Nordic Journal of Computing

GUIDED SYNTHESIS OF CONTROL PROGRAMS USING UPPAAL*

THOMAS HUNE[1] KIM G. LARSEN[2]
PAUL PETTERSSON[2]

[1]*BRICS*[†]*, Department of Computer Science, Aarhus University, Denmark*
E-mail: baris@brics.dk
[2]*BRICS*[†]*, Department of Computer Science, Aalborg University, Denmark*
E-mail: {kgl,paupet}@cs.auc.dk

Abstract. In this paper we address the problem of scheduling and synthesizing distributed control programs for a batch production plant. We use a timed automata model of the batch plant and the verification tool UPPAAL to solve the scheduling problem.

In modeling the plant, we aim at a level of abstraction which is sufficiently accurate in order that synthesis of control programs from generated timed traces is possible. Consequently, the models quickly become too detailed and complicated for immediate automatic synthesis. In fact, only models of plants producing two batches can be analyzed directly! To overcome this problem, we present a general method allowing the user to *guide* the model-checker according to heuristically chosen *strategies*. The guidance is specified by augmenting the model with additional guidance variables and by decorating transitions with extra guards on these. Applying this method have made synthesis of control programs feasible for a plant producing as many as 60 batches.

The synthesized control programs have been executed in a physical plant. Besides proving useful in validating the plant model and in finding some modeling errors, we view this final step as the ultimate litmus test of our methodology's ability to generate executable (and executing) code from basic plant models.

CR Classification: D.1.2, D.2.2, D.2.4, F.3.1, I.6.4, J.6

Key words: real-time verification, guided model-checking, scheduling, program synthesis, distributed systems

*This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid systems).

†**B**asic **R**esearch **I**n **C**omputer **S**cience, Centre of the Danish National Research Foundation.

Submitted February 2000

1. Introduction

In this paper we suggest a solution to the problem of synthesizing and verifying valid scheduling control programs for resource allocation, based on a batch plant of SIDMAR [Boel and Stremersch 1999],[Fehnker 1999], which is a case study of the VHS project[1]. We model the plant in a network of timed automata, with the different components of the plant (e.g. batches, recipes, casting machine, cranes, etc.) constituting the individual timed automata. The scheduling problem is formulated as a time-bounded reachability question allowing us to apply the real-time model-checking tool UPPAAL [Larsen *et al.* 1995],[Larsen *et al.* 1997] to derive a schedule. An overview of the methodology is shown in Figure 1.1.

UPPAAL offers a trace with actions of the model and timing information of the actions. The remaining effort required in transforming such a model trace into an executable control program depends heavily on the accuracy of the model with respect to the control programming language and the physical properties of the plant. Given a sufficiently high level of accuracy of the plant model, a schedule can be obtained from a trace by projection, and synthesis of the control program from a schedule amounts to textual substitution. However, a model suitable for such program synthesis becomes very detailed as all the necessary information about the plant, such as the timing bounds and the physical constraints for movements of loads, cranes etc, has be to specified. As an immediate drawback, synthesizing schedules for several batches quickly becomes infeasible.

To deal with this (unavoidable) problem we introduce a method, allowing the user to *guide* the model-checking according to certain chosen *strategies*. Each strategy will contribute with a reduction of the search-space, but in contrast to fully automatic reduction methods it is up to the user to 'guarantee' preservation of schedulability. However, if a schedule is identified via the guided search, the schedule is indeed a valid one for the original model.

To be able to run the generated control programs in a physical plant, we consider a LEGO® MINDSTORMS™ plant, instead of the original plant of SIDMAR. We have used the plant to successfully run synthesized control programs and by doing so increased our confidence in the plant model. We view this final, scientifically rather simple step as the ultimate litmus test of our methodology's ability to generate executable (and executing) code from rather natural plant models.

The SIDMAR plant has been studied by several other researchers. Our timed automata model is based on the model in [Fehnker 1999], which is similar to ours but more abstract in the sense that some information, such as delays for the moving of batches, is not included. A Petri net model of the plant is presented in [Boel and Stremersch 1999]. In [Stobbe 2000], constraint programming techniques are used to generate schedules of the SIDMAR plant for up to 30 batches. To obtain this techniques similar to ours are used for reducing the size of the search space. Other work applying the model of timed automata and UPPAAL to analyze and solve planning problems of batch plants include [Kristoffersen *et al.* 1999] in which an experimental batch plant is studied.

The rest of this paper is organized as follows: In the next two sections we describe the scheduling problem and how it has been modeled in UPPAAL. In Section 4 and 5 we present the guiding techniques and evaluate their effect on the plant model. In

[1] See the web site `http://www-verimag.imag.fr//VHS/main.html`.

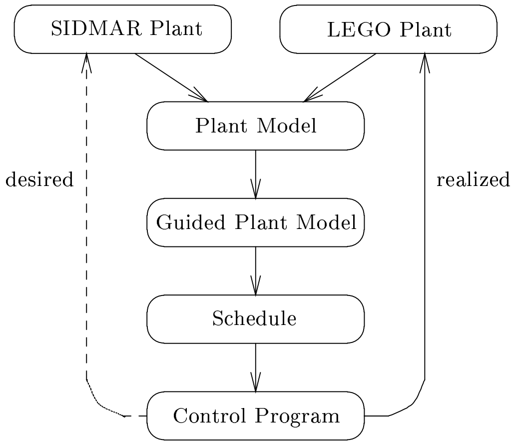

Fig. 1.1: Overview of methodology.

Section 6 we describe experiments with the LEGO® plant and how programs are synthesized for the plant. Section 7 concludes the paper. Finally, timed automata descriptions of four plant components are enclosed in the appendix.

2. The Scheduling Problem

Our plant is based on a part of the SIDMAR steel production plant located at Gent in Belgium. We will consider the part of the plant between the blast furnace and the continuous casting machine where molten pig iron is converted into steel of different qualities. The process is started when pig iron being poured into ladles by one of two converter vessels. The iron is transported in the ladles while it is being processed. By treatments in different machines the iron is converted into steel and finally casted in the casting machine. Depending on the machines used and how long the treatment in the machines last, different qualities of steel are produced. When the steel in a ladle has been casted the empty ladle must be moved to a storage place. From here the ladles are cleaned and reused. However, this is not part of our model, where ladles are stored at the storage place but not reused. The physical components of the process are: two converter vessels where molten iron is poured into ladles, five machines, tracks connecting these, two cranes running on one overhead track, a buffer place, a storage place for empty ladles, and one casting machine. The layout of the plant can be seen in Figure 2.1.

Machines number one and four are of the same type and so are machines number two and five. Each crane can only hold one ladle and they cannot overtake each other. On each track and in each machine there is room for at most one ladle. This means that the ladles cannot overtake each other without using one of the cranes.

The steel must sustain a minimum temperature during the process. This gives an upper bound on the time a batch is allowed to spend in the plant from it is poured and until it is casted. Casting takes a fixed time and must be continuous.

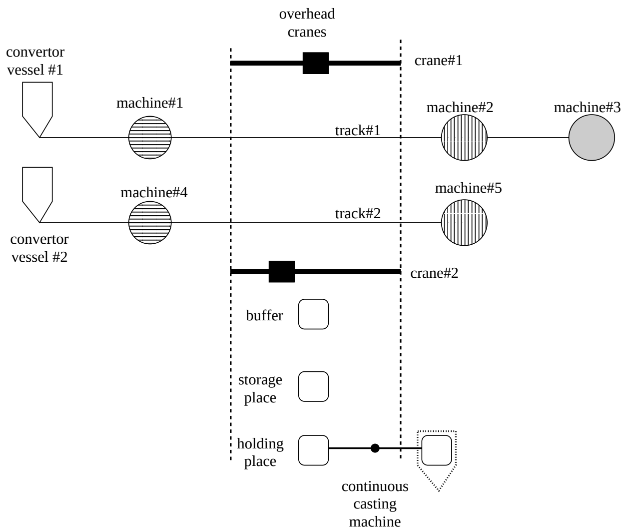

Fig. 2.1: Layout of the plant.

Therefore a new ladle filled with steel must be waiting in the holding place of the casting machine when casting of a ladle has finished.

Steel of different qualities can be produced depending on which types of machines are visited and for how long. For each batch this is specified by a recipe. The problem to be solved can now be stated as:

> *Given an ordered list of recipes, if possible synthesize a control program for the plant such that steel specified by the recipes are produced in the right order and within a given time.*

The major part of solving this problem is finding a schedule for the production if one exists. A schedule for the plant defines which action takes place in the plant e.g. moving of batches and cranes, and when the actions take place.

3. Scheduling with Timed Automata

Finding a schedule for producing an ordered list of steel qualities is the main part of the problem. It can be solved in a number of ways. Here we chose to model the plant using timed automata [Alur and Dill 1994] and use the verification tool UPPAAL [Larsen *et al.* 1995],[Larsen *et al.* 1997] to solve the scheduling problem [2]. For a discussion of this approach to scheduling see [Fehnker 1999].

The modeling language in UPPAAL is networks of timed automata extended with data variables [Larsen *et al.* 1997]. To meet requirements from various case-studies the language has been further extended with the notion of committed locations [Bengtsson *et al.* 1996], urgent synchronization actions [Larsen *et al.* 1997],

[2] See the web site `http://www.uppaal.com/` for more information about UPPAAL.

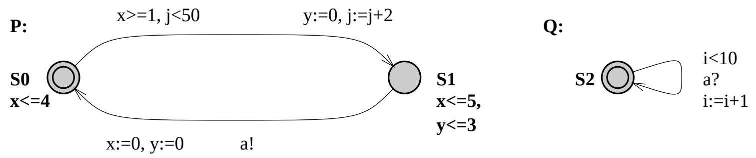

Fig. 3.1: A Network of Timed Automata.

and data structures such as arrays of data-variables etc. In this section we give a brief informal description of the modeling language of UPPAAL. For a detailed description we refer the reader to [Larsen *et al.* 1997].

3.1 Networks of Timed Automata

Consider the network of timed automata P and Q shown in Figure 3.1. Automaton P has two control locations **S0** and **S1**, two real-valued clocks x and y, and a data variable j. A *state* of the automaton is of the form (l, s, t, k), where l is a control location, s and t are non-negative reals giving the value of the two clocks x and y, and k is a natural number giving value to the data variable j. A control location is labelled with a condition (the location invariant) on the clock values that must be satisfied for states involving this location. Assuming that the automaton starts to operate in the state $(\mathbf{S0}, 0, 0, 0)$, it may stay in location **S0** as long as the invariant $\mathtt{x} \leq 4$ of **S0** is satisfied. During this time the values of the clocks increase synchronously. Thus from the initial state, all states of the form $(\mathbf{S0}, t, t, 0)$, where $t \leq 4$, are reachable. The edges of a timed automaton may be decorated with a condition (guard) on the clocks and the data variable values that must be satisfied in order for the edge to be enabled. Thus, only for the states $(\mathbf{S0}, t, t, k)$, where $1 \leq t \leq 4$ and $\mathtt{k} < 50$, is the edge from **S0** to **S1** enabled. Additionally, edges may be labelled with assignments and synchronization labels. An assignment may reset the value of the clocks and update the data variables. For example, when following the edge from **S0** to **S1** the clock **y** is reset to 0 and the data variable j is incremented by 2, leading to states of the form $(\mathbf{S1}, t, 0, 2)$, where $1 \leq t \leq 4$. The synchronization label is used to establish synchronization between automata. For example the transition from **S1** to **S0** of automaton P is labeled with a!, requiring the transition to be synchronized with the transition of automaton Q offering the complementary action a?.

In general, a timed automaton is a finite-state automata extended with a finite collection $\mathcal{C}$ of real-valued clocks ranged over by $\mathtt{x}, \mathtt{y}$ etc. and a finite set of data variables $\mathcal{D}$ ranged over by $\mathtt{i}, \mathtt{j}$ etc. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to stand for the set of formulas that can be an atomic constraint of the form: $\mathtt{x} \sim n$ or $\mathtt{x} - \mathtt{y} \sim n$ for $\mathtt{x}, \mathtt{y} \in \mathcal{C}$, $\sim \in \{<, \leq, =, \geq >\}$ and n being a natural number, or a conjunction of such formulas. Similarly, we use $\mathcal{B}(\mathcal{D})$ to stand for the set of *data-variable constraints* that are the conjunctive formulas of $\mathtt{i} \sim \mathtt{j}$ or $\mathtt{i} \sim k$, where $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and k is an integer number. To denote the set of formulas that are conjunctions of clock constraints and a data-variable constraints we use $\mathcal{B}(\mathcal{C}$,$\mathcal{D})$ (ranged over by g). The elements of $\mathcal{B}(\mathcal{C}$,$\mathcal{D})$ are called *constraints* or *guards*.

An assignment in UPPAAL is a sequence of operations of the form $\mathtt{x} := 0$, or $\mathtt{i} :=$ *Expr*, where x is a clock, i is a data variable, and *Expr* is an integer expression, e.g. $2 * (\mathtt{i} - \mathtt{j}) + 3$ (where j is a data variable). We shall use $\mathcal{R}$ to denote the set of assignments. Furthermore, we use $\mathcal{A}ct$ to denote a finite set of actions ranged over by a, a?, a!, b?, b!, etc.

DEFINITION 1. (TIMED AUTOMATA) *A timed automaton A over clocks $\mathcal{C}$ and data variables $\mathcal{D}$ is a tuple $\langle N, l_0, \longrightarrow, I\rangle$ where N is a finite set of (control-)locations, l_0 is the initial location, $\longrightarrow \subseteq N \times \mathcal{B}(\mathcal{C}, \mathcal{D}) \times \mathcal{A}ct \times \mathcal{R} \times N$ corresponds to the set of edges and finally, $I : N \mapsto \mathcal{B}(\mathcal{C})$ assigns invariants to locations. In the case, $\langle l, g, a, r, l'\rangle \in\longrightarrow$, we write $l \overset{g,a,r}{\longrightarrow} l'$.* □

To formalize the semantics we use variable assignments. A variable assignment is a mapping which maps the clocks $\mathcal{C}$ to the non-negative reals and the data variables $\mathcal{D}$ to integers. A semantical *state* of an automaton A is now a triple (l, u), where l is a location of A and u is a an assignment for $\mathcal{C}$ and $\mathcal{D}$, and the semantics of A is given by a transition system with the following two types of transitions (corresponding to delay-transitions and action-transitions):

- $(l, u) \longrightarrow (l, u \oplus d)$ if $I(l)(u)$ and $I(l)(u \oplus d)$
- $(l, u) \longrightarrow (l', u')$ if there exist g and r such that $l \overset{g,a,r}{\longrightarrow} l'$, $g(u)$, $u' = r[u]$ and $I(l')(u')$

where d is a non-negative real number, $u \oplus d$ denotes the assignment which maps each clock x in $\mathcal{C}$ to the value $u(\mathtt{x}) + d$ and leaves each data variable i with the unchanged value $u(\mathtt{i})$, and $r[u]$ denotes the result of updating the clocks $\mathcal{C}$ and the data-variables in $\mathcal{D}$ according to $r \in \mathcal{R}$.

Finally, we briefly introduce the notion of *networks of timed automata* [Yi *et al.* 1994],[Larsen *et al.* 1995]. A network is a finite set of automata composed in parallel with a CCS-like parallel composition operator [Milner 1989]. For a network with the timed automata $A_1, \ldots, A_n$ the intuitive meaning is similar to the CCS parallel composition of $A_1, ..., A_n$ with *all* actions being restricted, that is, $(A_1|...|A_n)\backslash\mathcal{A}ct$. Thus an edge labelled with action a *must* synchronize with an edge labelled with an action complementary to a, and edges with the silent τ action are internal, so they do not synchronize. In UPPAAL '?' and '!' are used to represent complementary actions, so a? and a! are considered complementary and can synchronize.

Given a network of timed automata and a set of states, UPPAAL can analyze whether or not one of the states is reachable from the initial state of the network. If the answer is positive, UPPAAL produces a trace with action- and delays-transitions leading from the initial state to one of the specified states.

For the model of the plant, which will be presented in the following, a trace defines a schedule for the plant since it specifies *what* happens in the plant (the synchronization actions) and *when* (the delays). From a schedule a working program controlling the plant may be generated. The level of detail in the trace (and therefore in the schedule) influences the work needed to generate the program. In [Fehnker 1999] the traces generated did not include time for the moving of batches, making the generation of executable programs from the schedules hard. To minimize the effort needed during the translation, we produce traces with detailed and precise information about timing of all actions in the plant.

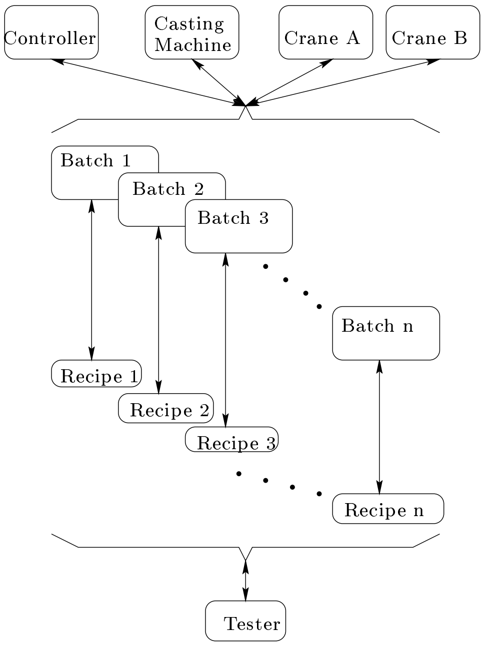

Fig. 3.2: Synchronization between the automata of a model.

3.2 A Model for Scheduling the Plant

An instance of the problem is given by a list of qualities of steel (or recipes) and a maximal production time. A model of a problem instance consists of: for each recipe one automaton representing the recipe and one automaton representing the movement of the batch; one automaton for each of the two cranes; one automaton testing that the recipes finish in the correct order; one automaton for making some actions synchronizing; and one automaton modeling the casting machine. Figure 3.2 shows the synchronizations between the different automata. The batch automata communicate with each other through two shared arrays and the two cranes also share an array. These arrays will be described in more detail later.

The most complex of the automata is the one modeling the possible behaviors of a batch (see Figure 7.4 of the appendix[3])[4]. The batch automaton reflects the topology of the plant (shown in Figure 2.1) as well as the physical constraints on the movements of a batch. Basically, there is one location for each position of the plant a batch can be located at. A position is either a machine, a track segment, the storage place, the casting machine, or a position on the overhead track. Positions on the overhead track are over one of the two tracks, the storage place, the casting machine, or in between any of these. A batch automaton has a clock named `x` associated to it which is used to measure the time spend on moving along a track. The time spend is the worst case time measured in the

[3] Unless stated otherwise, guided versions of the automata are shown since these have been used for most of the experiments.
[4] Pictures of all the automata and the LEGO® plant can be found at the web site `http://www.brics.dk/~baris/CaseStudy/`.

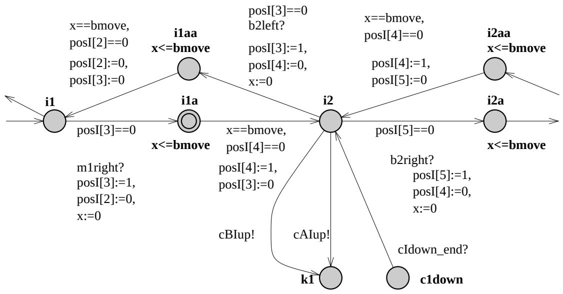

Fig. 3.3: Part of the unguided batch automaton.

physical plant which is given by the constant `bmove`. Shared among all the batch automata in a model are the two binary arrays `posI` and `posII`, which are used for storing which positions are occupied on the two tracks. Figure 3.3 shows the part of the unguided batch automaton modeling the position named **i2**, between machines number one and two on track one. Moving a batch between positions in the model is done in two steps. First a transition is taken to an intermediate position, e.g. from **i2** to **i1aa**. A batch can only start to move to a position if this position is free, which in this case is ensured by checking the array `posI` using the guard $\texttt{posI}[3] == 0$. Taking the transition resets the clock `x` and updates which positions are occupied by the assignment $\texttt{posI}[3] := 1, \texttt{posI}[4] := 0$. The batch can stay in the intermediate position at most `bmove` time units because of the invariant $\texttt{x} \leq \texttt{bmove}$ in the location. However, it cannot leave the location before `bmove` time units have passed because of the guard $\texttt{x} == \texttt{bmove}$ on the transition leaving the intermediate location. This means that moving a batch along a track is modeled as taking exactly `bmove` time units. A batch can also move when it is carried by a crane. The time spend during such moving is measured by the crane automaton.

Each batch has a recipe associated to it (a recipe using machine type one and two is shown in Figure 3.4). The recipe defines which machines should be visited, in which order, and for how long. It also measures the overall time the batch has spend in the plant. A recipe has two clocks associated to it. One is reset as the batch starts in the plant and measures the overall time spend in the plant by the batch. The other clock is used for measuring the time of the different treatments the batch goes through. When a batch is located at a machine of the right type according to the recipe, the batch and the recipe can synchronize to start the machine. This resets the clock measuring the time of treatments. When the specified time for the treatment has passed the recipe and the batch synchronize to turn the machine off. When the treatments are completed and the batch is ready to be casted the recipe synchronizes with the test automaton to ensure that the production order is kept. Here it is also checked that the batch has not spend too much time in the plant.

As mentioned the positions of a crane are over the two tracks, over the storage place, over the casting machine and in between these. An automaton modeling a crane has two locations for each of these positions, one modeling the crane being empty and one modeling the crane carrying a batch. The automaton modeling the

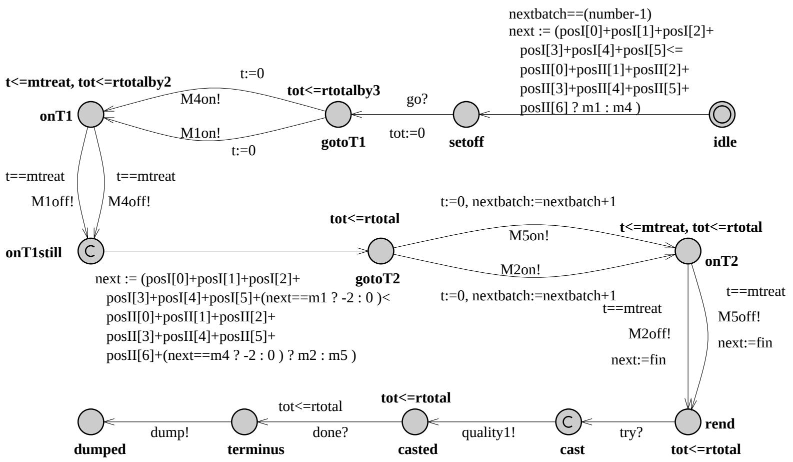

Fig. 3.4: An example recipe automaton.

upper crane which is only moving between the two tracks is shown in Figure 3.5 (the automaton modeling the other crane can be seen in Figure 7.2 of the appendix.) A crane picking up a batch is modeled by the two automata synchronizing. Similarly when a crane moves or sets a batch down. Each crane automaton has one clock which is used for measuring time when the crane is moving. The movement of a crane between two positions is modeled like movement between two positions in the batch automaton with an intermediate location where the time for the movement passes. The two crane automata share a binary array like the batch automata for storing which positions are occupied

The test automaton synchronizes with a recipe automaton just before the recipe allows the batch to enter the casting machine. This ensures that the order of the production as stated in the problem description is kept (Figure 7.3 in the appendix shows a test automaton).

There is also one automaton which has no influence on the overall behavior of the model (shown in Figure 3.6). However, since we will use the traces from the model for generating schedules, it is important that the all actions of the plant affecting the schedule appear directly in the trace. Some of these actions are internal actions in the batch automaton and will therefore not appear in the generated traces. An example is the movements of a batch on the belts. The purpose of this automaton is to synchronize with the internal actions (modified to external actions) to make them appear in the traces.

Finally there is an automaton modeling the casting machine (see Figure 7.1 of the appendix). It synchronizes with a batch to start the casting. After a specific time when the batch has been casted, the casting machine and the batch should synchronize again to let the batch leave the casting machine and to let a new one enter.

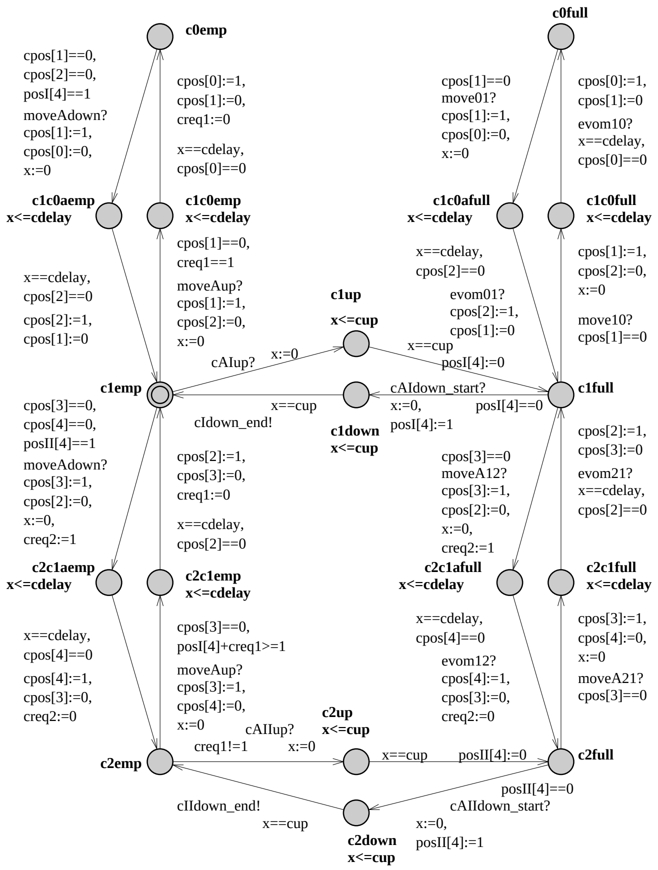

Fig. 3.5: The upper crane.

4. Guiding Timed Automata

The timed automata described in the previous section models the steel production plant at a high level of accuracy. The details in the model are needed to allow generation of schedules from model traces by projection, and to allow generation of control programs from schedules by textual substitution. However, the fact that the model is detailed and consisting of a many parallel timed automata with several clocks is also a serious problem, as the model is too big and complicated for automatic analysis. In fact, finding traces of a plant model with just a few batches is infeasible in practice (see Section 5). The limiting factor is the amount of time and memory consumed during the analysis to (symbolically) explore and store the reachable state-space of the analyzed model. To solve this problem we introduce a way of user directed *guiding* of a state-space exploration algorithm according to a number of certain chosen *strategies*.

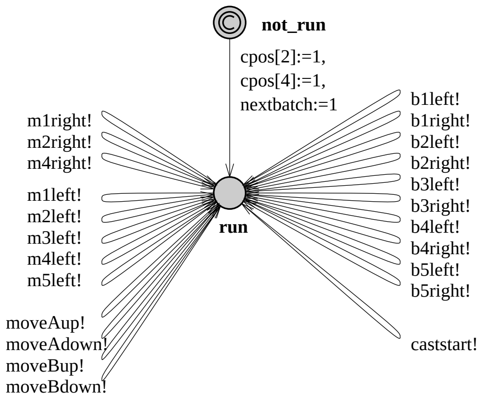

Fig. 3.6: The automaton ensuring synchronization.

4.1 Guiding

The overall idea of guiding an automata model is to let the user implement reduction strategies by augmenting the automata with a set of additional clocks, data variables, constraints and assignments [5]. Each strategy will contribute to the reduction of the state-space by constraining the behavior of the model. However, in contrast to automatic state-space reduction techniques, the guiding technique trust the user to preserve schedulability of the plant model.

Assume a network of timed automata over clocks $\mathcal{C}$ and data variables $\mathcal{D}$. The automata are guided by introducing a set of new clocks $\mathcal{C}_{\mathrm{G}}$ and integer variables $\mathcal{D}_{\mathrm{G}}$. We call $\mathcal{C}_{\mathrm{G}} \cup \mathcal{D}_{\mathrm{G}}$ *guiding variables*. A guide is implemented by conjuncting new constraints from $\mathcal{B}(\mathcal{C}_{\mathrm{G}} \cup \mathcal{C}, \mathcal{D}_{\mathrm{G}} \cup \mathcal{D})$ to the existing guards of the automata, new clock constraints from $\mathcal{B}(\mathcal{C}_{\mathrm{G}} \cup \mathcal{C})$ to the location invariants, and adding new assignments of variables in $\mathcal{C}_{\mathrm{G}} \cup \mathcal{D}_{\mathrm{G}}$ to the resets. Thus, the guides may test the values of all the clocks and the data variables in the transition guards and the location invariants of the automata. A guide may also assign the guiding variables in the reset sets. However, the original clocks and data variables of the timed automata (i.e. $\mathcal{C} \cup \mathcal{D}$) should not be assigned. This ensures the essential property that a trace generated from a guided network of timed automata indeed is a valid trace of the original network of timed automata. In the plant model this means that the schedules generated from the guided plant model is guaranteed to also be valid in the original plant model.

4.2 Implemented Strategies

We have used guiding to implement a number of strategies in the plant model. In the following we describe the strategies abstractly, in terms of the physical plant,

[5] The technique of adding guiding variables presented in this paper is reminiscent of the notion of history and prophesy variables used in traditional program verification, as in the work of Abadi and Lamport [Abadi and Lamport 1991].

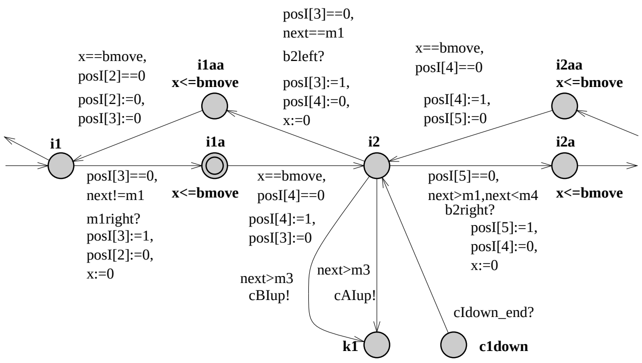

Fig. 4.1: Guided part of the batch automaton.

and give some detailed examples of how the guides are introduced in the plant model. We emphasize that many of the strategies are heuristics and most of them could in fact reduce the number of valid schedules of the plant model. However, this is not a problem as long as it is still possible to generate valid schedules from the model (as we are not concerned with finding optimal schedules).

The implemented strategies are based on the general observation that the plant model described in the previous section models *all* possible behaviors of the plant. This also includes several behaviors that should not (or are unlikely to) appear in a valid schedule. The implemented strategies aim at reducing these 'unwanted' behaviors.

Strategy 1: Ordering of Batches. When the scheduling problem is stated the production order of the steel qualities is given. One strategy is to use this order when starting new batches in the plant. To implement the strategy we introduce the guiding variable `nextbatch` in the recipe automaton associated to each batch, to control which batch is allowed to start next. According to the engineers at SIDMAR the same strategy is used there.

A recipe automaton is shown in Figure 3.4. The guide is implemented in the guard `nextbatch==(number-1)` on the first transition of the automaton, where `number` is a unique constant number associated to each recipe. The guide ensures that the recipe starts the batch when the value of `nextbatch` is equal to `number-1`. The recipe automaton increments the `nextbatch` variable on a transition from location **goT2** to **onT2** (see also Strategy 2 below) to allow the next batch to start.

Strategy 2: Delaying of Batches. Related to the first strategy is the starting time of batches. Since there is an upper bound on the time a batch is allowed to spend in the plant, all batches should not be started at the same time. Therefore, we prevent a batch from starting based on the progress of the batch just before it. The strategy is implemented in the recipe automata by delaying the update of the `nextbatch` variable. In the recipe1 automaton shown in Figure 3.4 the `nextbatch` guiding variable is incremented on the transition from location **goT2** to **onT2** instead of immediately after the test on the first transition. This prevents the next batch to start before the batch has been treated by two machines.

Strategy 3: Global Routing of Batches. To guiding the movements of the batches we introduce a new guiding variable named `next` for each batch. The value of `next` specifies where the batch should go next, based on the next machine treatment specified in its recipe. When there is a choice of machines the recipe will chose the machine on the track with fewest batches present. For example the choice of the first machine is implemented by a guiding expression on the first transition of the recipe automaton:

$$\textbf{if } (track1 \leq track2) \textbf{ then } \texttt{next}{:=}\texttt{m1} \textbf{ else } \texttt{next}{:=}\texttt{m4}$$

where $track1$ is the number of batches present on track one and $track2$ the number of batches on track two. In the recipe automaton in Figure 3.4 the value of $track1$ and $track2$ are computed as the sum of active bits in the bit vectors `posI` and `posII` respectively (recall from the previous section that `posI` and `posII` are used to ensure mutex on the positions of the two production tracks).

Strategy 4: Local Routing of Batches. The possible movements of the batches are further reduced by a strategy deciding how a batch should move between two given position. The implemented strategy selects the only direct route between two positions. To implement the strategy in the plant model we use the guiding variable `next`. A guard constraining the value of `next` is added to all transitions of the batch automata leaving a location modeling a physical position in the plant. Figure 4.1 shows a part of the guided batch automaton corresponding to the partial original automaton shown in Figure 3.3. Machine one is the only machine located to the left of position **i2** on track 1. Therefore, the guides require the `next` variable to have value `m1` (representing machine 1) to move in the left direction. This is ensured by the guard `next==m1` on the transition from location **i2** to **i1aa**. The transitions from location **i2** to **k1** represents the batch being picked up by one of the cranes. When this is the case the next destination of the batch should not be a machine on track one (i.e. not machine 1, 2, or 3) therefore `next` is required to be greater than `m3`.

Strategy 5: Moving of Cranes. When a crane is carrying a batch it always follows the strategy of the batch. If a crane is empty, the strategy is to move only when something is ready to be picked up or if it is blocking the other crane. Guiding guards in the crane automata testing bits in `posI` and `posII` ensure that the cranes move towards the pick up positions on the tracks when a batch is waiting to be picked up (see e.g. the transition from location **c2emp** to **c2c1emp** in Figure 7.2 of the Appendix).

To allow an empty crane to move in other situations the guiding variables `creq1` and `creq2` are introduced. Guards testing their value are introduced on some transitions to allow the crane to move from certain positions in a specified directions when the variables are non-zero. The variables are typically assigned by the other crane to indicate that it is moving towards a (possibly) occupied position that must be empty. For example, in the craneB automaton shown in Figure 7.2 the variable `creq1` is assigned on the transitions from location **c2emp** to **c1emp** to allow crane 1 to leave crane position 1 (modeled by the locations **c1emp**, **c1up**, **c1down**, and **c1full** in the craneB automaton).

5. Experimental Results

The plant models have been analyzed in the validation and verification tool Uppaal [Larsen *et al.* 1995],[Larsen *et al.* 1997]. In this section we present the results of an

#	All Guides						No Guides					
	BFS		DFS		BSH		BFS		DFS		BSH	
	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
1	0.1	0.9	0.1	0.9	0.1	0.9	3.2	6.1	0.8	2.2	3.9	3.3
2	18.4	36.4	0.1	1.0	0.1	1.1	-	-	19.5	36.1	-	-
3	-	-	3.2	6.5	3.4	1.4	-	-	-	-	-	-
4	-	-	4.0	8.2	4.6	1.8	-	-	-	-	-	-
5	-	-	5.0	10.2	5.5	2.2	-	-	-	-	-	-
10	-	-	13.3	25.3	16.1	9.3	-	-	-	-	-	-
15	-	-	31.6	51.2	48.1	22.2	-	-	-	-	-	-
20	-	-	61.8	89.6	332	46.1	-	-	-	-	-	-
25	-	-	104	144	87.2	83.3	-	-	-	-	-	-
30	-	-	166	216	124.2	136	-	-	-	-	-	-
35	-	-	209	250	-	-	-	-	-	-	-	-

TABLE I: Time and space requirements for generating schedules.

experiment where two versions of the plant model have been analyzed: the version with no guides described in Section 3, and a version with all guides described in Section 4. To evaluate the effect of adding guides, we use the standard UNIX programs `time` and `top` to measure the CPU time and the memory consumed by UPPAAL when generating a trace from the two models.

UPPAAL offers a number of options to control the internal verification algorithm applied in the tool [Larsen *et al.* 1997]. When analyzing the plant models we have used the compact data-structure for constraints [Larsson *et al.* 1997], the control-structure reduction [Larsson *et al.* 1997], and a recently implemented version of the (in-)active clock reduction [Daws and Tripakis 1998]. In addition we experiment with using breadth-first (BFS), depth-first search strategy (DFS), and depth-first search in combination with bit-state hashing (BSH) [Holzmann 1991] [6].

Table I shows the time (in seconds) and space (in MB) consumed by UPPAAL version 3.0.12 [7] when generating schedules from the two models. The numbers in the leftmost column corresponds to the number of batches in the model (and in the generated schedule). We use the marker "-" to indicate that the corresponding execution requires more than 256MB of memory, more than two hours of execution time, or that a suitable hash table size has not been found [8].

As can be seen in Table I, the use of guides significantly increases the size of models that can be analyzed. In the guided model, schedules can be generated for 35 batches using 250 MB in 3.5 minutes, whereas no schedule can be generated for three batches (or more) when no guides are used. It can also be observed that the bit-state hashing technique does not allow analysis of larger models in

[6] The bit-state hashing technique generates a sub set of the reachable state-space. A feasible schedule found with this technique is guaranteed to also be feasible in the original plant model.
[7] The tool was installed on a Linux Redhat 5.2 machine equipped with a Pentium III processor and 256MB of memory.
[8] When applying the hash table technique, we have used table sizes from 1048577 to 33554441 bits. The reported results corresponds to the most suitable hash table sizes found.

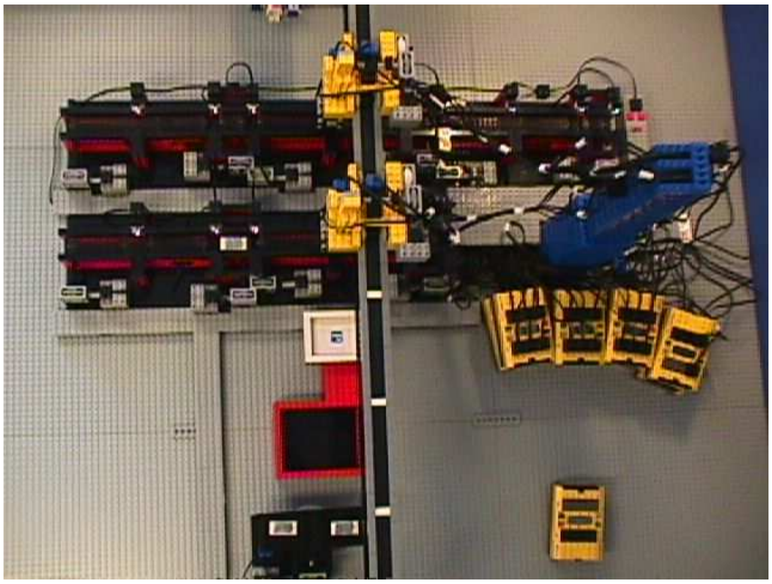

Fig. 6.1: The LEGO® plant.

this experiment, even though it performs well space-wise on most models. We experienced that finding suitable hash table sizes is very tedious for large system models. The largest system analyzed in the experiment is therefore a guided model using depth-first search strategy but without the bit-state hashing technique.

We have also installed Uppaal on a Sun Ultra machine equipped with 1024 MB of memory. On this machine, a schedule for 60 batches can be generated from the guided model in 2 257 seconds.

6. Synthesis of Control Programs

We did not expect to be able to run the generated control programs in the original plant of SIDMAR. Therefore we have used a LEGO® plant (see Figure 6.1) to run the synthesized programs in. This allows for experimenting with the plant to validate the model and it also makes it easy to find answers to a number of questions about the plant (e.g. measuring time bounds). The plant consists of a number of distributed units, each controlled locally by one RCX™ [LEGO 1998] brick. There are three types of units used in the plant: a crane, a machine with a track segment, and the casting machine. For the cranes there is an overhead track. The interface to the units consists of a set of commands like *MoveTrackRight*, *TurnOnMachine*, and *LiftBatch*. Commands are send to the local units by one central controller which is running the synthesized program. Ideally, one would want the local controllers to give feedback to the central controller when actions have finished or when an error occurs. However, since the communication between the RCX™ bricks is slow and unreliable especially if more than one brick tries to send at one time, the only feedback from the local controllers are acknowledgements of commands received from the global controller. This has big influences on the generated programs.

```
 ...                 Delay(5)
Load1.Track1Right    Crane1.Move1Left
Delay(10)            Delay(5)
Load1.Machine1On     Load1.Machine2On
Load2.Track5Right    Delay(1)
Delay(4)             Crane1.Move1Left
Crane1.Move1Left     Delay(6)
Delay(6)             Crane1.Move1Left
Load1.Machine1Off    Delay(3)
Load1.Track2Right    Load1.Machine2Off
Crane1.Pickup1        ...
```

Fig. 6.2: Part of a generated schedule.

As a result of the model checking in UPPAAL a trace containing information about synchronizations between automata and delays is obtained. Some of the synchronizations in the model, like the recipe synchronizing with the test automaton, are not relevant for the generated schedule. To get a schedule for the plant we project the trace to the actions relevant for the plant. Given some numbering of tracks and machines, part of a schedule looks like in Figure 6.2. There is a one-to-one correspondence between a schedule of this kind and the commands of the synthesized central control program. Each line with a `Delay` action is translated into a delay in the control program (in RCX™ code there is a `Wait` instruction doing this). For the rest of the lines only the second part is used, which defines what unit the command should be send to and what the command is. For example in the line `Load1.Track2Right`, the part `Track2Right` is translated to a command *MoveTrackRight* and sent to the local controller of track two.

The projection and the translation have been implemented using the pattern scanning and processing language `gawk`. Since the RCX™ language does not offer reliable communication primitives, each line in the schedule is translated into a code segment implementing such communication.

The synthesized programs have been executed in the plant. This was mainly intended as validation of the UPPAAL model of the plant. During the validation we found three errors in the model: the crane started to move horizontally too early when an empty ladle was picked up from the casting machine, causing the crane to collide with the casting machine and accidently drop the lifted ladle, so here a delay was missing in the model; when two cranes were located at positions next to each other and started to move in the same direction they could collide because the crane 'in front' was started last; in systems with only one batch the casting machine did not turn correctly. These problems were corrected in the model and new control programs were synthesized.

At one point during the experiments with the plant the batteries running the crane started to wear out. This meant that the initial timing information obtained from the plant was inaccurate because the cranes were moving slower. At this point having the complete process from generating traces to synthesizing control programs fully automated proved especially useful. New times for the moving of the cranes were measured and put into the model. Since scheduling still was possible, new programs were quickly synthesized and were running in the plant as expected.

Performing the experiments also validate the implementation of the translation from schedules to programs and here no problems were found. Our confidence in the correctness of the model has been significantly increased by the experiments.

7. Conclusion

In this paper, we have used timed automata and the verification tool Uppaal to synthesize control programs for a batch production plant. To deal with the unavoidable complexity of a plant model suitably accurate for program synthesis, we suggest and apply a general approach of guiding a model according to certain strategies. With this technique, we have been able to synthesize schedules for as many as 60 batches on a machine with 1024 MB of memory. Applying bit-state hashing the space consumption may be decreased even further.

Based on traces from the model checking tool Uppaal, schedules are generated. From theses schedules, control programs are synthesized and later executed in a physical plant. During execution a few modeling errors were detected. After correction, new schedules were generated and correct programs were synthesized and executed in the plant.

The presented method for guiding model-checking has proved very successful in significantly increasing the size of models which can be analyzed. The largest model we analyze consists of 125 timed automata and a total of 183 clocks. The notion of guides allows the user to add heuristics for controlling the behavior of the plant, and we believe that the approach is applicable and useful for model checking in general and reachability checking in particular. The validation of the model by running the synthesized programs also proved useful: having access to the (a) physical plant during the design of the model, allowed a number of questions to be readily answered.

Based on the traces generated from the Uppaal model other types of control programs can be synthesized. Here it would be especially interesting to study how more communication between the distributed controllers can be used, e.g. for generating more optimal programs, and for detecting run-time errors.

Acknowledgements: The authors wish to thank Ansgar Fehnker and Kåre Jelling Kristoffersen for fruitful discussions and many useful suggestions.

References

Abadi, Martin and Lamport, Leslie. 1991. The existence of refinement mappings. *Theoretical Computer Science 82*, 253–284.

Alur, R. and Dill, D. 1994. Automata for Modelling Real-Time Systems. *Theoretical Computer Science 126*, 2 (April), 183–236.

Bengtsson, Johan, Griffioen, W.O. David, Kristoffersen, Kåre J., Larsen, Kim G., Larsson, Fredrik, Pettersson, Paul, and Yi, Wang. 1996. Verification of an Audio Protocol with Bus Collision Using Uppaal. In *Proc. of the 8th Int. Conf. on Computer Aided Verification*, Number 1102 in Lecture Notes in Computer Science. Springer–Verlag, 244–256.

Boel, Rene and Stremersch, Geert. 1999. VHS Case Study 5: Timed Petri net model of steel plant at SIDMAR. Tech. report, SYSTeMS Group, Universiteit Gent, Technologiepark-Zwijnaarde 9, B-9052 Ghent, Belgium.

Daws, Conrado and Tripakis, Stavros. 1998. Model Checking of Real-Time Reachability Properties Using Abstractions. In *Proc. of the* 4th *Workshop on Tools and*

Algorithms for the Construction and Analysis of Systems, Number 1384 in Lecture Notes in Computer Science. Springer–Verlag, 313–329.

Fehnker, Ansgar. 1999. Scheduling a Steel Plant with Timed Automata. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA99)*. IEEE Computer Society, 280–286.

Holzmann, Gerard. 1991. *The Design and Validation of Computer Protocols*. Prentice Hall.

Kristoffersen, K., Larsen, K., Pettersson, P., and Weise, C. 1999. VHS Case Study 1 - Experimental Batch Plant using UPPAAL.

Larsen, Kim G. Pettersson, Paul, and Yi, Wang. 1995. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 76–87.

Larsen, Kim G. Pettersson, Paul, and Yi, Wang. 1997. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer 1*, 1–2 (Oct.), 134–152.

Larsson, Fredrik, Larsen, Kim G. Pettersson, Paul, and Yi, Wang. 1997. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 14–24.

LEGO. 1998. *Software developers kit*.

Milner, R. 1989. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs.

Stobbe, M. 2000. Results on Scheduling the Sidmar Steel Plant Using Constraint Programming.

Yi, Wang, Pettersson, Paul, and Daniels, Mats. 1994. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*. North–Holland, 223–238.

Appendix

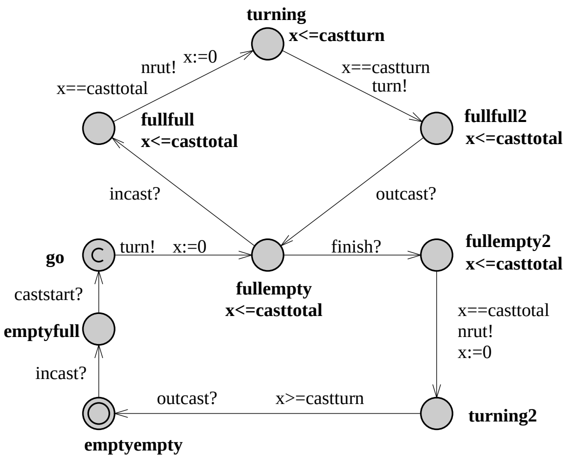

Fig. 7.1: The casting machine.

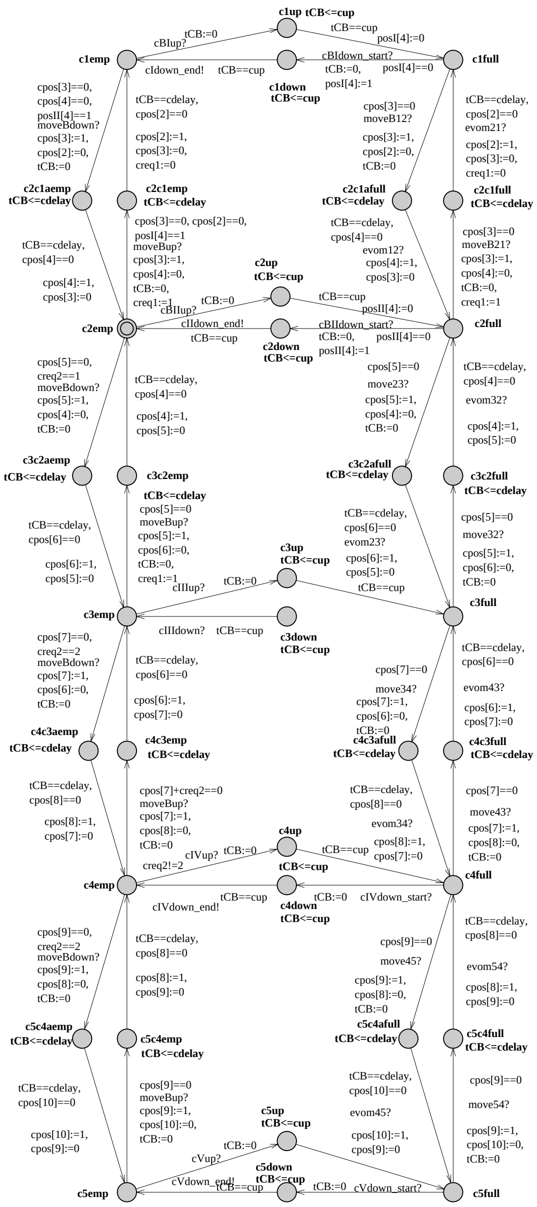

Fig. 7.2: The lower crane.

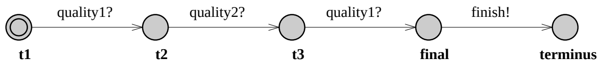

Fig. 7.3: A test automaton for producing three batches.

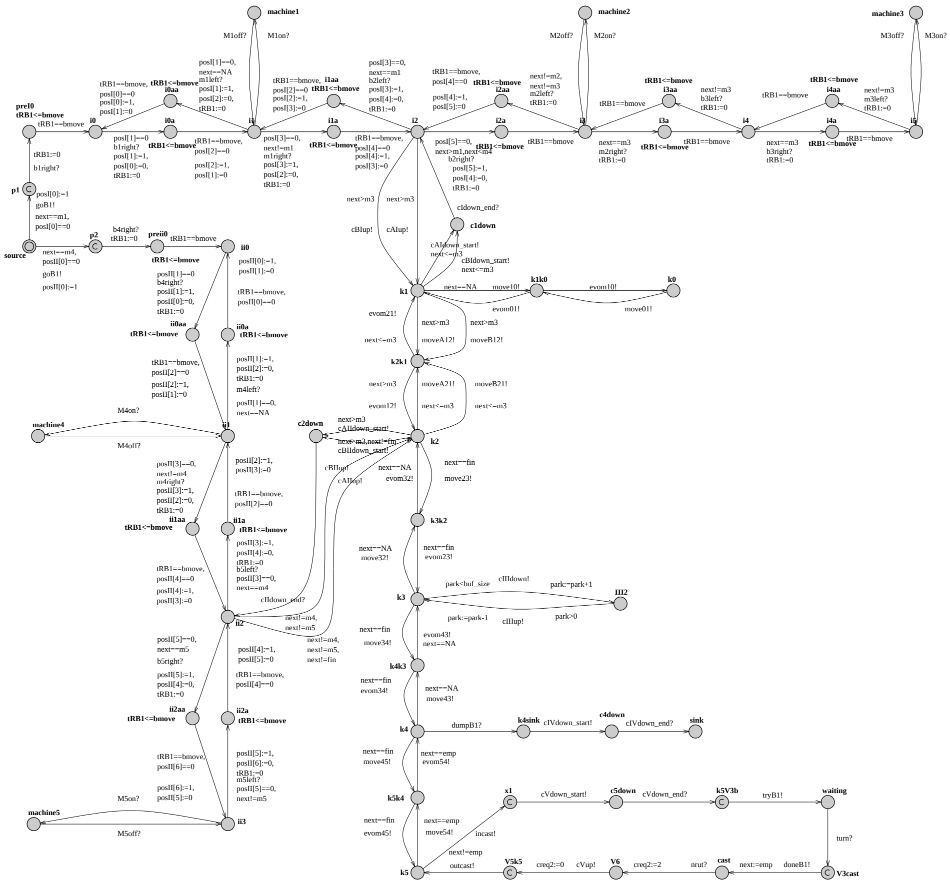

Fig. 7.4: The batch automaton.