint `E[A] #= T #=> B[T-1,A,FV] #= 0` states that if agent `A`'s end time is `T`, then agent `A` does not occupy the goal vertex `FV` at time `T-1`. The inner `foreach` loop enforces that the agent `A` occupies the goal vertex `FV` from time `T` through `M+1`.

A straightforward approach to modify the program in Figure 2 to make it minimize the overall sum-of-costs is changing the nondeterministic call `between(LB,UB,M)` to `M = UB`, adding the call `end_time(B,As,K,M,E)` before the call to `solve`, and changing `solve(B)` to `solve([$min(sum(E))],B)`,[4] which passes the objective `min(sum(E))` to the solver. In this way, the plan found will have the minimum SOC for all possible makespans. This naive approach requires many variables in the model if the upper bound for makespan is large. A more advanced approach was proposed in [24] that is based on idea of increasing the makespan together with increasing the limit for the cost function. This reduces the size of the model (the number of variables), but increases the number of calls to the solver (the solver is invoked for every makespan tried).

One can also devise a hybrid objective function: minimizing makespan as a primary objective and minimizing SOC as a secondary objective, i.e., among plans with the same makespan prefer the plan that has the smallest SOC. In order to modify the program in Figure 2 to perform this two-stage optimization, we just need to make the same changes as above for optimizing overall SOC, but leaving `between(LB,UB,M)` as it is.

### B. Priorities

The sum of costs objective assumes that the cost of moving each agent is equal. In many scenarios, however, moving different agents may incur different costs. For example, consider a MAPF problem where all agents represent taxis moving people in a city except for one agent that represents an ambulance driving to the hospital. Clearly, it is more important to minimize the time it takes the ambulance to reach the hospital than to minimize the time spent by the other taxi agents. We can represent such scenarios in a general way by assuming a given vector of weights $\mathbf{w} = (w_1, \ldots, w_k)$, where $w_i$ is the cost of having agent $a_i$ spend one time step, and the corresponding objective function is the *weighted sum-of-costs*.

Adjusting the sum of costs model to support priorities in such a ways is trivial. We define a variable $WE = \mathbf{w} \cdot E$, which holds the weighted cost of each agent, and pass to the solver the objective `min(sum(WE))` instead of `min(sum(E))`.

[4]The preceding dollar symbol indicates that `min(sum(E))` is a term, not a function call.

```
foreach(T in 1..M1, A in 1..K, V in 1..N)
    B[T,A,V] #=> sum([B[Prev,A2,V] :
                A2 in 1..K, A2!=A,
                Prev in max(1,T-L)..T]) #= 0
end
```

Fig. 4. The constraint needed to support finding a $L$-robust solution

### C. Robust Plans

Since agents may get delayed unexpectedly, it is sometimes desirable to create solutions to MAPF problems that are robust to such changes. One form of such robust planning that was suggested in the context of MAPF is *$L$-robustness* [1]. A $L$-robust solution to a MAPF problem is a problem that minimizes some objective function (e.g., sum of costs or makespan) while ensuring that the solution can be executed as long as there is no agent that will experience more than $L$ unexpected delays.

Adjusting the Picat model to return $L$-robust solutions is also very simple. It involves modifying a constraint that forbids two agents from occupying the same vertex at the same time, such that the constraint forbids every two agents from occupying the same vertex in times that are no more than $L$ time steps from each other. Fig. 4 shows the exact Picat code for this constraint. We implemented this variant and observed that in some settings it is comparable to a CBS-based $k$-robust solver [1].

### D. Edge Conflicts and Train-like Motion

Some variants of the problem impose additional constraints on valid paths. One constraint requires that when an agent moves from vertex $u$ to $v$ ($u \neq v$) at time $t$, $v$ cannot be occupied by another agent at time $t$. Without this constraint agents can move in a train-like formation, while having this constraint bans such a motion and bans agents from making cyclic rotations in an empty-space-free area [27]. Encoding this constraint on top of the Picat model is very easy. In fact, it is equivalent to searching for a 1-robust solution.

Another constraint that have been used in prior works is to mandate that no two agents can cross each other on an edge at any time step. Under this constraint, when an agent moves from vertex $u$ to vertex $v$ at time step $t \rightarrow (t+1)$, no agent can move from vertex $v$ to vertex $u$ at the same time step. This is known as the *edge conflict* constraint. Adjusting the

model to avoid edge conflicts can be done by adding an explicit constraint that checks for every two agents, every location, and every pair of consecutive time steps that these agents did not swap locations. We implemented this extension and observed that it causes substantial increase in runtime, as this constraint translates to many constraints to the underlying solver. An alternative implementation can be to maintain a variable for each edge, time step, and agent, to keep track of which edges are being used by the agents.

Note that we do not claim that all the above is the best encoding for these variants. More sophisticated encodings might yield shorter runtimes. However, the above encodings are sound and easily implemented in our declarative model.

## VI. Experimental Results

This section gives experimental results comparing a range of solvers, encodings, and MAPF variants. Following prior work, we generated random problem instances by randomly adding obstacles to an NxN grid and set random start and goal locations to the agents in that grid [23], [19]. Of the set of problems we have generated, we chose 20 representative instances under different settings. Each instance name takes the form `gN_pP_aK`, where `N×N` is the grid size, `P` is the percentage of obstacles, or blocked squares, in the grid, and `K` is the number of agents. For example, `g16_p10_a40` represents an instance with 40 agents on a 16x16 grid where 10% of its cells are obstacles. Unless stated otherwise, all experiments were run on Cygwin notebook computer with 2.60GHz Intel i7 and 64GB RAM, and we measured CPU time in seconds. A time limit was set to 600 seconds.

Table I compares the performances of finding makespan-optimal plans with our declarative model using different underlying solvers, namely SAT and MIP. For SAT, the generated CNF code was solved with Lingeling (version 587f) [3]. For MIP, the generated code was solved with Gurobi version 6.5.1 [9]. We evaluated the solvers' performance with (columns $\text{SAT}_p$ and $\text{MIP}_p$) and without (columns $\text{SAT}_{nop}$ and $\text{MIP}_{nop}$) the preprocessing described in Section IV-C.

A CP model and a planner model, both available in Picat version 2.1, were also evaluated. The CP model uses a domain variable for each agent and each time, which indicates the vertex that the agent occupies at the time. The planner model searches the state space using tabled backtracking search. The results of the CP and planner models were very poor on large instances so we do not present them in this section.

The results show two clear trends. First, as expected, the preprocessing described in Section IV-C is significantly useful for both SAT and MIP solvers. Second, the SAT solver (with preprocessing) is always almost the same as or faster than the MIP solver for our instances. Further testing showed that the SAT program successfully solved all of the instances with 100 agents generated for the grid size of $32 \times 32$ within the time limit. Note that the MIP solver we used – Gurobi – is commercial and not open source. Thus, it is difficult to analyze why our SAT solver was faster than our MIP solver.

TABLE I
A comparison of solvers' performance for minimizing makespan (CPU time, seconds)

| Instance (M) | $\text{SAT}_{nop}$ | $\text{SAT}_p$ | $\text{MIP}_{nop}$ | $\text{MIP}_p$ |
|---|---|---|---|---|
| g16_p10_a05 (14) | 0.59 | **0.27** | 1.51 | 0.43 |
| g16_p10_a10 (20) | 2.29 | 1.37 | 4.96 | **0.88** |
| g16_p10_a20 (23) | 5.73 | 2.76 | 13.14 | **2.54** |
| g16_p10_a30 (23) | 8.89 | **3.11** | 19.88 | 3.50 |
| g16_p10_a40 (30) | 24.40 | **8.25** | 119.07 | 72.93 |
| g16_p20_a05 (24) | 1.70 | 1.01 | 3.23 | **0.83** |
| g16_p20_a10 (24) | 2.29 | **1.50** | 5.86 | 1.81 |
| g16_p20_a20 (23) | 4.56 | **2.12** | 11.26 | 2.50 |
| g16_p20_a30 (28) | 11.63 | **4.37** | 38.31 | 7.06 |
| g16_p20_a40 (22) | 12.19 | **3.48** | 31.35 | 10.24 |
| g32_p10_a05 (37) | 7.45 | **1.98** | 24.86 | 2.14 |
| g32_p10_a10 (34) | 13.63 | **3.08** | 49.74 | 4.15 |
| g32_p10_a20 (42) | 40.67 | **8.71** | 219.65 | 14.05 |
| g32_p10_a30 (55) | 82.02 | **34.48** | 380.28 | 167.94 |
| g32_p10_a40 (47) | 79.49 | **34.95** | >600 | 150.59 |
| g32_p20_a05 (53) | 11.10 | **5.75** | 54.97 | 9.86 |
| g32_p20_a10 (36) | 14.03 | **2.97** | 50.56 | 4.99 |
| g32_p20_a20 (49) | 42.80 | **16.93** | 266.25 | 24.69 |
| g32_p20_a30 (43) | 53.30 | **12.98** | 248.05 | 43.38 |
| g32_p20_a40 (42) | 86.66 | **16.51** | 351.55 | 30.89 |
| Total solved | **20** | **20** | 19 | **20** |

For both SAT and MIP, the running time increases with the increase of the number of agents, but the increase rate for SAT is usually much smaller than that for MIP. The impact of increasing the percentage of obstacles increases from 10% to 20% is less clear, but in general having more obstacles reduces the runtime. This is because having more obstacles reduces the graph size, and ultimately leads to the decrease of the number of variables in the models.

Next, we evaluated the performance of some of the variants of our declarative model described in Section V. Table II gives the CPU time taken to find a plan for three different objective functions: (1) minimum SOC among those with the shortest makespan (columns "makespan+SOC"), (2) minimum SOC (regardless of the makespan, reported in the "SOC" columns), and (3) minimum SOC with priorities (i.e., weighted SOC, reported in the "SOC with priorities" columns). For the SOC and SOC with priorities results, we present results for the naive model (denoted with a $naive$ subscript), in which the makespan is set to its upper bound, and the more advanced model mentioned (denoted with a $adv$ subscript), in which the makespan increases synchronously with the bound for SOC [24] (see Section V-A).

Consider the results for makespan+SOC. While the SAT solver was sometimes slower, it was in general more effective, solving all 20 instances within the time limit while the MIP solver could solve only 11. These results are counter-intuitive because this objective function involves multiple integer-domain variables and there is a perception that MIP is more suited to arithmetic constraints than SAT. Nonetheless, this experiment shows that log-encoded SAT code [30] is competitive and even better for this problem setting.

Next, consider the results for the SOC objective. Following prior work, we observe that optimizing SOC is harder than optimizing makespan [25]. Indeed, none of the solvers where able to solve all 20 instances. Here too the SAT solver

TABLE II
A COMPARISON OF SOLVERS' PERFORMANCE FOR MINIMIZING SUM-OF-COSTS (CPU TIME, SECONDS)

| Instance (SOC) | makespan+SOC | | SOC | | | | SOC with priorities | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | SAT | MIP | $SAT_{naive}$ | $SAT_{adv}$ | $MIP_{naive}$ | $MIP_{adv}$ | $SAT_{naive}$ | $SAT_{adv}$ | $MIP_{naive}$ | $MIP_{adv}$ |
| g16_p10_a05 (55) | 0.85 | **0.35** | 19.70 | 5.68 | 8.07 | **2.46** | 24.45 | 64.57 | **10.85** | 71.67 |
| g16_p10_a10 (126) | 8.48 | **1.10** | 64.80 | 35.82 | 69.38 | **24.49** | 127.67 | >600 | **96.50** | >600 |
| g16_p10_a20 (240) | 21.60 | **3.05** | 453.06 | **143.35** | 82.38 | 336.93 | >600 | >600 | >600 | >600 |
| g16_p10_a30 (376) | **23.38** | >600 | >600 | **495.04** | >600 | >600 | >600 | >600 | >600 | >600 |
| g16_p10_a40 (526) | **97.03** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g16_p20_a05 (66) | 3.84 | **0.97** | 26.16 | 16.20 | 10.76 | **8.14** | 33.64 | 90.28 | **14.04** | 109.80 |
| g16_p20_a10 (141) | 11.62 | **2.70** | 96.74 | **92.16** | 224.21 | 153.02 | **121.09** | >600 | 231.45 | >600 |
| g16_p20_a20 (278) | **16.91** | 17.66 | 528.72 | **209.74** | >600 | >600 | >600 | >600 | >600 | >600 |
| g16_p20_a30 (423) | **60.02** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g16_p20_a40 (520) | **46.87** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p10_a05 (124) | 14.70 | **2.65** | 230.50 | **29.91** | 93.46 | 42.08 | 264.88 | >600 | **115.94** | >600 |
| g32_p10_a10 (218) | **18.99** | 34.95 | >600 | **84.92** | >600 | 176.39 | >600 | >600 | >600 | >600 |
| g32_p10_a20 (446) | 76.22 | **18.53** | >600 | **586.71** | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p10_a30 (738) | **421.21** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p10_a40 (960) | **361.32** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p20_a05 (125) | 41.26 | **13.55** | 181.86 | **58.27** | 131.99 | 196.74 | 260.22 | >600 | **151.30** | >600 |
| g32_p20_a10 (253) | **25.44** | 66.38 | >600 | **112.20** | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p20_a20 (481) | **195.75** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p20_a30 (852) | **123.50** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| g32_p20_a40 (1050) | **197.33** | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 | >600 |
| Total solved | **20** | 11 | 8 | **12** | 7 | 8 | **6** | 2 | **6** | 2 |

TABLE III
A COMPARISON WITH STATE-OF-THE-ART MAKESPAN AND SOC (CPU TIME, SECONDS)

| Instance | Makespan | | | Sum of costs | | |
|---|---|---|---|---|---|---|
| | Picat | MDD | ASP | Picat | MDD | ICBS |
| g16_p10_a05 | 0.27 | **0.02** | 10.86 | 5.68 | **0.01** | **0.01** |
| g16_p10_a10 | 1.37 | **0.14** | 9.58 | 35.82 | **0.01** | **0.01** |
| g16_p10_a20 | 2.76 | **0.76** | 26.06 | 143.35 | **0.01** | **0.01** |
| g16_p10_a30 | 3.11 | **0.79** | >600 | 495.04 | 0.52 | **0.02** |
| g16_p10_a40 | 8.25 | **4.71** | >600 | >600 | **107.95** | >600 |
| g16_p20_a05 | 1.01 | **0.16** | 5.96 | 16.2 | **0.01** | **0.01** |
| g16_p20_a10 | 1.5 | **0.31** | 18.59 | 92.16 | 1.58 | **0.16** |
| g16_p20_a20 | 2.12 | **0.46** | 20.71 | 209.74 | 0.6 | **0.05** |
| g16_p20_a30 | 4.37 | **1.45** | >600 | >600 | >600 | >600 |
| g16_p20_a40 | 3.48 | **1.15** | >600 | >600 | >600 | >600 |
| g32_p10_a05 | 1.98 | **0.53** | 12.93 | 29.91 | **0.01** | **0.01** |
| g32_p10_a10 | 3.08 | **1.21** | 31.34 | 84.92 | **0.01** | **0.01** |
| g32_p10_a20 | 8.71 | **6.8** | 105.47 | 586.71 | 0.03 | **0.01** |
| g32_p10_a30 | **34.48** | 40.13 | 274.11 | >600 | 0.22 | **0.02** |
| g32_p10_a40 | 34.95 | **24.87** | >600 | >600 | 1.81 | **0.34** |
| g32_p20_a05 | 5.75 | **2.77** | 11.99 | 58.27 | **0.01** | **0.01** |
| g32_p20_a10 | 2.97 | **1.11** | 33.22 | 112.2 | 0.09 | **0.01** |
| g32_p20_a20 | 16.93 | **13.73** | 101.84 | >600 | 2.5 | **0.22** |
| g32_p20_a30 | 12.98 | **4.54** | 199.69 | >600 | 1.78 | **0.05** |
| g32_p20_a40 | 16.51 | **8.17** | 418.56 | >600 | 3.24 | **0.13** |
| Total solved | 20 | 20 | 15 | 12 | 18 | 17 |

performed best. Also, the advanced model [24] is clearly more efficient than the naive one, Finally, consider the results for the SOC with priorities objective. Adding priorities to agents increases the role of numerical objective and indeed the MIP solver was the best there. Notice also that the naive model is better in this setting as it requires a single call to the solver, while the advanced model tries many makespans. This indicates that there is an interesting relation between the satisfiability component (path finding, no-conflict constraints) and numerical objective, which is a topic for future work.

Table III compares the performance of our best model ($SAT_p$ and $SAT_{adv}$) with other state-of-the-art MAPF solvers. In particular, we compared with an ASP-based solver [8], MDD-SAT for optimizing makespan [22], MDD-SAT for optimizing SOC [24], and Improved Conflict-Based Search (ICBS) [4]. The ASP-based solver was the program given in [8] solved with the `clingo 4.5.4` ASP solver [6]. The MDD-SAT solver uses encodings very similar to ours, including a similar reachability-based preprocessing. MDD-SAT for SOC includes an additional optimization that reduces significantly the number of variables and constraints by counting only movements of agents that contribute to the cost above the sum of individual costs (sum of lengths of shortest paths connecting start and goal per individual agents). ICBS is the best solver from CBS family of search-based MAPF solvers. Note that there is no available ASP-based MAPF solver for minimizing SOC and no version of ICBS designed for minimizing makespan. For all solvers, the CPU time included all relevant preprocessing (including grounding time for ASP) and the solving time. The best result for each objective and instance was marked in bold.

Consider first the makespan objective. The results show that the ASP solver is clearly the slowest. It was able to find at least one plan for each of the instances, but failed to find optimal plans for 5 of the instances within the time limit. These results are consistent with the ones reported in [8]. The performance of our model and MDD-SAT is very similar, which is reasonable since the models are very similar. The difference can be attributed to the at-most-one constraint encoding, which is intensively used by both models: MDD-SAT uses the sequential counter encoding [18] while Picat uses the product encoding [5].

For the SOC objective, MDD-SAT has a clear advantage over our solution. We conjecture that this is due to (1) the more advanced preprocessing employed by MDD-SAT where the full shortest path to prune variables instead of the Manhattan distance, (2) the encoding of the sum-of-costs bound based on sequential counter that enables efficient Boolean constraint propagation within the SAT solver, and (3) the above-mentioned additional optimization that counts only movements over the sum of individual costs. ICBS, which is a highly optimized MAPF-specific search-baseed solver is understandably the fastest in most cases, except for one case which was solved by MDD-SAT and not by ICBS. Note that we can improve our SOC model using the MDD-

SAT optimizations. However, the focus of this work is not in devising the best declarative model for each MAPF variant, but to demonstrate the flexibility of our declarative model. For example, there is no ASP, MDD-SAT, or ICBS versions for supporting different agent priorities, while we have shown that it is trivial to do so with our declarative model.

## VII. Related Work and Discussion

Our declarative model follows the planning-as-satisfiability framework [10], [11], [13]. Declarative models based on the same framework have been proposed for SAT [23], ASP [8], and CP [15] solvers. The ASP encoding is arguably the most concise and elegant one. As we showed experimentally, our model is more efficient. More importantly, the downside of ASP is its lack of flexibility in expressing low-level constraints and control knowledge. For example, in ASP, it is difficult to implement the use of greater-than rather than equality in the transition constraint $B_{tav} = 1 \Rightarrow \Sigma_{u \in neibs(v)}(B_{(t+1)au}) \geq 1$, the preprocessing technique for eliminating variables, and the generate-and-test algorithm utilized to optimize makespan. The preprocessing technique for eliminating variables we describe in Section IV-C is well-used in constraint programming for maintaining consistency of some of the global constraints, such as the *regular* constraint [12].

Many MAPF solvers that are not based on a declarative model. Yu and LaValle formulates and solve MAPF as a network flow problem [28]. Others used sophisticated solvers based on heuristic search. A mini-survey of both algorithmic and declarative solutions to MAPF can be found in [16].

## VIII. Conclusion

We presented a constraint-based declarative model and its implementation in Picat for the MAPF problem. The model is easy and the implementation is concise. Through the Picat implementation, we provide for the first time a direct comparison of SAT and MIP solvers for MAPF. Our declarative solution using SAT is more competitive than existing declarative solutions and is comparable to state-of-the-art MAPF solvers. Also, while declarative models tend to suffer from poor performance, our model is comparable with the state-of-the-art. We also show how our solution can be easily altered for other variants and objectives, some of which were never addressed before. Further work includes comparison on more benchmarks and exploitation of domain knowledge, such as graph structures, for better performance.

## Acknowledgements

Roman Barták is supported the Czech Science Foundation under the project P202/12/G061 and together with Roni Stern and Pavel Surynek by the Czech-Israeli Cooperative Scientific Research Project 8G15027. Neng-Fa Zhou is supported in part by the NSF under grant number CCF1618046.

## References

[1] Dor Atzmon, Ariel Felner, Roni Stern, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. k-robust multi-agent path finding. In *International Symposium on Combinatorial Search (SoCS)*, pages 157–158, 2017.

[2] Dimitris Bertsimas and Robert Weismantel. *Optimization over integers*. Athena Scientific, 2005.

[3] Armin Biere. Lingeling, http://fmv.jku.at/lingeling/, 2014.

[4] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. ICBS: improved conflict-based search algorithm for multi-agent pathfinding. In *IJCAI*, pages 740–746, 2015.

[5] Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *the Int. Workshop of Constraint Modeling and Reformulation*, 2010.

[6] Clingo. Potassco, the potsdam answer set solving collection, https://potassco.org, 2016.

[7] Kurt M. Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res. (JAIR)*, 31:591–656, 2008.

[8] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *AAAI Conference on Artificial Intelligence*, 2013.

[9] Gurobi. http://www.gurobi.com/, 2016.

[10] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A novel transition based encoding scheme for planning as satisfiability. In *AAAI Conference on Artificial Intelligence*, 2010.

[11] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[12] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP*, pages 482–495, 2004.

[13] Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.

[14] Gabriele Röger and Malte Helmert. Non-optimal multi-agent pathfinding is solved (since 1984). In *Symposium on Combinatorial Search (SoCS)*, 2012.

[15] Malcolm Ryan. Constraint-based multi-robot path planning. In *ICRA*, pages 922–928, 2010.

[16] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artif. Intell.*, 195:470–495, 2013.

[17] David Silver. Cooperative pathfinding. In *the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pages 117–122, 2005.

[18] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *CP*, pages 827–831, 2005.

[19] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, volume 1, pages 28–29, 2010.

[20] Pavel Surynek. An optimization variant of multi-robot path planning is intractable. In *AAAI Conference on Artificial Intelligence*, 2010.

[21] Pavel Surynek. On propositional encodings of cooperative path-finding. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 524–531. IEEE, 2012.

[22] Pavel Surynek. A sat-based approach to cooperative path-finding using all-different constraints. In *Symposium on Combinatorial Search (SoCS)*, 2012.

[23] Pavel Surynek. A simple approach to solving cooperative path-finding as propositional satisfiability works well. In *PRICAI*, pages 827–833, 2014.

[24] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*, pages 810–818, 2016.

[25] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *Symposium on Combinatorial Search (SoCS)*, 2016.

[26] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.

[27] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI Conference on Artificial Intelligence*, 2013.

[28] Jingjin Yu and Steven M LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5):1163–1177, 2016.

[29] Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In *PADL*, pages 48–62, 2016.

[30] Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, page 15 pages, 2017.

[31] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.