International Journal on Software Tools for Technology Transfer (2018) 20:515–528
https://doi.org/10.1007/s10009-018-0498-1

SPIN 2016

A case study of planning for smart factories

Model checking and Monte Carlo search for the rescue

Stefan Edelkamp[1] · **Christoph Greulich**[2]

Published online: 7 August 2018

Abstract
In this work, we propose the application of the SPIN software model checker to a multiagent system that controls the industrial production of goods. The flow of material is buffered on a production line with assembling stations. As the flow of material is asynchronous at each station, queuing is required as long as buffers provide waiting room. Besides validating the design of the system, the core objective of this work is to find concurrent plans that optimize the throughput of the system. In the mapping of the production system to the model checker, we model the production line as a set of communicating processes, with the movement of items modeled as channels. Experiments show that the model checker is able to analyze the system, subject to the partial ordering of the product parts. It derives valid and optimized plans with several thousands of steps using constraint branching in branch-and-bound search. We compare the results with a randomized exploration based on recent advances in Monte Carlo search.

Keywords Model checking · Action planning · Flow production · Monte Carlo search

1 Introduction

The ongoing transformation of production industries causes a paradigm shift in manufacturing processes toward new technologies and innovative concepts, called cyber, smart, digital or connected factory. The sector is entering its fourth revolution, characterized by a merging of computer networks and factory machines. At each link in production and supply chains, tools and workstations communicate constantly via Internet and local networks. Machines, systems and products exchange information both among themselves and with the outside world.

Such production systems are installed for goods that are produced in high quantities but in different configuations. By optimizing the flow of production, manufacturers hope to speed up individualized production at a lower cost and in a more environmentally sound way. In manufacturing practice of smart factories, there are not only lines with stations arranged one behind the other, but also more complex networks of stations at which assembly operations are performed. The considerable difference from flow lines, which can be analyzed by known methods, is that a number of required components are brought together to form a single unit for further processing.

For the optimization of the production, we explore the state space induced as a system of reactive modules, probably the most widely used simulation technique. Modern software systems support lightweight processes or threads. By the growing amount of non-determinism, however, such systems encounter their limits to optimize the concurrent acting of individual processes.

Agent-based simulation is a relatively new technique and consists of autonomous agents (self-directed objects which move about the system) and rules (which the agents follow to achieve their objectives). Agents move about the system interacting with each other and the environment and are used to model situations in which the entities have some form of intelligence. While the underlying implementation to control the production is in fact a multiagent system with goods that are associated with software agents that have desires of what they eventually become, and intension to avoid too much

✉ Stefan Edelkamp
stefan.edelkamp@kcl.ac.uk

[1] Department of Informatics, King's College London, 30 Aldwych, London WC2B 4BG, UK

[2] Faculty for Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany

queuing, in the optimization we rather look at the network of deterministic reactive processes.

Performance analysis of production systems is generally needed during the planning phase regarding the system design, when the decision for a concrete configuration of such a system has to be made. The planning problem arises, e.g., with the introduction of a new model or the installation of a new manufacturing plant. Because of the investments involved, optimization of the system is crucial. The expenditure for new machines, for buffer or handling equipment, and the holding costs for the expected work-in-process face revenues from sold products. The performance of a concrete configuration is characterized by the throughput, i.e., the number of items that are produced per time unit. Other performance measures are the expected work in process or the idle times of machines or workers.

In this paper, we utilize the state-of-the-art model checker SPIN [31] as a performance analysis and optimization tool for such manufacturing systems, together with its input language Promela to express the flow production of goods. There are some twists needed to adapt SPIN to the optimization of the system that are uncovered in the sequel of the text. Language features from the latest version of the model checker (including loops and native c-code verification) are exploited. The text pushes forward the exchange of modeling and exploration between model checking and action planning, while studying an industrial case for the techniques to reach out to the real world.

Our running case study is the Z2, a physical monorail system for the assembling of tail lights. Unlike most production systems, Z2 employs agent technology to represent autonomous products and assembly stations. We formalize the production floor in this manufacturing process as a system of communicating processes and apply SPIN to analyze its behavior. Using optimization mechanisms implemented on top of SPIN, additional to the verification of the correctness of the model, we exploit its exploration process for the optimization of the production.

Instead of a simple objective like the number of steps in the plan, we look at a more complex cost function that computes the makespan based on travel and queue waiting times over a set of transport agents.

The algorithmic contribution of this text is general cost-optimization via constraint branch-and-bound. The optimization approaches originally invented by Ruys for SPIN were designed for traversing state space trees [45,46], while the proposed approaches also support the traversal of state space graphs. Our argument in favor of using SPIN in this case study is that there is a natural correspondance between the queuing of messages in a communication channel and of products on a monorail to be processed in assembling stations. We also look into a framework of Monte Carlo search for comparison.

The paper is structured as follows. First, we review related work on industrial production and planning via model checking. Next, we introduce our Z2 case study of tail light assembling with its various manufacturing stages. The modeling as a distributed set of communicating processes (agents) is assisted by one possible formalization. We define the problem objective to be minimized and give a proof that the production flow problem is at least NP-hard. Afterward, we turn to the intricacies of the Promela model specification, introducing the rail switches as proctypes and the monorails as communication channels. The deterministic optimization scheme and its extensions are discussed, studying depth-first branch-and-bound, guarded branching, c-expression and process synchronization. Then, we describe an alternative for analyzing the Z2 model applying Monte Carlo search. In the empirical part, we study the effectiveness of the proposed approaches. In addition to this single-player Monte Carlo game framework implementation, we look at the parameterization of SPIN, and at two different models of simulation time.

2 Related work

Since the origin of the term artificial intelligence, the automated generation of plans for a given task has been seen as an integral part of problem solving in a computer. In action planning [23], we are confronted with the descriptions of the initial state, the goal (states) and the available actions. Based on these, we want to find a plan containing as few actions as possible (in case of unit-cost actions, or if no costs are specified at all) or with the lowest possible total cost (in case of general action costs).

The process of fully automated property validation and correctness verification is referred to as model checking [12]. Given a formal model of a system and a property specification in some form of temporal logic (such as LTL [22]), the task is to validate, whether or not the specification is satisfied in the model. If not, the model checker returns a counterexample trace as a witness for the falsification of the property.

Planning and model checking have much in common [10, 24]. Both rely on the exploration of a potentially large state space of system states. Usually, model checkers only search for the existence of specification errors in the model, while planners search for a path from the initial state to one of the goal states. Moreover, there is rising interest in planners that prove insolvability [30], and in model checkers to produce minimal counterexample [18].

Such form of planning via model checking has been pioneered by Abbeddaim and Maler, looking at job-shop scheduling problems with timed automata [1]; by Brinksma and Mader [6] in the verification and optimization of a PLC control schedule; and by Wijs [51] in a general treatment on analyzing and optimizing system behavior in time.

In terms of leveraging state space search, over the last decades there has been much cross-fertilization between fields. For example, based on Satplan citev32 bounded model checkers exploit SAT and SMT representations [2,4] of the system to be verified, while directed model checkers [16,35] exploit planning heuristics to improve the exploration for falsification; partial-order reduction [25,50] and symmetry detection [20,38] limit the number of successors, while symbolic planners [11,17,33] apply functional data structures like BDDs to represent sets of states succinctly.

Especially in open, unpredictable, dynamic, and complex environments, multiagent systems are applied to determine adequate solutions for transport problems. For example, agent-based commercial systems are used within the planning and control of industrial processes [13,29] as well as within other areas of logistics [7,19]. A comprehensive survey is provided by [42].

Flow line analysis is often done exploiting queuing theory [8,39]. Pioneering work in analyzing assembly queuing systems with synchronization constraints analyzes assembly-like queues with unlimited buffer capacities [27]. It shows that the time an item has to wait for synchronization may grow without bound, while limitation of the number of items in the system works as a control mechanism and ensures stability. Work on assembly-like queues with finite buffers all assume exponential service times [3,32,36].

Model checking production flow is rare. Timed automata were used for simulating material flow in agricultural production [28]. There are, however, numerous attempts to apply model checking to validate the work of multiagent systems. The LORA framework [52,53] uses labeled transition and Kripke systems for characterizing the behavior of the agents (their belief, their desire and their intention), and temporal logics for expressing their interplay, as well as for the progression of knowledge. Alternatives consider a multiagent system as a game, in which agents —either in separation or cooperatively—optimize their individual outcome [47]. Communication between the agent is available via writing to and reading from channels or via common access to shared variables. Other formalization approaches include work in the context of the MCMAS tool by Lomuscio1. Recently, there has been some approaches to formalize multiagent systems as planning problems [41].

3 Case study: Z2

One of the few successful real-world implementations of multiagent flow production is the so-called Z2 production floor unit [21,40]. The Z2 unit consists of six workstations where human workers assemble parts of automotive taillights. The system allows production of certain product variations and reacts dynamically to any change in the current order situation, e.g., a decrease or an increase in the number of orders of a certain variant. As individual production steps are performed at the different stations, all stations are interconnected by a monorail transport system. The structure of the transport system is shown in Fig. 1. On the rails, autonomously moving shuttles carry the products from one station to another, depending on the products requirements. The monorail system has multiple switches which allow the shuttles to enter, leave or pass workstations and the central hubs.

The goods transported by the shuttles are also autonomous, which means that each product decides on its own which variant to become and which station to visit. This way, a decentralized control of the production system is possible.

The modular system consists of six different workstations, each is operated manually by a human worker and dedicated to one specific production step. At production steps III and V, different parts can be used to assemble different variants

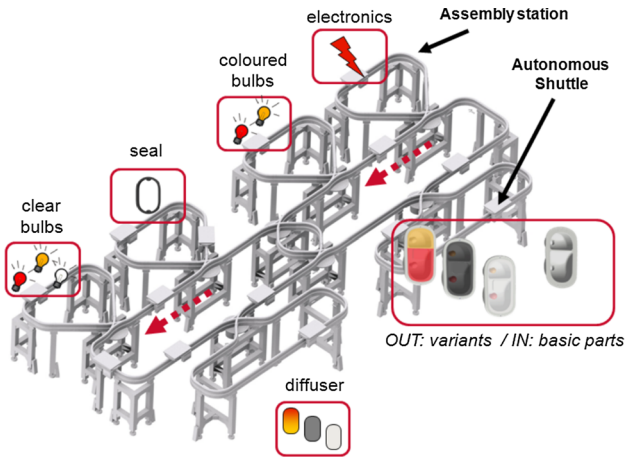

Fig. 1 Assembly scenario for taillights [40]

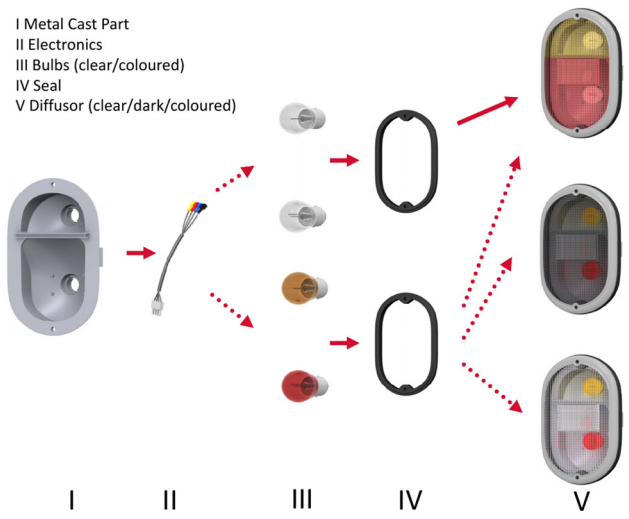

Fig. 2 Assembly states of taillights [21]

of the taillights as illustrated in Fig. 2. At the first station, the basic metal-cast parts enter the monorail on a dedicated shuttle. The monorail connects all stations; each station is assigned to one specific task, such as adding bulbs or electronics. Each taillight is transported from station to station until it is assembled completely.

From the given case study, we derive a more general notation of flow production for an assembly line network. System progress is non-deterministic and asynchronous, while the progress of time is monitored.

Definition 1 (*Flow production*)
A *flow production* is tuple $F = (A, P, G, \prec, S, Q)$ where

- A is a set of all possible assembling actions;
- P is a set of n products; each $P_i \in P, i \in \{1, \ldots, n\}$, is a set of assembling actions, i.e., $P_i \subseteq A$;
- $G = (V, E, w, s, t)$ is a graph with start node s, goal node t, and weight function $w : E \to I\!R_0$;
- $\prec = (\prec_1, \ldots, \prec_n)$ is a vector of assembling plans with each $\prec_i \in A \times A, i \in \{1, \ldots, n\}$, being a partial order;
- $S \subseteq E$ is the set of assembling stations induced by a labeling function $\rho : E \to A \cup \{\emptyset\}$, so that $S = \{e \in E \mid \rho(e) \neq \emptyset\}$
- Q is a set of (FIFO) queues, all of finite size together with a labeling $\chi : E \to Q$.

Products $P_i, i \in \{1, \ldots, n\}$, travel through the network G, meeting their assembling order $\prec_i \subseteq A \times A$ of the assembling actions A.

Definition 2 (*Run, plan, path*)
Let $F = (A, P, G, \prec, S, Q)$ be a flow production. A run π is a set of triples (e_j, t_j, l_j) of edges e_j, queue insertion positions l_j, and execution time-stamp t_j, $j \in \{1, \ldots, n\}$. The run partitions into a set of n plans $\pi_i = (e_1, t_1, l_1), \ldots, (e_{m_i}, t_{m_i}, l_{m_i})$, one for each product $P_i, i \in \{1, \ldots, n\}$. Each plan π_i corresponds to a path, starting at the initial node s and terminating at goal node t in graph G.

In the real-world implementation of the Z2 system, every assembly station, every monorail shuttle and every product is represented by a software agent. Even the RFID readers which keep track of product positions are represented by software agents which decide when a shuttle may pass or stop. The agent representation is based on the well-known Java Agent Development Kit (JADE) and relies heavily on its Foundation for Intelligent Physical Agents (FIPA)-compliant messaging components.

Most agents in this system resemble simple reflex agents. These agents just react to requests or events which were caused by other agents or the human workers involved in the manufacturing process. In contrast, the agents which represent products are actively working toward their individual goal of becoming a complete taillight and reaching the storage station. In order to complete its task, each product has to reach sub-goals which may change during production as the order situation may change. The number of possible actions is limited by sub-goals which already have been reached, since every possible production step has preconditions as illustrated in Fig. 3.

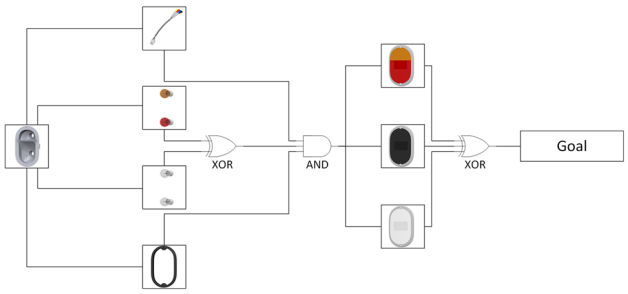

Fig. 3 Preconditions of the various manufacturing stages

The product agents constantly request updates regarding queue lengths at the various stations and the overall order situation. The information is used to compute the utility of the expected outcome of every action which is currently available to the agent. High utility is given when an action leads to fulfillment of an outstanding order and takes as little time as possible. Time, in this case, is spent either on actions, such as moving along the railway or being processed, or on waiting in line at a station or a switch. Each agent individually makes its decisions [43].

More generally, the objective of products in such a flow production system can be formalized as follows.

Definition 3 (*Optimization objective, travel and waiting time*)
Let $F = (A, P, G, \prec, S, Q)$ be a flow production. The optimization objective is to minimize

$$\max_{1 \leq i \leq n} wait(\pi_i) + time(\pi_i)$$

over all possible paths with initial node s and goal node t, where

- $time(\pi_i) = \sum_{e \in \pi_i} w(e)$ is the travel time of product P_i, alias the sum of edge costs
- $wait(\pi_i) = \sum_{(e,t,l)(e',t',l') \in \pi_i, e' \in Pred(e)} t - (t' + w(e'))$ the waiting time, where $Pred(e) = \{e' = (u, v) \in E \mid e = (v, w)\}$ is the set of predecessor edges of e.

It is not difficult to show that the production flow problem is at least NP-hard. To obtain a decision rather than an optimization problem, we choose the time threshold large enough, so that only goal achievement is of interest. The reduction to 3-SAT then is as follows.

We take an instance of 3-SAT with n variables and m clauses. Each clause is a disjunction of literals $l_1 \cup l_2 \cup l_3$, where each literal is a variable in $\{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$. Every clause shuttle (of m shuttles) consists of variable assemblies, representing its 3 literals. At $2n$ stations, the variables are set either to true or false (like a clear or colored bulb). The assignment, therefore, avoids a variable being set to both true or false. If one variable x_i is fixed to a value true and false, all assembling plans of the other shuttles are dynamically adjusted to only allow this truth value assignment to the variable x_i (and the opposite assignment to $\neg x_i$).

The terminal state is to have all shuttles arrive at the target location, so the terminal state is reached if and only if the 3-SAT formula is fulfilled.

The Z2 multiagent system was developed strictly for the purpose of controlling the Z2 monorail hardware setup. Nonetheless, due to its hardware abstraction layer [40], the Z2 multiagent system can be adapted into other hardware or software environments. By replacing the hardware with other agents and adapting the monorail infrastructure into a directed graph, the Z2 multiagent system can be transferred to a virtual simulation environment [26]. Such an approach, which treats the original Z2 agents like black boxes, can easily be hosted by a JADE-based event-driven multiagent system simulation platform. Experiments show that the simulated and the real-world scenarios match. For this study [26], the authors extended the model with timers to measure the time taken between two graph nodes. For the analysis, we simplify the system to reactive processes.

4 Promela model

Promela is the input language of the model checker SPIN, the ACM-awarded popular open-source software verification tool, designed for the formal verification of multi-threaded software applications, and used by thousands of people worldwide. Promela defines asynchronously running communicating processes, which are compiled to finite-state machines. It has a c-like syntax, and supports bounded channels for sending and receiving messages.

Channels in Promela follow the FIFO principle. They implicitly maintain order of incoming messages and can be limited to a certain buffer size. Consequently, we are able to map edges to communication channels. Unlike the original Z2 multiagent system, the products are not considered to be decision-making entities within our Promela model. Instead, the products are represented by messages which are passed along the node processes, which resemble switches, station entrances and exits.

We provided the physical model with timers to measure the time taken between two graph nodes. Since the hardware

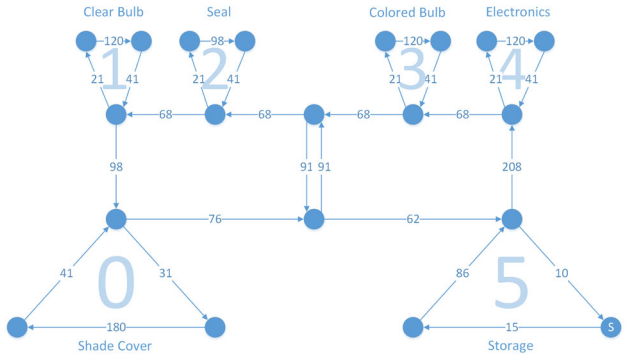

Fig. 4 Weighted graph model of the assembly scenario

includes many RFID readers along the monorail, which all are represented by an agent and a node within the simulation, we simplified the graph and kept only three types of nodes: switches, production station entrances and production station exits. The resulting abstract model of the system is a weighted graph (see Fig. 4), where the weight of an edge denotes the traveling/processing time of the shuttle between two respective nodes.

The Promela model is designed to apply branch-and-bound optimization in order to evaluate the optimal throughput of the original system. Instead of local decision making, the various processes have certain non-deterministic options of handling incoming messages, each leading to a different system state. The model checker systematically computes these states and memorizes paths to desirable outcomes when it ends up in final states. As mentioned before, decreasing production time for a given number of products increases the utility of a final state.

The important aspect is that for branch-and-bound optimization SPIN does not stop with proving but with disproving, and if it is forced to go on searching by its termination code. Either there is no continuation, then time is progressed automatically (by the so-called watchdog), or the goal of the simulation has been reached in which an assertion `assert(!final)` is thrown, and a counterexample is generated, the trail is copied and the number of trails are counted.

We derive a Promela model of the Z2 as follows. First, we define global settings on the number of stations and number of switches. We also define the data type storing the index of the shuttle/product to be byte. In the model, switches are realized as processes and edges between the units by the following channels.

```
chan entrance2exit[STATIONS]=[1] of {shuttle};
chan exit2switch[STATIONS]=[BUFSIZE] of {shuttle};
chan switch2switch[SWITCHES]=[BUFSIZE] of {shuttle};
chan switch2entrance[STATIONS]=[BUFSIZE] of {shuttle};
```

As global variables, we have bit-vectors for marking the different assemblies.

```
bit metalcast[SHUTTLES];
bit electronics[SHUTTLES];
bit bulb[SHUTTLES];
bit seal[SHUTTLES];
bit cover[SHUTTLES];
```

Additionally, we have a bit-vector that denotes when a shuttle with a fully assembled item has finally arrived at its goal location. A second bit-vector is used to set for each shuttle whether it has to acquire a colored or a clear bulb.

```
bit goals[SHUTTLES];
bit color[SHUTTLES];
```

A switch is a process that controls the flow of the shuttles. In the model, a non-deterministic choice is added to either enter the station or to continue traveling onwards on the cycle. Three of four switching options are made available, as immediate re-entering a station from its exit is prohibited.

```
proctype Switch(byte in; byte out; byte station) {
 shuttle s;
 do
 :: exit2switch[station]?s; switch2switch[out]!s;
 :: switch2switch[in]?s; switch2switch[out]!s;
 :: switch2switch[in]?s; switch2entrance[station]!s;
 od
}
```

The entrance of a manufacturing station takes the item from the according switch and moves it to the exit. It also controls that the manufacturing complies with the capability of the station.

First, the assembling of product parts is different at each station; in the stations 1 and 3, we have the insertion of bulbs (Station 1 provides colored bulbs, Station 3 provides clear bulbs), Station 2 assembles the seal, Station 4 the electronics and Station 0 the cover. Station 5 is the storage station where empty metal casts are placed on the monorail shuttles and finished products are removed to be taken into storage. Secondly, there is a partial order of the respective product parts to allow flexible processing and a better optimization based on the current load of the ongoing production.

```
proctype Entrance(byte station) {
 shuttle s;
 do
 :: switch2entrance[station]?s;
    entrance2exit[station]!s
 if
 :: (station == 4) -> electronics[s] = 1;
 :: (station == 3 && !color[s]) -> bulb[s] = 1;
 :: (station == 2)-> seal[s] = 1;
 :: (station == 1 && color[s]) -> bulb[s] = 1;
 :: (station == 0 && seal[s] && bulb[s] &&
     electronics[s])-> cover[s] = 1;
 :: (station == 5 && cover[s]) -> goals[s] = 1;
 :: else
 fi
 od
}
```

An exit is a node that is located at the end of a station, at which assembling took place. It is connected to the entrance of the station and the switch linked to it.

```
proctype Exit(byte station){
 shuttle s;
 do
 :: entrance2exit[station]?s; exit2switch[station]!s;
 od
}
```

A hub is a switch that is not connected to a station but provides a shortcut in the monorail network. Again, three of four possible shuttle movement options are provided

```
proctype Hub(byte in1;byte out1;byte in2;byte out2){
 shuttle s;
 do
 :: switch2switch[in1]?s; switch2switch[out1]!s;
 :: switch2switch[in1]?s; switch2switch[out2]!s;
 :: switch2switch[in2]?s; switch2switch[out1]!s;
 od
}
```

In the initial state, we start the individual processes, which represent switches and hereby define the network of the monorail system.

Moreover, initially, we have that the metal cast of each product is already present on its carrier, the shuttle. The coloring of the taillights can be defined at the beginning or in the progress of the production. Last, but not least, we initialize the process by inserting shuttles on the starting rail (at Station 5).

```
init {
 atomic {
  byte i;
  c_code { cost = 0; }
  c_code { best_cost = infinity; }
  for (i : 0 .. (SHUTTLES)/2)){ color[i] = 1; }
  for (i : 0 .. (SHUTTLES-1)) { metalcast[i] = 1; }
  for (i : 0 .. (STATIONS-1)) {
    run Entrance(i); run Exit(i);
  }
  run Switch(7 ,0 ,5); run Switch(0 ,1 ,4);
  run Switch(1 ,2 ,3); run Switch(3 ,4 ,2);
  run Switch(4 ,5 ,1); run Switch(5 ,6 ,0);
  run Hub(2 ,3 ,8 ,9); run Hub(6 ,7 ,9 ,8);
  for (i : 0 .. (SHUTTLES-1)) {
    exit2switch[5]!i;
  }
 }
}
```

We also heavily made use of the term atomic, which enhances the exploration for the model checker, allowing it to merge states within the search. In difference to the more aggressive d step keyword, in an atomic block all communication queue actions are blocking, so that we chose to use an atomic block around each loop.

5 Optimized scheduling

Inspired by [6,37] and [45], we applied and improved branch-and-bound (BnB) for the optimization. Branching is the process of spawning subproblems, while bounding refers to ignoring partial solutions that cannot be better than the current best solution. To this end, lower and upper bounds are maintained as global control values on the solution quality, which improves over time.

For applying BnB to flow production systems, we extend depth-first search (DFS) with upper (and lower) bounds. In this context, branching corresponds to the generation of successors, so that DFS can be casted as generating a branch-and-bound search tree. One way of obtaining a lower bound L for the problem state u is to apply an admissible heuristic h with $L(u) = g(u) + h(u)$, where g denotes the cost for reaching the current node from the root, and h is admissible; it always underestimates the remaining cost to reach a goal.

As with standard DFS, the first solution obtained might not be optimal. With depth-first branch-and-bound (DFBnB), however, the solution quality improves over time together with the global value U until eventually the lower bound $L(u)$ at some node u is equal to U. In SPIN, the trivial heuristic $h = 0$ is used, but in HSF-Spin [16], a number of heuristic functions have been implemented. We obtain the following result.

For an admissible heuristic function h, the DFBnB procedure will eventually find the optimal solution to the flow production problem $F = (A, E, G, \prec, S, Q)$. It compute costs for partial runs and extend partial schedules incrementally. The objective function is monotonically increasing. Only inferior paths that cannot be extended to a better than the currently best one are pruned. As the state space is finite, the search will eventually terminate and return the optimal solution.

There are different options for finding optimized schedules with the help of a model checker that have been proposed in the literature.

First, in the Soldier (alias Hippie) model of [?], rendezvous communication to an additional synchronized process has been used to increase cost, dependent on the transition chosen, and together with a specialized LTL property to limit the total cost for the model checking solver. This approach, however, turned out to be too limited for our purpose.

An alternative proposal for branch-and-bound search is based on the support of native c-code in SPIN (introduced in version 4.0) [45]. One running example is the traveling salesman problem (TSP), but the approach is generally applicable to many other optimization problems. However, as implemented, there are certain limitations to the scalability of state space problem graphs. Recall that the problem graph induced by the TSP is in fact a tree, generating all possible permutations for the cities.

Following citev13,v6 and [45], we applied improved branch-and-bound optimization within SPIN. Essentially, the model checker can find traces of several hundreds of steps and provides trace optimization by finding a minimized counterexample (if ran with the parameter i). As these traces are step-optimized, and not cost-optimized, Ruys [45] proposed the introduction of a variable cost that we extend as follows.

```
c_state ``int min_cost'' ``Hidden''
c_state ``int min_cost'' ``Hidden''
c_code { int cost; }
c_code { int cost[SHUTTLES ]; }
c_track ``cost'' ``sizeof(int)'' ``Matched''
c_track ``cost'' STRING ``Matched''
```

While the cost variable increases the amount of memory required for each state, it also limits the power of SPINs built-in duplicate detection, as two otherwise identical states are considered different if reached by different accumulated cost. If the search space is small, so that it can be explored even for the enlarged state vector, then this option is sound and complete, and finally returns the optimal solution to the optimization problem. However, there might be simply too many repetitions in the model so that introducing cost to the state vector leads to a drastic increase in state space size, so that otherwise checkable instances now become intractable. We noticed that even by concentrating on safety properties (such as the failed assertion mentioned), the insertion of costs causes troubles.

5.1 Guarded branching

For our model, cost is tracked for every shuttle individually. The cost of the most expensive shuttle ride fixes the duration of the whole production process. Furthermore, the total cost value provides insight regarding unnecessary detours or long waiting times. Hence, minimizing both criteria are possible optimization goals of this model.

In Promela, every non-deterministic choice (in do-od blocks) is allowed to contain an unlimited number of possible options for the model checker to choose from. In theory, the model checker randomly chooses between these options, but at least in the deterministic enumeration options of the state spaces, the SPIN model checker obeys to the ordering of statements in non-deterministic choices. We used this feature extensively to order the statements that SPIN is able to find a valid trace on one of its first execution branches.

It is also possible to add a condition: if the first statement of a do-od block holds, SPIN will start to execute the following statements; otherwise, it will pick a different option.

Since the model checker explores any possible state of the system, many of these states are technically reachable but completely useless from an optimization point of view. In order to reduce state space size to a manageable level, we

add constraints to the relevant receiving options in the do-od block of every node process.

Peeking into the incoming queue to find out, which shuttle is waiting to be received is already considered a complete statement in Promela. Therefore, we exploit C-expressions to combine several operations into one atomic statement. For every station t and every incoming channel q, a function *prerequisites*(t, q) determines, if the first shuttle in q meets the prerequisites for t, as given by Fig. 3.

```
shuttle s;
do
:: c_expr{ prerequisites (Px ->q,Px ->t)} ->
           channel[q]?s; channel[out ]!;
od
```

At termination of a successful run, we now extend the proposal of [45]. We use the integer array cost of the Promela model. It enables each process to keep track of its local cost vector and is increased by the cost of each action as soon as the action is executed. This enables the model checker to print values to the output, only if the values of the current maximum and total cost values have improved.

```
terminate:
 c_code {
  int max = 0, total = 0, j;
  for (j=0; j<SHUTTLES; j++) {
   total += cost[j];
   if (cost[j] > max) max = cost[j]; }
   if (max < min_cost) {
     min_cost = max; putrail(); Nr_Trails --;
   };
 }
```

For solution reconstruction, we write a file for each new cost value obtained, temporarily renaming the trail file as follows.

```
char mytrail[512];
sprintf(mytrail,``%s_t%d_st%d.pr'',base,min_cost,total);
char* y = mytrailfile;
swap (& TrailFile, &y);
putrail ();
swap (&y, &TrailFile );
```

5.2 Process synchronization

Due to the nature of the state space search of the model checker, processes in the Promela model itself do not make decisions. Nonetheless, the given model is distributed consisting of a varying number of processes, which potentially influence each other if executed in parallel.

We address this problem by introducing event-based progression to the Promela model. Whenever a shuttle s travels along one of the edges, the corresponding message is put into a channel and the cost of the respective shuttle is increased by the cost of the given edge.

```
shuttle s;
do
:: c_expr{ receives(channel,Px ->q,Px ->station) }
  -> channel[q]?s
 c_code { cost[s] += Px ->c; }
 channel[out] ! s;
od
```

We introduce another atomic function *receives*(q) that flags true if the first item s of q has minimal cost, changing the receiving constraint to the following.

```
c_code {
int receives(int type,int arrayidx,int station) {
int idx = -1;
switch(type) {
 case xyz: idx = now.xyz[arrayidx]; break; [...]
}
if(idx > -1 && q_len(idx) > 0) {
 int shuttle = qrecv(idx , 0, 0, 0);
 int minimum = infinity;
 for (int j=0; j<SHUTTLES; j++) {
  if (cost[j] < minimum) minimum = cost[j]; }
  return (minimum == cost[shuttle ]); }
 return 0;
}
```

Within SPIN, the global Boolean variable timeout is automatically set to true when all current processes are unable to proceed, e.g., because they cannot receive a message. Consequently, for every shuttle p, all processes will be blocked and timeout will be set to true. As suggested by Bosnacki and Dams [5], we add a process that enforces time progress, whenever timeout occurs (final is a macro for reaching the goal).

```
active proctype watchdog () {
 do
 :: timeout -> c_code{ increase(); }; assert(!final);
 od
}
```

Time delay is enforced as follows: if the minimum event in the future event list is blocked (e.g., a shuttle is not first in its queue), we compute the wake-up time of the second best event. If the two are of the same time, a time increment of 1 is enforced. In the other case, the second best event time is taken as the new one for the first. It is easy to see that this strategy eventually resolves all possible deadlocks. Its implementation is as follows.

```
int increase () {
 int j, l = 0, minimum = cost[0];
 for (j=1; j<SHUTTLES; j++)
 if (cost[j] < minimum) { minimum = cost[j]; l = j; }
 int second = infinity;
 for (j=0; j<SHUTTLES; j++) {
  if (cost[j] < second && cost[j] > minimum)
   second = cost[j]; }
 cost[l] = (second == infinity) ? minimum+1 : second;
}
```

As a summary, the constraint bounded depth-first exploration has turned into the automated generation of the underlying state space of the event system, using c-code to

preserve the causality of actions and to simulate the future event list.

6 Monte Carlo search

In the encoding of the production problem as a single-player game, the player starts at an empty floor and chooses in each step an agent and a case (one for entrance and exit, three for switches) until the goal is achieved or a predefined length is exceeded. The smaller the makespan for each agent found by the algorithm, the higher the score of the play. More formally, the game is consisting of all queue content, shuttle locations, and their respective cost values. The start position has all shuttles and all queues being empty and moves are the set of allowed actions for each queue. The set of final positions consists of all states in which either all the individual goals or the maximal step sized is reached, and the score function adds a constant for each individual unreached goal, on top of the maximum of the individual cost values.

The components of the game induce a search tree in the natural way with the final positions as leaves. A play(out) is a path from the root to some leaf.

6.1 UCT

The randomized optimization scheme we consider refers to the wider class of Monte Carlo search (MCS) algorithms. The main concept of MCS is the random playout (or rollout) of a position, whose outcome, in turn, changes the likelihood of generating successors in subsequent trials. Prominent members in this class of reinforcement learning algorithms are upper confidence bounds applied to trees (UCT) [34]. In each iteration of UCT, there are four stages: selection (following a tree policy), expansion (adding a leaf node), simulation (performing a rollout), and backpropagation for the adaptation of values at nodes based on the observed outcome. Cumulating in the success of AlphaGo and AlphaZero in winning matches against professional human Go player [48], and top Chess and Shogi engines [49], the impact of MCS in playing games can no longer be doubted.

6.2 Nested Monte Carlo search

Nested Monte Carlo Search (NMCS) [9] is a randomized search method that has been successfully applied to solve many challenging combinatorial problems, including Klondike Solitaire, Morpion Solitaire, Same Game, just to name a few. A large fraction of TSP instances have been solved efficiently at or close to the optimum. NMCS compares well with other heuristic methods that include much more domain-specific information. NMCS is parameterized with the recursion level of the search (which denotes the depth of the recursion tree) and with the number of iterations (that denotes the branching of the recursion tree). At each leaf of the recursive search a rollout is executed, which performs and evaluates a random run.

What makes the NMCS variant Nested Rollout Policy Adaptation (NRPA) [14,44] notably different is the concept of learning a policy through an explicit mapping of moves to selection probabilities.

The production flow problem is implemented in a Monte Carlo search framework for single-agent game, so that the interface adapts the nomenclature of a game, Agents meet in arenas, they have some legal moves to execute, they enter, they play, they score and they terminate.

```
class Arena {
 public:
 Move rollout[MaxLength];
 int length;
 Arena () {
  length = 0;
  length = 0;
  for (int i = 0; i< STATIONS; i++) {
   switch2entrance[i]->clear();
   exit2switch[i]->clear();
   entrance2exit[i]->clear();
  }
  for (int i = 0; i< STATIONS + 2*HUBS; i++)
   switch2switch[i]->clear();
  for (int i = 0; i < SHUTTLES; i++) {
   wait[i] = 0; cost[i] = i * 70;
   goals[i] = 0; color[i] = i%2;
   metalcast[i] = 1; diffusor[i] = 0;
   electronics[i] = bulb[i] = seal[i] = 0;
   switch2entrance[5]->push(i);
  }
 }
 int legalMoves (Move moves[MaxLegalMoves]) {
  int m[3], mvs = 0;
  while (mvs == 0) {
   for (int p = 0; p < agent.size (); p++) {
    int k = agent[p]->nextLegalMove(m);
    for(int l=0;l<k;l++)
     moves[mvs++] = p*3 + m[l];
   }
   if (mvs == 0) increase_time();
  }
  return mvs;
 }
 void play (Move m) {
  rollout[length++] = m;
  agent[m/3]->executeMove(m%3);
 }
 bool terminal () {
  int reached = 1;
  for (int j=0; j<SHUTTLES; j++)
  reached &= goals[j];
  return (reached) || length == MaxLength -1;
 }
 double score () {
  int maximum = 0, total = 0;
  for (int j=0; j<SHUTTLES; j++)
  if (cost[j] > maximum) maximum = cost[j];
  int reached = 0;
  for (int j=0; j<SHUTTLES; j++)
   reached += !goals[j];
  return (reached * 1000) + maximum;
 }
}
```

A move (or play) corresponds to a movement of shuttles and is a combination of agent ID and applied case. Moves are stored in the rollout which is bound by a Boolean condition (terminal). The length of the rollout and score are recorded and the score, which itself is the result of a maximization of costs over all agents, is minimized. Finding the potential set of successors (legalMoves), finalizes the implementation. In NRPA, each rollout (calling the constructor) is initially empty.

The abstract class that is needed for each individual agent looks as follows.

```
class Agent {
 public:
 Agent () {}
 virtual void executeMove(int m) = 0;
 virtual int nextLegalMove (int* moves) = 0;
};
```

Next we provide one example for implementing an agent. Operating the switch leads to three different cases.

```
class Switch : public Agent {
 public:
 int In, Out, Station, B, C;
 Switch(int in , int out , int s, int b, int c):
 Agent (),In(in),Out(out),Station(s),B(b),C(c) {}
 void executeMove(int move) {
 if (move == 0) {
  int Shuttle = switch2switch[In]->pop();
  wait[Shuttle] += C; cost[Shuttle] += C;
  switch2entrance[Station]->push(Shuttle);
 }
 if (move == 1) {
  int Shuttle = switch2switch[In]->pop();
  wait[Shuttle] += B; cost[Shuttle] += B;
  switch2switch[Out]->push(Shuttle);
 }
 if (move == 2) {
  int Shuttle = exit2switch[Station]->pop();
  wait[Shuttle] += B; cost[Shuttle] += B;
  switch2switch[Out]->push(Shuttle);
 }
 }
 int nextLegalMove(int* moves) {
  int mvs = 0;
  if (receives(SW2SW_EN,In,Station))
  moves[mvs++] = 0;
  if (receives(SW2SW_PASS,In,Station))
  moves[mvs++] = 1;
  if (receives(EX2SW,Station,Station))
  moves[mvs++] = 2;
  return mvs;
 }
};
```

Last, but not least, we provide the implementation of incrementing virtual time,

```
void increase_time() {
 int min = INF , d = 1;
 for (int p = 0; p < SHUTTLES; p++)
  if (0 < wait[p] && wait[p] < min) min = wait[p];
 if (min < INF) d = min;
 for (int p = 0; p < SHUTTLES; p++)
  if (wait[p] - d >= 0) {
   wait[p] -= d; cost[p] += d;
 }
 else wait[p] = 0;
}
```

and the channel requirements to obey imposed constraints on a valid operation. Most queues are unbounded but some are restricted to size 1, namely the ones responsible for assembling an item.

```
bool receives(int type, int i,int station) {
 int result = 0;
 Channel* channel = NULL;
 switch(type) {
 case EN2EX: channel = entrance2exit[i]; break;
 case EX2SW: channel = exit2switch[i]; break;
 case SW2SW_PASS: channel = switch2switch[i]; break;
 case SW2SW_EN: channel = switch2switch[i]; break;
 case SW2EN:
 if (entrance2exit[station]->length () >= 1)
  channel = NULL;
  else channel = switch2entrance[station];
  break;
 }
 if (channel != NULL && channel ->length () > 0) {
  int shuttle = channel ->front ();
  if (wait[shuttle] <= 0)
   result = 1;
 }
 return result;
}
```

7 Experiments

We first compare the results of the exploration minimizing local virtual time (LVT) [15] to the ones simulating the discrete event system (DES) described in this paper. For comparison, we also present timing results of simulation runs of the original multiagent system implementation (MAS) on Z2 hardware [26]. We use MCS for Monte Carlo search with nested rollout policy adaptation.

Unlike the original software system, the Promela models do not rely on local decision making and induce a systematic global search for an optimal solution. Therefore, in difference to a distributed multiagent system that is analyzed, both models are studied and explored a centralized planning via model checking.

For executing the model checking, we chose version 6.4.3 of SPIN. As a compiler, we used gcc version 4.9.3, with the posix thread model. For the standard setting of trace optimization for safety checking (option SAFETY), we compiled the model as follows.

```
./spin -a z2.pr;
gcc -O2 -DREACH -DSAFETY -o pan pan.c;
./pan -i -m30000
```

Parameter i stands for the incremental optimization of the counterexample length. We regularly increased the maximal tail length with option m, as in some cases of our running

Table 1 Simulated production times for n products in the original MAS and SPIN simulation, including the amount of RAM required to compute the given result

n	MAS	LVT		DES	
	Time	Time	RAM	Time	RAM
2	4:01	3:24	987 MB	2:53	731 MB
3	4:06	3:34	2154 MB	3:04	503 MB
4	4:46	3:56	557 MB	3:13	519 MB
5	4:16	4:31	587 MB	3:25	541 MB
6	5:29	4:31	611 MB	3:34	565 MB
7	5:18	5:08	636 MB	3:45	587 MB
8	5:57	5:43	670 MB	3:55	610 MB
9	6:00	5:43	692 MB	4:06	635 MB
10	6:08	5:43	715 MB	4:15	557 MB
20	9:03	8:56	977 MB	5:59	857 MB

example, the traces turned out to be longer than the standard setting of at most 10,000 steps (Table 1). Option REACH is needed to warrant minimal counterexamples.[1]

For smaller problems, we experimented with SPINs parallel BFS, as it computes optimal-length counterexamples. The hash table is shared based on compare-and-swap (CAS). We also tried sbitstate hashing (supertrace) as a trade-off. Unfortunately, BFS interacts with c track, so we had to drop the experiments for cost optimization. Swarm search (./swarm -c3 -m16G -t1 -f) found many solutions, some of them being shorter than the ones offered by option i (indicating ordering effects), but due to the increased amount of randomness, for the optimized scheduling problem of the Z2 in general no better results that ordinary DFS were found. In each experimental run, a number of $n \in \{2, \ldots, 20\}$ shuttles carry products through the facility. All shuttles with even IDs acquire clear bulbs; all shuttles with odd IDs acquire colored ones.

A close look at the experiment results of every simulation run reveals that, given the same number of products to produce, all three approaches result in different sequences of events. LVT and DES propose the same sequence of production steps for the product of each shuttle 4,2,1,0,5, while the MAS approach proposes 2,1,4,0,5. All three simulation models keep track of the local production time of each shuttles product. In MAS and LVT simulation, minimizing maximum local production time is the optimization goal. Steady, synchronized progress of time is maintained centrally after every production step. Hence, whenever a shuttle has to wait in a queue, its total production time increases. For the DES model, progress of time is managed differently. Results show that max. production time in DES is lower than LVT and MAS production times in all cases.

[1] To run experiments, we used a common notebook with an Intel(R) Core(TM) i7- 4710HQ CPU at 2.50 GHz, 16 GB of RAM and Windows 10 (64 Bit).

Table 2 Efficiency of MCS for a rising number of shuttles

n	MCS			LVT	DES
	Length	Time	CPU	Time	Time
2	48	2:54	<1s	3:24	2:53
3	72	2:59	<1s	3:34	3:04
4	99	3:08	<2s	3:56	3:13
5	123	3:13	<2s	4:31	3:25
6	153	3:22	<5s	4:31	3:34
7	186	3:38	<5s	5:08	3:45
8	213	3:45	<5s	5:43	3:55
9	240	3:52	<5s	5:43	4:06
10	267	3:52	<5s	5:43	4:15
20	540	5:16	<5s	8:59	5:59

For every experiment, the amount of RAM required by DES to determine an optimal solution is slightly lower than the amount required by LVT as shown in Table 2. While the LVT required several iterations to find an optimal solution, the first valid solution found by DES was already the optimal solution in any conducted experiment. However, the LVT model is able to search the whole state space within the 16 GB RAM limit (given by our machine) for $n \leq 3$ shuttles, whereas the DES model is unable to search the whole state space for $n > 2$. For every experiment with $n > 3$ (LVT) or $n > 2$ (DES) shuttles, respectively, searching the state space for better results was canceled, when the 16 GB RAM limit was reached.

The experiments indicate that the DES is faster and more memory efficient than the LVT approach. In our experiment, if one element in a process queue was delayed, all the ones behind it were delayed as well. While DES and LVT are both sound in resolving deadlocks for the given case study, LVT has the more accurate representation for the progress of time.

Table 2 shows that the MCS implementation also scales well. It shows the length of the plan, the simulation time (Time), and the runtime for a growing number of vehicles. Different to the other models, we do not enforce prerequisites, namely that shuttles are protected from driving into a station if they have not all required components available. Given that SPIN is a full-fledged model checker that analyzes the encoding of the problem on the source-code level (resulting of traces that have thousands of steps), the result could have been expected, even though the search space is huge.

The runtime for MCS was less than 5 seconds, the RAM requirements rather small (less than 4MB for the largest instance). As with the DES/LVT model, in cost we measure travel time plus some initial waiting time. To help the solver to find valid solutions, we extended the objective fuction $(1000 \cdot reached) + maximum$ by the term

Table 3 Efficiency of MCS and LVT for a rising number of shuttles in dynamic context

n	MAS	MCS		LVT	
	Time	Time	CPU	Time	CPU
2	4:01	3:04	<1s	3:09	<1s
3	4:06	3:13	<1s	3:27	<1s
4	4:46	3:24	<2s	3:45	<1s
5	4:16	3:35	<2s	4:03	<1s
6	5:29	4:10	<5s	4:21	<1s
7	5:18	4:20	<5s	4:39	<1s
8	5:57	4:31	<5s	4:57	<1s
9	6:00	5:11	<5s	5:15	<1s
10	6:08	5:22	<5s	5:33	<1s
20	9:03	8:25	<5s	8:33	<1s

$(10e_r) + (10b_r) + (10s_r) + (100d_r)$, where e_r, b_r, s_r, and d_r are the violations to the assembling status of electronics, bulbs, seals, and diffusors, respectively. We observe that there is a noticeable difference in the simulation times of LVT and DES even for two shuttles. Hence, we decided to reimplement LVT and have the two cost functions in a close match.

Table 3 shows that (due to RAM usage) the time to find the first solution for the manually tuned model in SPIN is extremely short, and the MCS remained sufficiently fast. The resulting cost values of the schedules are about the same with small advantages to MCS. However, this is not a deficiency in SPIN: we blame the differences to a slight shift in the evaluation of the terminal state: SPIN invokes the watchdog, while MCS extracts the shuttles from the rails.

In Table 3, we also added the measurements of the simulated multiagent system, where the agents chose the color of the lamp dynamically based on fuzzy logic decision rules that take the incoming orders and observed current queue lengths into account. Even though the task is slightly different in the real world with dynamically acting agents, we can derive that both analytical methods show a potential for optimization in the dynamic schedule.

At this point, we emphasize that both the model checker and the Monte Carlo solver generate traces from top to bottom, so it is possible to include dynamics such as a different choice of bulb colors based on incoming orders and queue length into the state space search. On the other hand, input models for SPIN and MCS are finite; they do not feature optimization in infinite open loops.

8 Conclusion and discussion

In this work, we presented a novel approach for optimizing a modern industrial production line. The research is motivated by our interest in finding and comparing centralized and distributed solutions to the optimization problems in autonomous manufacturing.

The formal model reflects the routing of shuttles in the monorail multiagent system. Using model checking for optimizing multiagent and discrete event systems is a relatively new playground for formal method tools in form of a new analysis paradigm. Switches of the rail network were modeled as active processes, the edges between the switches as communication channels.

Our results clearly indicate a lot of room for improvement in the decentralized solution, since the model checker found more efficient ways to route the shuttles on several occasions. Furthermore, the model checker could derive optimized plans of several thousand steps.

With all the additional c-code, we have worked hard to make SPIN do what it originally is not supposed to do: real-time model checking and optimization of the makespan of traveling agents in an industrial setting. As a surplus, however, we arrive at a larger generality of systems that can be optimized. As with directed model checking, additional heuristics are expected to guide the search toward finding a good plan even faster. Even though discrete event simulation is a powerful method, by looking at the limits and possibilities of representing time, alternatives to discrete time model checking based on ticks have to be looked at.

Monte Carlo search performed even better than systematic exploration. The advanced NRPA algorithm also has a smaller memory footprint than exhaustive methods (assuming that they store states for eliminating duplicates). The main advantage to a systematic enumeration of the state space is that the random choices do not rely on a fixed traversal ordering and that for a good performance, we do not had to assist the model checker by ordering the non-deterministic choices manually.

The success in many other application domains illustrates the effectiveness and generality of this search option: compared to depth-first search and branch-and-bound, Monte Carlo search is less dependent on finding a very good ordering of the successors. Even hand-coded pruning like checking prerequisites was not needed to arrive at high-quality solutions in a short amount of time. Nonetheless, there is noticeable impact of hand-coded information, as additional guidance information encoded in the cost function increased the performance of the search substantially, preventing the planner from exploring large parts of the state space.

So far, we haven't extended SPIN by Monte Carlo search but gave an alternative implementation based on an existing single-player game optimization framework. Thus, on a first glance, a head-to-head comparison is seemingly unfair: SPIN is aimed at general software verification, and with a granularity on lines of the running Promela code, it does produce

much longer traces than the search framework, which is based on non-deterministic action choices only.

Moreover, SPIN already supports randomization in its swarm search wrapper, but swarm algorithms are based on random depth-first search, while Monte Carlo search with policy adaptation improves over time, and, thus, learns the structure of the underlying problem. Nestedness of the search leads to exponential refreshment of policies, and, therefore, offers a trade-off between exploitation and exploration.

The interface for randomized search is simple and flexible. We are confident that the implementation of a Model Checker based on Monte Carlo search is only a matter of time.

References

1. Abdeddaïm, Y., Maler, O.: Job-shop scheduling using timed automata. In: Proceedings of the 13th International Conference on Computer Aided Verification, CAV 2001, Paris, France, July 18–22, 2001, pp. 478–492 (2001)
2. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. STTT **11**(1), 69–83 (2009)
3. Bhat, U.N.: Finite capacity assembly-like queues. Queueing Syst. **1**(1), 85–101 (1986)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22–28, 1999, pp. 193–207 (1999)
5. Bosnacki, D., Dams, D.: Integrating real time into SPIN: A prototype implementation. In: Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE XI/PSTV XVIII'98, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), 3–6 November, 1998, Paris, France, pp. 423–438 (1998)
6. Brinksma, E., Mader, A., Fehnker, A.: Verification and optimization of a PLC control schedule. STTT **4**(1), 21–33 (2002)
7. Bürckert, H., Fischer, K., Vierke, G.: Holonic transport scheduling with teletruck. Appl. Artif. Intell. **14**(7), 697–725 (2000)
8. Burman, M.H.: New results in flow line analysis. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1995)
9. Cazenave, T.: Nested Monte-Carlo search. In: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11–17, 2009, pp. 456–461 (2009)
10. Cimatti, A., Giunchiglia, F., Giunchiglia, E., Traverso, P.: Planning via model checking: A decision procedure for AR. In: Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24–26, 1997, Proceedings, pp. 130–142 (1997)
11. Cimatti, A., Roveri, M., Traverso, P.: Automatic OBDD-based generation of universal plans in non-deterministic domains. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26–30, 1998, Madison, Wisconsin, USA, pp. 875–881 (1998)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2001)
13. Dorer, K., Calisti, M.: An adaptive solution to dynamic transport optimization. In: 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25–29, 2005, Utrecht, The Netherlands - Special Track for Industrial Applications, pp. 45–51 (2005)
14. Edelkamp, S., Cazenave, T.: Improved diversity in nested rollout policy adaptation. In: KI 2016: Advances in Artificial Intelligence - 39th Annual German Conference on AI, Klagenfurt, Austria, September 26–30, 2016, Proceedings, pp. 43–55 (2016)
15. Edelkamp, S., Greulich, C.: Branch-and-bound optimization of a multiagent system for flow production using model checking. In: Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016), Volume 1, Rome, Italy, February 24–26, 2016, pp. 27–37 (2016)
16. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT **5**(2–3), 247–267 (2004)
17. Edelkamp, S., Reffel, F.: OBDDs in heuristic search. In: KI-98: Advances in Artificial Intelligence, 22nd Annual German Conference on Artificial Intelligence, Bremen, Germany, September 15–17, 1998, Proceedings, pp. 81–92 (1998)
18. Edelkamp, S., Sulewski, D.: Flash-efficient LTL model checking with minimal counterexamples. In: Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10–14 November 2008, pp. 73–82 (2008)
19. Fischer, K., Müller, J.P., Pischel, M.: Cooperative transportation scheduling: an application domain for DAI. Appl. Artif. Intell. **10**(1), 1–34 (1996)
20. Fox, M., Long, D.: The detection and exploitation of symmetry in planning problems. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, pp. 956–961 (1999)
21. Ganji, F., Kluge, E.M., Scholz-Reiter, B.: Bringing agents into application: Intelligent products in autonomous logistics. In: Artificial intelligence and Logistics (AiLog)—Workshop at ECAI 2010, pp. 37–42 (2010)
22. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995, pp. 3–18 (1995)
23. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning—Theory and Practice. Elsevier, Amsterdam (2004)
24. Giunchiglia, F., Traverso, P.: Planning as model checking. In: Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, September 8-10, 1999, Proceedings, pp. 1–20 (1999)
25. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Computer Aided Verification, 2nd International Workshop, CAV'90, New Brunswick, NJ, USA, June 18–21, 1990, Proceedings, pp. 176–185 (1990)
26. Greulich, C., Edelkamp, S., Eicke, N.: Cyber-physical multiagent-simulation in production logistics. In: Multiagent System Technologies—13th German Conference, MATES 2015, Cottbus,

Germany, September 28–30, 2015, Revised Selected Papers, pp. 119–136 (2015)
27. Harrison, J.: Assembly-like queues. J. Appl. Probab. **10**, 354–367 (1973)
28. Helias, A., Guerrin, F., Steyer, J.P.: Using timed automata and model-checking to simulate material flow in agricultural production systems—application to animal waste management. Comput. Electron. Agric. **63**(2), 183–192 (2008)
29. Himoff, J., Rzevski, G., Skobelev, P.: Magenta technology multi-agent logistics i-scheduler for road transportation. In: 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8–12, 2006, pp. 1514–1521 (2006)
30. Hoffmann, J., Kissmann, P., Torralba, Á.: "distance"? who cares? tailoring merge-and-shrink heuristics to detect unsolvability. In: ECAI 2014—21st European Conference on Artificial Intelligence, 18–22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), pp. 441–446 (2014)
31. Holzmann, G.J.: The SPIN Model Checker—Primer and Reference Manual. Addison-Wesley, Reading (2004)
32. Hopp, W.J., Simon, J.T.: Bounds and heuristics for assembly-like queues. Queueing Syst. **4**(2), 137–155 (1989)
33. Jensen, R.M., Veloso, M.M., Bowling, M.H.: OBDD-based optimistic and strong cyclic adversarial planning. In: European Symposium on Planning (2001)
34. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings, pp. 282–293 (2006)
35. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30–April 1, 2006, Proceedings, pp. 35–52 (2006)
36. Lipper, E.H., Sengupta, B.: Assembly-like queues with finite capacity: bounds, asymptotics and approximations. Queueing Syst. **1**(1), 67–83 (1986)
37. Liu, W., Gu, Z., Xu, J., Wang, Y., Yuan, M.: An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In: Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2009, Grenoble, France, October 11–16, 2009, pp. 61–70 (2009)
38. Lluch-Lafuente, A.: Symmetry reduction and heuristic search for error detection in model checking. In: Model Checking and Artificial Intelligence (MoChArt), pp. 77–86 (2003)
39. Manitz, M.: Queueing-model based analysis of assembly lines with finite buffers and general service times. Comput. OR **35**(8), 2520–2536 (2008)
40. Morales-Kluge, E., Ganji, F., Scholz-Reiter, B.: Intelligent products—towards autonomous logistic processes—a work in progress paper. In: Intern. PLM Conf. (2010)
41. Nissim, R., Brafman, R.I.: Cost-optimal planning by self-interested agents. In: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14–18, 2013, Bellevue, Washington, USA (2013)
42. Parragh, S.N., Doerner, K.F., Hartl, R.F.: A survey on pickup and delivery problems. J. Betriebswirtschaft **58**(2), 81–117 (2008)
43. Rekersbrink, H., Ludwig, B., Scholz-Reiter, B.: Entscheidungen selbststeuernder logistischer Objekte. Ind. Manag. **23**(4), 25–30 (2007)
44. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo tree search. In: IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16–22, 2011, pp. 649–654 (2011)
45. Ruys, T.C.: Optimal scheduling using branch and bound with SPIN 4.0. In: Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9–10, 2003, Proceedings, pp. 1–17 (2003)
46. Ruys, T.C., Brinksma, E.: Experience with literate programming in the modelling and validation of systems. In: Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, pp. 393–408 (1998)
47. Saffidine, A.: Solving Games and All That. PhD thesis, University Paris–Dauphine (2014)
48. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T.P., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
49. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T.P., Simonyan, K., Hassabis, D.: Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. CoRR arXiv:1712.01815 (2017)
50. Valmari, A.: A stubborn attack on state explosion. Form. Methods Syst. Des. **1**(4), 297–322 (1992)
51. Wijs, A.: What to do Next? Analysing and Optimising System Behaviour in Time. PhD thesis, Vrije Universiteit Amsterdam (2007)
52. Wooldridge, M.J.: Reasoning about Rational Agents. The MIT Press, Cambridge (2000)
53. Wooldridge, M.J.: An Introduction to MultiAgent Systems, 2nd edn. Wiley, New York (2009)