Solving the Rehearsal Problem with Planning and with Model Checking

Peter Gregory*, Alice Miller†, and Patrick Prosser†

*Computer and Information Sciences, University of Strathclyde, Scotland
peter.gregory@cis.strath.ac.uk
†Department of Computing Science, University of Glasgow, Scotland,
{alice,pat}@dcs.gla.ac.uk

Abstract. Planning problems have been modelled and solved as constraint satisfaction problems [1–4]. Similarly, model checking problems have been modelled and solved as constraint satisfaction problems [5, 6]. In this paper we show that, conversely, planning and model checking techniques can each be applied to a constraint satisfaction problem. We demonstrate this by modelling and solving what might generally be accepted as a constraint satisfaction problem, the rehearsal problem [7], using both planning and model checking technologies. Both of these technologies compete with the more *natural* encoding, the constraint programming solution, in terms of search time.

1 Introduction

Constraint satisfaction techniques have been applied to planning problems for at least 20 years: Stefik's MOLGEN [8] used constraint posting to model the interaction between subproblems. More recently, planning systems have taken a Graphplan representation of planning problems, mapped this to a constraint satisfaction problem (csp) and solved the csp with advanced constraint processing techniques [1–4]. Constraint technology has also been applied to model checking problems [6, 5]. This suggests that constraint technology has a wide application. Model checking technology has also been applied to planning problems, where the planning problem is reformulated as a model checking problem which is then solved using model checking technology [9]. This might suggest that we can solve a planning problem with any one of these techniques but that the planning technology is too specialised to be used in any other domain.

Figure 1 depicts what we believe to be the state of the art with respect to modelling and solving these three classes of problems with these three different technologies. All our mappings so far are in one direction: from constraint programming (CP) to model checking and planning, and from model checking to planning. Could we turn things around? Could we solve a constraint satisfaction problem with planning or model checking technology? Indeed we can, and we demonstrate this on a problem that is generally acknowledged to be a constraint satisfaction problem, the rehearsal problem [7]. In this paper we re-introduce the rehearsal problem and present a constraint model in the choco toolkit. We then cast the problem as a planning problem and solve it with a planning toolkit (Metric-FF) and finally model and solve the problem using the SPIN model checker. We compare the performance of each technology and discuss the merits of each approach.

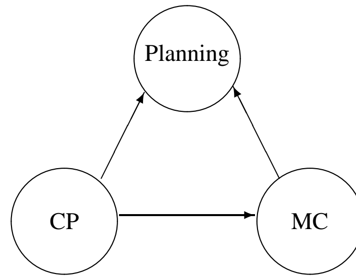

Fig. 1. Constraint Programming (CP) has been used to model and solve Planning problems and Model Checking (MC) problems. Model Checking (MC) has been used to solve planning problems. Could we turn some of these arrows around? Yes, we can.

2 A constraint satisfaction problem: the rehearsal problem

The rehearsal problem was introduced to the CP community by Barbara Smith, as cspLib problem number 39, and in APES report 67 [7]. An instance of the problem is shown in Figure 2 below. There are 9 pieces of music that have to be rehearsed; 5 players, p1 to p5; and a cost for performing each piece. The 0/1 entries indicate whether a player performs a given piece. For example, player p5 plays pieces 3, 5, 6, 7, and 8. Assuming the pieces are played in the order shown, player p5 needs to arrive at time 3 and can leave at time 9, and need not be paid for time slots 1, 2, and 9. However p5 will be paid 3 units for doing nothing while piece 4 is being performed.

piece	1	2	3	4	5	6	7	8	9
p1	1	1	0	1	0	1	1	0	1
p2	1	1	0	1	1	1	0	1	0
p3	1	1	0	0	0	0	1	1	0
p4	1	0	0	0	1	1	0	0	1
p5	0	0	1	0	1	1	1	1	0
cost	2	4	1	3	3	2	5	7	7

Fig. 2. An instance of the rehearsal problem, taken from [7] and also as cspLib entry 39. There are 9 pieces of music to be performed, and there are 5 players, p1 to p5. Each piece has an associated cost (the bottom line). An entry i,j is 1 if and only if the ith player performs the jth piece. A player should arrive in time to perform the first piece that they must play, and can leave only after performing the last piece that they must play. If a player is present when a piece is performed that player must be paid the cost of that piece, even if the player does not actually perform. The columns are to be permuted to minimise the cost of paying idle players.

The problem is then to permute the columns of the above table such that the total cost of idleness is minimised. A minimum cost ordering (there are at least 4) is

382715649 (i.e. start with piece 3, then piece 8, piece 2, ... finishing with piece 9), with a cost of 17.

3 A Constraint Programming Encoding

The encoding below is essentially the same as that of Barbara Smith except that channelling constraints are not used. Below we introduce the variables and the constraints (in *italics*).

- **M** a two dimensional array. M[i][j] = 1 if and only if player i performs piece j. The array M is essentially an array of constants. That is, M is the data read in initially and does not change.
- **s** a one dimensional array. If s[i] = j then piece i will be performed in time slot (column) j. That is, s gives us the permutation of the columns in the above figure. Each variable in s has a domain 1 to 9, and all the variables in s must take different values. Therefore we have a constraint amongst all the s variables, *allDiff(s)* [10]. This constraint allows us to dispense with the dual variables and channelling constraints used in [7].
- **T** is the timetable, and is a two dimensional array of 0/1 variables. *If s[j] = k then T[i][k] = M[i][j]*. That is, if piece j is being performed in time slot k (s[j] = k) and person i performs piece j (M[i][j]) then the person i is timetabled to perform piece j in time k (T[i][k] = M[i][j]).
- **C** is a two dimensional array, such that C[i][j] is the cost of player i performing piece j. The variable C[i][j] has a domain with two values, 0 and cost[i] (the cost of performing piece i) where cost[i] is the last row of the above table (e.g. cost[3] = 1).
- **arrived** is a two dimensional array of 0/1 variables. If arrived[i][k] = 1 then player i has arrived at time k. Furthermore, if a player has arrived at time k he has also arrived at time k+1. Therefore we have a *ladder* constraint[1] such that *if arrived[i][k] = 1 then arrived[i][k+1] = 1*. This has a rippling effect. Something must initiate that ripple, and that is when T[i][k] = 1. We then have the following constraints, *if T[i][k] = 1 then arrived[i][k] = 1*. That is, if player i is actually performing a piece at time k then he has arrived!
- **notLeft** is also a two dimensional array of 0/1 variables. If a player has not yet left at time k then notLeft[i][k] = 1. Again we have a ladder or rippling constraint, such that if at time k you have not yet left then at time k-1 you have also not left! Therefore we have the ladder/ripple constraints *if notLeft[i][k] = 1 then notLeft[i][k-1] = 1*. Again, we need something to kick-off this ripple, and again it is when a T[i][k] = 1. The constraint is then *if T[i][k] = 1 then notLeft[i][k] = 1*, i.e. if player i is actually performing at time k then he has not yet left.
- **CT** is the timetabling cost and is a two dimensional array of integer variables, similar to array C. Each variable in CT has a value in the range 0 to the maximum value in cost[i]. We now need to tie together arrays C (costs) and CT (the timetabled costs). If piece j is timetabled to be performed at time k then the timetabled cost at time k

[1] Peter Knightingale suggested this name.

is the cost at time j. This is expressed as follows: *if s[j] = k then C[i][j] = CT[i][k]*. The next constraint is the one that makes it all happen. If at time k a player has arrived, and the player has not yet left, and the player is not actually performing a piece then we incur the cost of that player being paid to do nothing. The constraint is then *if arrived[i][k] = 1 and notLeft[i][k] = 1 and T[i][k] = 0 then CT[i][k]* $\neq$ *0*. This constraint ties it all together; the bit about arriving, having not left, being timetabled but not actually doing anything, and the cost of it all.

total is the objective variable to be minimised, and is the sum of the variables in CT.

The above model was coded in choco which has a default search strategy. First, the search attempts to resolve all disjunctive constraints. Having done that, the variable with smallest domain is selected and the domain is split, until it reaches a singleton. This is hopeless for our model as choco will concentrate on dealing with the ripple constraints, trying to decide if someone has arrived or not yet left. We must therefore force choco to consider only the decision variables s[1] to s[9] and examine those variables in a good order. Barbara Smith proposed a static variable ordering heuristic for this problem where we select variables in the following order s[1], s[9], s[2], s[8], s[3], s[7], s[4], s[6], s[5], i.e. work from both ends of the timetable toward the middle. The reasoning behind this is that as soon as we have decided that someone has arrived and not yet left we can propagate and update the timetable cost CT. We can then use this in our branch-and-bound search.

3.1 Performance of the model

There are essentially three versions of the model. Version 1 does not use any heuristic ordering of the decision variables. Version 2 again does not use any heuristic ordering but does use a symmetry breaking constraint proposed in [7]. If we have a timetable with a given cost we can reverse the ordering of the pieces and obtain another timetable with the same cost. This can be prevented by adding the constraint $s[1] < s[9]$. Version 3 is the same as version 2 with the additional ends-to-middle variable ordering heuristic.

The choco code was compiled and run on a 755MHz machine with 256MB of ram. We measured nodes, backtracks and time in milliseconds to find and prove the optimal solution (cost = 17). The data is tabulated in Figure 3 and we see that Version 3 is by far the best. The symmetry breaking constraint makes only a marginal improvement in performance but the static variable ordering makes a significant difference, speeding up run time by a factor of 4.

4 A Planning Encoding

Planners are not regarded as versatile tools. Neither are they viewed, or even designed, as tools for solving optimization problems. However, by using the rehearsal problem as a test-case it can be shown that planners can be adapted to solve constraints problems optimally. At first glance the rehearsal problem does not seem to be a planning problem, or a problem that a planning approach could solve. However, similar problems do appear as subproblems of resource scheduling problems in which resources are to be scheduled in such a way as to be active for the shortest possible time.

	nodes	backtracks	time (ms)
Version 1	95171	197395	357990
Version 2	81170	149820	303100
Version 3	14417	36144	74230

Fig. 3. The performance of the choco model on a 755MHz machine with 256MB of ram. In Version 1 the variables s[1] to s[9] are selected in index order and there is no symmetry breaking. Version 2 uses index ordering of the variables and uses the symmetry breaking constraint s[1] $<$ s[9]. Version 3 selects variables in the order s[1], s[9], s[2], s[8], s[3], s[7], s[4], s[6], s[5] (i.e. working from both ends of the timetable into the middle) and uses the symmetry breaking constraint.

4.1 A PDDL encoding of the problem

Despite many 'benchmark' planning domains being viewed by the wider community as toy-domains, the planning community attempts to model realistic problems. Modelling the rehearsal problem requires several sophisticated features including numerics, quantified and disjunctive preconditions and effects. Many planners can handle a subset of these requirements but few can solve problems with all of them. However, Metric-FF [11] *can* deal with them all. Metric-FF uses a simple forward chaining search strategy; firstly using an 'enforced hill-climbing' algorithm then, on failure of that algorithm, best-first search. Since all solutions to the rehearsal problem are valid solutions it may be expected that Metric-FF would just return any arbitrary ordering for a solution. This is true unless certain restrictions are placed upon the problem. There are four different predicates necessary to define the rehearsal problem:

- `(present ?p - player)` musician `?p` has arrived.
- `(plays_in ?p - player ?pi - piece)` player `?p` plays in piece `?pi`.
- `(left ?p - player)` player `?p` has left (in contrast to having not yet arrived). This predicate is necessary so that musicians do not return once they have left.
- `(played ?p - piece)` piece `?p` has been played.

There were also two functions to evaluate:

- `(cost ?p - piece)` This is the time that a certain piece of music takes. It can be seen as a numeric cost associated with the piece.
- `(totalcost)` This function represents the summation of the wasted time during the entire rehearsal.

A representation is possible with just one operator but for clarity, we present a three operator version here. All of the code fragments are written in PDDL 2.1 [12]. The syntax is lisp-like, so, for example, `A or B` reads `(or A B)` in PDDL.

The three operators `(arrive ?p - player)`, `(leave ?p - player)`, and `(perform ?p - piece)` are defined as follows:

`(arrive ?p - player)` – The `arrive` operator has the precondition that the player has not already arrived and that he/she has not already left the rehearsal. The effect of the operation is that the musician is now present and able to perform.

```
(:action arrive
 :parameters
   (?p - player)
 :precondition
   (not
     (or (present ?p) (left ?p) )
   )
 :effect
   (present ?p)
)
```

`(leave ?p - player)` – The precondition states that a player has played all of the pieces he/she was involved in and that they are present. The effect of the action is that the player is no longer present and that they have left.

```
(:action leave
 :parameters
   (?p - player)
 :precondition
   (and
     (present ?p)
     (forall (?pi - piece)
       (imply (plays_in ?p ?pi) (played ?pi)) )
   )
 :effect
   (and
     (not (present ?p))
     (left ?p)
   )
)
```

`(perform ?p - piece)` – The precondition of the perform operator states that everybody who should be playing is present and that the piece hasn't already been played. The effect updates `totalcost` with the product of the cost of the piece and the amount of idle performers.

```
(:action perform
 :parameters
   (?p - piece)
 :precondition
   (and
     (not (played ?p))
     (forall (?pl - player)
       (imply (plays_in ?pl ?p) (present ?pl)) )
   )
 :effect
   (and
     (forall (?pl - player)
       (when (and (present ?pl)
                  (not(plays_in ?pl ?p)))
         (increase (totalcost) ( cost ?p ))
       )
     )
     ( played ?p )
   )
)
```

We exploit symmetry in the same way as that used in versions 2 and 3 of the choco encoding, described in section 3.1. When run initially, Metric-FF creates a trivial plan (the same ordering as in the initial state). Clearly a method of optimizing the plan is required.

4.2 Planning for Optimality

In their revision of the PDDL language, Fox and Long [12] created a method of supplying a metric within a planning problem (i.e. the domain-engineer has the option of specifying a function to optimize). If an upper-bound is placed on the possible value of `totalcost` then, as Metric-FF is a complete planner, eventually either a solution will be generated or no solution is possible. The optimisation problem can then be posed as a sequence of decision problems, where we iteratively reduce the value of `totalcost`. Of course, the planner must be executed from scratch for each of these iterations. A more efficient approach would be to employ a branch-and-bound technique to reduce the search. This would involve significant alteration to the planner and has not been implemented here.

4.3 Performance of Metric-FF

The following data was generated using Metric-FF V2.3 on a Pentium4 2600 with 512MB of system memory. Metric-FF gives an output of number of states visited. Although this number contains incomplete solutions it is the best comparison we have between the planning results and the nodes visited in the csp encoding. We report this measure as nodes, but only for proof of optimality, i.e. when `totalcost` $= 16$. We also report the time in milliseconds to solve the rehearsal problem to optimality, i.e.

the sum of the run times for the sequence of decision problems for $49 \geq$ `totalcost` ≥ 17, plus the time to prove optimality, i.e. when `totalcost` $= 16$. The initial value 49 of `totalcost` is the cost of the initial ordering in the problem as presented in Figure 2. Version 1 is solved without breaking symmetries. Version 2 uses the symmetry-breaking predicates. We also include the best nodes and time for the choco encoding, and the run time for the best ILOG Solver implementation in [7]. The reader should note that different hardware platforms have been used for the 3 studies, so we must be cautious in comparing results. However, if we assume that the planning technology is running on a processor 4 times faster than that used in the choco encoding, our best planning solution is at least an order of magnitude better than our best choco encoding.

	nodes	time (ms)
Version 1	10299	520
Version 2	6825	910
choco 3	14417	74230
Solver 4	-	990

Fig. 4. The performance of the planning solution. Version 1 and 2 use Metric-FF V2.3 on a Pentium4 2600 with 512 MB of ram. choco 3 is the best version of our choco encoding, running on a 755 MHz processor. Solver 4 is the best implementation in [7], running on a 600MHz Celeron PC.

With symmetry-breaking there are less nodes expanded, but the time is slower. It is often the case with symmetry-breaking that the effort required to break the symmetries make execution time less attractive. Since we order arbitrary pieces then it is fully conceivable that this choice affects the states visited and execution time.

Clearly the planning encoding performed better than the choco implementation. Compared to the encoding in ILOG Solver [7] our PDDL encoding is reasonably competitive (see the table above).

The choco encoding was greatly improved by a variable-ordering heuristic (Version 3, section 3.1). Similarly, Metric-FF makes two different heuristic estimates for its choice of action: one from the logical side of the problem and the other from the numeric. The user may alter the overall weightings of each of these functions. However, only the standard weightings were used here.

5 A Model Checking Encoding

Model checking is a technique whereby temporal logic properties of a system can be checked by building an abstract model of the system and checking whether the model satisfies the properties. The model is constructed using a specification language, and checked using an automatic model checker. Failure of the *model* to satisfy a property that the system is expected to satisfy indicates either that the model does not accurately reflect the behaviour of the system, or that there is an error (bug) in the original system.

Examination of counter-examples provided by the model checker enable the user to either refine the model or, more importantly, to debug the original system.

The SPIN verification system [13] is one of the most widely used model checkers. Developed at Bell labs, SPIN is used to verify high-level models of concurrent software systems. Promela is the specification language accepted by SPIN. It has a C-like syntax and has constructs which allow for the specification of non-determinism, concurrency and communication between processes. During model-checking the execution paths (sequences of states reachable from the initial state) are explored (usually) via depth-first search. All states are stored in a hash array. Once a previously visited state has been reached, the search backtracks. Additionally, all states in the current execution path are stored in a stack so that the path may be used to provide a counter example when an error state, for example, is discovered. There are no local search strategies available with SPIN, but by using a bounded search we can automatically find the shortest path to an error state (see below)

As well as checking for deadlock, livelock, process starvation etc. SPIN can perform verification of LTL (linear temporal logic) properties. These properties are assertions about every possible behaviour (or execution path) of the system and can be loosely divided into safety properties (a proposition θ_1 is always true) and liveness properties (eventually a proposition θ_2 will be true).

Suppose that we have a safety property ϕ of the form "(along all paths) θ_1 is always true". To show that ϕ holds for a given model, SPIN must perform a search of the state-space of the system to ensure that there are no states at which θ_1 is true. If, for a given model ϕ is not true, the search will only continue until a state s is found at which θ_1 does not hold. The current execution path, up to s is then provided as a counter-example.

If, on the other hand, we have a liveness property of the form "(along all paths) eventually θ_2 is true", things are not so simple. Suppose that s_0 is the initial state of the model. A counter-example in this case would consist of a path $s_0, s_1, \ldots, s_n$ say, where θ_2 does not hold at s_i, for $0 \leq i \leq n$ and where s_n is reachable from itself. This counter-example demonstrates that there exists an infinitely looping path along which θ_2 is never true (namely $s_0, s_1, \ldots, s_n, \ldots, s_n, \ldots, s_n, \ldots$) .

Although we give examples of LTL properties here, we do not provide details of the logic. A full description of LTL and its use in model checking can be found in [14]. In SPIN, LTL properties are converted to Büchi automata [15] expressed in the form of *never-claims*. Never-claims can be thought of as an additional process which executes a transition every time one of the other processes in the model has executed a transition. If no such transition is possible (if the current path can not possibly lead to an error for example) the current path is blocked, and the search backtracks.

In order to solve the rehearsal problem, very few of the features of SPIN (and of model checking in general) need to be exploited. For example there is no communication involved, or any true concurrency. We will only consider a single "process" - the rehearsal scheduler. The only other process is the never-claim, which can be thought of as recording the behaviour of the scheduler process, rather than as an independent process. Similarly, we are only interested in finding paths along which a given condition (or proposition) eventually holds (not that the condition will eventually hold along all

paths). This is equivalent to finding a counter-example to a safety property. Thus we are not considering liveness properties here.

One of the features of the SPIN model checker that we do make use of is the REACH algorithm, which allows us to determine whether, for a safety property, an error exists within a (user-defined) maximum depth, MAX say. (For a safety property "always θ", an error is said to exist at depth D if there is a path $s_0, s_1, \ldots, s_{D-1}$ and θ does not hold at s_{D-1}.) Once it has been determined that an error exists within a given bound, the SHORTEST_PATH algorithm can be used to find the shortest path to the error. During the SHORTEST_PATH algorithm the value of MAX is successively decremented until a depth MIN is found such that an error exists at depth MIN but no error exists at any depth less than MIN.

5.1 Modelling the rehearsal problem in Promela

A Promela program consists of process template *proctype* descriptions together with an *init* process in which instantiations of the (parameterised) proctypes are initiated. In this case we have a single process template, the *scheduler* proctype, which is instantiated via the init process. A simple scheduler process declaration is provided below. Note that *choose_next_piece* and *play_piece* are calls to (user-defined) *inline* functions. (In Promela an inline is similar to a procedure in an imperative language like C.)

```
proctype scheduler()
{byte current_cost=0;
 byte count=no_pieces;
        do
        ::(no_played==no_pieces)->break
        ::else->choose_next_piece(count);
                play_piece(count);
                no_played++;
                count=no_pieces
        od;
        STOP=1;
}
```

A call to the *choose_next_piece* inline results in the next piece to be chosen non-deterministically from the remaining pieces. The *play_piece* inline is as follows:

```
inline play_piece(j)
{byte count1=0;
 byte temp=0;
 do
 ::atomic{(count1==no_players)->played_piece[j]=1;
    count1=0;break}
 ::else->
  if
  ::((plays[count1].piece[j]==0)&&
                (present[count1]==1))
    ->temp=time_to_play[j];
     current_cost=current_cost+temp;temp=0
  ::atomic{(plays[count1].piece[j]==1)->
     present[count1]=1; pieces_left[count1]--;
    if
    ::(pieces_left[count1]==0)->present[count1]=0
    ::else->skip
    fi}
  ::else->skip
  fi;
  count1++
od}
```

When a call to the *play_piece* inline is made, for each player currently present who is not involved in the current piece, the cost is increased accordingly. Any player who is involved in the piece is made present (unless already present), and sent home if this is the final piece in which they are involved.

The $current_cost$ variable records the current cost and the (i, j)th element of the $plays.piece$ array records whether performer i is involved in piece j. The $present$, $pieces_left$ and $time_to_play$ arrays record which performers are currently present, how many pieces each performer has yet to play, and how long each piece takes to play (the cost of each piece) respectively. Initial values of these variables are either zero, in which they are set upon declaration within a preamble, or are set within the *init process* as seen below. Note that the scheduler process is initiated also from within *init*.

```
init
{ time_to_play[0]=2;time_to_play[1]=4;
      . . .;time_to_play[8]=7;
  plays[0].piece[0]=1; plays[0].piece[1]=1;
      . . .;plays[4].piece[7]=1;
  pieces_left[0]=6;. . . ;pieces_left[4]=5;

  run scheduler()
}
```

Leaving aside for the moment the issue of the *least* cost, how can we use SPIN to show that it is possible to schedule the rehearsal so that the total cost is less than a given value N say? Note that when all of the pieces have been played the $STOP$ variable takes the value 1. At all other times it remains set to 0.

For a specific value of N we can capture the proposition "all pieces have been played and the total waiting time is less than N" in LTL via the proposition p where p is $((STOP == 1)\&\&(current_cost < N))$. Checking that the property "always not p" holds will produce a counter-example if there exists a path in which p becomes true. It is therefore possible to determine the least waiting time by performing a series of model checking runs with decreasing values of N, until no error is found. This approach is similar to that used in the planning approach, described in section 4.2.

However, it would be more desirable for the model checker to perform this iteration automatically. Indeed, it seems sensible to use the iterative search facility provided by the SHORTEST_PATH algorithm, described above, and to find the shortest path to a violation of the property "always q", where q is $(STOP == 0)$.

In order to exploit the SHORTEST_PATH algorithm however, we must ensure that there is a direct correspondence between the length of a given path and the waiting time associated with that path. To do this, we have introduced a further inline function $wait$ which forces a new state to be reached for every unit of waiting time (cost). When piece j is played, for each non-participating player i, instead of increasing the value of $current_cost$ by $time_to_play[j]$ in one step, it is now increased in $time_to_play[j]$ single steps. Thus in the $play_piece$ inline we replace the statement $current_cost = current_cost + temp$ with a call to the $wait$ inline, via the statement $wait(temp)$ where the $wait$ inline is given by

```
inline wait(j)
{byte count2=0;
 do
 ::atomic{(count2==j)->count2=0;break}
 ::else->current_cost++;count2++
 od}
```

Every unit of waiting time (cost) will now involve an execution of this loop, and a new state in the current path.

(Atomic statements are used in Promela to reduce the amount of possible interleavings between concurrent processes. In this case it simply reduces the number of times the never-claim monitors the value of the $current_cost$ and $STOP$ variables.)

In order to enable us to determine the performance sequence corresponding to the shortest path, the $scheduler$ proctype is enhanced with print statements to announce which pieces are being played, and the final total cost (the value of the $current_cost$ variable when $STOP$ is set to 1). The performance sequence and total waiting time can then be extracted (via a Perl script for example) from an output file associated with the error trace.

5.2 Performance of SPIN

For all verification runs we used a PC with a 2.4GHz Intel Xenon processor, 3GB of available main memory, running Linux (2.4.18).

To perform SPIN verification we performed some initial exploratory runs to find a reasonable bound below which an error (or solution in this case) was known to exist. Having found that there was an error within depth 700 the SHORTEST_PATH algorithm was used to find the shortest path to the error, corresponding to the solution with

minimal waiting time. The SHORTEST_PATH search found a solution at depth 670, took 3.9 seconds (user + system time), and used 90.7 MB of main memory. There were 818164 stored states and 24308 matched states. Note that stored states correspond to nodes in a constraints solution, and matched states to backtracks.

It was very simple to implement the symmetry reduction used in both the constraints and planning approaches (see sections and 3.1 and 4.1 respectively). When the next piece to be played was chosen, piece 8 was only chosen if piece 0 had already been played. In this case a solution was found at depth 522 in 2.3 seconds, using 53.4 MB of main memory. In this case there were 473377 stored states and 13540 matched states. Note that the same rehearsal timetable was produced using the implementation with symmetry reduction, and that without.

It should be noted that, since model checking involves saving every new state, the memory requirements associated with model checking can become prohibitive. However, in this example memory requirements are small. In addition, there are compression algorithms available with SPIN which reduce the amount of memory required, at the expense of time.

6 Comparison of Results

In the following table (Figure 5), we compare the number of nodes (or stored states in the model checking case), number of backtracks (where available), and search time associated with the constraints, planning and model checking encodings. In each case, Version 1 is a preliminary encoding without symmetry, and Version 2 the corresponding encoding with symmetry. There is also a third version for the constraints encoding, in which the ends-to-middle variable ordering heuristic has been applied. As noted earlier, each of our experiments took place on different machines. However, if we optimistically assume that the model checking and planning solutions were running on machines 4 times faster than the choco encoding we can see that the non-CP technologies perform very well indeed. Measures of nodes and backtracks are reported only to give some indication of the size of the state space explored by each technology, but we can't really compare these figures.

Returning to the original results in [7], the best ILOG Solver encoding took 990ms on a 600MHz Celeron PC. Again, if we assume that our best performer (the planning solution), was running on a machine four times faster than the platform in [7] our best planning solution would be about half the speed of the ILOG solution. We think this is rather good, as ILOG solver is a highly optimised commercial toolkit whereas Metric-FF is a freely available planner and was not intended to be used for problems like the rehearsal problem.

Why did Metric-FF do so well? Metric-FF, like FF [16], employs a novel search strategy combining hillclimbing with complete search, whilst making use of powerful heuristic pruning techniques. In [16] Hoffmann and Nebel suggest that FF's good performance over the planning benchmarks has as much to do with the texture of the state space of these problems as it has to do with FF's technology. Maybe all that we can conclude is that the rehearsal problem has a state space that suits FF.

Could we improve the model checking approach? The algorithms used are rather basic. In order to allow us to detect the shortest path a new node is created per unit of

		nodes	backtracks	time(ms)
Constraints:				
	Version 1	95171	197395	357990
	Version 2	81170	149820	303100
	Version 3	14417	36144	74230
Planning:				
	Version 1	10299	--	521
	Version 2	6825	--	910
Model checking:				
	Version 1	818164	24308	3900
	Version 2	437377	13540	2300

Fig. 5. The performance of the three approaches. Note that hardware platforms differ across technologies. Nevertheless, the planning technology is at least an order of magnitude faster than any of the other approaches.

waiting time. On the one hand, this has the negative effect that the number of nodes is substantially more than in the other cases. However, the solution with the shortest waiting time could be found automatically, rather than as a sequence of decision problems as in the planning approach. The Promela description was fairly intuitive, and the hard work was left entirely to the model checker. It would certainly be beneficial to be able to direct the model checking search towards a solution faster, as in the csp approach.

Our choco encoding was the slowest. One explanation for this is the runtime environment: it was run within the choco interpreter rather than directly on the machine. Another explanation may be that that the model was not as efficient as it might have been. The fastest solution continues to be Barbara Smith's ILOG Solver encoding, when we factor in processor speed. Her solution was run on a machine approximately 4 times slower than the planning solution, and when we take that into consideration we should expect that it will run about twice as fast as the Metric-FF solution.

7 Conclusion

As Mark Twain said *To a man with a hammer, everything looks like a nail*. And to a constraint programmer (our third author), does everything look like a constraint satisfaction problem? We cannot afford to ignore other technologies. When we solve a planning problem with constraint programming we should also give some thought to doing the reverse: solving a csp with planning (as did our first author). The same holds with model checking (our second author). As we have seen, model checking and planning both represent and solve the rehearsal problem with little difficulty. What is more, they have outperformed our choco constraint programming solution in terms of time.

We have made no attempt to generalise from our study. All we have done is give an existence proof, that we can indeed model and solve a constraint satisfaction prob-

lem quite competitively using two technologies that are very different from constraint programming. Hopefully this work may be extended such that we can learn general lessons, and maybe identify the features of csp's that are best solved with these very different methods.

Acknowledgements

We would like to thank our reviewers for there useful and encouraging comments.

References

1. Van Beek, P., Chen, X.: CPlan: A Constraint Programming Approach to Planning. In: AAAI'99. (1999) 585–590
2. Do, M.M., Kambhampati, S.: Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. Artifi cial Intelligence **132** (2001) 151–182
3. Lopez, A., Bacchus, F.: Generalizing GraphPlan by Formulating Planning as a CSP. In: IJCAI'03. (2003)
4. Jefferson, C., Miguel, A., Miguel, I., Tarim, A.: Modelling and Solving English Peg Solitaire. In: CPAIOR'03. (2003)
5. Fribourg, L.: Constraint Logic Programming Applied to Model Checking. In: LOPSTR'99. (1999) 31–42
6. Delzanno, G., Podelski, A.: Model Checking in CLP. In: TACAS/ETAPS'99. (1999) 223–239
7. Smith, B.: Constraint Programming in Practice: Scheduling a Rehearsal. Technical report, APES (2003)
8. Stefi k, M.: Planning with Constraints. Artifi cial Intelligence **16** (1981) 111–140
9. Giunchiglia, F., Traverso, P.: Planning as Model Checking. In: ECP'99. (1999)
10. Jean-Charles Regin: A Filtering Algorithm for Constraints of Difference in CSPs. In: AAAI'94. (1994) 362–367
11. Hoffmann, J.: The Metric-FF planning system: Translating "ignore delete lists" to numeric state variables. Journal of Artifi cial Intelligence Research **20** (2003) 291–341
12. Fox, M., Long, D.: An extension to PDDL for expressing temporal planning domains. Journal of Artifi cial Intelligence Research **20** (2003) 61–124
13. Holzmann, G.J.: The SPIN model checker: primer and reference manual. Addison Wesley (2003)
14. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
15. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verifi cation (preliminary report). In: Proceedings, Symposium on Logic in Computer Science, Cambridge, Massachusetts, IEEE Computer Society (1986) 332–344
16. Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation through Heuristic Search. Journal of Artifi cial Intelligence Research **14** (2001) 253–302