Branch-and-Bound Optimization of a Multiagent System for Flow Production using Model Checking

Stefan Edelkamp and Christoph Greulich
University of Bremen, Institute for Artificial Intelligence, Am Fallturm 1, 28359 Bremen, Germany

Keywords: Multiagent Systems, Model Checking, Optimization, Branch-and-Bound, Autonomous Production.

Abstract: In this paper we propose the application of a model checker to evaluate a multiagent system that controls the industrial production of autonomous products. As the flow of material is asynchronous at each station, queuing effects arise as long as buffers provide waiting room. Besides validating the design of the system, the core objective of this work is to find plans that optimize the throughput of the system. Instead of mapping the multiagent system directly to the model checker, we model the production line as a set of communicating processes, with the movement of items modeled as communication channels. Experiments shows that the model checker is able to analyze the movements of autonomous products for the model, subject to the partial ordering of the product parts. It derives valid and optimized plans with several thousands of steps using constraint branch-and-bound.

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* (Bracht et al., 2011). The sector is entering its fourth revolution, characterized by a merging of computer networks and factory machines. At each link in the production and supply chains, tools and workstations communicate constantly via the 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, an optimization problem arises. 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 com-

Edelkamp, S. and Greulich, C.
Branch-and-Bound Optimization of a Multiagent System for Flow Production using Model Checking.
DOI: 10.5220/0005705100270037
In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016) - Volume 1*, pages 27-37
ISBN: 978-989-758-172-4

municating processes and apply the state-of-the-art model checker *Spin* (Holzmann, 2004) 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 (Ruys and Brinksma, 1998; Ruys, 2003), 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. In the experiments we validate the conciseness and effectiveness of the model and the taken approach.

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 (Dorer and Calisti, 2005; Himoff et al., 2006), as well as within other areas of logistics (Fischer et al., 1996; Bürckert et al., 2000). A comprehensive survey is provided by (Parragh et al., 2008).

Flow line analysis is often done with queuing theory (Manitz, 2008; Burman, 1995). Pioneering work in analyzing assembly queuing systems with synchronization constraints analyzes assembly-like queues with unlimited buffer capacities (Harrison, 1973). 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 (Bhat, 1986; Lipper and Sengupta, 1986; Hopp and Simon, 1989).

2.1 Model Checking Multiagent Systems

Model checking production flow is rare. Timed automata were used for simulating material flow in agricultural production (Helias et al., 2008). There are, however, numerous attempts to apply model checking to validate the work of MASs.

The LORA framework (Wooldridge, 2000; Wooldridge, 2002) 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 (Saffidine, 2014). 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 Lomuscio[1]. Recently, there has been some approaches to formalize MASs as planning problems (Nissim and Brafman, 2013).

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* (Nau et al., 2004), 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* (Clarke et al., 2000). Given a formal model of a system M and a property specification ϕ in some form of temporal logic like LTL (Gerth et al., 1995), the task is to validate, whether or not the specification is satisfied in the model, $M \models \phi$. 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 (Giunchiglia and Traverso, 1999; Cimatti et al., 1997). Both rely on the exploration of a potentially large state space of system states. Usually, model

[1] http://vas.doc.ic.ac.uk/software/mcmas/

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 (Hoffmann et al., 2014), and in model checkers to produce minimal counterexamples (Edelkamp and Sulewski, 2008).

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 (Kautz and Selman, 1996) *bounded model checkers* exploit SAT and SMT representations (Biere et al., 1999; Armando et al., 2006) of the system to be verified, while *directed model checkers* (Edelkamp et al., 2001; Kupferschmid et al., 2006) exploit panning heuristics to improve the exploration for falsification; partial-order reduction (Valmari, 1991; Godefroid, 1991) and symmetry detection (Fox and Long, 1999; Lluch-Lafuente, 2003) limit the number of successor states, while symbolic planners (Cimatti et al., 1998; Jensen et al., 2001; Edelkamp and Reffel, 1998) 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 (Ganji et al., 2010; Morales Kluge et al., 2010). 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 Figure 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

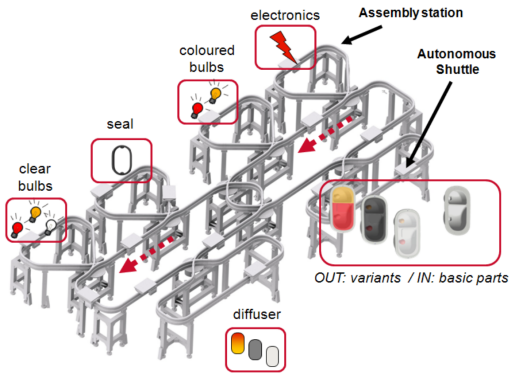

Figure 1: Assembly scenario for tail-lights (Morales Kluge et al., 2010).

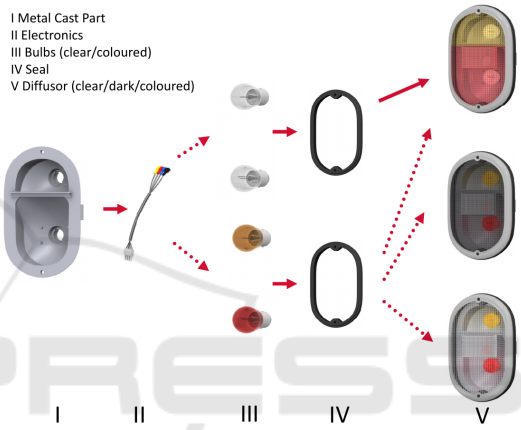

Figure 2: Assembly states of tail lights.(Ganji et al., 2010).

used to assemble different variants of the tail-lights 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 tail-light 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 floor *is a 6-tuple* $F = (A, E, G, \prec, S, Q)$ *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* $w : E \to \mathbb{R}_{\geq 0}$
- $\prec = (\prec_1, \ldots, \prec_n)$ *is a vector of assembling plans with each* $\prec_i \subseteq A \times A$, $i \in \{1, \ldots, n\}$, *being a par-*

tial order

- *$S \subseteq E$ is the set of assembling stations induced by a labeling $\rho : E \to A \cup \emptyset$, i.e., $S = \{e \in E \mid \rho(e) \neq \emptyset\}$*
- *Q is a set of (FIFO) queues of finite size $|Q| < \infty$ together with a labeling $\psi : E \to Q$*

Products P_i, $i \in \{1,\ldots,n\}$, travel through the network G, meeting their assembling plans/order $\prec_i \subseteq A \times A$ of the assembling actions A. For defining the cost function we use the set of predecessor edges $Pred(e) = \{e' = (u,v) \in E \mid e = (v,w)\}$.

Definition 2 (Run, Plan, and Path)**.** *Let $F = (A,E,G,\prec,S,Q)$ be a flow production floor. A* run π *is a schedule 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 set of all runs is denoted as Π. The run partitions into a set of n* plans *$\pi_i = (e_1,t_1,l_1),\ldots,(e_m,t_i,l_m)$, 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 G.*

3.1 Multiagent System Simulation

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 FIPA-compliant messaging components.

Most agents in this MAS resemble simple reflex agents as defined by Russell and Norvig (2010) . 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 figure 3.

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

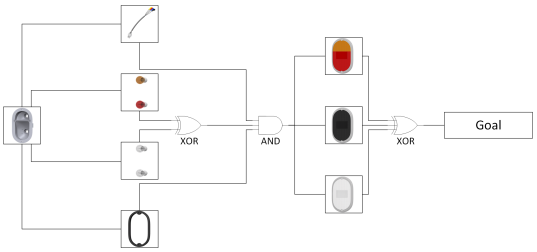

Figure 3: Preconditions of the various manufacturing stages.

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 (Rekersbrink et al., 2007).

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

Definition 3 (Product Objective, Travel and Waiting Time)**.** *The* objective *for product i is to minimize*

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

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

- $\text{time}(\pi_i)$ *is the* travel time *of product P_i, defined as the sum of edge costs* $\text{time}(\pi_i) = \sum_{e \in \pi_i} w(e)$*, and*
- $\text{wait}(\pi_i)$ *the* waiting time*, defined as* $\text{wait}(\pi_i) = \sum_{(e,t,l),(e',t',l') \in \pi_i, e' \in Pred(e)} t - (t' + w(e'))$.

The Z2 MAS was developed strictly for the purpose of controlling the Z2 monorail hardware setup. Nonetheless, due to its hardware abstraction layer (Morales Kluge et al., 2010), 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 (Greulich et al., 2015). 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 PlaSMA[2]. 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:

[2]http://plasma.informatik.uni-bremen.de/

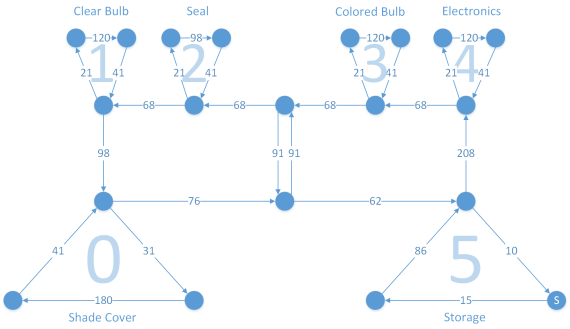

Figure 4: Weighted graph model of the assembly scenario.

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.

4 FORMAL SPECIFICATION

Promela is the input language of the model checker Spin[3], 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. 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

[3]http://spinroot.com/spin/whatispin.html

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 (Ruys and Brinksma, 1998), 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) (Ruys, 2003). 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 (Edelkamp et al., 2001; Brinksma and Mader, 2000) and (Ruys, 2003) 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 (2003) 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 Constrained Branching

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. Again, a more general formalization can be derived from our case study as follows.

Definition 4 (Overall Objective)**.** *With* $\text{cost}(\pi_i) = \text{wait}(\pi_i) + \text{time}(\pi_i)$*, as* overall objective function *we have* $\min_{\pi\in\Pi} \max_{1\le i\le n} cost(\pi_i)$

$$\begin{aligned} &= \min_{\pi\in\Pi} \max_{1\le i\le n} \textstyle\sum_{e\in\pi_i} w(e) \\ &\quad + \textstyle\sum_{(e,t,l),(e',t',l')\in\pi_i, e'\in Pred(e)} t - (t' + w(e')) \\ &= \min_{\pi\in\Pi} \max_{1\le i\le n,(e,t,l)\in\pi_i} t + w(e) \end{aligned}$$

subject to the side constraints that

- *time stamps on all runs* $\pi_i = (e_1,t_1,l_1)\dots(e_m,t_m,l_m)$*,* $i \in \{1,\dots,n\}$ *are monotonically increasing, i.e.,* $t_l \le t_k$ *for all* $1 \le l < k \le m$*.*
- *after assembling all products are complete, i.e., all assembling actions have been executed, so that for all* $i \in \{1,\dots,n\}$ *we have* $P_i = \cup_{(e_j,t_j,l_j)\in\pi_i}\{\rho(e_j)\}$
- *the order of assembling product* P_i *on path* $\pi_i = (e_1,t_1,l_1)\dots(e_m,t_m,l_m)$*,* $i \in \{1,\dots,n\}$*, is preserved, i.e., for all* $(a,a') \in \prec_i$ *and* $a = \rho(e_j), a' = \rho(e_k)$ *we have* $j < k$*,*
- *all insertions to queues respect their sizes, i.e., for all* $\pi_i = (e_1,t_1,l_1)\dots(e_m,t_m,l_m)$*,* $i \in \{1,\dots,n\}$*, we have that* $0 \le l_j < |\psi(e_j)|$*.*

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 Figure 3.

```
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 (Ruys, 2003). 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--;
  };
}
```

5.2 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.

Consequently, we introduce an integer array `waittime[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. However, 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 (Fujimoto, 2000) emerges, as soon as products part ways at any switch node.

We addressed this problem by introducing an event-based time progress to the Promela model. Whenever a shuttle *s* travels along one of the edges, the corresponding message is put into a channel and the waiting time *waittime(s)* of the respective shuttle is increased by the cost of the given edge. The receiving process is not allowed to take the message out of the channel, until the waiting time of the shuttle has passed.

Again, we introduce an atomic C function *canreceive*(q), which returns true only if the first element *s* of *q* has *waittime(s)* ≤ 0, changing the receiving constraint to the following.

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

Within Spin, a global Boolean variable `timeout` is defined, which is automatically set to *true* when all current processes are unable to proceed, e.g. because they cannot receive a message. Consequently, when *waittime(p)* > 0 for every shuttle *p*, all processes will be blocked and `timeout` will be set to *true*. As suggested by Bošnački and Dams (1998) , we add a process that computes time progress whenever `timeout` occurs. Unlike Bošnački and Dams, however, we apply an event-driven discrete time model as described in Algorithm 1. 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
}
```

Algorithm 1: Increase simulation time.

```
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
```

6 EVALUATION

In this section, we present results of a series of experiments executing the Promela model. We compare the results with the outcomes of the JADE-based simulation of the original hardware implementation. Unlike the original MAS, the Promela model does not rely on local decision making but searches for an optimal solution systematically. Therefore, the Promela model resembles a centralized planning approach. Consequently, we compare the centralized solution with the original distributed MAS solution. The comparison should be dealt with care: while a simulation executes one run in a complex system, model checking explores all possible runs in a simplified system.

For executing the model checking, we chose version 6.4.3 of Spin. 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.

We used two different machines for the experiments: First, a common notebook with an Intel(R)

Table 1: Simulated production times for n products in PlaSMA and Spin simulation, including the amount of RAM required to compute the given result. (* indicates that the whole state space was searched.)

	PlaSMA	Spin (Inflexible)	
Products	Sim. Time	Sim. Time	RAM
2	4:01	3:24	987 MB*
3	4:06	3:34	2154 MB*
4	4:46	3:56	557 MB
5	4:16	4:31	587 MB
6	5:29	4:31	611 MB
7	5:18	5:08	636 MB
8	5:57	5:43	670 MB
9	6:00	5:43	692 MB
10	6:08	5:43	715 MB
20	9:03	8:56	977 MB

Core(TM) i7-4710HQ CPU at 2.50 GHz, 16 GB of RAM and Windows 10 (64 Bit). Second, a server with an Intel(R) Xeon(R) CPU E5-4627 v2 at 3.30 GHz 128 GB of RAM and Windows Server 2012 R2 (64 Bit).

6.1 Inflexible Product Variants

For the first series of experiments, we predefined production goals for each product: Products with even IDs acquire clear bulbs, products with odd IDs acquire colored ones.

Table 1 shows that in most cases the Spin model checker proposes an optimal solution that is up to one minute faster than the original MAS. While a certain deviation between both simulations is unavoidable, since the non-deterministic communication processes between shuttles, products and stations are not considered in the Promela model, results clearly indicate that the agents' decision making leaves a lot of room for improvement, especially, under consideration of the agents' flexibility: The autonomous products in the original MAS are able to decide, which product variant they want to become to counter waiting times at stations.

However, the Spin model checker also reveals considerable limitations. While the state space of experiments with $n \in \{2,3\}$ products can be searched completely even on a standard notebook, experiments with $n > 3$ shuttles easily exhaust 128 GB of RAM on our server without ever completing the search of the whole state space. Luckily, potentially good results can be found early on: Table 1 shows that even in experiments that exhausted 128 GB of RAM, the best results were found before the search space filled 2 GB of RAM. However, a valid solution for $n > 30$ shuttles could not be computed by the model checker before the RAM on our server was exhausted.

Table 2: Sequences of events for $n = 2$ products. ($Product \Rightarrow Station$, where $\Rightarrow$ indicates a finished production step.)

PlaSMA	Spin (Infl.)	Spin (flex.)
$0 \Rightarrow 4$	$0 \Rightarrow 4$	$0 \Rightarrow 4$
$1 \Rightarrow 2$	$1 \Rightarrow 4$	$1 \Rightarrow 4$
$0 \Rightarrow 3$	$2 \Rightarrow 4$	$2 \Rightarrow 4$
$2 \Rightarrow 1$	$0 \Rightarrow 3$	$0 \Rightarrow 3$
$0 \Rightarrow 2$	$2 \Rightarrow 3$	$1 \Rightarrow 3$
$1 \Rightarrow 4$	$1 \Rightarrow 2$	$0 \Rightarrow 2$
$0 \Rightarrow 0$	$1 \Rightarrow 1$	$2 \Rightarrow 2$
$2 \Rightarrow 4$	$2 \Rightarrow 2$	$1 \Rightarrow 2$
$0 \Rightarrow 5$	$1 \Rightarrow 0$	$0 \Rightarrow 0$
$1 \Rightarrow 1$	$0 \Rightarrow 2$	$2 \Rightarrow 1$
$2 \Rightarrow 2$	$2 \Rightarrow 0$	$1 \Rightarrow 0$
$1 \Rightarrow 0$	$0 \Rightarrow 0$	$2 \Rightarrow 0$
$2 \Rightarrow 0$	$1 \Rightarrow 5$	$1 \Rightarrow 5$
$1 \Rightarrow 5$	$2 \Rightarrow 5$	$0 \Rightarrow 5$
$2 \Rightarrow 5$	$0 \Rightarrow 5$	$2 \Rightarrow 5$

Regarding computation time, experiments in both Spin and the PlaSMA system provided results within few minutes. It is mentionable though, that Spin provides the above results in shorter computation time than the corresponding PlaSMA simulation. However, exhausting the servers RAM takes about 20 minutes and slightly exceeds PlaSMAs computation time.

6.2 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 either of our test machines. For $n = 2$ shuttles, the model checker proposes a solution that takes 3:21 seconds and therefore is 3 seconds faster than the inflexible solution. For $n = 3$ shuttles, the difference is 10 seconds, as the production takes 3:24 seconds of simulation time.

A closer look at the sequence of events reveals, that a flexible choice of product variants allows products to overtake stations more efficiently, as illustrated in Table 2.

7 CONCLUSIONS

In this paper, we presented a novel approach for model checking an industrial production line. The research was motivated by our interest in finding and comparing centralized and distributed solutions to the optimization problems in autonomous production systems.

The formal model reflects the routing and scheduling of shuttles in the multiagent system. Nodes of the rail network were modeled as processes, the edges between the nodes were modeled as communication channels. Additional constraints to the order of production steps enable to carry out a complex planning task.

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

In future work, we will consider a larger parameter space for the model checker. We are also thinking of applying an action planner or a general game player for comparison. We do not expect a drastic improvement in state space size, as the model languages (PDDL (Hoffmann and Edelkamp, 2005) and GDL (Love et al., 2006)) are considerably different and do not have native support for communication queues. However as in directed model checking (Edelkamp et al., 2001), the integration of informative heuristics might help to guide the search process towards finding the goal.

ACKNOWLEDGEMENTS

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

REFERENCES

Armando, A., Mantovani, J., and Platania, L. (2006). Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN*, pages 146–162. Springer.

Bhat, U. (1986). Finite capacity assembly-like queues. *Queueing Systems*, 1:85–101.

Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*.

Bošnački, D. and Dams, D. (1998). Integrating Real Time Into Spin: A Prototype Implementation. In Budkowski, S., Cavalli, A., and Najm, E., editors, *FORTE/PSTV*, volume 6 of *IFIP The International Federation for Information Processing*, pages 423–438. Springer.

Bracht, U., Geckler, D., and Wenzel, S. (2011). *Digitale Fabrik: Methoden und Praxisbeispiele*. Springer, Berlin/Heidelberg.

Brinksma, E. and Mader, A. (2000). Verification and optimization of a PLC control schedule. In *SPIN*, volume 1885, pages 73–92.

Bürckert, H.-J., Fischer, K., and Vierke, G. (2000). Holonic transport scheduling with teletruck. *Applied Artificial Intelligence*, 14(7):697–725.

Burman, M. (1995). *New results in flow line analysis*. PhD thesis, Massachusetts Institute of Technology.

Cimatti, A., Giunchiglia, E., Giunchiglia, F., and Traverso, P. (1997). Planning via model checking: A decision procedure for AR. In *ECP*, pages 130–142. Springer.

Cimatti, A., Roveri, M., and Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI*, pages 875–881.

Clarke, E., Grumberg, O., and Peled, D. (2000). *Model Checking*. MIT Press.

Dorer, K. and Calisti, M. (2005). An adaptive solution to dynamic transport optimization. In *AAMAS*, pages 45–51. ACM.

Edelkamp, S., Lluch-Lafuente, A., and Leue, S. (2001). Directed model-checking in HSF-SPIN. In *SPIN*, pages 57–79.

Edelkamp, S. and Reffel, F. (1998). OBDDs in heuristic search. In *KI*, pages 81–92.

Edelkamp, S. and Sulewski, D. (2008). Flash-efficient LTL model checking with minimal counterexamples. In *SEFM*, pages 73–82.

Fischer, K., Müller, J. R. P., and Pischel, M. (1996). Cooperative transportation scheduling: an application domain for dai. *Applied Artificial Intelligence*, 10(1):1–34.

Fox, M. and Long, D. (1999). The detection and exploration of symmetry in planning problems. In *IJCAI*, pages 956–961.

Fujimoto, R. (2000). *Parallel and Distributed Simulation Systems*. Wiley & Sons.

Ganji, F., Morales Kluge, E., and Scholz-Reiter, B. (2010). Bringing Agents into Application: Intelligent Products in Autonomous Logistics. In Schill, K., Scholz-Reiter, B., and Frommberger, L., editors, *Artificial intelligence and Logistics (AiLog) - Workshop at ECAI 2010*, pages 37–42.

Gerth, R., Peled, D., Vardi, M., and Wolper, P. (1995). Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, pages 3–18. Chapman & Hall.

Giunchiglia, F. and Traverso, P. (1999). Planning as model checking. In *ECP*, pages 1–19.

Godefroid, P. (1991). Using partial orders to improve automatic verification methods. In *CAV*, pages 176–185.

Greulich, C., Edelkamp, S., and Eicke, N. (2015). Cyber-Physical Multiagent-Simulation in Production Logistics. In *MATES*, pages 119–136. Springer.

Harrison, J. (1973). Assembly-like queues. *Journal of Applied Probability*, 10:354–367.

Helias, A., Guerrin, F., and Steyer, J.-P. (2008). Using timed automata and model-checking to simulate material flow in agricultural production systems – application to animal waste management. *Computers and Electronics in Agriculture*, 63(2):183–192.

Himoff, J., Rzevski, G., and Skobelev, P. (2006). Magenta technology multi-agent logistics i-scheduler for road transportation. In *AAMAS*, pages 1514–1521. ACM.

Hoffmann, J. and Edelkamp, S. (2005). The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579.

Hoffmann, J., Kissmann, P., and Torralba, Á. (2014). "distance"? Who cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In *ECAI*, pages 441–446.

Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

Hopp, W. and Simon, J. (1989). Bounds and heuristics for assembly-like queues. *Queueing Systems*, 4:137–156.

Jensen, R. M., Veloso, M. M., and Bowling, M. H. (2001). Obdd-based optimistic and strong cyclic adversarial planning. In *ECP*.

Kautz, H. and Selman, B. (1996). Pushing the envelope: Planning propositional logic, and stochastic search. In *ECAI*, pages 1194–1201.

Kupferschmid, S., Hoffmann, J., Dierks, H., and Behrmann, G. (2006). Adapting an AI planning heuristic for directed model checking. In *SPIN*, pages 35–52.

Lipper, E. and Sengupta, E. (1986). Assembly-like queues with finite capacity: bounds, asymptotics and approximations. *Queueing Systems*, pages 67–83.

Lluch-Lafuente, A. (2003). Symmetry reduction and heuristic search for error detection in model checking. In *MOCHART*, pages 77–86.

Love, N. C., Hinrichs, T. L., and Genesereth, M. R. (2006). General Game Playing: Game Description Language Specification. Technical Report LG-2006-01, Stanford Logic Group.

Manitz, M. (2008). Queueing-model based analysis of assembly lines with finite buffers and general service times. *Computers & Operations Research*, 35(8):2520 – 2536.

Morales Kluge, E., Ganji, F., and Scholz-Reiter, B. (2010). Intelligent products - towards autonomous logistic processes - a work in progress paper. In *PLM*, pages 348 – 357, Bremen.

Nau, D., Ghallab, M., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nissim, R. and Brafman, R. I. (2013). Cost-optimal planning by self-interested agents. In *AAAI*.

Parragh, S. N., Doerner, K. F., and Hartl, R. F. (2008). A Survey on Pickup and Delivery Problems Part II: Transportation between Pickup and Delivery Locations. *Journal für Betriebswirtschaft*, 58(2):81–117.

Rekersbrink, H., Ludwig, B., and Scholz-Reiter, B. (2007). Entscheidungen selbststeuernder logistischer Objekte. *Industrie Management*, 23(4):25–30.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach*. Pearson Education, 3rd edition.

Ruys, T. C. (2003). Optimal scheduling using branch and bound with SPIN 4.0. In *SPIN*, pages 1–17.

Ruys, T. C. and Brinksma, E. (1998). Experience with literate programming in the modelling and validation of systems. In *TACAS*, pages 393–408.

Saffidine, A. (2014). *Solving Games and All That*. PhD thesis, University Paris-Dauphine.

Valmari, A. (1991). A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–165.

Wooldridge, M. (2000). *Reasoning about Rational Agents*. The MIT Press.

Wooldridge, M. (2002). *An Introduction to Multi-Agent Systems*. Wiley and Sons, Chichester, UK.