Model Checker Aided Design of a Controller for a Wafer Scanner*

Martijn Hendriks[1] Barend van den Nieuwelaar[2†]
Frits Vaandrager[1]

[1]*Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, The Netherlands*
{martijnh,fvaan}@cs.kun.nl

[2]*Department of Mechanical Engineering, Eindhoven University of Technology, The Netherlands*
N.J.M.v.d.Nieuwelaar@tue.nl

Abstract

For a case-study of a wafer scanner from the semiconductor industry it is shown how model checking techniques can be used to compute (i) a simple yet optimal deadlock avoidance policy, and (ii) an infinite schedule that optimizes throughput. Deadlock avoidance is studied based on a simple finite state model using SMV, and for throughput analysis a more detailed timed automaton model has been constructed and analyzed using the UPPAAL tool. The SMV and UPPAAL models are formally related through the notion of a stuttering bisimulation. The results were obtained within two weeks, which confirms once more that model checking techniques may help to improve the design process of realistic, industrial systems. Methodologically, the case study is interesting since two models (and in fact also two model checkers) were used to obtain results that could not have been obtained using only a single model (tool).
Keywords: Resource allocation systems, deadlock avoidance policy, throughput optimization, model checking, finite and timed automata, stuttering bisimulation.

1 Introduction

Scheduling and resource allocation problems occur in many different domains, for instance (1) scheduling of production lines in factories to optimize costs and delays, (2) scheduling of computer programs in (real-time) operating systems to meet deadline constraints, (3) scheduling of micro instructions inside a processor with a bounded number of registers and processing units, (4) scheduling of trains (or airplanes) over limited quantities of railway tracks and crossroads, and (5) mission planning for autonomous robots on spacecrafts. Typically, in each of these

*Supported by the European Community Project IST-2001-35304 (AMETIST).
†Part-time software architect at ASML, Veldhoven, The Netherlands.

domain problems are solved using different approaches and mathematical tools. The EU IST project Ametist (see `http://ametist.cs.utwente.nl/`) envisages a unifying framework for time-dependent behavior and dynamic resource allocation that crosses the boundaries of application domains.

In the Ametist approach, components of a system are modeled as *dynamical systems* with a state space and a well-defined dynamics. All that can happen in a system is expressed in terms of *behaviors* that can be generated by the dynamical systems; these constitute the semantics of the problem. Verification, optimization, synthesis and other design activities explore and modify system structure so that the resulting behaviors are correct, optimal, etc. Preferably, the limitations of currently known computational solutions should not influence modeling too much: only after the semantics of a problem is properly understood, abstractions and specialization due to computational considerations can intervene. In such situations, the soundness of abstractions should ideally also be proved, either via deductive verification or model checking.

The mission of Ametist is to extend this approach, which underlies the successful domain of *formal verification*, to resource allocation, scheduling and other time-related problems. The mathematical carrier for the Ametist methodology is the *timed automaton* model [2, 3], a modeling framework for discrete event dynamical systems that can handle quantitative timing delays between events. Some tools for *model checking* timed automata already exist. Model checking is a method for formally verifying dynamical systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model and to check (fully automatically) if the specification holds or not. We aim at further improving model checking tools for timed automata, investigating the applicability of these tools, and establishing links to tools developed in specific domains whenever appropriate.

In this paper, as an illustration of the Ametist methodology, we use model checking techniques to solve the deadlock avoidance and throughput optimization problems for a realistic case of a wafer scanner from the semiconductor industry.

A major concern in the design of controllers for many resource allocation systems (RASs) is *deadlock*, a permanently blocking condition. There are three general ways of handling deadlock: (i) deadlock prevention, (ii) deadlock detection and resolution, and (iii) deadlock avoidance. Deadlock prevention restricts the system in such a way that deadlock is a priori impossible. As a consequence, performance may be unnecessarily low. Deadlock detection and resolution, on the other hand, is not restrictive at all and detects and resolves a deadlock at run-time. This, however, may be very expensive. Deadlock avoidance achieves a middle ground; it dynamically chooses the control actions to avoid the occurrence of deadlock. In this paper, we show how a least restrictive deadlock avoidance policy (DAP) for the wafer scanner can be easily computed using SMV, a model checker for finite automata. This DAP can be represented by a very short predicate over the states of the wafer scanner, which can be used by the controller for the wafer scanner.

In addition, we use the timed automaton tool UPPAAL to define a refined model that adds timing constraints to address the issue of throughput optimization.

We relate the UPPAAL model to the SMV model via the concept of *stuttering bisimulation* introduced by [5]. Since stuttering bisimulation preserves validity of CTL formulas (without nexttime operator), all properties (and in particular the DAP) that we established for the untimed model using SMV, carry over to the UPPAAL model. It is not possible to compute the least restrictive DAP directly for the UPPAAL model since (a) UPPAAL does not support full CTL, and (b) the state space of the UPPAAL model is so big that it cannot be fully explored. However, using heuristics we are able to use the UPPAAL model checker to to find an infinite schedules that optimize throughput.

Contribution. We obtained our results within two weeks, which confirms once more that model checking may help to improve the design process of realistic, industrial systems. Our deadlock avoidance policy computation approach is referred to in a patent application of ASML, which shows its significance for industry. Methodologically, the case study is interesting since two models (and in fact also two model checkers) were used to obtain results that could not have been obtained using only a single model (tool). Our approach illustrates ones more that building models that are just abstract enough for addressing a specific question, often provides a way to deal with the state space explosion problem. The SMV and UPPAAL models are formally related through the notion of a stuttering bisimulation. We are not aware of other work that addresses both deadlock avoidance and throughput optimization in (what essentially is) a single framework.

Related work. Other papers in which model checking tools are used to solve scheduling problems include a case study in which a control schedule for a smart card personalization system is synthesized using the SMV model checker [10], and a case study in which the UPPAAL model checker is used to find feasible schedules for a steel plant [9]. The present work is a follow-up on [4], which considers the same example as the present paper and uses suboptimal deadlock prevention heuristics to generate schedules that are not guaranteed to be optimal. The present work, however, gives a least restrictive (and thus optimal) deadlock avoidance policy and a schedule that optimizes stationary throughput in the absence of errors.

Much research has been devoted to deadlock avoidance in RASs, see for instance [17, 18]. Discouraged by the NP-completeness of optimal deadlock avoidance for many RAS classes, see for instance [13], this kind of work generally focuses either on computation of suboptimal but polynomial DAPs or on optimal policies for very specific sub classes. Much of this work uses the Petri net formalism [16] for the modeling and analysis of RASs.

In [11] a deadlock free controller is constructed by an iterative process. The parallel composition of the controller and the plant is checked against deadlock by SMV. If a deadlock state is found, then the controller is adjusted to exclude the counterexample and the verification is run again. Otherwise, the controller is deadlock free. Finally, the work presented in [20] deals with verification of several DAPs using SMV.

Outline. First, Section 2 informally presents the case study. Section 3 then presents the SMV model and shows two ways of obtaining an exact characterization of the set of safe states using SMV. In Section 4, a UPPAAL model of the wafer scanner is proposed and infinite schedules which optimize throughput

are computed. Finally, Section 5 draws some conclusions and gives directions for future work. The full SMV and UPPAAL models are available at the URL `http://www.cs.kun.nl/ita/publications/papers/martijnh/`.

2 The EUV Machine

Lithographic machines, called *wafer scanners*, are used within the semiconductor industry to project chip designs on slices of silicon which are called wafers. A key performance characteristic of wafer scanners is throughput. In order to maximize throughput, a controller must have a strategy that optimizes throughput in the absence of errors. Moreover, no deadlock may ever arise since resolution of the deadlock means that the machine is off-line for a relatively long period (or at least has a suboptimal throughput).

Figure 1 schematically depicts a possible design of the wafer flow within an *Extreme Ultra Violet machine* (EUV machine), which is a particular type of wafer scanner that is currently being developed by ASML (EUV refers to the kind of radiation that is used to expose the wafers). The inside of an EUV machine is kept vacuum as EUV light is absorbed by air.

The wafer flow is as follows. First, the track robot (which is not shown) puts a wafer in one of the four locks. This lock is depressurized, and then the wafer is picked up by one of the two internal robots. Each internal robot has two arms that can each hold a wafer and that are opposite to each other. The internal robot turns and puts the wafer on the closest chuck, which is in the so-called "measure position". The wafer is measured and a chuck swap is performed. The chuck with the measured wafer now is in the "expose position" and the wafer is exposed. After another chuck swap, the exposed wafer is picked up by one of the internal robots which turns and puts it in a depressurized lock. After the lock has been pressurized, the track robot removes the exposed wafer from the machine. Each wafer thus has a fixed recipe for its route: lock - internal robot - chuck - internal robot - lock. There is a choice which locks, internal robots and chucks are used by a wafer. An obvious question that arises is why we not let the unexposed wafers flow through the upper two locks and let the exposed wafers exit through the lower two locks. In that case there are no crossing material paths which means that there is no deadlock possible. The answer is twofold. First, if locks are unidirectional then filling the machine from the initial, empty, state takes unnecessarily long. Second, if locks are unidirectional then the depressurization operation might become critical instead of the exposure, since depressurization takes more than twice as long as exposure. This

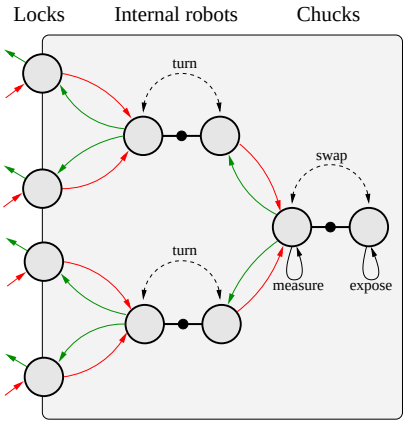

Figure 1: Wafer paths within the EUV machine.

is bad, since the lens which is used during the exposure is the most expensive part of the wafer scanner and must therefore have a maximal utilization. In Section 4, we will prove that indeed the exposure operation has maximal utilization in the design of Figure 1.

A typical example of a deadlock situation in the EUV machine would be a state in which all four robot arms hold unprocessed wafers, and both chucks hold processed wafers. A controller for the EUV machine should ensure that no such deadlock situation can ever be reached. The problem of finding such a control strategy is commonly referred to as the deadlock avoidance problem. The EUV machine is a disjunctive RAS according to the taxonomy of [14]. Instead of the traditional Petri net or graph based approaches to solving the deadlock avoidance problem, we will show in the next section how it can be tackled using the SMV model checker.

3 A Least Restrictive Deadlock Avoidance Policy

In this section, after a (very) brief introduction into SMV, we present our SMV model of the EUV machine, discuss how one can formalize the notion of deadlock as a temporal logic formula, and present the deadlock avoidance policy that we synthesized using SMV. The reader is referred to [7] and [15] for an extensive introduction into model checking and SMV.

3.1 SMV

In the approach supported by the SMV model checker, a system is modeled as a finite *transition system*, i.e. as a tuple $(S, s_{\text{init}}, \rightarrow)$ where S is a finite set of states, s_{init} is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation. We write $s \rightarrow s'$ instead of $(s, s') \in \rightarrow$. A state is defined as a valuation of a number of *state variables*. The value of state variable v in state s is denoted by $s(v)$. Furthermore, $s[v := c]$ denotes the state that is obtained by updating the value of v in state s to c. A *path* of a transition system is a sequence $s_0 s_1 s_2 \cdots$ such that for all i, $s_i \rightarrow s_{i+1}$. A state is *reachable* if it occurs on some path.

In SMV, specifications are described in *Computation Tree Logic* (CTL), a branching time temporal logic. Below some examples of CTL formulas are given, which should be sufficient to understand the present paper. The basic building blocks of CTL are *atomic formula*, which denote functions from the set of states to $\{true, false\}$. For instance, if p is a state variable, then $p = 2$ is an atomic formula, which denotes the function from states to $\{true, false\}$ that maps a state s to *true* iff $s(p) = 2$. In this case, we say state s *satisfies* formula $p = 2$, notation $s \models (p = 2)$. Every atomic formula is a *state formula*. State formulas can be combined with Boolean connectives and *path operators*. We show three path operators that are relevant for this paper. First, if ϕ is a state formula, then $\mathbf{AG}(\phi)$ also is a state formula. A state s satisfies $\mathbf{AG}(\phi)$, denoted by $s \models \mathbf{AG}(\phi)$, if for all paths $s_0 s_1 s_2 \ldots$ with $s = s_0$, and for all $i \geq 0$, $s_i \models \phi$. Second, if ϕ is a state formula, then $\mathbf{EF}(\phi)$ is also a state formula. We define $s \models \mathbf{EF}(\phi)$ if there exists a path $s_0 s_1 s_2 \ldots$ such that $s = s_0$ and $s_i \models \phi$, for some $i \geq 0$. Finally, if ϕ is a

```
module main ()
{
  -- state variables
  l : array 0..3 of {e,r,g};
  rb: array 0..1 of array 0..1 of {e,r,g};
  c : array 0..1 of {e,r,g};

  -- initialization
  for (i=0; i<4; i=i+1)
    init(l[i]):=e;
  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1)
      init(rb[i][j]):=e;
  for (i=0; i<2; i=i+1)
    init(c[i]):=e;

  -- system dynamics
  for (i=0; i<4; i=i+1)
    tl[i]: process entry_exit(l[i]);

  for (i=0; i<4; i=i+1)
    for (j=0; j<2; j=j+1)
      lr[i][j]: process move(l[i],rb[(i<2?0:1)][j]);

  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1)
      for (k=0; k<2; k=k+1)
        rc[i][j][k]: process move(rb[i][j],c[k]);

  for (i=0; i<2; i=i+1)
    exp[i]: process expose(c[i]);

}
```

```
module entry_exit (p)
{
  if (p=e)
    next(p):=r;
  else if (p=g)
    next(p):=e;
}

module move (lft,rgt)
{
  if (lft=r && rgt=e)
  {
     next(lft):=e;
     next(rgt):=r;
  }
  else if (lft=e && rgt = g)
  {
     next(lft):=g;
     next(rgt):=e;
  }
}

module expose (p)
{
  if (p=r)
     next(p):=g;
}
```

Figure 2: SMV model of EUV machine.

state formula, then $\mathbf{EG}(\phi)$ also is a state formula. We define $s \models \mathbf{EG}(\phi)$ if there exists a path $s_0 s_1 s_2 \ldots$ with $s = s_0$ such that for all $i \geq 0$, $s_i \models \phi$.

3.2 An SMV Model of the EUV Machine

The EUV machine can be modeled conveniently and concisely in SMV. In fact, the full code is displayed in Figure 2.

For each of the 10 positions in the machine our model contains a state variable: an array `l` of size 4 for the locks, a 2-dimensional array `rb` of size 2×2 for the robots, and an array `c` of size 2 for the chucks. These state variables can either take value `e` (*empty*), which means that the position is empty, value `r` (*red*), which means that the position is occupied by an unexposed wafer, or `g` (*green*), which means that the position is occupied by an exposed wafer. Initially, the machine is completely empty and all state variables have value `e`.

To model the system dynamics, i.e., the movement and exposure of wafers, we introduce 22 asynchronous processes, which are executed in an interleaving fashion:

- For each of the 4 locks `i` we have process `tl[i]`, which may either put an unexposed wafer in lock `i` if it is empty, or move an exposed wafer from

the lock to the track robot. In the definition of process `tl[i]` we use an auxiliary function `entry_exit` that describes the state change that results from running this process.

- For each of the 16 pairs of positions `i`, `j` such that `i` is on the left of `j` and a wafer can move directly from `i` to `j` (or back), we introduce a process that takes care of moving unexposed wafers from `i` to `j`, and exposed wafers from `j` back to `i`. In the definition of these processes we use a function `move(lft, rgt)` that describes the state change that results from moving a wafer from `lft` to `rgt` or vice versa.

- For each of the 2 chucks `i` we introduce a process `exp[i]` that models exposure of the wafer. An auxiliary function `expose` describes the state change that results from exposing the wafer at position `p`: the value of the corresponding state variable changes color from `r` (red) to `g` (green).

In the SMV model we abstract from the turning of internal robots. So a wafer can be picked up by both arms of an internal robot (possibly, the robot first has to turn). Similarly, the SMV model abstracts from chuck swaps and the measure operation. In Section 4, we present a more detailed model of the EUV machine in which we do not abstract from these aspects.

As it turns out, our SMV model has 57116 reachable states, which is close to the total number of states which equals $3^{10} = 59049$. An example of an unreachable state is one in which the machine is completely filled with exposed wafers. Transition systems of this size can very easily be handled by SMV and the computer hardware that is available today. In fact, SMV routinely handles systems with 10^{20} states and beyond, so we expect that our approach can also be applied to considerably larger designs.

3.3 Defining Deadlock and Safety in SMV

Standard textbooks on operating systems, e.g. [19], state four conditions for deadlock in systems that consist of *processes* that compete for *resources*. The first three conditions concern the model itself and are necessary, and the fourth condition concerns the states of the model and is necessary and sufficient when the first three are met:

1. *Mutual exclusion*: only one process may use a resource at a time.

2. *Hold and wait*: a process may hold allocated resources while awaiting assignment of others.

3. *No preemption*: no resource can be forcibly removed from a process that is holding it.

4. *Circular wait*: a closed chain of processes exists such that each process holds at least one resource needed by the next resource in the chain.

In the EUV machine, the wafers are the processes and they compete for the positions in the machine that constitute the resources. Clearly, the EUV machine satisfies the first three conditions for deadlock. The fourth condition, which thus is necessary and sufficient for deadlock, can be formalized with help from a *needs* function, that specifies for each wafer the set of positions it may move to. Let P denote the set of positions in the EUV machine. For $p \in P$ and $c \in \{\mathtt{r}, \mathtt{g}\}$, we define $\mathit{needs}(p, c) \subseteq P$ to be the set of positions (different from p) to which a wafer with color c at position p may move next. In particular, for p a chuck, $\mathit{needs}(p, \mathtt{r}) = \mathit{needs}(p, \mathtt{g}) = R$, where R is the set of positions of the internal robots. If s is a state and p a position then we use $\mathit{needs}^s(p)$ as an abbreviation for $\mathit{needs}(p, s(p))$. The circular wait property can now be defined as follows.

Definition 3.1 (Circular wait) *A state s has a* circular wait *in $Q \subseteq P$ iff, for all $q \in Q$,*

$$s(q) \neq \mathtt{e} \wedge \emptyset \neq \mathit{needs}^s(q) \subseteq Q \neq \emptyset.$$

It is not possible to directly formulate the circular wait property in terms of CTL, so some encoding is required. The basic idea is that the machine has a circular wait in a subset Q of positions iff the wafers in Q will never be able to move again. Observe that if in our model a transition $s \rightarrow s'$ moves a wafer from place p to place p', then p is empty in s'. Thus, the property that some wafer cannot move anymore can be formalized in CTL as follows.

Definition 3.2 (Jam) *A position p is* jammed *in state s iff $s \models \mathbf{AG}(p \neq \mathtt{e})$. A state s is* jammed *iff some position is jammed in s.*

Proposition 3.5 below asserts the equivalence of the circular wait and jammed properties, thereby providing us with a way to express deadlocks in CTL. In order to prove the proposition, we need two technical lemmas stating that (a) circular waits are preserved by the transition relation, (b) if a position p is jammed then also any position to which the wafer at p may move next is jammed. We prove Proposition 3.5 and the technical lemmas only for our model of the EUV machine, but from the proofs it should be clear that these results can be generalized to a whole class of resource allocation problems.

Lemma 3.3 *Suppose that state s has circular wait in Q and $s \rightarrow s'$. Then state s' has circular wait in Q.*

Proof. We consider three cases, corresponding to different types of transitions:

- If a process `entry_exit` takes a step, then this does not involve any position in Q: entry of a new wafer on positions in Q is not possible since all these positions are filled; also exit of a wafer in Q is not possible since for all positions in $q \in Q$ we have $\mathit{needs}^s(q) \neq \emptyset$. Since none of the variables in Q is modified, the fact that s has circular wait in Q implies that also state s' has circular wait in Q.

- Also if a process `move` takes a step then this does not involve any position in Q: entry of a new wafer on positions in Q is not possible since all these positions are filled; also exit of a wafer in Q is not possible since for all positions in $q \in Q$ we have $needs^s(q) \subseteq Q$. Hence the circular wait property is preserved by the transition.

- If a process `expose` takes a step, then this does not effect emptiness of positions, nor the value of the *needs* set. Hence the circular wait property is preserved by the transition, and also s' has circular wait in Q.

■

Lemma 3.4 *Suppose position p is jammed in state s and $p' \in needs^s(p)$. Then position p' is jammed in s.*

PROOF. By contradiction. Suppose p' is not jammed. Then there exists a path on which eventually p' is empty. If in this path, directly after p' becomes empty, we schedule a transition that empties p (this is possible since $p' \in needs^s(p)$), we obtain a path in which eventually p is empty. But we assumed no such path exists. Contradiction. ■

Proposition 3.5 *A state has a circular wait in some Q if and only if it is jammed.*

PROOF.

$\Rightarrow$ Assume that state s has a circular wait in Q. Pick an element $q \in Q$ (this exists since s has circular wait in Q). By Lemma 3.3, any state s' reachable from s in zero or more steps has circular wait in Q. Hence, $s' \models (q \neq \mathtt{e})$. It follows that $s \models \mathbf{AG}(q \neq \mathtt{e})$. Therefore, state s is jammed.

$\Leftarrow$ Assume that state s is jammed. Then there exists a position q such that q is jammed in s. Define q to be the least fixed-point $\mu Q(\{q\} \cup needs^s(Q))$. Then, by construction, $needs^s(q) \subseteq Q \neq \emptyset$. By Lemma 3.4, using an inductive argument, it follows that all positions in Q are jammed in s. This implies in particular that, for all $q \in Q$, $s(q) \neq \mathtt{e}$ and $needs^s(q) \neq \emptyset$ (the latter inequality follows because if $needs^s(q) = \emptyset$ this implies that q is a lock that is filled with an exposed wafer, so q can be emptied in a single transition, which contradicts the assumption that q is jammed). It now follows that state s has a circular wait in Q.

■

In the remainder of this paper, we will say that a state is *deadlocked* if it has circular wait, i.e., if it is jammed. The question that we need to answer is whether and how we can prevent the system of entering a deadlocked state. In Dijkstra's paper on the *banker's algorithm* [8], the first published deadlock avoidance algorithm, a state is defined to be *safe* if "all processes can be run to

completion". In our case, the wafers are the processes and "a wafer is run to completion" if it exits the machine. Thus, Dijkstra's definition can be translated to CTL as follows.

Definition 3.6 (Safe states) *A state s is* safe *iff* $s \models \mathbf{EF}\left(\bigwedge_{p\in P}(p = \mathsf{e})\right)$.

Note that in general safe and not being deadlocked are different things. If a state s is not deadlocked then $s \models \bigwedge_{p\in P} \mathbf{EF}(p = \mathsf{e})$, i.e., each individual position can be emptied, but it need not be the case that all positions can be emptied simultaneously. If a state is deadlocked it is unsafe, but if it is unsafe it need not be deadlocked. However, in many cases and (according to SMV) in particular for our model of the EUV machine, the following property does hold[1]:

$$\mathbf{AG}(\text{safe} \iff (\mathbf{EG}\ \neg\text{deadlock})). \tag{1}$$

This formula suggests a simple least restrictive deadlock avoidance policy (DAP): just keep the system in a safe state. This policy can be realized for the EUV machine. Every non-initial safe state has at least one safe successor (different from itself), otherwise it would not be not possible to return to the initial state. In addition, we verified using SMV that all successors of the initial state are again safe.

3.4 A Least Restrictive DAP

In order to actually build a controller that always keeps the system in a safe state, it would clearly be very helpful to have a simple, yet exact characterization of the set of safe states. We see two ways to obtain such a characterization.

1. When checking whether the initial state is safe, SMV computes a *binary decision diagram* (BDD, see [6]) which provides a compact representation of the set of safe states. With the available SMV releases it is not possible to get the BDD out. However, since there is an open-source distribution available solving this problem should just be a matter of programming.

2. The set of safe states can be manually characterized by the following iterative procedure:

   $$\begin{array}{l} S := \mathit{true} \\ \textbf{while}\,(s_{\text{init}} \not\models \mathbf{AG}(\text{safe} \iff S)) \\ \qquad S := S \wedge (\neg C) \end{array}$$

   where C is the characterization of the last state of the counter example that is generated by SMV.

[1] In fact, in the EUV machine a state is safe if and only if it has no deadlock. It is easy to come up with variations of the machine with states that are not safe and not deadlocked, for example a design in which the internal robots only have one arm. In such cases, in order to make formula (1) hold, we need to require weak fairness for all processes in the SMV model to exclude runs in which no progress is made due to infinite stuttering of some components.

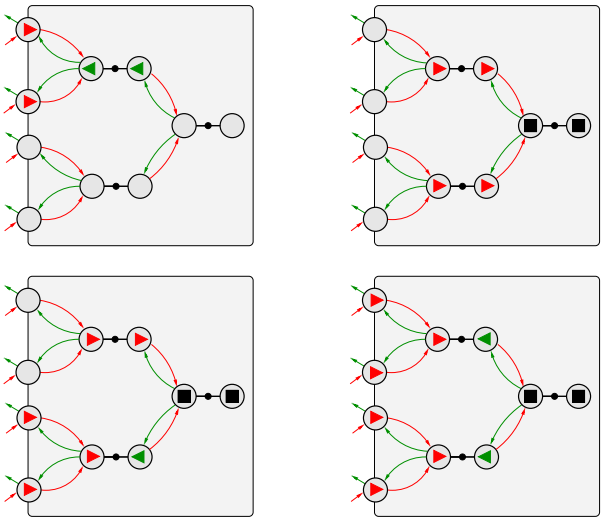

Figure 3: The four possible unsafe scenarios (modulo symmetry) in the EUV machine.

The first approach enables a least restrictive DAP with linear time complexity, since checking whether a state is included in a BDD takes $\mathcal{O}(n)$ operations, where n is the number of booleans from which the BDD is composed (20 in case of the EUV machine). The size of the BDD, however, can in the worst case be exponential in the number of booleans. A second drawback is that it can be difficult to derive individual unsafe and/or deadlock situations from a BDD, which may be required during the design phase of the system. The second approach can quickly become practically infeasible since all unsafe states are explicitly enumerated. If it is carried out manually, however, then it might be possible to abstract from irrelevant state information and to visualize the various unsafe situations in the system. This technique has been used to characterize the safe states of the EUV machine. With five iterations, we found four unsafe situations, depicted in Figure 3, which happen to characterize all deadlocks. A right-pointing arrow represents a red wafer, a left-pointing arrow represents a green wafer, and a black square represents a red or green wafer. The predicate S that exactly characterizes the set of safe states is the negation of the situations shown in Figure 3, and which are described by predicates `d1`, `d2`, `d3` and `d4` in Figure 4.

4 Throughput Analysis

A first objective for a controller of the EUV machine is to avoid deadlocks. In the previous section, using our SMV model, we synthesized a least restrictive control policy that achieves this. A second key objective for a controller of the machine of course is to maximize throughput. Our SMV model is not sufficiently detailed to address this issue since, for instance, relevant information about the delays in the locks and the speed of the robots has not been included. Also, the SMV model abstracts from the delays due to turning of the internal robots, measuring

```
#define empty (l[0]=e & l[1]=e & l[2]=e & l[3]=e & rb[0][0]=e & rb[0][1]=e &
               rb[1][0]=e & rb[1][1]=e & c[0]=e & c[1]=e)

#define safe (EF(empty))

#define d1a (l[0]=r & l[1]=r & rb[0][0]=g & rb[0][1]=g)
#define d1b (l[2]=r & l[3]=r & rb[1][0]=g & rb[1][1]=g)
#define d1 (d1a | d1b)

#define d2 (~c[0]=e & ~c[1]=e & rb[0][0]=r & rb[0][1]=r & rb[1][0]=r & rb[1][1]=r)

#define d3a (~c[0]=e & ~c[1]=e & rb[0][0]=r & rb[0][1]=r &
             ((rb[1][0]=r & rb[1][1]=g) | (rb[1][0]=g & rb[1][1]=r)) & l[2]=r & l[3]=r)
#define d3b (~c[0]=e & ~c[1]=e & rb[1][0]=r & rb[1][1]=r &
             ((rb[0][0]=r & rb[0][1]=g) | (rb[0][0]=g & rb[0][1]=r)) & l[0]=r & l[1]=r)
#define d3 (d3a | d3b)

#define d4 (~c[0]=e & ~c[1]=e & ((rb[0][0]=r & rb[0][1]=g) | (rb[0][0]=g & rb[0][1]=r)) &
            ((rb[1][0]=r & rb[1][1]=g) | (rb[1][0]=g & rb[1][1]=r)) &
            l[0]=r & l[1]=r & l[2]=r & l[3]=r)

safe_iff_p0: SPEC AG( safe <-> 1);
safe_iff_p1: SPEC AG( safe <-> ~(d1) );
safe_iff_p2: SPEC AG( safe <-> ~(d1 | d2) );
safe_iff_p3: SPEC AG( safe <-> ~(d1 | d2 | d3) );
safe_iff_p4: SPEC AG( safe <-> ~(d1 | d2 | d3 | d4) );
```

Figure 4: SMV characterization of the set of safe states.

of wafers, and swapping of the chucks. Therefore, in this section, we present a more refined *timed automata model* ([2, 3]), which contains sufficient information to address the throughput issue.

In order to define and analyze our model, we used the model checking tool UPPAAL. UPPAAL supports modeling of systems in terms of networks of timed automata which are extended by blocking synchronization and bounded integer variables. Similarly to SMV, the semantics of a UPPAAL model is defined by a transition system. In addition to the discrete part, the states also contain a real-valued clock valuation. For these models, the UPPAAL model checker can decide a subset of *Timed Computation Tree Logic* (TCTL, see [1]). For a detailed account of UPPAAL we refer to [12] and to `http://www.uppaal.com`.

After presenting the UPPAAL model of the EUV machine in Section 4.1, we discuss the relationship between the UPPAAL and SMV models in Section 4.2. Then, in Section 4.3, we use UPPAAL to derive a schedule for the EUV machine that optimizes throughput.

4.1 UPPAAL Model

The UPPAAL model of the EUV machine contains the same state variables as the SMV model for the positions in the machine: arrays `l`, `rb` and c, which may take the same values `e`, `r` and `g` to indicate that a position is respectively empty, filled with an unexposed wafer, or with an exposed wafer.

In addition, the UPPAAL model has a number of Boolean state variables to ensure "physical integrity":

```
const H     1480;    const S       260;    const PRES    120;    const DEPRES  670;
const LOAD    25;    const UNLOAD   25;    const L2R_T    23;    const R2L_T    23;
const R2C_T   40;    const C2R_T    54;    const TURN     10;    const MEAS    140;
const EXPO   250;    const SWAP     10;    const TR1      50;    const TR2      50;
```

Figure 5: Timing parameters in the UPPAAL model.

- For each lock `id` there is a Boolean `lbt`[`id`] which is *true* iff either pressure in the lock is not atmospheric or in case a track robot is busy loading or unloading a wafer.
- Similarly, for each lock `id` there is a Boolean `lb`[`id`] which is *true* iff either the lock is not vacuum or in case an internal robot is busy loading or unloading a wafer.
- For each chuck `id` there is a Boolean `cb`[`id`] which is *true* iff either an internal robot is accessing chuck `c`[`id`] or an internal robot may not access chuck `c`[`id`] because it is not in location `measure` (i.e., it is busy with something).

The model consists of 12 automata, of which 11 model physical components of the machine: the track robot, the four locks, the four robot arms (two for each of the robots), and the two chucks. These automata move wafers around with certain delays and according to the material paths as specified in Section 2. An additional automaton, the *observer*, is used to observe when exposed wafers exit the system (*unload events*). Within the model a number of timing parameters are used. Figure 5 lists the values for these parameters that were selected for the design of the EUV machine. Below, the individual templates of the model are explained. All templates have a local clock x. If a template is called a "process", then this template has no parameters.

Figure 6 shows the track robot process. Initially, the track robot is ready to load a wafer to a lock. From its initial location, the track robot may move instantaneously to a location where it is ready to unload a wafer from a lock, but the reverse transition takes time *TR1*. When the track robot is ready to load, it may actually start loading a wafer to one of the four locks, provided the lock is empty and has atmospheric pressure. Similarly, when the track robot is ready to unload, it may start to unload a wafer from one of the locks, provided the lock contains a processed wafer and has atmospheric pressure. Upon finishing an unload operation the track robot synchronizes over the channel *unload* with the observer, and after *TR2* time units returns to its initial state.

Figure 7 shows the UPPAAL template for a lock. It has one parameter `id`, that gives the identity of the locks. Initially, a lock has atmospheric pressure. A lock may then start depressurizing provided the track robot is not busy with it. Similarly, if a lock is at vacuum pressure, it may start pressurizing if the internal robot is not busy with it.

There are two internal robots in the system, each equipped with two arms. Initially, one arm points at the chucks and the other arm points at the locks. An internal robot may turn, which interchanges the positions of the arms. Figures 8

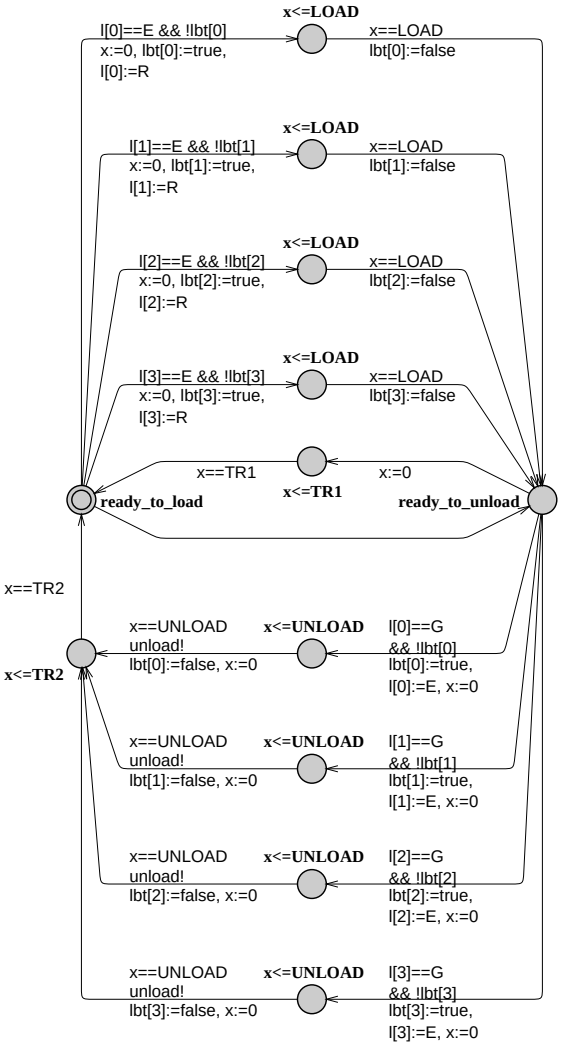

Figure 6: Process for the track robot.

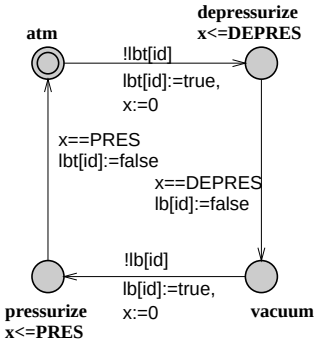

Figure 7: Template for a lock.

and 9 show the templates for the two types of robot arms. These templates have four parameters: a constant *id* that identifies the internal robot to which the arm belongs, two constants *l0* and *l1* that identify the locks to which the robot arm has access, and a channel *turn*. Note that the templates have different initial states, and that the turn time $TURN$ is only measured by the template for robot arm 0. When a robot arm is at the locks, then it can get a wafer from a lock (*L02R* and *L12R*), or it can put a wafer in a lock (*R2L0* and *R2L1*). Of course, it can only perform these actions if the lock is at vacuum pressure, and if the wafer flow is as specified in Section 2. Similarly, when a robot arm is at the chucks then it can load/unload a wafer to/from the chuck that is at the *measure* location. The *cb* variables are used to ensure that only one robot arm has access to the chuck at a time and that the chuck cannot execute a transition while the robot arm is loading/unloading a wafer.

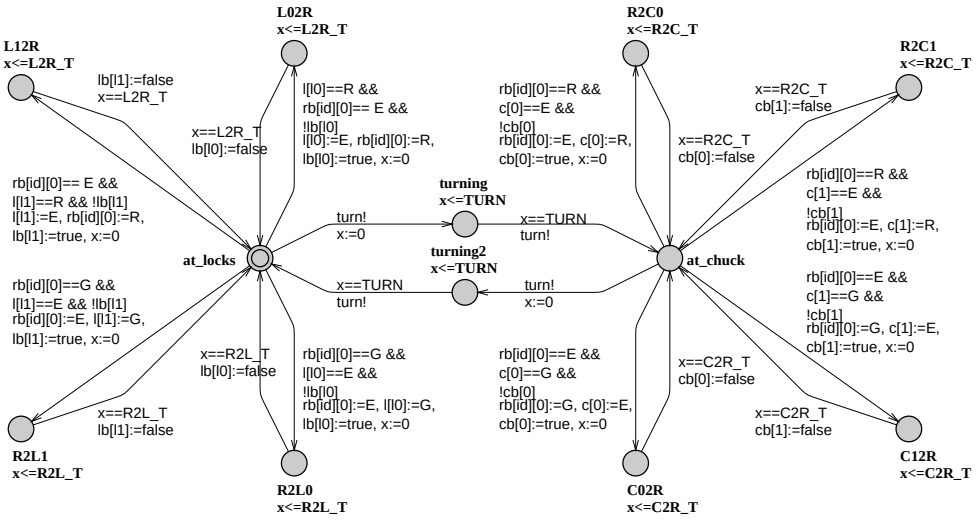

Figure 8: Template for a robot arm 0.

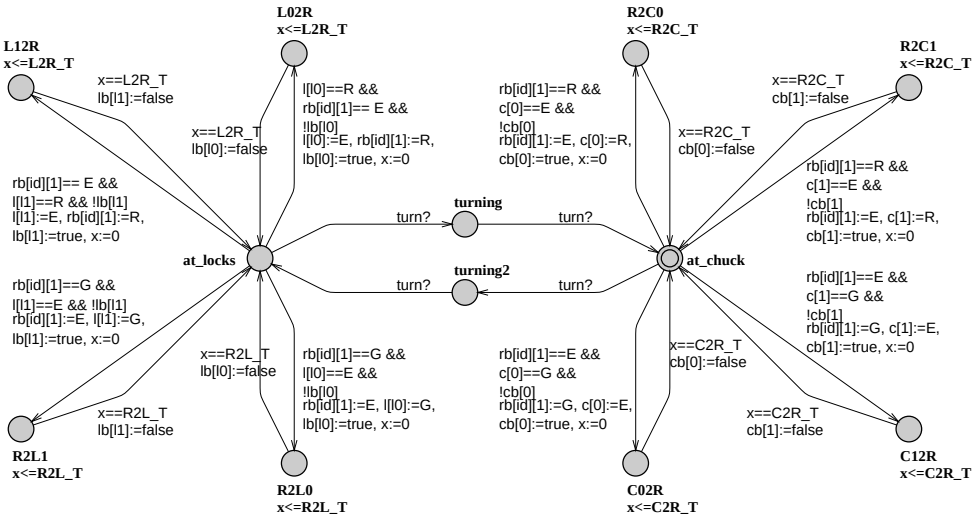

Figure 9: Template for a robot arm 1.

Figures 10 and 11 show the UPPAAL processes for Chuck 0 and Chuck 1 respectively. Chuck 0 is initially in the "measure" position while Chuck 1 initially is

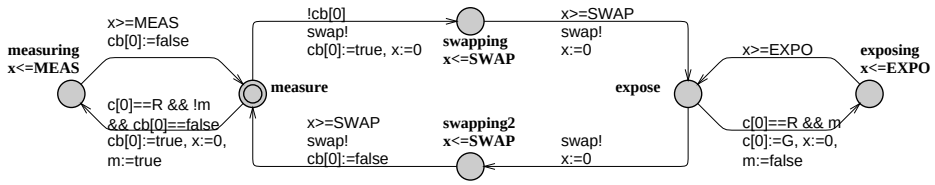

Figure 10: Process for chuck 0.

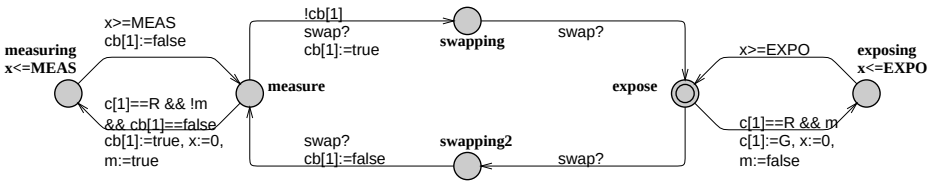

Figure 11: Process for chuck 1.

in the "expose" position. The chucks can simultaneously swap by synchronization over the channel *swap*. Note that chuck 0 measures the time that is needed for swapping. The *cb* variables are used by the chucks and the robot arms to prevent faulty behavior: (i) a robot can only access a chuck if it is in the measure position and not measuring (thus, the chuck must be in location *measure*), and (ii) when a robot is accessing a chuck, then the chuck may not perform any transitions. Each chuck has a local Boolean variable `m` which is *true* iff there is a measured wafer on the chuck; only a measured wafer can be exposed.

Figure 12 shows the observer process which, as we will explain in more detail in Section 4.3, is used to ensure progress in the model. This process measures the time until the first unload event in location *L0*, and the time between two unload events in location *L1*.

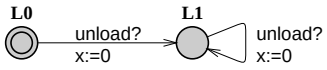

Figure 12: Process for the observer.

4.2 Bisimulation between SMV and UPPAAL models

Clearly, there is a relationship between the SMV model and the UPPAAL model. The SMV model is an abstraction from the UPPAAL model, which has the property that every transition in the UPPAAL model can be simulated in the SMV model, and vice versa. Formally, the relationship between the two models can be expressed as a *stuttering bisimulation* relation in the sense of [5]. Stuttering bisimulations are defined in terms of *Kripke structure*, an extension of transition systems in which to each state a set of atomic propositions is associated that hold in that state.

Definition 4.1 (Kripke Structures) *Let* **AP** *be a set of* atomic proposition symbols*. A Kripke structure is a structure* $(S, s_{init}, \rightarrow, l)$*, where* $(S, s_{init}, \rightarrow)$ *is a transition system and function* $l : S \rightarrow 2^{\mathbf{AP}}$ *associates to each state a set of atomic proposition symbols.*

In this paper, we let **AP** be the set of equations of the form $p = v$, where p is a position in the EUV machine and $v \in \{\mathtt{e}, \mathtt{r}, \mathtt{g}\}$. For the transition systems induced by the SMV and UPPAAL models, the labeling is obvious: we label a state s with $p = v$ iff this equation holds in s. For the SMV model the labelling function

is injective: different states have different labels. For the UPPAAL model this is clearly not the case.

A stuttering bisimulation relates states from two Kripke structures. Initial states are related, and related states are labeled with the same proposition symbols. If two states are related and from one state a transition is possible, then it should be possible to simulate this transition from the related state, after first doing zero or more *stuttering transitions*, i.e., transitions that do not change the labeling.

Definition 4.2 (Stuttering Bisimulation) *A* stuttering bisimulation *between Kripke structures* $(S, s_{init}, \rightarrow, l)$ *and* $(S', s'_{init}, \rightarrow', l)$ *is a relation* $R \subseteq S \times S'$ *such that*

1. $(s_{init}, s'_{init}) \in R$,
2. *If* $(r, s) \in R$ *then* $l(r) = l(s)$,
3. *if* $(r, s) \in R$ *and* $r \rightarrow r'$ *then there exist, for some* $n \geq 0$, $s_0, s_1, \ldots, s_n$ *such that* $s_0 = s$ *and, for all* $i < n$, $s_i \rightarrow' s_{i+1}$, $(r, s_i) \in R$ *and* $(r', s_n) \in R$.
4. *if* $(r, s) \in R$ *and* $s \rightarrow s'$ *then there exist, for some* $n \geq 0$, $r_0, r_1, \ldots, r_n$ *such that* $r_0 = r$ *and, for all* $i < n$, $r_i \rightarrow r_{i+1}$, $(r_i, s) \in R$ *and* $(r_n, s') \in R$.

Proposition 4.3 *Consider the projection function* π *from states of the Kripke structure induced by the* UPPAAL *model to states of the Kripke structure induced by the* SMV *model. Function* π *only preserves the values of the arrays* `l`, `rb` *and* `c`. *Let* R *be the relation consisting of pairs* $(s, \pi(s))$, *for* s *a reachable state from the* UPPAAL *model. Then* R *is a stuttering bisimulation between the* UPPAAL *and* SMV *Kripke structures.*

PROOF (SKETCH). Function π maps the initial state of the UPPAAL model, in which the machine is completely empty, to the initial state of the SMV model.

By definition π, and hence R, preserves labeling of states.

Transfer property (3) follows by inspection of all the transitions in the UPPAAL model: each transition either does not affect the labeling, in which case it can be simulated by a stuttering transition in the SMV model, or it does affect the labeling but then a process in the SMV model is enabled that results in the same change of labels.

Proving transfer property (4) is somewhat more involved.

We need a number of auxiliary invariants on the UPPAAL model. These include the integrity constraints mentioned at the beginning of Section 4.1 that restrict the values of the Booleans `lbt`[`id`], `lb`[`id`] and `cb`[`id`]. Also, we need some obvious invariants that relate the locations of connected robot arms, and the locations of the two chucks. The full set of invariants is listed in the file `EUV-invariants.q` at `http://www.cs.kun.nl/ita/publications/papers/martijnh/`. The state space of the UPPAAL model is too big too establish these invariants directly. However, we were able to prove them automatically for an abstraction of the model

in which we remove all clock variables, the arrays `l`, `rb` and `c`, as well as all references to these variables in transitions and locations. This is a valid abstraction in the sense that each invariant of the abstract model also holds for the original full UPPAAL model.

A key observation in the proof of transfer property (4) is that from any reachable state of the UPPAAL mode we can drive the system back to its initial state — except for the values of arrays `l`, `rb` and `c`, the values of the local clocks `x`, and the value of the `m` variables — by doing stuttering steps only. More specifically, let ρ be the mapping from states of the SMV model to states of the UPPAAL model such that, for any r, in state $\rho(r)$ the values of arrays `l`, `rb` and `c` are equal to the values in r, and all locations and Boolean variables have their initial value, except the two `m` variables, which are *true* iff the corresponding chuck contains an unprocessed wafer. Then we claim that, for any reachable state s of the UPPAAL model with $\pi(s) = r$, there exists a path with only stuttering steps from s to a state which, up to the values of local clocks, is equal to $\rho(r)$.[2] To see why this claim holds, first observe that each process in a non-quiescent location (a location with a nontrivial invariant) may evolve to a quiescent state by some stuttering transitions with guards that only refer to the local clock `x`, and (possibly) with synchronization output labels (!) for which a corresponding input (?) is always enabled. After all processes have reached a quiescent state we can, one by one, drive each process back to its initial location:

1. The track robot only has two quiescent locations: ready to load and ready to unload. Via two successive internal transitions, we can drive the track robot from ready to unload to ready to load in time `TR1`.

2. Each lock has two quiescent locations corresponding to atmospheric and vacuum pressure. If a lock `id` is vacuum then, since the robotarms are in a quiescent state, due to the invariants, `!lb[id]`. Hence we can drive the lock to its initial location (atmospheric pressure) by to successive transitions in time `PRES`.

3. In order to bring the robot arms to their initial location, we may need to turn them around. The invariants for the robot arms imply that we can bring all arms in their initial location simultaneously.

4. For the chucks, we also have to ensure that `m` is *true* in case a chuck contains an unprocessed wafer. This can be achieved by driving the automaton through the measuring location, which may require swapping of the chucks. After the `m` variables have been set to the appropriate value, me may need to swap the chucks again. The invariants for the chucks imply that we can bring both chucks to their initial location.

If all processes are in their initial location, then the invariants imply that also the Boolean arrays `lb`, `lbt` and `cb` have their initial values. From this the claim follows.

[2]Technically, the values of the clocks are irrelevant ("inactive") in the initial locations, and UPPAAL also abstracts from their value.

Next, we claim for any state r from the SMV model that if s enables some transition, this can be simulated from $\rho(r)$, possibly after some stuttering steps. This follows from a routine case distinction. For instance, a transition moving a wafer from a lock `0` to an internal robot can be simulated by first depressuring lock `0`, possibly turning the robot arm, and then moving the wafer to the robot via the transition to location `L02R`. We leave it to the reader to check the details of all the cases. ■

The significance of the above result stems from the fact that validity of CTL formulas without nexttime operator (i.e. all the formulas used in this paper) is preserved by stuttering bisimulation equivalence (see [5]). Thus all the results on deadlock avoidance established using SMV in Section 3 carry over to the UPPAAL model. It is not possible to obtain these results directly using the UPPAAL tool since (a) UPPAAL does not support full CTL, and (b) the state space of the UPPAAL model is so big that it cannot be fully explored.

4.3 Finding an Optimal Schedule

As mentioned above, the *observer* process of Figure 12 observes unload events. It starts in location *L0* and upon the first unload event it resets its local clock x and enters location *L1*. In location *L1* the clock is reset whenever an unload event takes place. The observer is used to find an infinite schedule that takes at most H time units until the first unload event, and that has at most S time units between two unload events. Such a schedule is specified by the following TCTL property that can be checked by UPPAAL.

$$\mathbf{EG}((observer.L0 \Rightarrow observer.x \leq H) \wedge (observer.L1 \Rightarrow observer.x \leq S)) \quad (2)$$

If this property is satisfied, then UPPAAL can return an example execution that consists of a path followed by a cycle. Such an execution thus gives an infinite control schedule for the wafer scanner with a *stationary* throughput of at least one wafer per S time units. Unfortunately, the size of the reachable state space prevents UPPAAL from finding such an execution directly. We therefore added heuristics to the model to prune the state space:

1. The DAP derived in the previous section has been used to avoid unsafe material configurations of the machine.

2. Some transitions are useless (or suboptimal) in certain states, e.g., an internal robot can always turn, but this is useless if it does not hold wafers. The state space has been reduced by adding guards that prevent such useless behavior.

3. The optimal behavior of the locks in the initial phase (the filling of the machine) differs from their optimal behavior in the stationary phase. Therefore a heuristic has been added to enforce this difference: a lock can pressurize when it contains either an exposed wafer, or it is empty and the machine is not yet filled with enough wafers to be in the stationary state.

4. Some transitions have been made *urgent* (greedy): they must be taken as soon as they are enabled. For instance, if the DAP allows loading a wafer to a lock, then this must be done immediately.

Note that using urgent transitions without the DAP may be an unwise idea, since this can result in many deadlocks with the effect that an execution satisfying Property 2 does not exist anymore in the model. Also note that at least the last three heuristics may remove good schedules.

A lower bound on the time until the first unload event, min_h, can easily be derived from the model. It is also easy to see that the minimal separation time between exposed wafers that appear at the chuck that is in the measure position (and can therefore be picked up by an internal robot) equals $min_s = EXPOSE + SWAP$, where the former is the time needed for the expose operation and the latter is the time needed for the chuck swap. Therefore, the theoretical maximal stationary throughput of the machine is at most one wafer per min_s time units. For the UPPAAL model with heuristics it is possible to find an execution that satisfies Property 2 for a value of H that is 5% larger than min_h and for $S = min_s$. Figure 13 shows this schedule that optimizes the stationary throughput of the EUV machine.

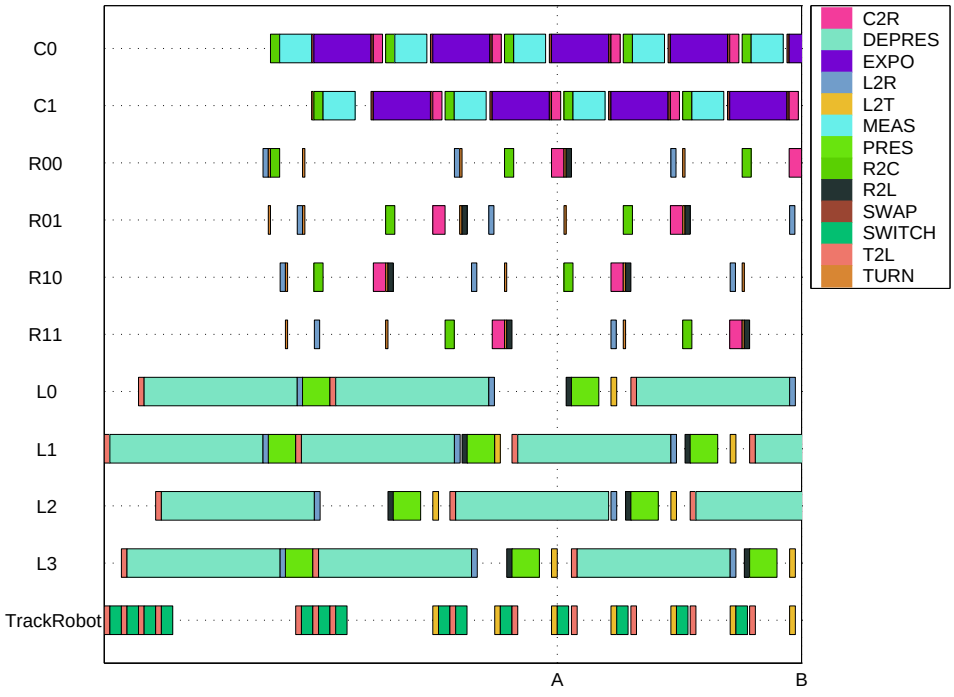

Figure 13: A schedule that optimizes the stationary throughput of the EUV machine. The cyclic part of the schedule consists of the interval between points A and B. The operation of the chucks is critical in the cyclic part.

Two other machine lay-outs have also been investigated w.r.t. throughput. First, the incoming wafers have been restricted to the upper two locks and the outgoing wafers to the lower two locks (to prevent deadlock a priori; see Sec-

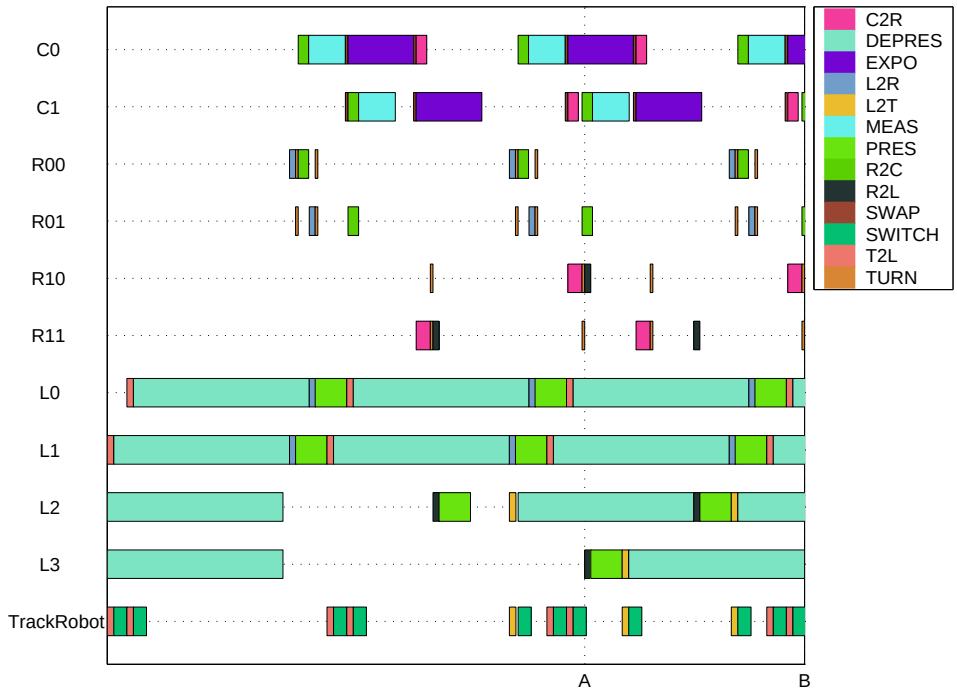

Figure 14: A schedule that optimizes the stationary throughput of the EUV machine in which unexposed wafers enter through the upper two locks (L0 and L1) and exposed wafers exit through the lower two locks (L2 and L3). The cyclic part of the schedule consists of the interval between points A and B. Clearly, the operation of the locks is critical in the cyclic part.

tion 2). Note that one lock has a wafer throughput of one wafer per $min_l = LOAD + PRES + DEPRES + L2R\_T$ time units, where $LOAD$ is the time needed by the track robot to place a wafer in the lock, *(DE)PRES* is the time needed to (de)pressurize a lock, and $L2R\_T$ is the time needed by an internal robot to grab a wafer from a lock. Thus, two locks have a throughput of at most one wafer per $\frac{1}{2}min_l$ time units. Since $\frac{1}{2}min_l > min_s$, a better upper bound on the throughput is 1 wafer per $\frac{1}{2}min_l$ time units. We are able to find a schedule for a value of H that is 11% larger than min_h and for $S = \frac{1}{2}min_l$. Therefore, this schedule optimizes the stationary throughput of this alternative machine lay-out. Concluding, the optimal stationary throughput is 61% smaller than the optimal stationary throughput of the original machine, and not the expose operation but the locks have become critical. This confirms our line of thought in Section 2. Figure 14 shows this alternative schedule.

The second alternative lay-out consists of only two locks and one internal robot. Again, an upper bound on the throughput of this machine is 1 wafer per $\frac{1}{2}min_l$ time units. The throughput loss compared to the original machine thus is *at least* 61%. However, the best schedule we are able to find with UPPAAL has a stationary throughput that is 83% worse than the optimal schedule for the original machine. Figure 15 shows this alternative schedule.

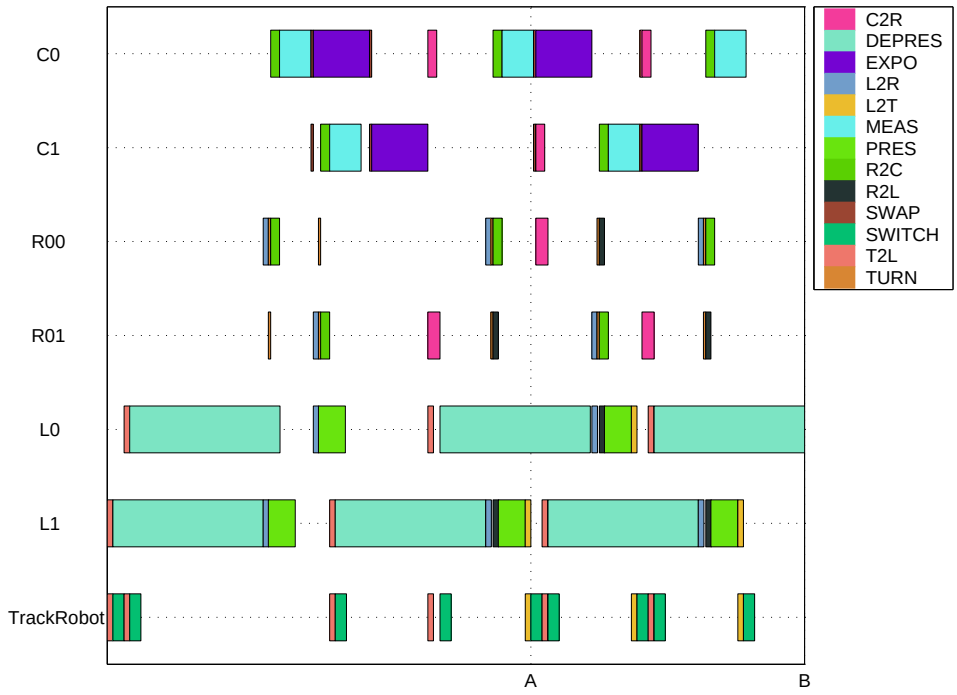

Figure 15: A schedule for the EUV machine with only two locks and one internal robot. The cyclic part of the schedule consists of the interval between points A and B.

5 Conclusions

The SMV model checker has successfully been used to characterize the set of safe states of the EUV machine. This characterization consists of a very short boolean expression over the places in the machine and is useful for the design of an actual controller since deadlock can easily be avoided by examining the possible successor states of the current state. Since the characterization is exact, the controller implements a least restrictive (optimal) deadlock avoidance policy. Furthermore, we used the UPPAAL model checker to compute infinite schedules for the EUV machine that optimize stationary throughput. The throughput of two alternative machine configurations has also been analyzed and it is shown that the original configuration has a throughput that is at least 61% higher. In theory, our approach can be applied to a broad class of resource allocation systems. As always when using model checking, the state space explosion is the main problem for scalability. Altogether, in our view, the present work nicely illustrates the usefulness of model checking techniques to support the design process of applications that involve resource allocation and scheduling. Building models that are just abstract enough for addressing a specific question, often provides a good way to deal with the state space explosion problem.

Acknowledgements. The authors thank Biniam Gebremichael for his useful suggestions concerning the SMV model.

References

[1] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.

[2] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages and Programming*, pages 322–335, 1990.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] N. C. W. M. Braspenning. Scheduling and behavior verification of machines based on task-resource models. Master's thesis, Department of Mechanical Engineering, Eindhoven University of Technology, The Netherlands, October 2003. Confidential.

[5] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1,2):115–131, 1988.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, August 1986.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[8] E. W. Dijkstra. Cooperating sequential processes. Technical report, Eindhoven University of Technology, The Netherlands, 1965.

[9] A. Fehnker. Scheduling a steel plant with timed automata. In *Proceedings of the sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.

[10] B. Gebremichael and F. W. Vaandrager. Control synthesis for a smart card personalization system using symbolic model checking. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, number 2791 in LNCS, pages 189–203. Springer–Verlag, 2004.

[11] V. Hartonas-Garmhausen, E. M. Clarke, and S. Campos. Deadlock prevention in flexible manufacturing systems using symbolic model checking. In *IEEE Conference on Robotics and Automation*, volume 1, pages 527–532, 1996.

[12] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1/2):134–152, 1997.

[13] M. Lawley and S. A. Reveliotis. Deadlock avoidance for sequential resource allocation systems: Hard and easy cases. *International Journal of Flexible Manufacturing Systems*, 13(4):385–404, 2001.

[14] M. Lawley, S. A. Reveliotis, and P. Ferreira. Design guidelines for deadlock handling strategies in flexible manufacturing systems. *International Journal of Flexible Manufacturing Systems*, 9(1):5–30, January 1997.

[15] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 1992.

[16] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[17] J. Park and S. A. Reveliotis. Deadlock avoidance in sequential resource allocation systems with multiple resource acquisitions and flexible routings. *IEEE Transactions on Automatic Control*, 46(10):1572–1583, 2001.

[18] S. A. Reveliotis, M. Lawley, and P. Ferreira. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Transactions on Automatic Control*, 42(10):1344–1357, 1997.

[19] W. Stallings. *Operating Systems – Internals and Design Principles*. Prentice–Hall, 1998.

[20] Y. Wang and Z. Wu. Deadlock avoidance control synthesis in manufacturing systems using model checking. In *IEEE American Control Conference*, volume 2, pages 1702–1704, 2003.