Using SPIN for the Optimized Scheduling of Discrete Event Systems in Manufacturing

Stefan Edelkamp and Christoph Greulich

Faculty 3 – Mathematics and Computer Science, Bremen University, Bremen, Germany

Abstract. A discrete event system (DES) is a dynamic system with discrete states the transitions of which are triggered by events. In this paper we propose the application of the Spin software model checker to a discrete event system that controls the industrial production of autonomous products. The flow of material is asynchronous and buffered. The aim of this work is to find concurrent plans that optimize the throughput of the system. In the mapping the discrete event system directly to the model checker, we model the production line as a set of communicating processes, with the movement of items modeled as channels. Experiments shows that the model checker is able to to analyze the DES, 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

Discrete event (dynamic) systems (DES) provide a general framework for systems where the system dynamics not only follow physical laws but also additional firing conditions. DES research is concerned about performance analysis, evaluation, and optimization of DES. As the systems are often only available as computer programs, it turns out to be difficult to describe the dynamics of these systems using closed-form equations.

In many cases, *discrete event system simulation* (DESS) is chosen to describe the DES dynamics and for performance evaluation. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next. Each simulation activity is modeled by a process. The idea of a process is similar to the notion in model checking, and indeed one could write process-oriented simulations using independent processes. Most DESS systems store information about pending events in a data structure known as an *event queue*. Each item in the queue would at minimum contain the following information: a timestamp and a piece of software for executing event. The typical operations on an event queue are: inserting a new event and removing the next event (the one with the lowest timestamp) from the queue. It may also be necessary to cancel a scheduled event.

DESS is probably the most widely used simulation technique. Similar approaches are *system dynamics* (SD), and *agent-based simulation* (ABS). As the name suggests DES model a process as a series of discrete events. They are built using: *entities* (objects that move through the system; *events* (processes which the entities pass through); and *resources* (objects, which are needed to trigger event). SD are related to DES, focusing on *flows* around networks rather than *queueing systems*, it considers: *stocks* (basic stores of objects); *flows* (movement of objects between different stocks in the system); *delays* (between the measuring and then acting on that measurement). ABS is a relatively new technique in OR and consists of: *autonomous agents* (self-directed objects which move about the system) and *rules* (which the agents follow to achieve their objectives). Agents move about the system interacting with each other and the environment. ABS are used to model situations in which the entities have some form of *intelligence*.

Earlier simulation software was efficient but platform-dependent, due to the need for stack manipulation. Modern software systems, however, support lightweight processes or threads. By the growing amount of non-determinism, however, DESS encounters its limits to optimize the concurrent acting of individual processes.

With the advances in technology, more and more complex systems were built, e.g., transportation networks, communication and computer networks, manufacturing lines. In these systems, the main dynamic mechanism in task succession stems from synchronization and competition in the use of common resources, which requires a policy to arbitrate conflicts and define priorities, all kinds of problems generally referred to under the generic terminology of *scheduling*. This type of dynamics hardly can be captured by differential equations or by their discrete time analogues. This is certainly the reason why those systems, which are nevertheless true dynamic systems, have long been disregarded by formal method experts and have been rather considered by operations researchers and specialists of manufacturing with no strong connections with system theory. The dynamics are made up of *events*, which may have a continuous evolution imposed by some called software once they start, but this is not what one is mainly interested in: the primary focus is on the beginning and the end of such events, since ends can cause new beginnings. Hence, the word *discrete* includes *time* and *state*.

In this paper, we utilize the state-of-the-art model checker Spin [25] as a performance analysis and optimization tool, together with its input language Promela to express the flow production of goods. There are several twists needed to adapt *Spin* to the optimization of DES(S) that are uncovered in the sequel of the text. Our running case study is the *Z2*, a physical monorail system for the assembling of tail-lights. Unlike most production systems, Z2 employs agent technology to represent autonomous products and assembly stations. 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 *Spin* 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 the production.

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 optimization approach originally invented for Spin was designed for state space trees [37, 36], while the proposed approach also supports state space graphs. Scheduling via model checking has been pioneered by Mahler [1, 7] and Wijs [40].

The paper is structured as follows. First, we introduce discrete event simulation and industrial (flow) production. Then, we review related work including scheduling via model checking. Next, we introduce the industrial case study, and its modeling as well as its simulation as a DES. The simulator is used to measure the increments of the cost function to be optimized. Afterwards, we turn to the intricacies of the Promela model specification, to the parameterization of SPIN, as well as to the novel branch-and-bound optimization scheme. In the experiments, we study the effectiveness of the approach.

2 Preliminaries

2.1 Discrete Event Simulation

An *entity* is an object of interest in the system, and an *attribute* is a (relevant) property of an entity. Attributes are *state variables*, while *activities* form part of the *model specification* and *delays* form part of the simulation result. The *(system) state* is a variable needed to describe the state (e.g., length of a queue), which is aimed to be complete and minimal at any point in time. The occurrence of a *primary event* (e.g. arrival) is scheduled at a certain time, while a *secondary event* (e.g. queueing) is triggered by a certain condition becoming true. An *event* is an occurrence which is instantaneous may change the state of the system. The *(future) event list PQ* controls the simulation: it contains all future events that are scheduled, and is ordered by increasing time of events. Operations on the *PQ* are: *insert* an event into *PQ* (at an appropriate position!), *remove* first event from *PQ* for processing, and *delete* an event from *PQ*. Thus, *PQ* is a priorty queue. As operations must be performed efficiently, the common implementation of an event queue is a (binary) *heap*. With such a data structure, access to the next event requires $O(1)$ time, while inserting/deleting an event requires $O(\log(n))$ time, where n is the number of events currently in the queue. Depending on the implementation (e.g., Fibonacci heaps), there are other trade-offs, with constant-time insertion and $O(\log(n))$ (amortized) deletion. The generic DES simulation algorithm looks as follows:

1. IF (*PQ* empty) THEN exit
2. remove & process 1st primary event e from *PQ*
3. IF (conditional event e' enabled) THEN remove & process e', goto 3. ELSE goto 1.

We assume exact timing, i.e., deterministic time. However, by different choices points for generating successor events, the simulated DES itself is non-deterministic. Events inserted with priority t are generally assumed to remain unchanged until deletion at time t.

2.2 Flow Manufacturing

Flow manufacturing systems are DES 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 manufacturing 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.

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.

In a general notation of flow manufacturing, system progress is non-deterministic and asynchronous, while the progress of time is monitored.

Definition 1 (Flow Manufacturing System). *A* flow manufacturing system *is a 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 partial 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 manufacturing system. 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 *$\pi_i = (e_1, t_1, l_1), \ldots, (e_m, t_m, l_m)$, 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.*

The objective in a flow manufacturing 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

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

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

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

subject to the side constraints that

– *time stamps on all runs $\pi_i = (e_1,t_1,l_1) \ldots (e_m,t_m,l_m)$, $i \in \{1,\ldots,n\}$ 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 $i \in \{1,\ldots,n\}$ 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 $\pi_i = (e_1,t_1,l_1) \ldots (e_m,t_m,l_m)$, $i \in \{1,\ldots,n\}$, 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 $\pi_i = (e_1,t_1,l_1) \ldots (e_m,t_m,l_m)$, $i \in \{1,\ldots,n\}$, we have that $0 \le l_j < |\psi(e_j)|$.*

3 Related Work

One of the most interesting problems in manufacturing is *job shop scheduling* [3]. When solving the scheduling problem, a set of n jobs has to be assigned to a set of m machines. Consequently, the total number of possible solutions is $(n!)^m$. The problem complexity grows when the number of required ressources increases, e.g. by adding specific tools or operators to run machines. For an additional set k of necessary ressources, the number of possible solution increases to $((n!)^m)^k$ [38]. In the related *flow shop scheduling* problem, a fixed sequence of tasks forms a job [17]. It is applicable to optimize the so called *makespan* on assembly lines.

Flow line analysis is a more complex setting, often done with queuing theory [33, 8]. Pioneering work in analyzing assembly queuing systems with synchronization constraints analyzes assembly-like queues with unlimited buffer capacities [22]. 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 [4, 30, 26]. A rare example of model checking flow production are timed automata that were used for simulating material flow in agricultural production [23].

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* [35], 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 [18], 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 [19, 9]. 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 [24], 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 [28] *bounded model checkers* exploit SAT and SMT representations [5, 2] of the system to be verified, while *directed model checkers* [13, 29] exploit panning heuristics to improve the exploration for falsification; partial-order reduction [39, 20] and symmetry detection [16, 32] limit the number of successor states, while symbolic planners [10, 27, 14] apply functional data structures like BDDs to represent sets of states succinctly.

4 Case Study

We consider the simulation of the real-world *Z2* production floor unit [34]. 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. 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. Different parts can be used to assemble different variants of the tail-lights. 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. In the DESS implementation of the Z2 system, every assembly station, every monorail shuttle and every product is represented by a software process. Even the RFID read-

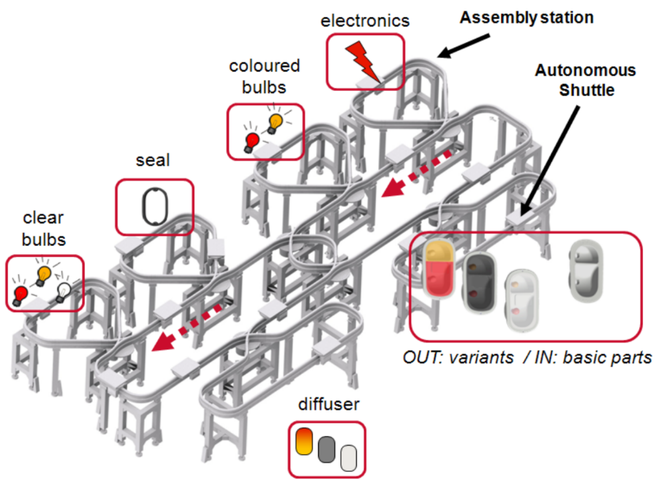

Fig. 1: Assembly scenario for tail-lights.

ers which keep track of product positions are represented by software processes, which decide when a shuttle may pass or stop.

Most processes in this DESS resemble simple reflex methods. These processes just react to requests or events which were caused by other processes or the human workers involved in the manufacturing process. In contrast, the processes 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 individual preconditions.

The product processes 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. 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 beeing processed, or on waiting in line at a station or a switch.

The Z2 DES was developed strictly for the purpose of controlling the Z2 monorail hardware setup. Nonetheless, due to its hardware abstraction layer [34], the Z2 DES can be adapted into other hardware or software environments. By replacing the hardware with other processes and adapting the monorail infrastructure into a directed graph, the Z2 DES has been transferred to a DESS [21]. Such an environment, which treats the original Z2 modules like black boxes, can easily be hosted by a DESS. Experiments showed how close the simulated and the real-world scenarios match.

For this study, we provided the 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

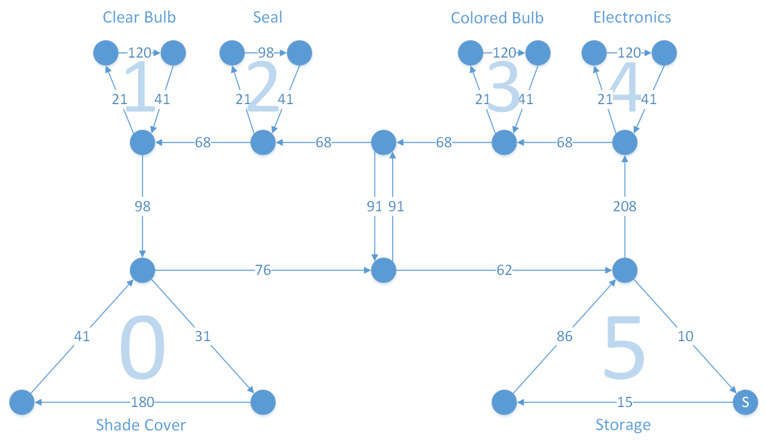

Fig. 2: Weighted graph model of the assembly scenario.

(see Fig. 2), where the weight of an edge denotes the traveling/processing time of the shuttle between two respective nodes.

5 Promela Specification

Promela is the input language of the model checker Spin[1], 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 DESS, 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 processes 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 Promela model of the Z2 as follows. First, we define global setting on the number of stations and number of switches. We also define the data type storing the

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

index of the shuttle/product to be `byte`. In the model, switches are realized as processes and edges between the units by the following channels.

```
chan 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 have bit-vectors for marking the different assemblies.

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

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

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

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

```
proctype Switch(byte in; byte out; byte station) {
  shuttle s;
  do
  :: 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 switches and hereby define the network of the monorail system. Moreover, initially, we have that the metal cast of each product is already present on its carrier, the shuttle. The coloring of the 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 = infinity; }
     for (i : 0 .. (SHUTTLES)/2)){ color[i] = 1; }
     for (i : 0 .. (SHUTTLES-1)) { metalcast[i] = 1; }
     for (i : 0 .. (STATIONS-1)) { run Entrance(i); run Exit(i); }
     run Switch(7,0,5); run Switch(0,1,4);
     run Switch(1,2,3); run Switch(3,4,2);
     run Switch(4,5,1); run Switch(5,6,0);
     run Hub(2,3,8,9); run Hub(6,7,9,8);
     for (i : 0 .. (SHUTTLES-1)) { 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 actions are blocking, so that we chose to use an atomic block around each loop.

6 Optimized Scheduling

Inspired by [7, 31] and [36] we applied and improved branch-and-bound (BnB) optimization. *Branching* is the process of spawning subproblems, while *bounding* refers to ignoring partial solutions that cannot be better than the current best solution. To this end, lower and upper bounds and are maintained as global control values on the solution quality, which improves over time.

Algorithm 1 DFBnB Algorithm.

```
DFBnB($F = (A,E,G,\prec,S,Q)$)
  Initialize upper bound $U$
  $\pi' \leftarrow \pi \leftarrow \emptyset$
  $DFS(F,(s,\dots,s),0,U)$
  return $\pi'$

DFS($F,u,\pi,U$)
  $\pi \leftarrow extend(\pi,u)$
  if $(u = (t,\dots,t))$
    if $(cost(\pi) < U)$ $\pi' \leftarrow \pi$; $U \leftarrow cost(\pi)$
  else for each $v$ in $successors(u)$
      if $(cost(\pi) + h(v) < U)$ $DFS(F,v,\pi,U)$
```

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

As with standard DFS, the first solution obtained might not be optimal. With *depth-first branch-and-bound* (DFBnB), however, the solution quality improves over time together with the global value U until eventually the lower bound $L(u)$ at some node u is equal to U. The pseudo-code of this approach is shown in Alg. 1. In standard Spin, the trivial heuristic is $h \equiv 0$ used, but in HSF-Spin [13], a few heuristic functions have been implemented. We obtain the following result.

Theorem 1. *(Optimality of Branch-and-Bound for Flow Manufacturing) For a admissible heuristic function h, the DFBnB procedure in Alg. 1 will eventually find the optimal solution to the flow manufacturing problem* $F = (A,E,G,\prec,S,Q)$.

Proof. We can compute costs for partial runs and extend partial schedules incrementally. The objective function to be minimized over all possible runs Π in the system is monotone increasing. Only inferior paths that cannot be extended to a better path than the currently best known one are pruned. As the state space is finite, the search will eventually terminate and return the optimal solution. q.e.d.

There are different options for finding optimized schedules with the help of a model checker that have been proposed in the literature. First, in the *Soldier* model of [37], 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 for our purpose. An alternative proposal for branch-and-bound search is based on the support of native c-code in Spin (introduced in version 4.0) [36]. One running example is the traveling salesman problem (TSP), but the approach is generally applicable to many other optimization problems. However, as implemented, there are certain limitations to the scalability of state space problem graphs. Recall that the problem graph induced by the TSP is in fact a tree, generating all possible permutations for the cities.

Following [13, 7] and [36] we applied 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 ran with the parameter `./pan -i`. As these traces are step-optimized, and not cost-optimized, Ruys [36] proposed to introduce a variable *cost* that we extend as follows.

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

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

6.1 Guarded 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.

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 these 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.

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

At termination of a successful run, we now extend the proposeal of [36]. We use the integer array `cost[SHUTTLES]` of the Promela model. It enables each process to keep track of its local cost vector and is increased by the cost of each action as soon as the action is executed. This enables the model checker to print values to the output, only if the values of the current `max` cost and `total` cost have improved.

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

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

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

6.2 Process Synchronization

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

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 cost of the respective shuttle is increased by the cost of the given edge.

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

We introduce an atomic C function *canreceive*(q) that returns true only if the first item *s* of *q* has minimal *cost(s)*, changing the receiving constraint to the following.

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

Within Spin, the global Boolean variable `timeout` is automatically set to *true* when all current processes are unable to proceed, e.g., because they cannot receive a message. Consequently, for every shuttle *p*, all processes will be blocked and `timeout` will be set to *true*. As suggested by Bošnački and Dams [6], we add a process that enforces time progress, whenever `timeout` occurs (*final* is a macro for reaching the goal).

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

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

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

As a summary, the constraint bounded depth-first exploration has turned into the automated generation of the underlying state space of the DES, using c-code to preserve the causality of actions and to simulate the future event list.

7 Evaluation

In this section, we present results of a series of experiments executing two different Promela models. We compare the results of the exploration minimizing local virtual time (LVT) [12] to the ones simulating the discrete event system (DES) described in this paper. For comparison, we also present results of simulation runs of the original implementation on hardware [21].

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).

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	$0 \Rightarrow 4$	$1 \Rightarrow 4$	$2 \Rightarrow 4$	$0 \Rightarrow 3$	$1 \Rightarrow 2$	$2 \Rightarrow 3$	$0 \Rightarrow 2$	$1 \Rightarrow 1$	$2 \Rightarrow 2$	$0 \Rightarrow 0$	$1 \Rightarrow 0$	$2 \Rightarrow 0$	$0 \Rightarrow 5$	$1 \Rightarrow 5$	$2 \Rightarrow 5$

For smaller problems we experimented with Spin's parallel BFS (`-DBFS_PAR`), as it computes optimal-length counterexamples. The hash table is shared based on compare-and-swap (CAS). We also tried supertrace/bitstate hashing (-DBITSTATE) as a trade-off. Unfortunately, BFS interacts with `c_track`, so we had to drop the experiments for cost optimization. Swarm tree search (`./swarm -c3 -m16G -t1 -f`) found many solutions, some of them being shorter than the ones offered by option `-i` (indicating ordering effects), but due to the increased amount of randomness, for the optimized scheduling in general no better results that ordinary DFS were found.

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 the the same sequence of production steps for the product of each shuttle. The example given in figure 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.

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 Section 6.2. In the DES model, actual travelling 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. Consequently, results 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 any conducted experiment. However, the LVT model is able to search the whole state space within the 16 GB RAM limit (given by our machine) for $n \leq 3$ shuttles, whereas the DES model is unable to search the whole state space for $n > 2$. For every experiment with $n > 3$ (LVT) or $n > 2$ (DES) shuttles respectively, searching the state space for better results was cancelled, when the 16 GB RAM limit was reached.

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.)

	MAS	LVT		DES	
Products	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

In general, experiments indicate that the DES model is faster and more memory efficient than the LVT model even though both approaches propose 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 beeing processed at a station. The LVT model includes idle time, hence, it minimizes the total time spent in the production system.

The DES model follows the assumption that once an event is scheduled to occur at a specific time, the event is added to the event list and its time does not change. This assumption poses a limitation in comparison with the more flexible LVT approach. In our real-world setup, if one element in a process queue is delayed, all the ones behind it are delayed as well. In future work, extentions of the event list have to be looked at in order to support the priority queue operation *increaseKey*. In general, we observe that DES and LVT are both sound in resolving deadlocks and computing optimal plans. DES has a slightly improved performance and LVT has a more accurate representation for the progress of time.

8 Conclusions

Simulation provides a method to approximate the behaviour in a real system (and, hence, can be used for testing scenarios). Constructing the model can prove useful in achieving greater understanding of the system. In this paper, we presented a novel approach for model checking (instead of simulating) DES. The research is motivated by our interest in finding and comparing centralized and distributed solutions to the optimization problems in autonomous manufacturing.

Using model checking for optimizing DES is a relative new playground for formal method tools in form of a new analysis paradigm. The formal model in Promela reflects the routing and scheduling of entities in the DES. Switches of the rail network were modeled as processes, the edges between the switches as communication channels. Additional constraints to the order of production steps enable to carry out a complex planning and scheduling 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. As with directed model checking, heuristics are expected to guide the search towards finding a good schedule faster. A close look at the limits and possibilities of LVT and DES revealed that both approaches have certain advantages and disadvantageds. In future work, we will consider alternative approaches to represent time manufacturing optimization.

References

1. Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In *CAV*, pages 478–492, 2001.
2. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN*, pages 146–162. Springer, 2006.
3. T. P. Bagchi. Multiobjective Scheduling by Genetic Algorithms. sep 1999.
4. U. Bhat. Finite capacity assembly-like queues. *Queueing Systems*, 1:85–101, 1986.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
6. D. Bosnacki and D. Dams. Integrating real time into spin: A prototype implementation. In *FORTE / PSTV*, pages 423–438, 1998.
7. E. Brinksma and A. Mader. Verification and optimization of a PLC control schedule. In *SPIN*, volume 1885, pages 73–92, 2000.
8. M. Burman. *New results in flow line analysis*. PhD thesis, Massachusetts Institute of Technology, 1995.
9. A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *ECP*, Lecture Notes in Computer Science, pages 130–142. Springer, 1997.
10. A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI*, pages 875–881, 1998.
11. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
12. S. Edelkamp and C. Greulich. Branch-and-bound optimization of a multiagent system for flow production using model checking. In *ICAART*, 2016.
13. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *SPIN*, pages 57–79, 2001.
14. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *KI*, pages 81–92, 1998.
15. S. Edelkamp and D. Sulewski. Flash-efficient LTL model checking with minimal counterexamples. In *SEFM*, pages 73–82, 2008.
16. M. Fox and D. Long. The detection and exploration of symmetry in planning problems. In *IJCAI*, pages 956–961, 1999.
17. M. R. Garey, D. S. Johnson, and R. Sethi. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, 1(2):117–129, may 1976.
18. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, pages 3–18, 1995.
19. F. Giunchiglia and P. Traverso. Planning as model checking. In *ECP*, pages 1–19, 1999.
20. P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, pages 176–185, 1991.

21. C. Greulich, S. Edelkamp, and N. Eicke. Cyber-physical multiagent simulation in production logistics. In *German Conference on Multiagent System Technologies (MATES)*, 2015.
22. J. Harrison. Assembly-like queues. *Journal of Applied Probability*, 10:354–367, 1973.
23. A. Helias, F. Guerrin, and J.-P. Steyer. 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, 2008.
24. J. Hoffmann, P. Kissmann, and Á. Torralba. ”Distance”? Who Cares? Tailoring Merge-and-Shrink Heuristics to Detect Unsolvability. In *ECAI*, pages 441–446, 2014.
25. G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
26. W. Hopp and J. Simon. Bounds and heuristics for assembly-like queues. *Queueing Systems*, 4:137–156, 1989.
27. R. M. Jensen, M. M. Veloso, and M. H. Bowling. OBDD-based optimistic and strong cyclic adversarial planning. In *ECP*, 2001.
28. H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *ECAI*, pages 1194–1201, 1996.
29. S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *SPIN*, pages 35–52, 2006.
30. E. Lipper and E. Sengupta. Assembly-like queues with finite capacity: bounds, asymptotics and approximations. *Queueing Systems*, pages 67–83, 1986.
31. W. Liu, Z. Gu, J. Xu, Y. Wang, and M. Yuan. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *CODES+ISSS*, pages 61–70, 2009.
32. A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *MOCHART*, pages 77–86, 2003.
33. M. Manitz. Queueing-model based analysis of assembly lines with finite buffers and general service times. *Computers & Operations Research*, 35(8):2520 – 2536, 2008.
34. E. Morales Kluge, F. Ganji, and B. Scholz-Reiter. Intelligent products - towards autonomous logistic processes - a work in progress paper. In *Intern. PLM Conf.*, 2010.
35. D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
36. T. C. Ruys. Optimal scheduling using branch and bound with SPIN 4.0. In *Model Checking Software, 10th International SPIN Workshop*, pages 1–17, 2003.
37. T. C. Ruys and E. Brinksma. Experience with literate programming in the modelling and validation of systems. In *TACAS*, pages 393–408, 1998.
38. W. Shen, L. Wang, and Q. Hao. Agent-Based Distributed Manufacturing Process Planning and Scheduling: A State-of-the-Art Survey. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 36(4):563–577, jul 2006.
39. A. Valmari. A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–165, 1991.
40. A. Wijs. *What to do Next? Analysing and Optimising System Behaviour in Time*. PhD thesis, Vrije Universiteit Amsterdam, 1999.