Optimal Scheduling using Branch and Bound with SPIN 4.0

Theo C. Ruys*

Department of Computer Science, University of Twente.
P.O. Box 217, 7500 AE Enschede, The Netherlands.
http://www.cs.utwente.nl/~ruys/

Abstract. The use of model checkers to solve discrete optimisation problems is appealing. A model checker can first be used to verify that the model of the problem is correct. Subsequently, the same model can be used to find an optimal solution for the problem. This paper describes how to apply the new PROMELA primitives of SPIN 4.0 to search effectively for the optimal solution. We show how Branch-and-Bound techniques can be added to the LTL property that is used to find the solution. The LTL property is dynamically changed during the verification. We also show how the syntactical reordering of statements and/or processes in the PROMELA model can improve the search even further. The techniques are illustrated using two running examples: the Travelling Salesman Problem and a job-shop scheduling problem.

1 Introduction

SPIN [10,11,12] is a model checker for the verification of distributed systems software. SPIN is freely distributed, and often described as one of the most widely used verification systems. During the last decade, SPIN has been successfully applied to trace logical design errors in distributed systems, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. [13]. This paper discusses how SPIN can be applied effectively to solve discrete optimisation problems.

Discrete optimisation problems are problems in which the decision variables assume discrete values from a specified set; when this set is set of integers, we have an integer programming problem. Combinatorial optimisation problems are problems of choosing the best combination out of all possible combinations. Most combinatorial problems can be formulated as integer programs.

In recent years, model checkers have been used to solve a number of non-trivial optimisation problems (esp. scheduling problems), reformulated in terms of reachability, i.e. as the (im)possibility to reach a state that improves on a given optimality criterion [2,5,7,8,15,20]. Techniques from the field of operations research [22] – e.g. Branch-and-Bound [3] techniques – are being applied to prune

* This work is partially supported by the European Community IST-2001-35304 Project AMETIST (Advanced Methods for Timed Systems).

parts of the search tree that are guaranteed not to contain optimal solutions. Model checking algorithms have been extended with optimality criteria which provided a basis for the guided exploration of state spaces [2,15].

Though SPIN has been used to solve optimisation problems (i.e. scheduling problems [5,20]), the procedures used were not very efficient and the state space was not pruned in any way. This paper shows how the new version of SPIN can be used to effectively solve discrete optimisation problems, especially integer program problems. We show how Branch-and-Bound techniques can be added to both the PROMELA model and – even more effectively – to the property ϕ that is being verified with SPIN. To improve efficiency we let the property ϕ dynamically change during the verification. We also show how the PROMELA model can be reordered syntactically to guide the exploration of the state space. The paper does not compare existing techniques to solve optimisation problems to the one presented here; we only show how one might use *vanilla* SPIN to solve (small) optimisation problems.

The paper tries to retain the tutorial style of presentation of [18,19] to make the techniques easy to adopt by intermediate SPIN users. The techniques are explained by means of running examples of two classes of optimisation problems. The effectiveness of the techniques is illustrated by some experiments.

The paper is structured as follows. In Section 2 we introduce the Travelling Salesman Problem (TSP) and show how SPIN can be used to find the optimal solution for this problem. Section 3 briefly describes the new primitives of SPIN 4.0. In section 4 we show how the new primitives can be used to solve a TSP more effectively. In section 5 we apply the same techniques to a job-shop scheduling problem and show how Branch-and-Bound techniques can elegantly be isolated in the property which is being verified. The paper is concluded in Section 6.

Experiments. All verification experiments for this paper were run on a Dell Inspiron 4100 Laptop computer driven by a Pentium III Mobile/1Ghz with 384Mb of main memory. All `pan` verification runs were limited to 256Mb though. The experiments were carried out under Windows 2000 Professional and Cygwin 1.3.6; the `pan` verifiers were compiled using `gcc` version 2.95.3-5. For our experiments we used SPIN version 4.0.1 (7 Jan 2003). To compile the `pan` verifiers, we used the following options for `gcc`:

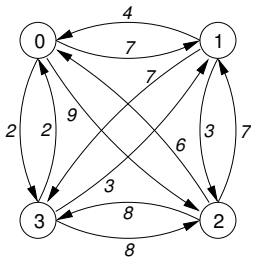
```
GCC_OPTIONS="-w -D_POSIX_SOURCE -DMEMLIM=256 -DSAFETY -DXUSAFE -DNOFAIR"
```

For PROMELA models without a `never`-claim, we added the `-DNOCLAIM` option. We executed the `pan` verifiers using the following directives:

```
PAN_OPTIONS="-m1000 -w20 -c1"
```

2 TSP with plain Spin

The Traveling Salesman Problem (TSP) [16,17] is a well known optimisation problem from the area of operations research [22]. In a TSP, n points (*cities*) are given, and every pair of cities i and j is separated by a distance (or cost) c_{ij}. The



	0	**1**	**2**	**3**
0	-	7	9	2
1	4	-	3	7
2	6	7	-	8
3	2	3	8	-

Fig. 1. Graph and matrix representation of the 4×4 example TSP.

problem is to connect the cities with the shortest closed *tour*, passing through each city exactly once. A specific TSP can be specified by a distance (or cost) matrix. An entry c_{ij} in row i and column j specifies the cost of travelling from city i to city j. The entries could be the Euclidean distances between cities in a plane, or simply costs – making the problem non-Euclidean. Extensive research has been devoted to heuristics for the Euclidean TSP (see e.g. [17]). Construction heuristics for the non-Euclidean TSP are much less investigated. This paper considers non-Euclidean TSPs only. The TSP is NP-complete.

Modelling a TSP in PROMELA is straightforward. To illustrate the idea we develop a PROMELA model for the sample TSP of Fig. 1. Fig. 1 shows both a graph- and matrix-representation of a 4×4 TSP. Fig. 2 shows the PROMELA model of the TSP of Fig. 1. The salesman itself is modelled by a single process `TSP`. For each place i that the man has to visit, there is a label `P`i in the process `TSP`. The salesman starts at label `P0`. From each label `P`i the salesman can (non-deterministically) go to any label `P`j that has not been visited yet. A `bit`-array `visited` is used to keep track of the places that have already been visited.[1] If, after reaching place `P`i, it turns out that all places have been visited, the salesman has to go back to place `P0`. To keep track of the travelling costs, a variable `cost` is used. This variable is initialised on `0`. When we move from place `P`i to `P`j, this variable is updated with the cost c_{ij} from the cost-matrix of Fig. 1.

Now that we have a PROMELA model of the TSP, we want to use SPIN to find the optimal route of the TSP. Fig. 3 shows a general algorithm for finding an optimal solution for an optimisation problem using a model checker. The procedure has been used in [5,20]. The algorithm iteratively verifies whether 'the `cost` will *eventually* be greater than `min`'. Each time this property is violated, SPIN has found a path leading to a final state for which the cost is less than `min`. For each error, SPIN generates an error trail which corresponds with the better route. As the number of possible routes is finite, at a certain point SPIN will not find a route for which the `cost` is less than the `min` found so far. Consequently, the error trace which was generated last (corresponding with this optimal `min`) is the optimal route.

[1] In this example we use PROMELA's built-in support for `bit`-arrays. In our experiments, however, we used the `bit`-vector library as discussed in [18], which are more efficient.

```
bit  visited[3];
int  cost;

active proctype TSP()
{
P0: atomic {
        if
        ::  !visited[1] -> cost = cost + 7 ; goto P1
        ::  !visited[2] -> cost = cost + 9 ; goto P2
        ::  !visited[3] -> cost = cost + 2 ; goto P3
        fi ;
    }
P1: atomic {
        visited[1] = 1;
        if
        ::  !visited[2] -> cost = cost + 3 ; goto P2
        ::  !visited[3] -> cost = cost + 7 ; goto P3
        ::  else        -> cost = cost + 4 ; goto end
        fi ;
    }
P2: atomic {
        visited[2] = 1;
        if
        ::  !visited[1] -> cost = cost + 7 ; goto P1
        ::  !visited[3] -> cost = cost + 8 ; goto P3
        ::  else        -> cost = cost + 6 ; goto end
        fi ;
    }
P3: atomic {
        visited[3] = 1;
        if
        ::  !visited[1] -> cost = cost + 3 ; goto P1
        ::  !visited[2] -> cost = cost + 8 ; goto P2
        ::  else        -> cost = cost + 2 ; goto end
        fi ;
    }
end:
}
```

Fig. 2. PROMELA model of the sample TSP of Fig. 1.

This approach works, but is (highly) inefficient: the complete state space already contains the most optimal solution. After a single run over the state space one should be able to report on the optimal solution. The problem, however, is that we cannot compare information (e.g. the `cost`) obtained via different execution paths in standard SPIN. This is inherent to the application of model checkers as a black box for solving optimisation problems.

3 Spin version 4.0

SPIN version 4.0 [10] – available from [11] – supports the inclusion of embedded C code into PROMELA models through five new primitives:

- `c_decl`: to introduce C types that can be used in the PROMELA model;
- `c_state`: to add new C variables to the PROMELA model. Such new variables can have three possible scopes:
  - *global* to the PROMELA model;
  - *local* to one of the processes in the model; or

```
    input: PROMELA model M with cost added to the states.
    output: the optimal solution min for the optimisation problem of M.

1   min ← (worst case) maximum cost
2   do
3      use SPIN to check M ⊨ ◇(cost > min)
4      if (error found)
5      then min ← cost
6   while (error found)
```

Fig. 3. Algorithm to find the optimal solution for an optimisation problem using SPIN.

  - *hidden*, which means that the variable will not end up in the state vector, but can be accessed in `c_expr` or `c_code` fragments.

- `c_expr`: to evaluate a C expression whose return value can be used in the PROMELA model (e.g. as a guard);
- `c_code`: to add arbitrary C code fragments as an atomic statement to the PROMELA model. For example, the `c_code` primitive enables to include useful `printf`-statements in the verifier for debugging purposes.
- `c_track`: to include (external) memory into the state vector.

The purpose of the new primitives is to provide support for automatic model extraction from C code. And although "it is not the intent of these extensions to be used in manually constructed models" [10], the extensions are helpful for storing and accessing global information of the verification process.

Within `c_expr` of `c_code` fragments one can access the global and local variables of the currrent state through the global C variable `now` of type `State`. The global variables of the PROMELA model are fields in a `State`. For example, if the PROMELA model has a global variable `cost`, the value of this variable in the current state can be accessed using `now.cost`.

As of version 4.0, the `pan`-verifier generated by SPIN also contains a guided simulation mode. It is no longer needed to replay error trails with SPIN.

For more details on the new features of SPIN 4.0 the reader is deferred to [10]. The rest of this paper only uses the primitives `c_state`, `c_expr` and `c_code`.

4 TSP with Branch-and-Bound

In this section, we will discuss how the new C primitives of SPIN 4.0 can be used to compute the optimal solution of a TSP more efficiently. We show how SPIN can be used to obtain the optimal solution in a single verification run. Branch-and-Bound techniques can be used to prune the search tree. We also show how heuristics can be used to further improve the search.[2]

[2] This paper only applies heuristics on the PROMELA level. Edelkamp et. al. [6] use a more powerful approach in HSF-SPIN, where heuristics are applied in the internals of SPIN.

SPIN 4.0 allows us to add *hidden* `c_state` variables to the `pan` verifier within the PROMELA model. Consequently, while exploring the state space, each time SPIN finds a better solution it can save this solution in such a *hidden* variable. To get the best route for our TSP problem with SPIN 4.0, the TSP model needs to be altered as follows:

- Add a hidden, global variable `best_cost` to the PROMELA model and consequently to the `pan` verifier. Initialise this variable `best_cost` on a worst-case estimate of the cost of a schedule, e.g.,
  ```
  c_state "int best_cost = 1000" "Hidden"
  ```
  Due to the scope `"Hidden"`, the variable `best_cost` will *not* be stored in the state vector and will be global to all execution runs. The declaration and initialisation of `best_cost` is copied verbatim to the `pan.c` file.
- Whenever a new solution is found (i.e. when the label `end` is reached), the `cost` for that new route is compared with the `best_cost` sofar. If `cost` is smaller, we have found a better solution, so the variable `best_cost` is updated and the trace is saved:
  ```
  ...
  end:
      c_code {
          if (now.cost < best_cost) {
              best_cost = now.cost;
              printf("\n> best cost sofar: %d ", best_cost);
              putrail();
              Nr_Trails--;
          }
      }
  ```
  The function `putrail` saves the trace to the current state (i.e. it writes the states in the current depth-first search stack to a `trail`-file). The statement `Nr_Trails--` makes sure that a subsequent call of `putrail` will overwrite a previous (less optimal) trail. Both `putrail` and `Nr_Trails` are defined in the generated `pan.c` file.

Branch-and-Bound in the model. Branch-and-Bound (B&B) [3,22] is an approach developed for solving discrete and combinatorial optimisation problems. The essence of the B&B-approach is the following:

- Enumerate all possible solutions and represent these solutions in an *enumeration tree*. The leaves are end-points of possible solutions and a path from the start node to a leaf represents a solution.
- While building the tree (i.e. the state space), we can stop considering descendents of an interior node, if it is certain that all paths via this node will (i) either lead to an *invalid solution* or (ii) will have *higher costs* than the best path found so far.

The B&B-approach is not a heuristic or approximating procedure, but it is an exact, optimising procedure that finds an optimal solution.

In our PROMELA model of the TSP problem, the B&B-approach can be applied to 'prune' the state space. If in a place `P`*i* the current `cost` is already higher than the best cost so far (i.e. `best_cost`), it is not useful to continue searching. So at the beginning of every place `P`*i* of our model we add the following `if`-statement with `c_expr` expression:

```
if
::  c_expr { now.cost > best_cost } -> goto end
::  else
fi;
```

Branch-and-Bound in the property. Recall the original idea of the algorithm of Fig. 3 which iteratively checks $\Diamond$(`cost` $>$ `min`) to find an optimal solution. Although inefficient, due to SPIN's on-the-fly model checking algorithm, for each subsequent iteration, less of the state space will be checked. For each execution path, SPIN will stop searching as soon as it finds a state for which `cost` $>$ `min` holds. Furthermore, SPIN will exit with an error as soon as it finds an execution path for which the final `cost` is lower than `min`. So, in a way, SPIN's on-the-fly verification algorithm already performs some B&B-functionality by default.

Using the possibilities of SPIN 4.0, we can improve the verification of the $\Diamond$-property by replacing `min` with the *hidden* global variable `best_cost`. We define the following macro using a `c_expr` statement:

```
#define higher_cost (c_expr { now.cost >= best_cost })
```

and we check $\Diamond$`higher_cost`. As the variable `best_cost` is changed during the verification, the property that is being checked is dynamically changed during the verification!

No cycles. SPIN translates the property $\Diamond$`p` to the following `never`-claim:

```
never { /* !<>p */
accept_init:
T0_init:
    if
    ::  (!((p))) -> goto T0_init
    fi;
}
```

Given this `never`-claim, the `pan` verifier will (i) search for states where `!p` does *not* hold (i.e. where `p` does hold, and thus the `if`-statement blocks) or (ii), due to the `accept_init`-label, for each state s where `!p` holds, the verifier will try to find a cycle from s to itself (i.e. an acceptance cycle). For discrete optimisation problems, however, the search space will be a tree without cycles. Consequently, the search for an acceptance cycle will always be in vain and thus unneeded. It is therefore enough to let the `pan` verifier run in safety mode, without the `-a` option to check for acceptance cycles.

Note that due to SPIN's *smart* double-nested depth-first search [14], this optimisation is more effective on the verification time than on the memory needed to store the state space.

```
1  procedure dfs(s: state)
2     if error(s) then report error fi
3     add s to Statespace
4     foreach successor t of s do
5        if t not in Statespace then dfs(t) fi
6     od
7  end dfs
```

Fig. 4. Basic depth-first search algorithm [14].

Nearest Neighbour Heuristic. When using B&B-methods to solve TSPs with many cities, large amounts of computer time may be required. For this reason, heuristics, which quickly lead to a good (but not necessarily optimal) solution to a TSP, are often used. One of such heuristics is the "Nearest Neighbour Heuristic" (NN-heuristic) [22]. To apply the NN-heuristic, the salesman begins at any city and then visits the nearest city. Then the salesman goes to the unvisited city *closest* to the city it has most recently visited. The salesman continues in this fashion until a tour is obtained.

In order to apply the NN-heuristic to SPIN we must control the order in which neighbour places are selected. In order words, we must control the order of successor states in the state space exploration algorithm of SPIN. The algorithm of Fig. 4 from [14] shows a basic depth-first search algorithm which generates and examines every global state that is reachable from a given initial state. Although SPIN uses a slightly different (nested) depth-first search algorithm, for the discussion here, Fig 4 suffices.

There is only one place in the algorithm where we can influence SPIN's depth-first search: line 4, where the algorithm iterates over the successor states of state `s`. SPIN always uses the same well-defined routine to order the list of successors:

- *Processes.* SPIN arranges the processes in *reverse* order of creation. That is, the process with the highest process id (`pid`) will be selected first.
- *Statements.* Within each process, SPIN considers all possible executable statements. For a statement without guards, there is at most one successor. For an `if` or `do` statement, the list of possible successors is the list of executable guards in the same order as they appear in the PROMELA model.

As the PROMELA processes can be created in any order and we are also free to order the guards within `if` and `do` clauses, we have limited control over SPIN's search algorithm from within the PROMELA model.

Fortunately, the control over the order of the guards within `if`-clauses is enough to apply the NN-heuristic to SPIN. To make sure that in every place `P`i, SPIN will first consider the place `P`j for which the cost c_{ij} is the lowest, the guards of all `if`-clauses are sorted on the cost c_{ij}, such that the guard with the lowest cost c_{ij} is at the top and the highest cost is at the bottom.

Experimental results. To compare the different approaches w.r.t. the TSP, we have carried out some experiments with some randomly generated TSPs. The

	dim=11	*dim=12*	*dim=13*	*dim=14*	*dim=15*
no B&B	572729	1878490	5459480	o.m.	o.m.
unsorted, B&B in model	278753	212984	514332	2478440	2820880
unsorted, B&B in property	111920	72022	173309	1050580	1010080
sorted, B&B in model	132517	54924	140075	1748130	1388100
sorted, B&B in property	49801	16662	43240	737107	480572

Table 1. Verification results (number of states) of verifying PROMELA models of five randomly generated TSP cost matrices using different types of optimisation schemes.

original approach which lets SPIN iteratively check $M \vDash \Diamond(\texttt{cost} > \texttt{min})$ has been left out of the experiments for obvious reasons. Table 1 lists the results of the experiments for randomly generated TSPs of dimension 11–15. We used a script to generate the cost-matrix for these TSPs where each c_{ij} was randomly chosen from the interval 1-100.[3] We used another script to generate the PROMELA models for the particular TSP as described in this section. The entry 'o.m.' stands for 'out of memory'.

From the experiments we can learn that B&B in the property is more advantageous than B&B in the model. This does not come as a suprise as due to the addition of B&B-functionality in the PROMELA model, the number of states of the `TSP` process increases. It is also interesting to see that the NN-heuristic really pays of. As the cost matrices are randomly generated, we cannot compare the results for the different dimensions.

5 Personalisation Machine

In this section, we discuss the application of the B&B-approach to a job-shop scheduling problem. We will extend the 'Branch-and-Bound in the property' technique as discussed in Section 4 by adding more bounding conditions to the property.

Problem description. The problem itself is a simplified version of a case study proposed by Cybernetix (France) within the Advanced Methods for Timed Systems (AMETIST, IST-2001-35304) project [1]. Cybernetix is a company manufacturing machines for the personalisation of smart cards. These machines take piles of blank smart cards as raw material, program them with personalised data, print them and test them.

Fig. 5 shows a schematic overview of the personalisation machine that we discuss in this paper. Cards are transported by a *Conveyer* belt. There are `NPERS` *Personalisation Stations* where cards can be personalised. The conveyer is `NPERS+2` positions long. The *Unloader* puts empty cards on the belt. The *Loader* removes personalised cards from the belt. The order in which the cards

[3] If the interval from which the different costs c_{ij} is (much) smaller, e.g. 1–10, the number of states drops significantly due to SPIN's state matching.

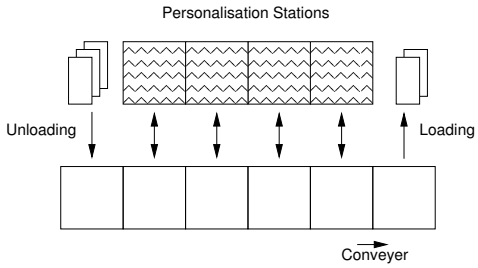


Fig. 5. Schematic overview of the personalisation machine.

are loaded *from* the belt should be same as the order in which they were unloaded *onto* the belt.

The conveyer can only move a step to the right which takes `tRIGHT` time units. If cards are unloaded onto the belt or loaded from the belt, the conveyer cannot move. Unloading and loading can be done in parallel. Unloading and loading takes `tUNLOAD` resp. `tLOAD` time units. If after a conveyer move, an empty card is under a personalisation station, the card might be taken of the belt by the personalisation station and personalisation of the card will start immediately. The personalisation of a card takes `tPERSONALISE` time units. In the original case study description, `tRIGHT` is equal to `1`, `tUNLOAD` and `tLOAD` are both `2`, whereas `tPERSONALISE` lies between `10` and `50`.

Goal. Given `NPERS` personalisation stations, the goal is to find an optimal schedule to personalise `NCARDS` cards.

PROMELA *model.* Modelling the personalisation machine in PROMELA is straightforward. The conveyer belt is modelled by an array of `NCELLS=NPERS+2` cells. A cell is represented by a `short`. If a cell has the value `0` it is empty. If a cell contains a value `n>0` the cell contains an unpersonalised card with number `n`. If `n<0`, the card has been personalised by one of the stations. There is one global variable `time` which is updated by the processes that 'consume time'. So the PROMELA model contains the following global variables:

```
short  belt[NCELLS];
short  time;
```

Apart from the global variables of the model, we also define a *hidden* `c_state` variable `best_time` which holds the time of the best schedule found so far. The behaviour of the model is specified by several parallel processes. The process `Conveyer` just moves the conveyer belt one step to the right. After updating the `belt`, the process increases the variable `time` with `tRIGHT` steps. The `Conveyer` process is modelled as follows.

```
proctype Conveyer() {
    byte i = 0;
    do
    ::  d_step {(belt[NCELLS-1] == EMPTY) && CARD_ON_BELT ->
            i=NCELLS-1;
            do
            ::  (i > 0) -> belt[i] = belt[i-1]; i=i-1
            ::  else    -> break
```

```
            od;
            belt[0] = EMPTY;
            time    = time + tRIGHT;
        }
    od
}
```

The macro `CARD_ON_BELT` returns `1` if there is a card on the belt.

The other two *logical* processes that 'consume time' are the *Unloading* and *Loading* process. Because unloading and loading might happen concurrently, the behaviour of both processes is modelled by a single process `UnloaderLoader`. The unloading part just puts cards on the belt. The loading part will remove cards from the belt and will check that the order of the cards is still correct. If not, it sets the `time` to -1.

Below we only include fragments of the loading part of the `UnloaderLoader` process. If the last card has been taken from of the belt, we check whether the schedule found is faster than the best schedule so far. If this is the case, we update the *hidden* `c_state`-variable `best_time`.

```
 ::  atomic { (belt[LAST] == expectedCard) ->
        belt[LAST]   = EMPTY;
        expectedCard = expectedCard-1;
        time         = time + tLOAD;
        if
        ::  expectedCard <  -(NCARDS+1) -> assert(false)
        ::  expectedCard == -(NCARDS+1) ->
            atomic {
                c_code { if (now.time < best_time) {
                             best_time = now.time;
                             Nr_Trails=0;
                             putrail();
                         }
                };
                break;
            }
        ::  else
        fi
    }
::  atomic { (belt[LAST] !=0 && belt[LAST] != expectedCard) ->
        time = -1;
        break;
    }
```

Each personalisation station is modelled by a process `PersStation(i)`. When an unpersonalised card `n` is in `belt[i]`, a personalisation station might start personalising this card `n`. Unlike the other processes, the process `PersStation` waits for time to pass. After it has taken an card from the belt it sets its `finish_time[i]` to the time that it will have finished the personalisation of `n` (i.e. `time + tPERSONALISE`). Then the process starts waiting till the `time` has reached `finish_time[i]`.

Variable time advance. Because either the conveyer or unloader might have to wait for a personalisation station to finish, we also need a process which consumes 'idle' time. In our initial, naive model we used a process `Tick` which just increases the `time` by 1 time unit. The total number of *ticks* was bounded by a constant. The obvious disadvantage of this method is that the process `Tick` can always do

a `time` tick; even when there are no personalisation stations currently 'waiting' for the `time` to reach their finishing time.

Therefore, in our current model we follow Brinksma and Mader [5], who use the well-known *variable time advance* procedure [21]. With a variable time advance procedure, simulated time goes forward to the next moment in time at which some event triggers a state transition, and all intervening time is skipped. With respect to the personalisation machine this means that we let `time` jump to the `finish_time[i] > 0` which is the earliest.

Heuristics. In the discussion on the algorithm of Fig. 4 we noted that we can guide SPIN's depth-first search by changing the order in which SPIN considers successor states of a state `s`. SPIN arranges the processes in *reverse* order of creation. That is, the process that is created last, will be selected first in considering the next successor state.

For optimal schedules for the 'personalisation machine' it is clear that the number of idle time steps by the `TimeAdvance` process should be minimized. So a step of the `TimeAdvance` process should be the last step to be considered by SPIN. Furthermore, as personalisation takes the longest time, starting the personalisation card should be considered first by SPIN.

Branch-and-Bound. Following the conclusions on the TSP, we want to apply the B&B-approach using a dynamic bound in the property. We will check $\Diamond$`too_late_or_wrong_schedule`, where the macro is defined as

```
#define too_late_or_wrong_schedule \
  (c_expr { (now.time >= best_time) || \
            (now.time < 0) || \
            (will_be_too_late()) || \
            (wrong_schedule()) \
  })
```

The macro expands to a `c_expr` expression which apart from the now familiar bound on the `time` and the test on negative `time` due to an incorrect schedule, contains two additional function calls: `will_be_too_late` and `wrong_schedule`. These two functions try to decide at an early stage whether the current schedule leads to an inferior or incorrect schedule. Both C functions only use the current state (i.e. `now`) and the `best_time` found so far.

- The function `will_be_too_late` checks whether the minimum time to finish the cards that are still in the machine already exceeds the `best_time` so far. The function only looks at the last card (i.e. the card with sequence number `NCARDS`) in the machine and computes the minimal time left for this card to reach the *Loader*.
- To signal incorrect schedules, the `UnloaderLoader` sets the `time` to `-1` whenever a card is to be loaded from the belt which is out of order. It will be more advantageous, however, to discover such incorrect schedules (much) earlier. The function `wrong_schedule` returns `1` if either one of the two conditions hold:

- *Two personalised cards on the* `belt` *are out-of-order*:
  $\exists\, 1 \leq i, j \leq \texttt{NPERS} + 1 :$
  $(i < j) \wedge (\texttt{belt[i]} < 0) \wedge (\texttt{belt[j]} < 0) \wedge (\texttt{belt[i]} < \texttt{belt[j]})$
- *An personalised card is under a personalisation station containing a card with a lower original sequence number*:
  $\exists\, 1 \leq i \leq \texttt{NPERS} : (\texttt{belt[i]} < 0) \wedge (-\texttt{belt[i]} > \texttt{card\_in\_pers[i]})$

Both functions together are coded in less than 70 lines of C code.

Get all optimal schedules. Due to the structure of the problem, SPIN will always find just a single (optimal) schedule for a given `time`. The reason for this is that for all schedules with the same end-time, in the last-but-one state, the last card will be under the *Loader*. Due to state matching of SPIN all these states will be regarded to be the same. To obtain *all* optimal schedules, an extra 'magic number' can be added to each state. The magic number ensures that each state will be unique. It is obvious that making the states unique will have a negative impact on the number of states.

Experimental results. To compare the various optimisations on the model of the 'personalisation machine', we have carried out experiments with several combinations of the B&B-optimisations discussed. For these experiments we used the following values for the time-constants: `tRIGHT=1`, `tUNLOAD=tLOAD=2` and `tPERSONALISE=10`. We have verified six different versions of the model. The models can be characterised as follows:

- v1 Model which uses a naive ordering of the creation of processes: `UnloaderLoader`, the `PersStation`-processes, `Conveyer` and finally `TimeAdvance`. The B&B-functionality is isolated in the property, but we only bound on: "`now.time >= best_time || now.time < 0`"
- v2 Model with an improved ordering of the processes: `TimeAdvance`, `UnloaderLoader`, `Conveyer` and finally the `PersStation`-processes. Version v2 uses the same B&B-property as v1.
- v3 = v2, but adding "`|| wrong_schedule()`" to the B&B property
- v4 = v2, but adding "`|| will_be_too_late()`" to the B&B property
- v5 = v2, but adding "`|| wrong_schedule() || will_be_too_late()`" to the B&B property (so, v5 = v3 + v4)
- v6 = v5, but adding a 'magic number' to each state (and thus obtaining all optimal schedules)

Table 2 shows the results of verifying the different versions of the PROMELA model for different values of `NPERS` and `NCARDS`. It is clear that the optimisations discussed can be quite effective. The difference between the version with no optimisations at all (v1) and all optimisations enabled (v5) is nearly two orders of magnitude. Note that from Table 2 alone we cannot conclude much on the relative effectiveness of the different optimisations. Only the results between v1 and v2 and between v4 and v5 can be compared directly as apart from the different optimisations nothing has changed in the models. Looking at the results

	NPERS=3 NCARDS=4			NPERS=4 NCARDS=4			NPERS=4 NCARDS=5		
	states	mem	time	states	mem	time	states	mem	time
v1	213760	23.2	1.9	1182600	122.6	12.3	o.m	o.m	o.m
v2	161140	18.4	1.4	869594	91.3	8.6	o.m	o.m	o.m
v3	125501	15.4	1.1	677040	72.1	6.4	o.m	o.m	o.m
v4	9709	<5.0	0.1	46600	9.0	0.5	457395	50.1	4.1
v5	6463	<5.0	0.1	33000	7.7	0.3	304731	34.8	2.6
v6	59715	9.8	0.5	477057	54.0	3.6	o.m	o.m	o.m

Table 2. Verification results (number of states, memory consumption in Mb and verification time in seconds) of finding the optimal schedule for PROMELA models of the 'personalisation machine' using several different optimisations.

for v4 and v5, it is clear that discarding the schedules that will be too late (v5) is more effective than discarding incorrect schedules beforehand (v4). Also note that changing the creation order of the processes (v1 vs. v2) has a considerable impact on the number of states. As predicted, adding a magic number to states to obtain all optimal schedules (v6) is expensive.

6 Conclusions

The use of model checkers for optimisation problems is appealing. A model checker can first be used to verify that the model of the problem is correct. Subsequently, the same model can be used to find an optimal solution for the problem. Iteratively checking $\Diamond(\texttt{cost} > \texttt{best\_so\_far})$ will eventually deliver the optimal solution, but the approach is highly inefficient. We have shown that with the new C primitives of SPIN 4.0, the optimal solution can be found in a single verification run with some minor modifications to the PROMELA model.

The search for an optimal solution can be greatly improved using B&B-techniques in the PROMELA model and/or property. A clear advantage is that the alterations can be done on the level of the PROMELA model. One does not have to alter the source code of SPIN or the verifier `pan`. We have seen that specifying the B&B-optimisations in the property has several advantages. First of all, all optimising code can nicely be isolated within the property: the PROMELA model does not have to be altered. But more importantly, specifying the B&B-behaviour in the property is more effective (w.r.t. the number of states) than adding it to the PROMELA model. Note that the LTL property that is being verified is dynamically changed during the verification.

The B&B-approach is most effective if SPIN can be guided into finding a good solution as soon as possible. Therefore it is advantageous to apply heuristics to the PROMELA model such that promising successor states are selected first in SPIN's depth-first search algorithm. On the PROMELA level, the user can reorder the guards in `if-` and `do` statements and/or can change the order of process creation (and thus the scheduling of the processes).

Earlier approaches (cite: [2,7,15]) have extended existing model checking algorithms with optimality criteria to guide the exploration of states. Behrmann et. al. [2], for example, annotate each state with the estimated minimum cost to reach the goal state and explore the state space by always selecting the state with the smallest minimum cost. Compared to such *local* approaches, this paper applies a more *global* approach in the sense the pruning is isolated in the property that is being checked.

The work might be extended in several ways:

- It would be interesting to see how the *global* approach with SPIN compares to the *local* approaches of [2,15] and other, more traditional techniques for obtaining optimal solutions for optimisation problems.
- The use of model checkers to solve optimisation problems is limited to the number of states that is needed to find an (optimal) solution. For most classes of discrete optimisation problems, however, there is no need to store the complete state space as the state space is just a tree without loops. The exploration algorithm of SPIN might be changed in such a way that not all states are stored. One might turn off SPIN's state matching functionality all together, or apply a garbage collection algorithm to remove states that are not longer needed, or use a state cache.
- In this paper we used PROMELA to prune the search space of optimisation problems. SYMMSPIN [4] is a symmetry reduction package on top of standard SPIN. The idea is to prune parts of the state space for which there is no need to visit them due to the symmetric nature of the PROMELA model. A drawback of extensions of SPIN like SYMMSPIN is that they are implemented by changing the original source of SPIN or by modifying the source code of the generated `pan` verifier. Consequently, with each new version of SPIN, the extension might cease to work. Now that extensions can be implemented from within the PROMELA model this opens doors to packages on top of SPIN which are easier to maintain.

Acknowledgements. The author wishes to thank Angelika Mader for insightful discussions on the 'personalisation machine' case study. Joost-Pieter Katoen is thanked for valuable comments to improve both the contents and the presentation of the paper. Tomas Krilavičius is thanked for making available UPPAAL-models of the 'personalisation machine' and Kim G. Larsen for showing pointers to related work.

References

1. Advanced Methods for Timed Systems (AMETIST) Project. IST-2001-35304. Homepage: http://ametist.cs.utwente.nl/.
2. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Procs. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 174–188, Genova, Italy, April 2001. Springer.

3. O. S. Benli. The Branch-and-Bound Approach. In Anil Mital, editor, *Industrial Engineering Applications and Practice: Users' Encyclopedia*, 1999. CD-ROM edition, chapter available from http://http://www.csulb.edu/~obenli/.
4. D. Bošnački, D. Dams, and L. Holenderski. Symmetric Spin. In Havelund et al. [9], pages 1–19.
5. E. Brinksma and A. Mader. Verification and Optimization of a PLC Control Schedule. In Havelund et al. [9], pages 73–92.
6. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed Explicit Model Checking with HSF-SPIN. In M. B. Dwyer, editor, *Model Checking Software, Procs. of the 8th Int. SPIN Workshop*, volume 2057 of *LNCS*, pages 57–79, Toronto, Canada, May 2001. Springer.
7. A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Procs. of the 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA 1999)*, pages 280–286. IEEE Computer Society, 1999.
8. A. Fehnker. *Citius Vilius Melius – Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* PhD thesis, University of Nijmegen, The Netherlands, April 2002.
9. K. Havelund, J. Penix, and W. Visser, editors. *SPIN Model Checking and Software Verification, Procs. of the 7th Int. SPIN Workshop (SPIN'2000)*, volume 1885 of *LNCS*, Stanford, California, USA, August 2000. Springer.
10. G. J. Holzman. *The SPIN Model Checker – Primer and Reference Manual.* Addison-Wesley, Boston, USA, 2003.
11. G. J. Holzmann. Spin homepage: http://spinroot.com/.
12. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.
13. G. J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
14. G. J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *The SPIN Verification System, Procs. of the 2nd Int. SPIN Workshop (SPIN'96)*, volume 32 of *DIMACS Series*, Rutgers University, New Jersey, USA, August 1996. AMS.
15. K. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As Cheap As Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In G. Berry, H. Comon, and A. Finkel, editors, *Procs. of the 13th Int. Conf. on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 493–505, Paris, France, July 2001. Springer.
16. Princeton University, Mathematics Department. Traveling Salesman Problem – Homepage. http://www.math.princeton.edu/tsp/.
17. G. Reinelt. *The Travelling Salesman – Computational Solutions for TSP Applications*, volume 840 of *LNCS*. Springer, 1994.
18. T. C. Ruys. Low-Fat Recipes for Spin. In Havelund et al. [9], pages 287–321.
19. T. C. Ruys. *Towards Effective Model Checking.* PhD thesis, University of Twente, Enschede, The Netherlands, March 2001. *Available from the author's homepage.*
20. T. C. Ruys and E. Brinksma. Experience with Literate Programming in the Modelling and Validation of Systems. In B. Steffen, editor, *Procs. of the 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, number 1384 in LNCS, pages 393–408, Lisbon, Portugal, April 1998.
21. G. S. Shedler. *Regenerative Stochastic Simulation.* Academic Press, Boston, 1993.
22. W. L. Winston. *Operations Research – Applications and Algorithms.* Duxbury Press, Belmont, California, USA, third edition, 1994.