

Two Model Checking Approaches to Branch-and-Bound Optimization of a Flow Production System

Christoph Greulich^(✉) and Stefan Edelkamp

Institute for Artificial Intelligence, University of Bremen, Bremen, Germany
{greulich,edelkamp}@cs.uni-bremen.de

Abstract. In this paper we introduce a novel application of model checking to find optimal planning solutions for a flow production system. Originally controlled by a multiagent system, the production system consists of autonomous products and asynchronous production stations with limited space for waiting products. In this work, we present two different approaches of application of the Spin model checker to optimize throughput in the given production system. Instead of mapping the multiagent system directly, we model the production line itself as a set of communicating processes. Each communication channel between two processes represents a one-way monorail connection from one station to another. Experiments show that both approaches derive valid and optimized plans with several thousands of steps using constrained branch-and-bound. However, experiments also indicate individual advantages of both approaches.

1 Introduction

The ongoing transformation of production industries causes a paradigm shift in manufacturing processes towards new technologies and innovative concepts, called *cyber*, *smart*, *digital* or *connected factory* [5]. 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.

Flow production systems are installed for products that are produced in high quantities. By optimizing the flow of production, manufacturers hope to speed up production at a lower cost, and in a more environmentally sound way. In manufacturing practice there are not only series flow lines (with stations arranged one behind the other), but also more complex networks of stations at which assembly operations are performed (assembly lines). 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 at the assembly stations. An assembly operation can begin only if all required parts are available.

Performance analysis of flow 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 consider *assembly-line networks with stations*, which are represented as a directed graph. Between any two successive nodes in the network, we assume a buffer of finite capacity. In the buffers between stations and other network elements, work pieces are stored, waiting for service. At assembly stations, service is given to work pieces. Travel time is measured and overall time is to be optimized.

Our running case study is the so called 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. The techniques developed, however, will be applicable to most flow production systems. We formalize the production floor as a system of communicating processes and apply the state-of-the-art model checker *Spin* [29] for analyzing 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 optimization of production flow.

For the *optimization via model checking* we use many new language features from the latest version of the Spin model checker including loops and native c-code verification. The main contribution of this text, however, is *general cost-optimization via branch-and-bound*. The optimization approach originally invented for Spin was designed for state space trees [43,44], while the proposed new approach also supports state space graphs, crucially reducing the running time and memory consumption of the algorithm, rendering otherwise intractable models to become analyzable.

The paper is structured as follows. First, we consider related work on agent-based industrial (flow) production, on model checking multiagent systems (MASs), and on planning via model checking. Next, we introduce the industrial case study, and its modeling as well as its simulation as an MAS. The simulator is used to measure the increments of the cost function to be optimized. Then, we turn to the intricacies of the Promela model specification and the parameterization of Spin, as well as to the novel branch-and-bound optimization scheme. Furthermore, we give a detailed overview over two different strategies to manage process synchronization and progression of time within the model. In the experiments we validate the conciseness and effectiveness of the model and the taken approaches.

2 Related Work

Especially in open, unpredictable, dynamic, and complex environments, MASs 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 [12, 27], as well as within other areas of logistics [7, 17]. A comprehensive survey is provided by [40].

Flow line analysis is often done with queuing theory [8, 36]. Pioneering work in analyzing assembly queuing systems with synchronization constraints analyzes assembly-like queues with unlimited buffer capacities [25]. 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 [2, 30, 34].

2.1 Model Checking Multiagent Systems

Model checking production flow is rare. Timed automata were used for simulating material flow in agricultural production [26]. There are, however, numerous attempts to apply model checking to validate the work of MASs.

The LORA framework [47, 48] 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 an MAS as a game, in which agents – either in separation or cooperatively – optimize their individual outcome [45]. Communication between the agents 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 MASs as planning problems [39].

2.2 Planning and Model Checking

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* [38], 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* [11]. Given a formal model of a system M and a property specification ϕ in some form of temporal logic like LTL [21], the task is to validate, whether or not the specification is satisfied in the model,

¹ <http://vas.doc.ic.ac.uk/software/mcmass/>.

$M \models \phi$. If not, a model checker usually returns a counterexample trace as a witness for the falsification of the property.

Planning and model checking have much in common [9, 22]. 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 short path from the initial state to one of the goal states. Nonetheless, there is rising interest in planners that prove insolvability [28], and in model checkers to produce minimal counterexamples [15].

In terms of leveraging state space search, over the last decades there has been much cross-fertilization between the fields. For example, based on Satplan [32] *bounded model checkers* exploit SAT and SMT representations [1, 3] of the system to be verified, while *directed model checkers* [13, 33] exploit panning heuristics to improve the exploration for falsification; partial-order reduction [23, 46] and symmetry detection [18, 35] limit the number of successor states, while symbolic planners [10, 14, 31] apply functional data structures like BDDs to represent sets of states succinctly.

3 Case Study: Z2

One of the few successful real-world implementations of a multiagent flow production is the so called Z2 production floor unit [20, 37]. The Z2 unit consists of six workstations where human workers assemble parts of automotive tail-lights. 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(a). 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,

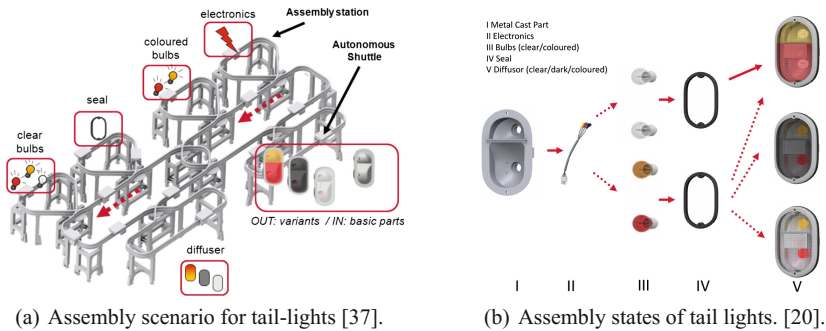


Fig. 1. Z2 Case study setup.

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 of the tail-lights as illustrated in Fig. 1(b). 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 tail-light is transported from station to station until it is assembled completely.

3.1 Multiagent System Simulation

In the real-world implementation of the Z2 system, every assembly station, every mono-rail 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 FIPA-compliant messaging components.

Most agents in this MAS resemble simple reflex agents as defined by Russell and Norvig [42]. 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 towards their individual goal of becoming a complete tail-light 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. 2.

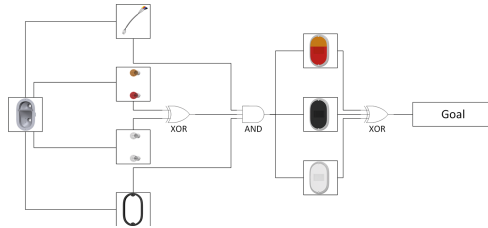


Fig. 2. Preconditions of the various manufacturing stages.

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. By inferring a MATLAB server, each agent individually makes its decisions by applying a Fuzzy Logic model [41].

The Z2 MAS was developed strictly for the purpose of controlling the Z2 monorail hardware setup. Nonetheless, due to its hardware abstraction layer [37], the Z2 MAS 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 MAS can be transferred to a virtual simulation environment [24]. Such an environment, which treats the original Z2 agents like black boxes, can easily be hosted by the JADE-based event-driven MAS simulation platform PlaSMA². Experiments show how close the executions of the simulated and the real-world scenarios match.

For this study, we provided the PlaSMA model with timers to measure the time taken between two graph nodes. Since the hardware 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. 3), where the weight of an edge denotes the traveling/processing time of the shuttle between two respective nodes.

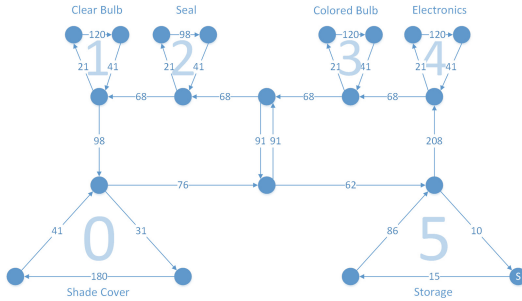


Fig. 3. Weighted graph model of the assembly scenario.

4 Formal Specification

Promela is the input language of the model checker Spin³, 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.

² <http://plasma.informatik.uni-bremen.de/>.

³ <http://spinroot.com/spin/whatispin.html>.

Channels in Promela follow the FIFO principle. Therefore, 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 MAS, 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.

Unlike the original MAS and the resembling PlaSMA simulation, the Promela model is designed to apply a branch-and-bound optimization to evaluate the optimal throughput of the original system. Instead of local decision making, the various node agents have certain nondeterministic 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 a final state. As mentioned before, decreasing production time for a given number of products increases the utility of the final state.

We derive a formal model of the Z2 multiagent systems as follows. First, we define global setting 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 Promela model, production nodes are realized as processes and edges between the nodes by the following channels.

```
chan entrance_to_exit[STATIONS]=[1] of {shuttle};
chan exit_to_switch[STATIONS]=[BUFFERSIZE] of {shuttle};
chan switch_to_switch[SWITCHES]=[BUFFERSIZE] of {shuttle};
chan switch_to_entrance[STATIONS]=[BUFFERSIZE] of {shuttle};
```

As global variables, we also have bit-vectors for the different assemblies being processed.

```
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
  :: exit_to_switch[station]?s; switch_to_switch[out]!s;
  :: switch_to_switch[in]?s; switch_to_switch[out]!s;
  :: switch_to_switch[in]?s; switch_to_entrance[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
  :: switch_to_entrance[station]?s;
     entrance_to_exit[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
  :: entrance_to_exit[station]?s;
     exit_to_switch[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
  :: switch_to_switch[in1]?s; switch_to_switch[out1]!s;
  :: switch_to_switch[in1]?s; switch_to_switch[out2]!s;
  :: switch_to_switch[in2]?s; switch_to_switch[out1]!s;
  od
}
```


In the initial state, we start the individual processes, which represent nodes 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 tail-lights 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 = 100000; }
    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)) { exit_to_switch[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 action are still blocking, so that we chose to use an atomic block around each loop.

5 Constrained Branch-and-Bound Optimization

There are different options for finding optimized schedules with the help of a model checker that have been proposed in the literature. First, as in the *Soldier* model of [44], rendezvous communication to an additional synchronized process has been used to increase cost, dependent on the transition chosen, together with a specialized LTL property to limit the total cost for the model checking solver. This approach, however, turned out to be limited in its ability. An alternative proposal for branch-and-bound search is based on the support of native c-code in Spin (introduced in version 4.0) [43]. 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.

Inspired by [6, 13] and [43] we applied and 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 the shortest path towards a counterexample if run with the parameter `./pan -i`. However, these traces are step-optimized, and not cost-optimized. Therefore, Ruys [43] proposed the introduction of a variable `cost`.

```

c_state "int best_cost" "Hidden"
c_code { int cost; }
c_track "cost" "sizeof(int)" "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, as with our model, it might be that there are 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 Optimization Goal

For our model, cost has to be tracked for every shuttle individually. The variable cost of the most expensive shuttle indicates the duration of the whole production process. Furthermore, the cost total provides insight regarding unnecessary detours or long waiting times. Hence, minimizing both criteria are the optimization goals of this model.

In Promela, every `do`-loop is allowed to contain an unlimited number of possible options for the model checker to choose from. The model checker randomly chooses between the options, however, it is possible to add an *if*-like condition to an option: If the first statement of a `do` option 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`-loops 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 (`c_expr`) 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. 2.

```

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

```

For branch-and-bound optimization, we now follow the guidelines of [43]. This enables the model checker to print values to the output, only if the values of the current `max` cost and `sum` cost have improved.

```

c_code {
  if (max < best_cost ||
      (max == best_cost && sum < best_sum_cost) {
    best_cost = max;
    best_sum_cost = sum;
    putrail();
    Nr_Trails--;
  }
};
}

```

6 Process Synchronization

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

In parallel simulation, different notions of time have to be considered. Physical time is the time of occurrence of real world events, simulation time (or virtual time) is the adaptation of physical time into the simulation model. Furthermore, wall clock time refers to the real-world time which passes during computation of the simulation.

Parallel execution allows faster processes to overtake slower processes, even though the LVT of the slower process is lower. While Spin maintains the order of products and their respective costs implicitly by the FIFO queues as long as the products are passed along in a row, the so called causality problem [19] emerges, as soon as products part ways at any switch node.

We addressed this problem by examining two different approaches of process synchronization in order to maintain simulation consistency. Both approaches ensure that a product p can only be removed from a queue q if it is its turn to move. Therefore, we introduce an atomic boolean function $canreceive(q)$ which only holds if the first element p in q is allowed to move. The function $canreceive(q)$ is added to the prerequisite check at every node entrance.

```

shuttle s;
do
  :: c_expr{canreceive(Px->q) &&
          prerequisites(Px->q, Px->t)} ->
     channel[q]?s;
     waittime[s]+=next_step_cost;
     channel[out]!s;

```

When no product p is allowed to make a move, all current processes are unable to proceed. Within Spin, a global Boolean variable `timeout` is defined, which is automatically set to `true` whenever this situation occurs. Following a suggestion by Bošnački and Dams [4], we add a process that computes time progress whenever `timeout` occurs. Unlike Bošnački and Dams, however, we examine two event-driven discrete time models. To further constrain branching, the time-managing process also asserts that the time does not exceed the `best_cost`, since worse results do not need to be explored completely.

```

active proctype timemanager() {
  do
    :: timeout -> c_code{ increasetime(); };
    assert(currenttime < best_cost);
  od
}

```

6.1 Discrete Event System

For the first approach, we created a discrete event system (DES) with event-based time progress [16]. Whenever a product p 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.

To maintain consistency in the DES, $\text{canreceive}(q)$ returns true for a product p only if no $p_i \neq p$ exists with $\text{cost}(p_i) < \text{cost}(p)$. Consequently, the first item p of q can only be moved if it has minimal $\text{cost}(p)$.

Time progress is enforced as follows: if the minimum event is blocked (e.g., because it 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. Algorithm 1.1 illustrates the procedure.

```

1: procedure INCREASETIME
2:    $first \leftarrow p_0 \in products$ 
3:    $min_a \leftarrow cost(p_0)$ 
4:    $min_b \leftarrow \infty$ 
5:   for all  $p \neq p_0 \in products$  do
6:     if  $cost(p) < min_a$  then
7:        $min_a \leftarrow cost(p)$ 
8:        $first \leftarrow p$ 
9:   for all  $p \in products$  do
10:    if  $cost(p) < min_b \wedge cost(p) > min_a$  then
11:       $min_b \leftarrow cost(p)$ 
12:   if  $min_b = \infty$  then
13:      $cost(first) \leftarrow min_a + 1$ 
14:   else
15:      $cost(first) \leftarrow min_b$ 

```

Algorithm 1.1. DES time progress.

6.2 Local Virtual Time

While the DES approach already maintains consistency within the simulation model, it only considers actual traveling costs per edge for each shuttle while costs for waiting in queues are not taken into account. In order to be able to include them into the total production cost, we introduce an integer array $\text{waittime}[\text{SHUTTLES}]$ to the Promela model. It enables each shuttle to keep

track of its local virtual time (LVT), as the wait time will be increased by the cost of each action as soon as the action is executed.

Again, we introduce a function *canreceive*(*q*), which returns true only if the first element *s* of *q* has *waittime*(*s*) ≤ 0. Furthermore, we apply an event-driven discrete time model as described in Algorithm 1.2. In this model, whenever a **timeout** occurs, the waiting time until the earliest event is determined and subtracted from waiting times of every product simultaneously.

```

1: procedure INCREASETIME
2:   minimum ← ∞
3:   delta ← 1
4:   for all p ∈ products do
5:     if 0 < waittime(p) < minimum then
6:       minimum ← waittime(p)
7:   if minimum < ∞ then
8:     delta ← minimum
9:   for all p ∈ products do
10:    if waittime(p) − delta ≥ 0 then
11:      waittime(p) ← waittime(p) − delta
12:    else
13:      waittime(p) ← 0

```

Algorithm 1.2. LVT time progress.

7 Evaluation

In this section, we present results of a series of experiments executing both synchronization models. For comparison, we also present results of simulation runs of the original MAS implementation [24].

Unlike the original system, the Promela models do not rely on local decision making but searches for an optimal solution systematically. Therefore, both Promela models resemble a centralized planning approach.

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 **-DSAFETY**), 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 example, the traces turned out to be longer than the standard setting of at most 10000 steps. Option **-DREACH** is needed to warrant minimal counterexamples at the end. 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).

7.1 Inflexible Product Variants

In each experiment run, a number of $n \in \{2 \dots 20\}$ 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. However, LVT and DES propose that the same sequence of production steps for each product. The example given in Table 1 shows that for all shuttles $0 \dots 2$ the scheduling sequence is exactly the same in LVT and DES, while the original MAS often proposes a different schedule. In the given example, both LVT and DES propose a sequence of 4, 2, 1, 0, 5 for shuttle 1. To the contrary, the MAS approach proposes 2, 1, 4, 0, 5 for shuttle 1. The same phenomenon can be observed for every $n \in \{2 \dots 20\}$ number of shuttles.

Table 1. Sequences of events for $n = 3$ products (*Product* \Rightarrow *Station*, where \Rightarrow indicates a finished production step).

MAS	0 \Rightarrow 4	1 \Rightarrow 2	0 \Rightarrow 3	2 \Rightarrow 1	0 \Rightarrow 2	1 \Rightarrow 4	0 \Rightarrow 0	2 \Rightarrow 4	0 \Rightarrow 5	1 \Rightarrow 1	2 \Rightarrow 2	1 \Rightarrow 0	2 \Rightarrow 0	1 \Rightarrow 5	2 \Rightarrow 5
LVT	0 \Rightarrow 4	1 \Rightarrow 4	2 \Rightarrow 4	0 \Rightarrow 3	2 \Rightarrow 3	1 \Rightarrow 2	1 \Rightarrow 1	2 \Rightarrow 2	1 \Rightarrow 0	0 \Rightarrow 2	2 \Rightarrow 0	0 \Rightarrow 0	1 \Rightarrow 5	2 \Rightarrow 5	0 \Rightarrow 5
DES	0 \Rightarrow 4	1 \Rightarrow 4	2 \Rightarrow 4	0 \Rightarrow 3	1 \Rightarrow 2	2 \Rightarrow 3	0 \Rightarrow 2	1 \Rightarrow 1	2 \Rightarrow 2	0 \Rightarrow 0	1 \Rightarrow 0	2 \Rightarrow 0	0 \Rightarrow 5	1 \Rightarrow 5	2 \Rightarrow 5

All three simulation models keep track of the local production time of each shuttle’s product. However, 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, as illustrated in Sect. 6.1. In the DES model, actual traveling costs per edge are summarized for each shuttle but costs for waiting in queues are not considered. Consequently, time in MAS and LVT includes idle time while time in DES does not. Therefore, results always show that max. production time in DES is lower than LVT and MAS production times in all cases.

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 every 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 $n > 3$. For every experiment with $n > 3$ (LVT) or $n > 2$ (DES) shuttles respectively, searching the state space for better results was cancelled, when the 16 GB RAM limit was reached.

In general, experiments indicate that the DES model is faster and more memory efficient than the LVT model even though both approaches propose

Table 2. Simulated production times for n products in the original MAS and Spin simulation, including the amount of RAM required to compute the given result. (* indicates that the whole state space was searched within the given RAM usage.)

Products	MAS	LVT		DES	
	Max. Prod. Time	Max. Prod. Time	RAM	Max. Prod. 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

the same optimal production schedules for each shuttle. Both models follow a different notion of time and, therefore, a slightly different optimization goal. By excluding idle time, the DES model focuses strictly on minimizing the time spent moving along edges and being processed at stations. The LVT model includes idle time, hence, it minimizes the total time spent in the production system.

7.2 Flexible Product Variants

In a second series of experiments, we allowed the model checker to decide, which products to provide with a colored or clear bulb. In these experiments, a desirable final state is reached when all products have returned to the storage station (station 5) and the difference d between the amount of both product variants is $0 \leq d \leq 1$.

In these experiments, the model checker has even more possibilities to branch its search space. Therefore, it is hardly surprising that problems with $n > 3$ shuttles could not be computed on our test machine. However, for $n = 2$ shuttles, the LVT model proposes a solution that takes 3:21s and therefore is 3s faster than the inflexible solution. For $n = 3$ shuttles, the difference is 10s, as the production takes 3:24s of simulation time.

8 Conclusions

In this paper, we introduced two different approaches to apply branch-and-bound optimization to a flow production system by employing model checking software. Our research is motivated by our interest in creating a benchmarking baseline for optimization of decentralized autonomous manufacturing.

Using model checking for optimizing DES is a relatively new playground for formal method tools in form of a new analysis paradigm. Our Promela model reflects the routing and scheduling of entities in a flow production system. We successfully adapted the monorail structure of our case study into a network of communicating channels which connect a number of concurrent processes. Additional constraints to the order of production steps enable to carry out a complex planning and scheduling task.

We introduced two different synchronization strategies. A close look at the limits and possibilities of LVT and DES revealed that both approaches have certain advantages and disadvantages.

In future work, we will consider applying an action planner or a general game player for comparison, even though we do not expect a drastic improvement in state space size. Also, we will use the baseline established in this paper as a reference to improve the decentralized planning for the original MAS implementation.

Acknowledgements. This research was partly funded by the International Graduate School for Dynamics in Logistics (IGS), University of Bremen, Germany.

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006). doi:[10.1007/11691617_9](https://doi.org/10.1007/11691617_9)
2. Bhat, U.: Finite capacity assembly-like queues. *Queueing Syst.* **1**, 85–101 (1986)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). doi:[10.1007/3-540-49059-0_14](https://doi.org/10.1007/3-540-49059-0_14)
4. Bošnački, D., Dams, D.: Integrating real time into spin: a prototype implementation. In: Budkowski, S., Cavalli, A., Najm, E. (eds.) FORTE/PSTV, vol. 6, pp. 423–438. Springer, New York (1998)
5. Bracht, U., Geckler, D., Wenzel, S.: *Digitale Fabrik: Methoden und Praxisbeispiele*. Springer, Heidelberg (2011)
6. Brinksma, E., Mader, A.: Verification and optimization of a PLC control schedule. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 73–92. Springer, Heidelberg (2000). doi:[10.1007/10722468_5](https://doi.org/10.1007/10722468_5)
7. Bürckert, H.J., Fischer, K., Vierke, G.: Holonic transport scheduling with teletruck. *Appl. Artif. Intell.* **14**(7), 697–725 (2000)
8. Burman, M.: New results in flow line analysis. Ph.D. thesis, Massachusetts Institute of Technology (1995)
9. Cimatti, A., Giunchiglia, E., Giunchiglia, F., Traverso, P.: Planning via model checking: a decision procedure for AR. In: Steel, S., Alami, R. (eds.) ECP 1997. LNCS, vol. 1348, pp. 130–142. Springer, Heidelberg (1997). doi:[10.1007/3-540-63912-8_81](https://doi.org/10.1007/3-540-63912-8_81)
10. Cimatti, A., Roveri, M., Traverso, P.: Automatic OBDD-based generation of universal plans in non-deterministic domains. In: AAAI, pp. 875–881 (1998)
11. Clarke, E., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (2000)

12. Dorer, K., Calisti, M.: An adaptive solution to dynamic transport optimization. In: AAMAS, pp. 45–51. ACM (2005)
13. Edelkamp, S., Lafuente, A.L., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Dwyer, M. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001). doi:[10.1007/3-540-45139-0_5](https://doi.org/10.1007/3-540-45139-0_5)
14. Edelkamp, S., Reffel, F.: OBDDs in heuristic search. In: Herzog, O., Günter, A. (eds.) KI 1998. LNCS, vol. 1504, pp. 81–92. Springer, Heidelberg (1998). doi:[10.1007/BFb0095430](https://doi.org/10.1007/BFb0095430)
15. Edelkamp, S., Sulewski, D.: Flash-efficient LTL model checking with minimal counterexamples. In: SEFM, pp. 73–82 (2008)
16. Edelkamp, S., Greulich, C.: Using SPIN for the optimized scheduling of discrete event systems in manufacturing. In: Bošnački, D., Wijs, A. (eds.) SPIN 2016. LNCS, vol. 9641, pp. 57–77. Springer, Heidelberg (2016). doi:[10.1007/978-3-319-32582-8_4](https://doi.org/10.1007/978-3-319-32582-8_4)
17. Fischer, K., Müller, J.R.P., Pischel, M.: Cooperative transportation scheduling: an application domain for DAI. *Appl. Artif. Intell.* **10**(1), 1–34 (1996)
18. Fox, M., Long, D.: The detection and exploration of symmetry in planning problems. In: IJCAI, pp. 956–961 (1999)
19. Fujimoto, R.: *Parallel and Distributed Simulation Systems*. Wiley, Hoboken (2000)
20. Ganji, F., Morales Kluge, E., Scholz-Reiter, B.: Bringing agents into application: intelligent products in autonomous logistics. In: Schill, K., Scholz-Reiter, B., Frommberger, L. (eds.) *Artificial Intelligence and Logistics (AiLog) - Workshop at ECAI 2010*, pp. 37–42 (2010)
21. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV, pp. 3–18. Chapman & Hall (1995)
22. Giunchiglia, F., Traverso, P.: Planning as model checking. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS (LNAI), vol. 1809, pp. 1–20. Springer, Heidelberg (2000). doi:[10.1007/10720246_1](https://doi.org/10.1007/10720246_1)
23. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991). doi:[10.1007/BFb0023731](https://doi.org/10.1007/BFb0023731)
24. Greulich, C., Edelkamp, S., Eicke, N.: Cyber-physical multiagent-simulation in production logistics. In: Müller, J.P., Ketter, W., Kaminka, G., Wagner, G., Bulling, N. (eds.) MATES 2015. LNCS (LNAI), vol. 9433, pp. 119–136. Springer, Heidelberg (2015). doi:[10.1007/978-3-319-27343-3_7](https://doi.org/10.1007/978-3-319-27343-3_7)
25. Harrison, J.: Assembly-like queues. *J. Appl. Probab.* **10**, 354–367 (1973)
26. 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)
27. Himoff, J., Rzevski, G., Skobelev, P.: Magenta technology multi-agent logistics i-scheduler for road transportation. In: AAMAS, pp. 1514–1521. ACM (2006)
28. Hoffmann, J., Kissmann, P., Torralba, Á.: “Distance”? Who cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In: ECAI, pp. 441–446 (2014)
29. Holzmann, G.J.: *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, Boston (2004)
30. Hopp, W., Simon, J.: Bounds and heuristics for assembly-like queues. *Queueing Syst.* **4**, 137–156 (1989)
31. Jensen, R.M., Veloso, M.M., Bowling, M.H.: OBDD-based optimistic and strong cyclic adversarial planning. In: ECP (2001)
32. Kautz, H., Selman, B.: Pushing the envelope: planning propositional logic, and stochastic search. In: ECAI, pp. 1194–1201 (1996)

33. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006). doi:[10.1007/11691617_3](https://doi.org/10.1007/11691617_3)
34. Lipper, E., Sengupta, E.: Assembly-like queues with finite capacity: bounds, asymptotics and approximations. *Queueing Syst.* **1**, 67–83 (1986)
35. Lluch-Lafuente, A.: Symmetry reduction and heuristic search for error detection in model checking. In: MOCHART, pp. 77–86 (2003)
36. Manitz, M.: Queueing-model based analysis of assembly lines with finite buffers and general service times. *Comput. Oper. Res.* **35**(8), 2520–2536 (2008)
37. Morales Kluge, E., Ganji, F., Scholz-Reiter, B.: Intelligent products - towards autonomous logistic processes - a work in progress paper. In: PLM, Bremen, pp. 348–357 (2010)
38. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
39. Nissim, R., Brafman, R.I.: Cost-optimal planning by self-interested agents. In: AAAI (2013)
40. Parragh, S.N., Doerner, K.F., Hartl, R.F.: A survey on pickup and delivery problems Part II: transportation between pickup and delivery locations. *J. für Betriebswirtschaft* **58**(2), 81–117 (2008)
41. Rekersbrink, H., Ludwig, B., Scholz-Reiter, B.: Entscheidungen selbststeuernder logistischer Objekte. *Ind. Manag.* **23**(4), 25–30 (2007)
42. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach, 3rd edn. Pearson Education, Upper Saddle River (2010)
43. Ruys, T.C.: Optimal scheduling using branch and bound with SPIN 4.0. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 1–17. Springer, Heidelberg (2003). doi:[10.1007/3-540-44829-2_1](https://doi.org/10.1007/3-540-44829-2_1)
44. Ruys, T.C., Brinksma, E.: Experience with literate programming in the modelling and validation of systems. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 393–408. Springer, Heidelberg (1998). doi:[10.1007/BFb0054185](https://doi.org/10.1007/BFb0054185)
45. Saffidine, A.: Solving games and all that. Ph.D. thesis, University Paris-Dauphine (2014)
46. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991). doi:[10.1007/BFb0023729](https://doi.org/10.1007/BFb0023729)
47. Wooldridge, M.: Reasoning About Rational Agents. The MIT Press, Cambridge (2000)
48. Wooldridge, M.: An Introduction to Multi-agent Systems. Wiley, Chichester (2002)