Synthesis and stochastic assessment of schedules for lacquer production*

H.C. Bohnenkamp[1] H. Hermanns[2,1] R. Klaren[1] A. Mader[1] Y.S. Usenko[1]

[1]Faculty of Electrical Engineering, Mathematics and Computer Science,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

[2]Department of Computer Science, Saarland University, D-66123 Saarbrücken, Germany

Abstract

The Modest *modeling language pairs modeling features from stochastic process algebra and from timed and probabilistic automata with light-weight notations such as exception handling. It is supported by the* Motor *tool, which facilitates the execution and evaluation of* Modest *specifications by means of the discrete event simulation engine of the* Möbius *tool. This paper describes the application of* Modest, Motor *and* Möbius *to a highly nontrivial case. We investigate the effect of faulty behavior on a hard real-time scheduling problem from the domain of lacquer production. The scheduling problem is first solved using the timed model-checker* Uppaal. *The resulting schedules are then embedded in a* Modest *failure model of the lacquer production line, and analyzed with the discrete event simulator of* Möbius. *This approach allows one to assess the quality of the schedules with respect to timeliness, utilization of resources, and sensitivity to different assumptions about the reliability of the production line.*

1. Introduction

Scheduling problems are an important area of research, where typically strict response requirements with limited resources have to be met. Their solution is usually based on techniques from classical scheduling theory, which is a well-established branch of operations research [7]. Industrial scheduling problems, e.g. for pharmaceutical manufacturing lines, are highly complex, and only heuristic solutions are used in practice.

More recently, the application of search techniques from model checking and mixed integer linear programming have been proposed as a complementary approach to schedule synthesis. The potential advantage of this approach lies in its simplicity and genericity. Where classical scheduling solutions often rely on restricting assumptions that may not be fulfilled in practice, state space exploration techniques are generally applicable, and very robust under variation in the problem parameters. In the model checking approach scheduling feasibility is expressed as temporal or real-time property ("The production can be finished by friday afternoon."), and a model checker is used to check whether this property holds for a reachable state in a model of the overall system behavior.

The European AMETIST [3] project is a joint initiative from leading research groups on timed systems, and four industrial partners. One of the main strands of activities of AMETIST is to advance the power of schedulability analysis by tackling challenging industrial case studies provided by the industrial partners. One particularly challenging case - focusing on scheduling in the domain of lacquer production - originates from AXXOM AG, a German software company specialized in supply chain optimization. The scheduling synthesis for lacquer production has been successfully tackled using the real-time model-checker Uppaal. However, this analysis had to ignore two parameters of the original problem specification as provided by AXXOM. These two parameters types relate to quantitative stochastic influences due to failures, repairs, cleaning periods and other unforeseeable (and thus unplannable) events.

This paper reports on our activities to incorporate these types of parameters into an *a posteriori* analysis of the schedules generated with Uppaal [16]. To do so, we refine the timed automata model of the production units into a *stochastic* timed automata model in order to faithfully represent the stochastic perturbations, as described by the so-far not considered specification parameters. The assessment of the schedules is then done on the basis of the estimates obtained by discrete-event simulation.

Technically, we use the process algebra-based formalism Modest to specify the production units, as well as the schedules. The Modest models are evaluated by means of

* This work was supported by the NWO-DFG bilateral project Validation of Stochastic Systems (VOSS), the STW/PROGRESS project TES.4999 (HaaST), the EU IST project AMETIST, and the NWO Vernieuwingsimpuls project 016.023.010 (VoPaD).

Möbius, a very flexible multi-formalism/multi-solution performance evaluation environment, which offers a powerful simulator and statistical engines to evaluate different types of stochastic systems. The Modest tool Motor is used to connect Modest to Möbius.

In summary, this paper offers the following tangible contributions. *(i)* It shows how schedules for this type of schedulability problem can be synthesized using Uppaal. *(ii)* It shows the application of Modest to a highly non-trivial example. *(iii)* It shows that the integration of Motor and Möbius, as advertised in [5], is a sustainable link to carry out a *combined* qualitative *and* quantitative analysis. *(iv)* It provides a methodology to fruitfully complement (rather *en vogue*) timed automata-based analysis with (fairly well-established) stochastic simulation analysis. It is out of our imagination that any other related technique can be applied as fruitful as Modest in this context.

Organization of the paper. Section 2 discusses the methodology and tools used to carry out this case study. Section 3 introduces the scheduling problem and how it is modeled in Modest. Results of our analysis are presented in Section 4. Section 5 concludes the paper.

2. Methodology

2.1. Scheduling synthesis based on timed automata model checking.

The synthesis of schedules can be seen as a special case of control synthesis [13]. It was first introduced by Fehnker [9], and Abdeddaïm and Maler [1]. In general, a model class suitable for scheduler synthesis must provide the possibility to represent timing information in addition to the usual state/action/synchronisation mechanism. The underlying framework used in this paper is the one of timed automata as introduced by Alur and Dill [2] and represented in the model checker Uppaal. In addition to traditional automata there are clocks, whose valuations progress with time. Clocks can be reset and be used as guards for transitions and in state invariants. In general, the semantic model of a timed automaton is an infinite state space. The region automaton construction [2], however, shows that this infinite state space can be mapped to an automaton with a finite number of equivalence classes (regions) and the finite-state model checking approach can be applied to the so constructed finite region automaton.

Scheduling synthesis based on timed automata makes use of the search strategies of model checking. First, a model of the overall, uncontrolled system behavior has to be constructed In our case here the model consists of all possible production steps (of all orders) possible at every moment. Feasibility is formulated as a real-time property ("The production is finished by friday evening"). The model checker searches the reachable state space for a state where this property holds. If it has found one, it provides a diagnostic trace. The diagnostic trace contains a sequence of actions and delays from the initial state to the state found. The start of a processing step is encoded as an action and can be found in the diagnostic trace together with the timing information. This is enough to extract a feasible schedule from a diagnostic trace.

The advantage of this approach is its robustness against changes in the setting of parameters, due to the fact that timed automata provides a very general model class. The disadvantage lies in the well-known state-space explosion problem. For interesting cases the model checking approach as described above does not terminate. The way out is to add heuristics, or features of schedules, that reduce the search space to a size that can be traversed more easily. In Section 3.1 we discuss the heuristics used for the case study presented here.

2.2. Modest

We introduce the language features of Modest by a small example; we refer to [8] for an exhaustive language introduction. Figure 1 depicts a fragment of a Modest specification, describing a cashier in a discount market environment. All numbers refer to this figure.

The Modest language allows one to specify *processes* (1), and to compose them in *parallel* using a 'par' operator (2). Processes can manipulate *data variables* by *assignments* (3). Data variables are typed and must be declared, and the point of declaration (4) determines their *scope*. In particular, they may be *local* to a process (not shown in this example), or *global*, in which case they are shared between all processes. Standard data types in Modest are *bool*, *int* and *float*. A particular type of variable which can be declared is the *clock* type (4). Clocks can be read like an ordinary *float* variable, but advance their value linearly to system time. All clocks run at the same speed. Clocks can only be set to zero. The language provides generic constructs to sample values from a set of predefined probability distributions (5). For instance, '$xd = U[2.0]$' assigns a sample from the uniform distribution on the interval $[10, 20]$ to the variable '*xd*'. Other types of distributions are, *e.g.*, *Exponential(rate)* and *Normal(mean,var)*.

Apart from manipulating data, processes can interact with other parallel processes (or the environment) by means of *actions* (6). Their occurrence within a process can be guarded by a 'when(.)' clause (7), specifying a Boolean enabledness condition. In particular, the boolean expression in a 'when(.)' clause may refer to clock values. In that case, an action may be enabled as soon as the when() condition becomes true (and no other action becomes enabled earlier). Processes in the body of a 'par' (2) construct perform actions and assignments independently from each other, ex-

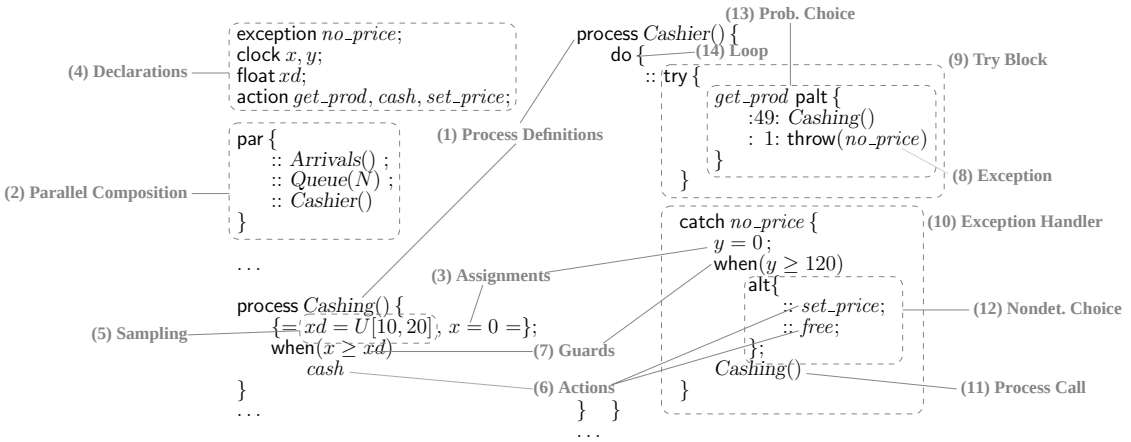

Figure 1. A MODEST specification

cept that common (non-local) actions need to be executed synchronously, à la CSP [11].

MODEST provides means to raise (8) *exceptions* in side a *try block* (9) and handle them (10). Exceptions must be declared (4). When an exception is raised (8), process control is handed over to the exception handler (10). Another, standard way of handing over process control is by a simple process call (11). Upon termination of the called process, the calling process gains back control, like in an ordinary procedure call.

The 'alt' construct (12) is used to specify choice between different possible behaviors. In general, it is allowed to make this choice nondeterministically. In this case the choice is between the execution of action *set_price* and action *free*. A variant thereof is the 'palt' construct (13), which provides a weighted *probabilistic choice*, where for each *weight* has the form :*w*:, with *w* a positive real number. A 'palt' must always be preceded by an action. The 'do' keyword (14) indicates a *repetitive behavior*. Upon termination of the body of this construct, the body is restarted, until a 'break' is encountered (not used in the example).

The combination of when() and alt has the features of a *test-and-set* operation, and does therefore enable the implementation of locks and semaphores via shared variables. In Section 3.2 we make extensive use of this feature.

2.3. MOTOR and MÖBIUS

The schedules assessed in this paper have been analyzed by means of the MODEST tool environment MOTOR and the performance evaluation environment MÖBIUS. In this section we discuss these two tools.

MÖBIUS. MÖBIUS is a performance evaluation tool environment developed at the University of Illinois at Urbana-Champaign, USA. MÖBIUS supports multiple input formalisms and several evaluation approaches for these models. Figure 2 shows an overview over the MÖBIUS architecture. Atomic models are specified in one of the available input formalisms. Atomic models can be composed by means of state-variable sharing, yielding so called composed models. Along with an atomic or composed model, the user specifies a reward model, which defines a reward structure on the overall model. On top of a reward model, the tool provides support to define experiment series, called *Studies*, in which the user defines the set of input parameters for which the composed model should be evaluated. Each combination of input parameters defines a so-called *experiment*. Before analyzing the experiments, a solution method has to be selected: MÖBIUS offers a powerful discrete-event simulator, and, for Markov models, explicit state-space generators and numerical solution algorithms. It is possible to analyze transient and steady-state reward models.

MÖBIUS currently supports four input formalisms: Bucket and Balls (an input formalism for Markov Chains), SAN (Stochastic Activity Networks) [14, 15], and PEPA (a Markovian Stochastic process algebra) [10]. Recently, the MODEST modeling language has been integrated into the MÖBIUS framework.

MOTOR. In order to facilitate the analysis of MODEST models, we have developed the prototype tool MOTOR [6]. The enormous expressiveness of MODEST implies that no generic analysis algorithm is at hand. Instead, MOTOR aims at supporting a variety of analysis algorithms tailored to the variety of analyzable submodels. The philosophy behind MOTOR is to connect MODEST to existing tools, rather than reimplementing existing analysis algorithms anew.

To complement the qualitative analysis of MODEST specifications using CADP we started joint efforts with the MÖBIUS developers [5] to link to the powerful solution techniques of MÖBIUS for quantitative assessment. The main objective was to simulate MODEST models by means of the MÖBIUS distributed discrete-event simulator, because a stochastic simulator can cope with one of the largest class of models ex-

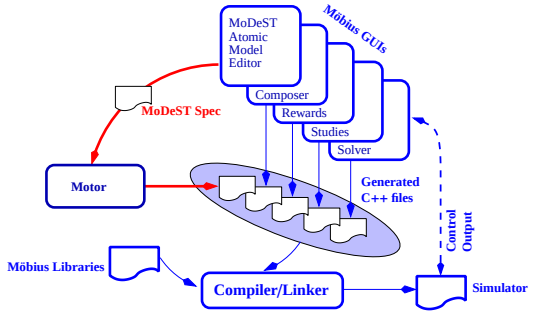

Figure 2. Möbius Architecture with Motor

pressible in Modest. A single semantic concept which is supported in Modest can not be dealt with in Möbius: nondeterminism. We address this restriction below.

Motor *and* Möbius. The integration of Modest into Möbius is done by means of Motor. For this integration, Motor has been augmented with a Modest-to-C++ compiler. From a user-perspective, the Möbius atomic model interface to design Modest specifications is an ordinary text editor. Whenever a new version of the Modest specification is saved to disk the Motor tool is called automatically in order to regenerate all C++ files (*cf.* Figure 2). Additionally, a supporting C++ library has been written for Möbius, which contains two components: first, a virtual machine responsible for the execution of the Modest model, and second, an interface to the simulator of Modest.

As with all other types of atomic models of Möbius is it possible to define reward models and studies on top of Modest models. The state variables which are accessible for reward specification are the *global variables* of the Modest specification. Additionally, it is possible to declare constants in the Modest specification as *extern*, meaning that these constants are actually input parameters of the model, pre-set according to the specified study.

Due to the possibility to specify non-Markov and non-homogeneous stochastic processes, only simulation is currently supported as a suitable evaluation approach for Modest models within Möbius. While it is in principle possible to identify sublanguages of Modest corresponding to Markov chain models, this has not been implemented in Motor yet.

Nondeterminism. In languages like Modest, specifying nondeterministic behavior is a vital concept and a basic requisite for compositionality. However, in stochastic evaluation of systems, nondeterminism can not be dealt with, since every possible state-change or time delay must be (stochastically) described. Since simulation is a form of stochastic evaluation, this would imply that a simulation model must not contain nondeterminism as well. However, it is in fact possible to take a more relaxed view here. Nondeterminism in simulation occurs if two (or more) events are scheduled for execution at the same time. There are two cases to distinguish: *(i)*, the order of execution matters; *(ii)*, the order does not matter. The first case occurs if the two events manipulate the same data, or are in a direct choice, such that the execution of one event would disable the respective other. If none of these dependencies exist between the two events, the second case holds and the simulator can just fire both in arbitrary order.

In the present case study, nondeterminism is heavily used as a convenient modeling means, and the possibility that nondeterminism of the first kind occurs can not be excluded. We will comment on this problem in Section 3.2.

3. Modeling

This section describes the case study considered in this paper, and how it was modeled with the tools in Modest. The case study originates from AXXOM AG (Munich, Germany) within the framework of the IST project AMETIST [3]. The focus of this case study is on synthesis of optimal schedules for a lacquer production plant, where various types of costs have to be optimized, e.g. for storage, material, delay, etc. In a first version of the problem we addressed only schedulability, i.e. the question whether there exist feasible schedules.

The description provided by AXXOM contains an abstract view on a pipeless production plant for lacquers of different kind. The plant comprises several resources (mixing vessels, filling lines, etc) which are used in different ways to produce different kinds of lacquer. These products need to be produced according to product orders which arrive at specified start times (from the customers), and possess due dates at which the production needs to be finished (and delivered to the customer).

Concretely, there are 29 product orders (jobs) that have to be delivered by their due dates and can not start before the earliest start time specified. Each product is based on one of three recipes (uni, metallic, and bronze lacquers). Each recipe gives information on the sequence of production steps, the type of resource that is needed for a production step, and the timing behavior, e.g. the processing time with a certain resource, offset times, that specify the time interval between succeeding production steps, etc. See Figure 3 for a graphical representation of the three recipes. The difference of the scheduling problem here to a classical job-shop problem is twofold. *(i)* There are multiple resources, e.g. 3 mixing vessels of the same type, and, *(ii)* there are timing constraints between processing steps, e.g. upper and lower bounds on the start time of a production step with respect to the start or end time of the previous production step.

The resources available for production are: 2 mixing vessels for uni-lacquers, 3 mixing vessels for metallic and

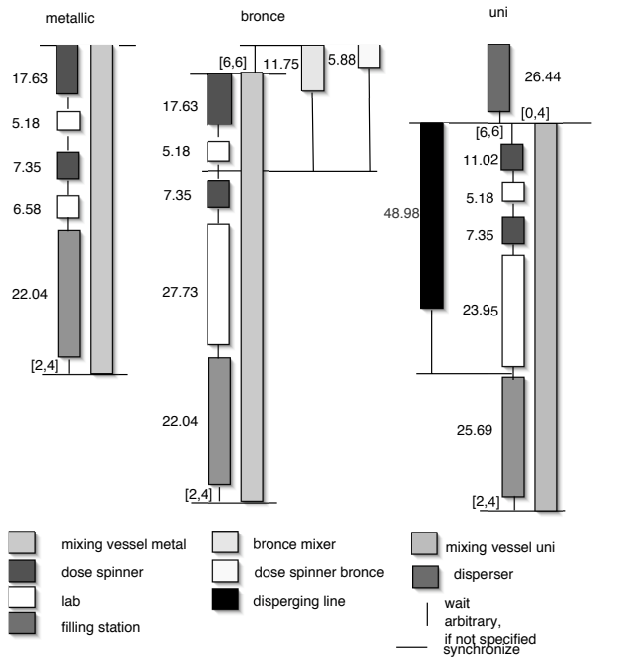

Figure 3. Recipes for different types of lacquers with timing information.

bronze lacquers, 2 filling stations, 1 disperser, 2 dose spinners, a special bronze dose-spinner, a bronze mixer, a dispersing line, and a lab with unlimited availability.

In the description provided by AXXOM, each of the resources comes along with two stochastic parameters, so-called performance and availability factors. Both are real numbers from the interval $(0, 1)$. A performance factor p indicates that the average fraction of time the resource is not operational due to unforeseeable or unplannable circumstances, such as resource breakdowns, is $1 - p$. The availability factors instead can best be considered as a means to abstract from system details. An availability factor a describes the fraction of operational time in which the resource can actually be used because necessary human support is present. A more detailed model could replace this factor by specifying the working hours (mon-fri, 9 a.m. to 5 p.m., for instance) explicitly. Thus, the availability factor can be influenced by increasing the person power, while the performance factor can not be influenced. Apart from these two factors, the description provided by AXXOM does not contain information about the frequency of resource breaks, the duration of down-times or repair-times, etc.

3.1. The UPPAAL model and experiments

UPPAAL [16] is a tool for modeling, simulation and verification of timed automata. To tackle the principle scheduler synthesis problem for the case study, we defined a (template) automaton for each of the recipes, including free parameters for earliest start time and due date. An order is then an instantiation of a recipe with these dates. The system consists of the parallel composition of the 29 instantiations of the recipes. Resources are modeled as integer variables shared between all the order automata. Each recipe requires one or two clocks measuring the durations of the processing steps and the intervals between subsequent processing steps. We dealt with the processing times in two different ways (giving two different models). In the first case we ignored the performance and availability factors. In the second case, following the suggestions of the case study provider, we extended the processing times by the corresponding factors, e.g., if a resource has a performance of 50% the production times on this resource are doubled.

The property to prove was whether all automata representing the orders can reach their final state, indicating that they are ready before the due date. In the first place the model checking runs suffered from state space explosion and further techniques (heuristics) had to be added. After some experimentation, the following heuristics were employed to generate successful runs:

1. start of the orders of one type (uni, metal, bronze) in order of their due dates;
2. orders of the same type do not overtake each other;
3. non-laziness (active), if possible, i.e. a order does not wait for a continuously unused resource, and takes it only after a (useless) waiting period. In the model such behavior leads to a deadlock and will be not considered (backtracking);
4. greedy strategy (alternatively to non-laziness): if a resource is available it is taken immediately - note that this strategy possibly excludes good schedules, but a feasible schedule found with this technique is still a valid one.

Several successful runs on models with the heuristics above were found with a random-depth first version of the verifier of UPPAAL, terminating within a few seconds (whereas the classical depth-first did not terminate). The diagnostic traces leading to the successful state provided the (feasible) schedules. For the work presented here, 20 schedules were synthesized, each of them meeting all 29 individual due dates. Ten synthesized schedules are based on the plain processing times, not taking performance and availability factors into account. The other ten are overapproximated in the sense that, processing times are extended by the performance and availability factors as follows. Instead of scheduling a job j on a resource R for processing time P, it is scheduled for $P/(p \cdot a)$ time units, where p and a are the above factors for this particular resource. This way of overapproximating was suggested (and is practiced) by the case study provider lacking other possibilities for dealing with stochastic parameters.

3.2. Modeling the case in MODEST

Ten of the schedules generated by model checking with UPPAAL contain overapproximations in the sense that it is assumed that the machine is reserved for a longer period than needed, in order to compensate the possible delays due to the breaks and repairs, and other unplanned influences. Since this way of overapproximation tries to compensate for random effects with a fixed amount of time, there is an obvious risk that the additionally assigned time may not suffice, and that a job may therefore miss its due date in reality. Intuitively, this risk is even higher for the schedules computed without overapproximations.

This is the phenomenon we are interested to study with a more faithful model of the resources, where the stochastic perturbations as described by the performance and availability factor are modeled explicitly. To this end, we execute the scheduled jobs on realistic machine models with stochastically distributed break and repair times. As a result of our analysis, we obtain the probabilities for each job to be finished on time. This analysis is performed for each schedule, thus enabling us to compare the schedules in view of this kind of robustness criterion. Note that all the synthesized schedules are perfect in the hard real-time interpretation, but they are expected to differ in the more realistic stochastic interpretation.

Structure of the model. The MODEST model is composed of several parallel processes. Each scheduled job (modeling a product order) is a process and each machine (modeling a resource) is a parallel process as well. All jobs and machines are independent, i.e. they do not communicate among themselves, but jobs do communicate (synchronously) with machines, and vice versa (see Figure 4). Each job process may

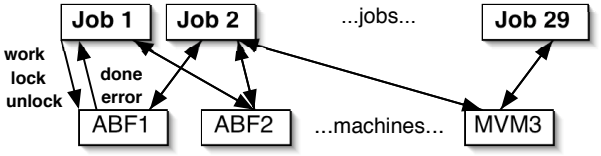

Figure 4. Structure of the MODEST model.

lock, *unlock* a machine, or give it a command to *work*. A machine process can produce a *done* or *error* action, so that the job process knows the result of performing a task.

Performance and availability factors. We intend to faithfully model the fact that the resources are not always available. This is achieved by a model where each resource may break and then can be repaired. For a specific resource, we use performance and availability factors p and a as an indication how often the breaks occur and how long it takes to repair it. The first question which arises in this context is whether a different treatment is needed for each of the factors. After studying the provided documentation, we came to the conclusion that the availability factor is conditional w.r.t the performance factor, so that by multiplying these two numbers we get the real performability factor of the machine, e.g. the fraction of time during which the machine works "full speed", on average.

With this decision, there is a close correspondence to the standard definition of availability given below:

$$p \cdot a = \frac{\text{mean up time (MUT)}}{\text{MUT+ mean time to repair (MTTR)}}$$

The case study documentation of a resource only contains the parameters p and a, but does not specify a quantity that could be interpreted as MUT or MTTR. For example, if a resource on average breaks every half a day and needs half a day to be repaired, it has the same factor $p \cdot a$ as the machine which on average breaks every week and needs another week to be repaired. To study the influence of this missing information, and to discuss its effect with the case study providers, we incorporate a *pace* parameter into the model that indicates how frequently events happen in the system. The pace may be viewed as the reciprocal of the basic time unit of our model, where the basic time unit is MUT+MTTR, so $pace^{-1} = \text{MUT} + \text{MTTR}$. We can thus derive $\text{MUT} = pace^{-1} \cdot p \cdot a$, and $\text{MTTR} = pace^{-1} \cdot (1 - p \cdot a)$. The final question that has to be answered before fixing the parameters of a resource model is how to represent these MUT and MTTR stochastically. Lacking any other specific information we chose to represent these delays by exponentially distributed random variables. Note that exponential distributions can be interpreted as the best guess (in the sense of maximizing the entropy) if only the mean of a distributions is available.

Resource model. In Figure 5 a schematic process for a resource (a filling line) is presented. This process waits for a command to start working. This happens when the variable `ABF1_work` is put to a non-zero value by a parallel process. This value represents the time during which the machine has to work. At this point we have already drawn a sample from the exponential distribution with the rate $r = 1/\text{MUT} = pace/(p \cdot a)$, where $p = 0.75$ and $a = 0.86$ are the two factors for the machine under consideration. This gives an average time of $1/r$ for the machine to break. If this time is larger than the time to work, then the process waits till the end of work is reached and then informs the other process that the job is done (act_done_ABF1). Otherwise, the process waits till the time to break and draws a sample from the exponential distribution to know the repair time. The mean value for this distribution is $pace/(1 - p \cdot a)$. After waiting for this time to repair, the process goes back to the working state. In the working state the work of the machine can be interrupted if the global time gets larger than

```
1  process ABF1_machine()  // Filling line 1
2  { clock w,y;
3    float br,r,work,deadline;
4    br=Exponential(pace/0.645) //next break time
5
6    do{::
7      when (ABF1_work>0) act_work_ABF1
8      {= work=ABF1_work, ABF1_work=0,
9         deadline=ABF1_deadline,
10        w=0 //current working time
11     =};
12     do
13     {::when (cjobs>=deadline) act_error_ABF1
14         {= ABF1_done=2, br-=w =}; break
15      ::when (br>=work && w>=work) act_done_ABF1
16         {= ABF1_done=1, br-=w =}; break
17      ::when (br<work && w>=br) act_break_ABF1
18         {= work-=br, y=0,
19            r=Exponential(pace/(1-0.645)) =};
20         when(y==r) act_repaired_ABF1
21         {          Exponential(pace/0.645) =}
22 } } }
```

Figure 5. A MODEST model of a resource.

the job deadline. We note here that this does not happen if the machine is being repaired.

Job model. The schedules derived from the synthesis with UPPAAL contain the precise times when each particular task of each job should start and when it finishes. There is some flexibility how to interpret these schedules in an environment where machines are occupied in a stochastic manner. In particular, a task may block a machine for less than the overapproximated time it has assigned, due to absence of breakdowns, for instance. This means that subsequent tasks can use the slack in the schedule, which is not apparent in the hard real-time interpretation. We therefore take a loose interpretation of the synthesized schedules. We interpret them as a totally ordered list of tasks, but only take into account the time when the whole job (the first task of that job) is scheduled to start, and also the time frame (the earliest starttime and the deadline) for the job execution. When executing a job we start a subsequent task immediately after the previous task has finished (if a machine is available), which means that we can even be ahead of the schedule being executed.

We use the global clock to know when to start the job (starttime parameter), and to know whether we have met the deadline or not at the end (of the job). A job process tries to lock a mixing vessel and then proceed to perform the tasks as they are shown on Figure 3. For each task it first locks the required machine, and then commands it to work. Then it awaits till the task is finished, and proceeds to the next task. At the end the job process compares the current time with the job deadline, and in this way determines if the deadline has been missed or not.

Synchronization. Apart from using action-based synchronization, another way to synchronize two processes in MODEST is by means of a shared variable and the `when` construction. The statement `when (ABF1_work>0) act_work_ABF1` means that the process waits till the value of the shared variable `ABF1_work` becomes larger than 0, and then executes `act_work_ABF1`. In this way another process just needs to assign a value bigger than 0 to `ABF1_work`, and the waiting process has to set it back to 0 afterwards. A symmetric synchronization is needed to make the synchronization in the other direction. By this kind of synchronization a data value is passed from one process to the other using the shared variable. Most of the synchronizations appearing in this case study use this feature.

```
1  //a filling line (and the mixing vessel) for 23
2  alt{
3   ::when(ABF1_lock==0) {= ABF1_lock=1,
4       ABF1_deadline=deadline-2, ABF1_work=23 =};
5     when(ABF1_done>0)
6      {= ABF1_done=0, ABF1_lock=0 =}
7   ::when(ABF2_lock==0) {= ABF2_lock=1,
8       ABF2_deadline=deadline-2, ABF2_work=23 =};
9     when(ABF2_done>0)
10     {= ABF2_done=0, ABF2_lock=0 =}
11 }
```

Figure 6. Implementation of locks in MODEST

Locks One of the ways to implement the locking mechanisms is based on test-and-set primitives [4, Page 43]. It has to do with the fact that the `when` construction is not an action, but a guard. This means that there is no interleaving between the `when` construction and the following action. The following action can have an assignment that performs the set part of test-and-set. As an example we show a part of the job process (Figure 6): Here the job tries to acquire the lock of one of the two filling lines. In case `ABF1_lock==0` it sets this lock (`ABF1_lock=1`) and no other job can access the value of `ABF1_lock` in the mean time. After that the work time and the deadline are set for `ABF1`, which is a command for the `ABF1` process to get out from the idle state. The order of the assignments is not important because all of them are executed in one atomic action.

It is important to mention here that this locking mechanism does not prevent a malicious or an incorrectly specified job from breaking the exclusivity constraint. A careful inspection of our model shows that, in fact, task executions are atomic even in this case, but, for example, if a job wants to lock a resource for two consecutive tasks, this cannot be guaranteed. Action synchronization with parameterized actions could provide a solution in this case.

It is also worthwhile to mention that the entire specification makes excessive use of nondeterminism induced by the `alt`-construct, but also by the interleaving semantics of the `par`-construct. This nondeterminism is a modeling means rather than a semantic entity, because the specification includes a schedule (generated by UPPAAL) which orders the tasks and jobs in a specific way, with jobs being bound to machines. Therefore, the nondeterminism is mostly "sched-

uled away", except for some cases, e.g. where scheduling decisions have to be made among multiple (identical) machines. We take a pragmatic approach to this issue, based on the discussion in Section 2.3. We apply a solution based on Bernoulli's principle of insufficient reason, which states that all events over a sample space should have the same probability unless there is evidence to the contrary. Practically, this means that the nondeterminism is actually treated as a probabilistic choice with a uniform distribution over the different alternatives, which is a maximum-entropy solution. It is realized by a simulator which is not biased, such that the different possibilities of choice are, on average, chosen with the same frequency.

4. Results

4.1. Experiments

Our primary goal is to evaluate the degree to which the schedules can tolerate the delays due to the breaks and repairs of the machines. We do evaluation by means of simulation, e.g. executing the schedules for many times and computing the average values of *performance variables*. We define the following performance variables. First of all we try to estimate the number of jobs meeting the deadlines (e.g. finishing before the due day). This number ranges from 0 to 29, so we are interested in the mean value as well as in the distribution of this performance variable among the values from 0 to 29. Another criterion is the success rate for each individual job. Here we want to estimate the probability of a particular job (with the number from 1 to 29) to finish before the deadline, for each particular schedule.

Following a suggestion of Kim Larsen [12], we are also investigating the total time that each particular machine is working, is being repaired, and is idle. We also estimate these for each of the schedules. We evaluated 20 schedules, 10 of them based on the plain processing times, 10 based on overapproximations, i.e. the extended processing times. We first experimented with a wide range of paces, but feedback from AXXOM made us focus on only two paces, corresponding to basic time units of 32 and 100 hours. Based on similar feedback we also distinguish two different deadline behaviors. We originally considered it wise to give up performing a job once it is known for sure that it will miss its due date. This makes resources available for future jobs, which can profit from this by starting tasks ahead of schedule. However, the case study providers considered it interesting, but unrealistic to abort a lacquer production with an unfinished product. Therefore we also consider all schedules by not giving up producing products even though their deadline has already passed. In total we simulated 80 experiments, each of which was executed for 20000 times.

The experiments were run on 4 Pentium 4 PC's. A total of 80 experiments were completed in 11 hours. The fastest machine (a PC running Linux with a Intel P4 3GHz CPU) completed one experiment in about 20 minutes. A running simulation took about 7MB of memory. As a result we obtained 80 files with the values of the performance variables. Postprocessing of these files was performed with python scripts and SQL queries to obtain the mean values and confidence intervals of the performance variables for each schedule. We further used GNU Plot and an office package to produce charts from the obtained data.

4.2. Evaluation

The experiments carried out with Motor and Möbius provide interesting insight into the scheduling problem. The first observation is that, both for "normal" and overapproximated schedule types, the probability distribution of the number of jobs finished on-time does not differ much from one particular schedule to another. This can be seen from Figure 7, where the significant portion of the probability distribution of the number of successful jobs is drawn for all 20 schedules. The chart on the left corresponds to the 10 normal schedules, and the chart on the right corresponds 10 overapproximated schedules. Each of the charts contains 4 clusters of lines, corresponding to the 2 paces and 2 kinds of deadline behaviors.

We recall that according to suggestions of the case study providers, overapproximated schedules were supposed to be more robust with respect to stochastic perturbations. However, as it turns out from the experiment, the normal schedules outperform the overapproximated ones, which is rather surprising at first sight. It can be explained by the fact that in the overapproximated schedules some jobs are scheduled to start late, because earlier jobs need to be scheduled longer. As a consequence, later jobs have less time remaining before the deadline. Note that in our model the jobs are not able to profit from slacks in a schedule, while the tasks are. As we can also see from the charts, the probability mass is shifted towards a lower number of jobs in the case when the job execution is continued, even if the deadline has already been missed. This can be explained by the fact that the "struggling" jobs continue to occupy resources, without improving their chance to reach the deadlines. It is also visible from Figure 7 that decreasing the pace (i.e., increasing both MTTR and MUT) makes the plot look steeper and thinner.

The probabilities of meeting the deadline for each individual job (for the case with pace 0.01 and without respecting the deadlines) are shown in the Figure 8. Here only the overapproximated schedules (11-20) are shown, because the differences among the schedules is slightly more significant in this case.

The jobs on the chart are ordered by their (earliest) starting time. This chart shows that most of the jobs with later starting times have less chance to finish on-time than the

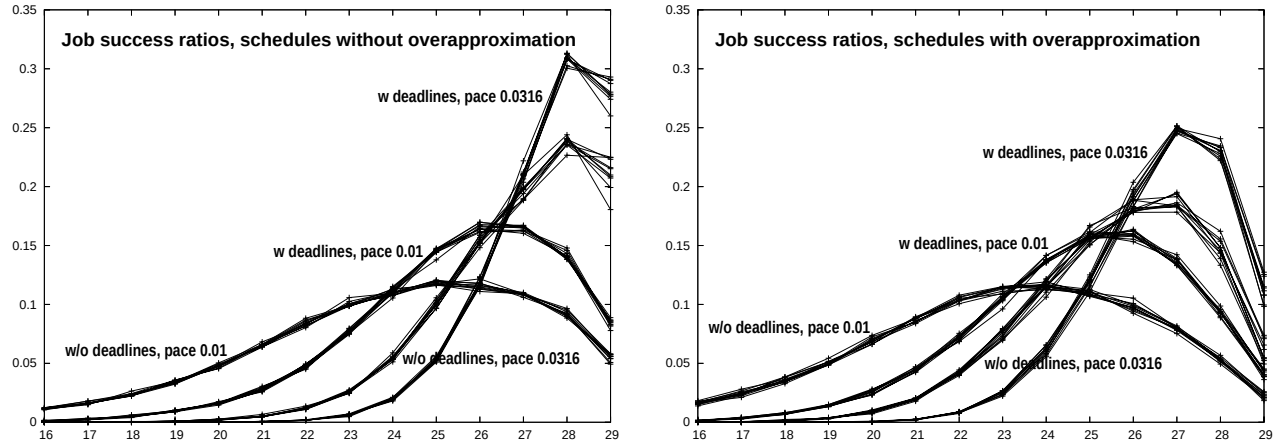

Figure 7. Success probabilities for a number of jobs.

jobs that can start earlier. This can be explained by the increase of stochastic perturbation of the model as time increases. Another observation concerns jobs of the same type. If two jobs of the same job type have close earliest starting time, one of them may have less chances to finish on-time, depending on the particular schedule (and the available resources). For instance, job 14 has only about 40% chance in the first three schedules, but about 73% by the following two. The situation is entirely reversed for job 18. A careful inspection of the schedules 11-15 reveals that the first three schedules launch job 18 before job 14, and the next two do the reverse. One can exploit this observation, e.g. if job 14 is a product for a more important customer than job 18, the schedules 14 and 15 are more preferable than schedules 11-13. Similar observations, though less significant, can also be made across jobs of different type, for example, jobs 20 and 19, or jobs 10 and 20.

As far as the average machine utilization/repair times are concerned, the results show that they do not differ at all from one schedule to another. Actually, this was to be expected, because the same amount of work is being done by all schedules and because the breaking behaviors of the machines is the same across the schedules.

5. Conclusions and Future Work

In this paper we have described a methodology to synthesize schedules, and to assess these schedules by means of simulation. We have used the tool UPPAAL and well-established model-checking techniques for the synthesis part, and MODEST/MOTOR/MÖBIUS for the simulation. The main contribution of this paper lies in the combination of real-time model-checking techniques and stochastic evaluation of results obtained by these techniques. To the best of our knowledge, this combination is unique. The MOTOR/MÖBIUS tandem provides excellent means to perform this kind of analysis, due to the fact that MODEST has strong roots in both worlds. Being an automata-based, compositional, easily extensible and modifiable formalism, MODEST enables a natural integration of the synthesized schedules and machine models into the simulation environment of MÖBIUS, in particular relative to more classical queuing-oriented tools such as QNAP.

The results of our simulation enabled us to detect which schedules are better and where the bottlenecks are (which jobs are risky), and to give feedback to the case study providers. We can also study which parameters influence the results and which do not. It became apparent that a naïve way to deal with performance and availability factors (i.e. overallocating processing times) as suggested and practiced by the case study provider does not lead to better results.

5.1. MODEST and MÖBIUS

This case study is the first touchstone for the MODEST/MÖBIUS tool tandem. Before the final evaluation of the presented model could take place, some practical obstacles had to be overcome. In the first version of the MODEST/MÖBIUS compiler, the generated C++ code was enormous: several 100.000 lines of code for a 700-line MODEST specification). Compilation of this code was quite difficult, since the used C++ compiler (g++ 3.3) uses a very memory intensive optimization algorithm. Actually, compilation was often infeasible doe to memory constraints. Since then the translation has been optimized so that the generated files became more than 10 times smaller.

Currently simulating MODEST models in the MÖBIUS simulator is somewhat inefficient since the generated files do not contain information about the interdependencies between actions and state variables. As a result the entire event queue needs to be re-examined after each action firing. Recent results from the same model including this dependency

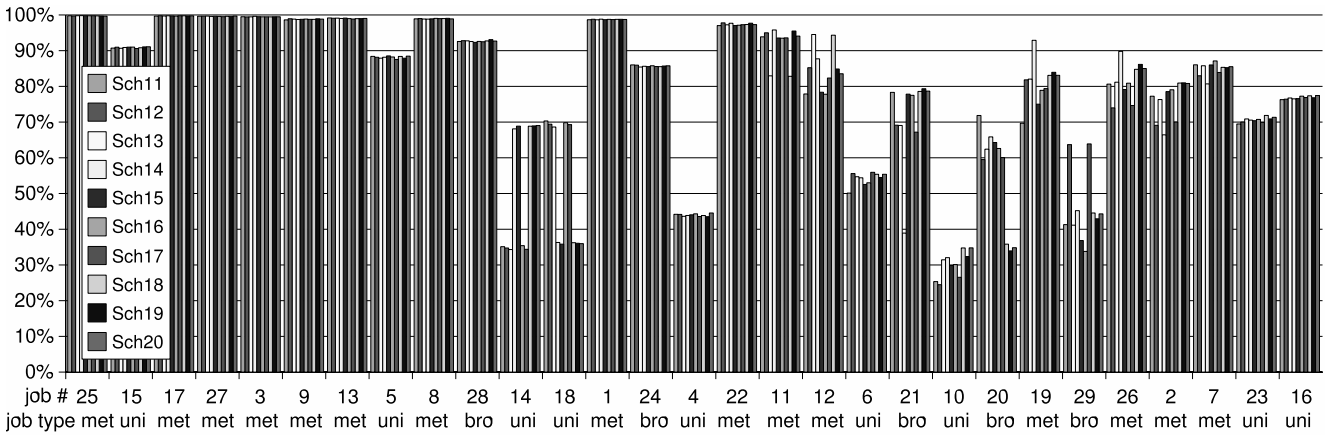

Figure 8. Success probabilities of the individual jobs.

information, generated with a development version of the compiler, indicate that the simulation can be done about 8 times faster and takes 3 times less memory.

In the future we plan to support more features for process parameterization and modularization. The current model could be substantially improved so that we do not repeat similar fragments of process definitions. One promising way to arrive there is based on an implementation of action synchronization augmented with value passing and matching mechanisms.

5.2. Model improvements

For a more consistent evaluation of the schedules a more realistic machine model is desirable. Mean up time, mean time to repair, factors like overall working time, working time since last repair, working time since last stop, overall idle time, idle time since last stop, etc., influence the breaking/repair time distributions of a machine. This information was not available, which forced us to assume simple exponential distributions as failure distributions, and varying the MUT and MTTR behavior with simple pace parameters. Our results are as good as these assumptions. To improve the results, a more sophisticated statistical analysis of a machine's breakdown behavior is needed, which is in the interest of the case study providers. Incorporating a more detailed description of a machines failure behavior into the MODEST model is not expected to pose problems.

The model can also be improved in a different direction by incorporating cost aspects. Using a machine can have a cost per time unit. Each repair could have an additional cost (also per time unit). Missing a job deadline also has a cost (depends on job type and the delay time). Meeting a deadline has a reward (constant per job type). We can also have storage costs if we finish earlier than the deadline (per time unit). In addition there can be costs for using the same equipment for different types of jobs, because the machines must be cleaned, etc. All these kind of costs can be conveniently incorporated into the model by means of the rate/impulse reward models that can be specified in MÖBIUS.

References

[1] Y. Abdeddaïm and O. Maler. Job-Shop Scheduling using Timed Automata. In *Proc. CAV'01*, 2001.

[2] R. Alur and D. Dill. A theory of timed automata. *TCS*, 138:183–335, 1994.

[3] AMETIST, IST project ist-2001-35304. `http://ametist.cs.utwente.nl/`.

[4] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall international series in computing science. Prentice Hall, 1990.

[5] H. Bohnenkamp, T. Courtney, D. Daly, S. Derisavi, H. Hermanns, J.-P. Katoen, V. V. Lam, and W. H. Sanders. On integrating the Möbius and MoDeST modeling tools. In *Proc. DSN'03*. IEEE Computer Society, 2003.

[6] H. Bohnenkamp, H. Hermanns, J.-P. Katoen, and R. Klaren. The MoDeST modeling tool and its implementation. In *TOOLS 2003*, LNCS 2794. Springer, 2003.

[7] G. Butazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[8] P. R. D'Argenio, H. Hermanns, J.-P. Katoen, , and R. Klaren. MoDeST - a modelling and description language for stochastic timed systems. In *Proc. PAPM-ProbmiV'01*, LNCS 2165, pages 87–104. Springer, 2001.

[9] A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Proc. RTCSA'99*. IEEE Computer Society Press, 1999.

[10] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994.

[11] C. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[12] K. G. Larsen. Personal communications, Dec. 2003.

[13] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. STACS'95*, LNCS 900. Springer, 1995.

[14] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic activity networks: structure, behavior and application. In *Proc. Int. Workshop on Timed Petri Nets*, pages 106–115, 1985.

[15] A. Movaghar and J. F. Meyer. Performability modeling with stochastic activity networks. In *Proc. Real-Time Systems Symposium*, 1984.

[16] UPPAAL home page. `http://www.uppaal.com`.