Guided Synthesis of Control Programs Using UPPAAL*

Thomas Hune
BRICS,† Department of Computer Science
Aarhus University
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark
E-mail: baris@brics.dk

Kim G. Larsen   Paul Pettersson
BRICS,† Department of Computer Science,
Aalborg University
Fredriks Bajersvej 7E,
DK-9220 Aalborg East, 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. The plant model aims at faithfully reflecting the level of abstraction required for synthesizing control programs from generated timed traces. Therefore it quickly becomes too detailed and complicated for automatic synthesis. To solve this problem we present a general way of adding guidance to a model by augmenting it with additional guidance variables and decorating the transitions with extra guards. Applying this technique have made synthesis of control programs feasible for a plant producing as many as 60 batches. In comparison, we could only handle plants producing two batches without using guides.*

The synthesized control programs have been executed in a physical plant. This proved useful in validating the plant model and in finding some modeling errors.

Keywords: real-time verification, guided model-checking, scheduling, program synthesis, distributed systems.

1 Introduction

In this paper we suggest a solution to the problem of synthesizing and verifying valid control programs for resource allocation, based on a batch plant of SIDMAR [3, 5]. We model the plant as 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 reachability question allowing us to apply the real-time model-checking tool UPPAAL [7, 8] to derive a schedule. An overview of the methodology is shown in Figure 1.

UPPAAL offers a trace with actions of the model and timing information of the actions. The remaining effort required in transforming 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 synthesizing 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, have to be 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 users intuition 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®[1] MINDSTORMS™ plant, instead of the original plant of SIDMAR. We have used the plant to successfully run synthesized control programs and to increase our confidence in the plant model.

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 i n UPPAAL. In Section 4 and 5 we present the guiding techniques and evaluate its effect on the plant model. In Section 6 we describe experiments with the LEGO® plant and how programs are synthesized for the plant. Sec-

*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.

[1] See the web site http://www.lego.com/.

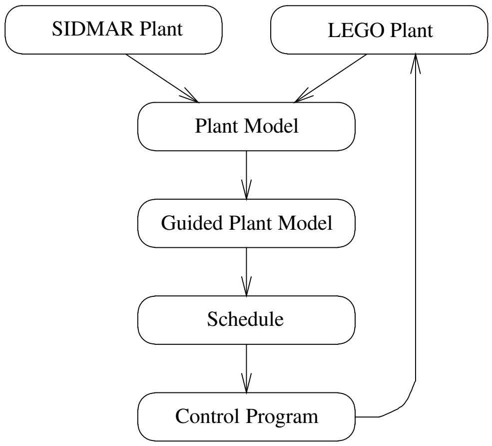

Figure 1. Overview of methodology.

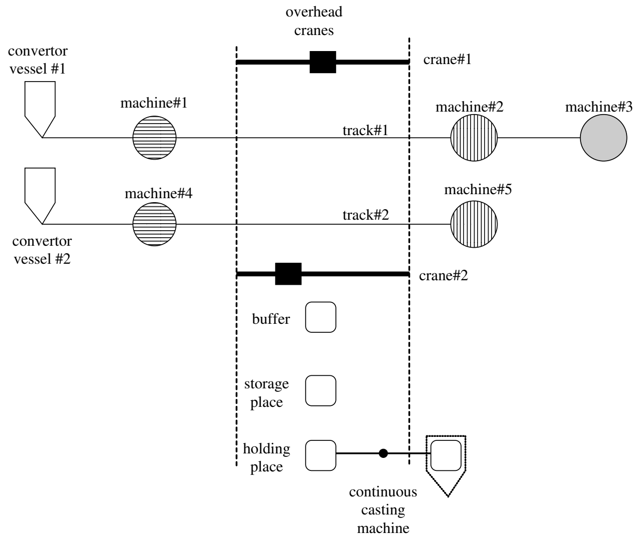

Figure 2. Layout of the plant.

tion 7 concludes the paper. Finally, timed automata descriptions of three 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. This is also a case study of the VHS project[2] (see [3, 5] for a description of the plant). Iron is poured into ladles which are used for transportation during the process. The iron is converted into steel by treatments in different machines and finally casted in the casting machine. Empty ladles must be moved to a storage place and then leave the system. 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.

The cranes can only hold one ladle and cannot overtake each other. Also on each track and in each machine there is room for at most one ladle. Only by using the crane the ladles can 'overtake' each other. Machines number one and four are of the same type and so are machines number two and five.

Because of the temperature of the steel there is an upper bound on the time a batch is allowed to spend in the plant from it is poured until it is casted. Casting of a ladle takes a specified time and must be continuous so when casting of one ladle has finished a new ladle must be waiting in the holding place of the casting machine.

Steel of different qualities can be produced depending on which types of machines are visited and for how long. For each batch this is decided by a recipe. The problem to be solved can now be stated as: *given an ordered list of qualities of steel, if possible synthesize a control program for the plant such that the qualities of steel are produced in the right order and within a given time*. A first step of solving this is to find a schedule for the plant.

3 Scheduling with Timed Automata

The scheduling problem can be solved in a number of ways. Here we use the real-time model checker UPPAAL [7, 8][3] (see [5] for a discussion of this approach). The plant is modeled using timed automata [2] allowing the scheduling problem to be stated as a reachability question that can be analyzed by UPPAAL. Timed automata are finite-state automata extended with clock variables, and guards over the clocks (a part of a timed automaton is shown in Figure 3). The result of a reachability analysis will be a trace defining a schedule for the plant. This trace will be translated into a working program controlling the plant. To make the translation as straightforward as possible, the produced trace should be as precise and detailed as possible, especially with respect to timing information.

Central to the model of the plant is the automaton representing the behavior of a batch (see Figure 9 of the appendix)[4]. The batch automaton reflects the topology as well as the physical constraints of the plant. Basically there is one location for each position in the plant, a position being either a machine,

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

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

[4] Pictures of all the automata and the LEGO® plant can be found at the web site `http://www.brics.dk/~baris/CaseStudy/`.

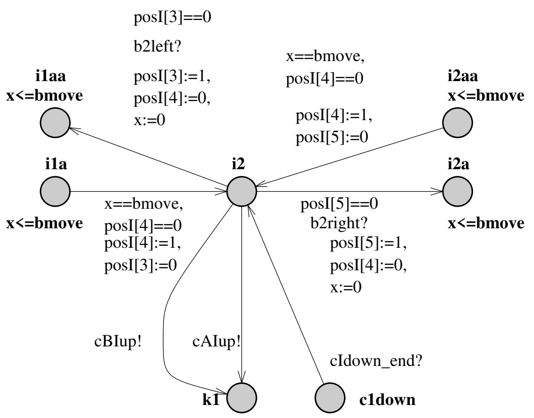

Figure 3. Part of the batch automaton.

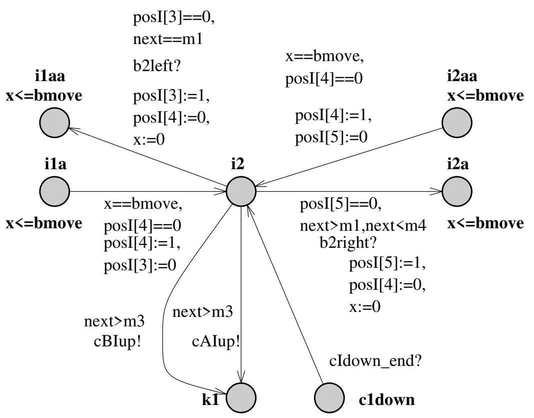

Figure 4. Guided part of the batch automaton.

the track between two machines, or a position on the overhead track when a batch is picked up by a crane. For each track there is one binary array used for storing which positions are occupied. There is one clock associated with the batch automaton used for ensuring that time passes when a batch is moving on a track. All timing constants in the model are worst case times for the movement. Figure 3 shows the part of the batch automaton modeling the position named **i2**, between machines number one and two on track one. Moving between positions is modeled in two steps. First a transition is taken to an intermediate position, e.g. from **i2** to **i1aa**, which resets the clock of the automaton (in this case, the clock named *x*). The worst case time for moving a batch between two positions is given by the constant *bmove*. The invariants in location **i1aa** and the guard on the transition leaving the location[5] ensures that exactly *bmove* time units passes in this location.

For each batch in the plant there is also an automaton modeling the recipe (a recipe using machine type one and two is shown in Figure 7 of the appendix). This defines which types of machines should be visited, for how long, and the order of the visits. When the batch is located at the right type of machine it has the possibility to synchronize with the recipe to have the machine turned on. When the specified time has passed the recipe and the batch synchronize again to have the machine turned off. The recipe also measures the overall time a batch has spend in the plant.

There is one automaton for each crane with two locations for each position a crane can be in, one representing the crane being empty and the other the crane carrying a batch (a crane automaton is shown in Figure 8 of the appendix). The time consuming movements of a crane are modeled using a clock and intermediate locations like in the batch automaton. In the same way as in the batch automaton, there is also a binary array for storing the positions occupied by the cranes. A crane and a batch automaton synchronize when a batch is picked up, moved, or set down.

Given a list of qualities of steel to be produced a model of the plant consists of one batch automaton and one recipe automaton for each quality of steel, two crane automata, an automaton representing the casting machine, and one automaton defining the list of qualities of steel. For example, a plant model with 60 batches consists of 125 timed automata (with 183 real-valued clocks).

4 Guiding Timed Automata

The plant model described in the previous section allows all possible behaviors of the physical plant. To keep the size of the state space manageable we need to restrict its behaviors. We will do this by guiding the state-space exploration according to a number of certain chosen strategies. The guides are implemented in the model by introducing a number of new variables, constraints (possibly also over existing clocks and variables), and assignments to the new variables[6]. Thus, in guiding a model we reduce its behavior. This ensures the essential property that any schedule generated for a guided model is indeed also a valid schedule of the original model.

We have implemented a number of strategies for guiding the state-space exploration. Due to space limitations we will only describe the guides abstractly, in terms of the physical plant, and give one detailed example of how the guides are introduced in the plant model. We emphasize that the strategies are heuristics. Most of them could in fact reduce the number of valid schedules of the plant model. However, this is not a

[5] The transition is not shown in the figure, it has the guard $x == bmove$ like the transition from **i1a** to **i2**.

[6] 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 [1].

#	All Guides						Some Guides						No Guides					
	BFS		DFS		BSH		BFS		DFS		BSH		BFS		DFS		BSH	
	sec	MB	sec	MB	sec	MB	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	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	-	-	2.1	4.4	7.8	1.2	-	-	19.5	36.1	-	-
3	-	-	3.2	6.5	3.4	1.4	-	-	72.4	92.1	901	3.4	-	-	-	-	-	-
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 1. Time and space requirements for generating schedules.

problem as long as it is still possible to generate valid schedules from the model.

When the scheduling problem is stated the production order of the steel qualities is given. One strategy is to use the order when starting new batches in the plant. To implement this we introduce the variable *nextbatch* to control which batch is allowed to start next. According to the engineers at SIDMAR the same strategy is used there.

Related to this strategy is the starting time of the batches. Since there is an upper bound on the time a batch is allowed to spend in the plant, all batches should not be allowed to start at the same time. Therefore, we prevent a batch from stating based on the progress of the batch just before it (the strategy is implemented by delaying the update of the *nextbatch* variable).

For guiding the movements of a batch we introduce a new variable named *next* for each batch. The value of *next* specifies where the batch should go next, based on its recipe. When there is a choice of machine the recipe will chose the machine on the track with fewest batches present. The choice of the first machine is implemented by an expression like:

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

where $track1$ is the number of batches present on track one and $track2$ the number of batches on track two.

A strategy for deciding how a batch moves from a given position to its destination has also been implemented. The strategy implemented selects the only direct route between the two positions. To implement this strategy in the plant model we use the *next* variable. A guard constraining on the value of *next* is added to all transitions leaving a position. Figure 4 shows the part of the batch automaton from Figure 3 with guiding guards added. Machine one is the only machine to the 'left' of position **i2** therefore *next* must have value $m1$ (machine 1) to move in that direction. This is ensured by the guard $next == m1$ on the transition from **i2** to **i1aa**. The transitions from **i2** to **k1** represents the batch being picked up by one of the cranes. If this is the case the 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 $m3$.

When a crane is carrying a batch it is always guided by the batch automaton. In general an empty crane should only move when something is ready to be picked up or if it is blocking the other crane moving a batch. Guards in the crane automata ensure that an empty crane only moves if a batch is waiting to be picked up or if the value of a guiding variable *cranereq* is non-zero. A crane moving to a position where the other crane could be blocking it, will set the *cranereq* variable to allow the blocking crane to leave.

It is possible to imagine other strategies and other experiments that would be interesting. However, the guides presented here have been very effective as shown by the results in the next section. Using the approach to guiding presented here allows for easy adding and changing of guides. This is important since guides are based on heuristics so experimenting is sometimes needed for finding good strategies.

5 Experimental Results

The plant models described in the previous sections have been analyzed in the validation and verification tool UPPAAL [7, 8]. In this section we present the results of the analysis for three versions of the model, with varying number of guides and batches. In particular, we present the measured time and space needed by UPPAAL to perform the analysis[7]. Comparing the requirements for the different models allows us to evaluate the benefits of the presented guiding techniques.

The three analyzed models are: the original plant model without the guides described in Section 3, the plant model with all guides added described in Section 4, and a model with all guides added except the once using the *nextbatch* variable described in Section 4.

[7] We use the standard UNIX programs `time` and `top` to measure the time and space consumptions.

UPPAAL offers a number of options to control the internal verification algorithm applied in the tool [8]. When analyzing the plant models we have used the compact data-structure for constraints [9], the control-structure reduction [9], and a recently implemented version of the (in-)active clock reduction [4]. In addition we experiment with using breadth-first (BFS), depth-first search strategy (DFS), or depth-first search in combination with bit-state hashing (BSH) [6][8].

In Table 1 we present the time (in seconds) and space (in MB) consumed by UPPAAL version 3.0.12 [9] when generating schedules. The positions marked with "-" indicate that the execution requires more than 256MB of memory, two hours of execution time, or that a suitable hash table size has not found[10].

As can be seen in Table 1, the use of guides allows us to generate schedules for 35 batches using 250 MB and 3.5 minutes, whereas no schedule can be generated for three batches when no guides are used. We also observe that adding some guides improves the situation by enabling analysis of systems with three batches. Finally, we observe that the bit-state hashing technique does not improve the situation when applied to model instances with guides, even though it performs well space-wise. We experienced that finding suitable hash table sizes is very tedious for large systems. Therefore the largest system analyzed in the experiment is a guided model using depth-first search without bit-state hashing. In fact, we have been able to generate schedules for models with as many as 60 batches on a Sun Ultra machine equipped with 1024 MB of memory.

6 Synthesis of Control Programs

As we did not expect to be able to run the generated control programs in the original plant of SIDMAR, we have used a LEGO® plant (see Figure 5) in which we run the synthesized programs. This allows for experimenting with the plant to validate the model and it also makes finding answers to a number of questions about the plant easy (e.g. measuring time bounds).

The plant consists of a number of distributed units, each controlled locally by one RCX™[10] brick. There are three types of units: a crane, a machine with a track segment, and the casting machine. Also an overhead track for the cranes exists in the LEGO® plant. Each unit is interfaced with a set of commands like *MoveTrackRight* and *LiftBatch*. The synthesized program will run in a central controller sending commands to the distributed local controllers. Since the communication between the RCX™ bricks is unreliable and slow, the only feedback from the local controllers are acknowledgements of commands received from the global controller. This has big influences on the kind of control programs we generate.

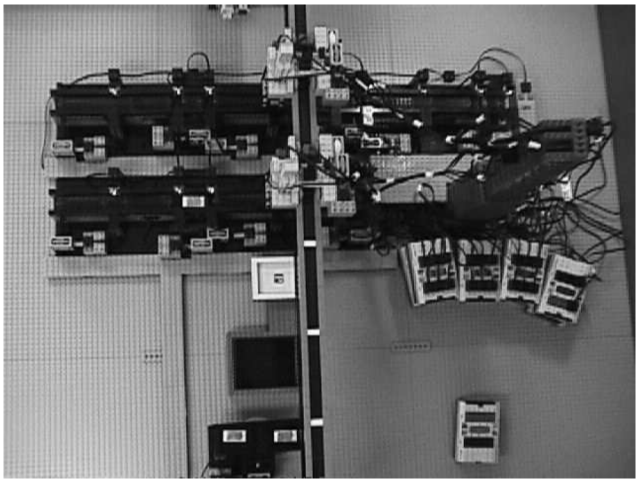

Figure 5. The LEGO® plant.

As a result of the model checking in UPPAAL a trace containing information about synchronizations between automata and delays is produced. Some of the synchronizations are not relevant for the scheduling. To get a schedule for the plant we project the trace to the actions relevant for the plant. Given some numbering of actions, part of a schedule looks like in Table 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 `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[11]. Figure 6 shows a part of a synthesized control program.

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: when the crane picked up an empty ladle from the casting machine it started to move horizontally at the same time

[8]The bit-state hashing technique generates a sub set of the reachable state-space. A feasible schedule found with this technique is therefore guaranteed to also be feasible in the original plant model.

[9]The tool was installed on a Linux Redhat 5.2 machine equipped with a Pentium III processor and 256MB of memory.

[10]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 size found.

[11]The code has to be in-lined as the language does not support function or procedure calls.

...	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	...

Table 2. Part of a generated schedule.

```
''''moveAup();
''''Crane A - Move UP
PB.PlaySystemSound 1
PB.SendPBMessage 2, 99 ' Move up, on C1
PB.SetVar 1, 15, 0 'Wait for ack
PB.While 0, 1, 3, 2, 99
  PB.Wait 2, 20
  PB.SetVar 1, 15, 0 'Read the message
  PB.ClearPBMessage
  PB.SumVar 2, 2, 1
  PB.If 0, 2, 2, 2, 20 'If looped 20 times
    PB.PlaySystemSound 1
    PB.SendPBMessage 2, 99 'Then Send message,
       again same as sendig 0
    PB.SetVar 2, 2, 0
  PB.EndIf
PB.EndWhile

''''Delay 12
PB.Wait 2, 1200
```

Figure 6. Part of a synthesized program.

as the pickup started, so here a delay was missing; 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; the casting machine did not turn correctly in systems with only one batch. These problems were corrected in the model and new control programs were synthesized.

Having the complete process from the model to synthesized control programs fully automated proved especially useful when the batteries got worn out. As a result the initial timing information, which was correct with new batteries, had to be changed. New times were measured and since scheduling was still possible, new programs were quickly generated and worked as expected.

Performing the experiments also validates the implementation of the translation from traces to programs and here no problems were found. Our confidence in the model has been significantly increased by conducting these 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

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

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

[3] Rene Boel and Geert Stremersch. VHS Case Study 5: Timed Petri net model of steel plant at SIDMAR. draft, 1999.

[4] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernard Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.

[5] Ansgar Fehnker. Scheduling a Steel Plant with Timed Automata. In *Proc. of the 6*th *International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 1999.

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

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

[8] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

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

[10] LEGO. *Software developers kit*, November 1998. See `http://www.legomindstorms.com/`.

Appendix

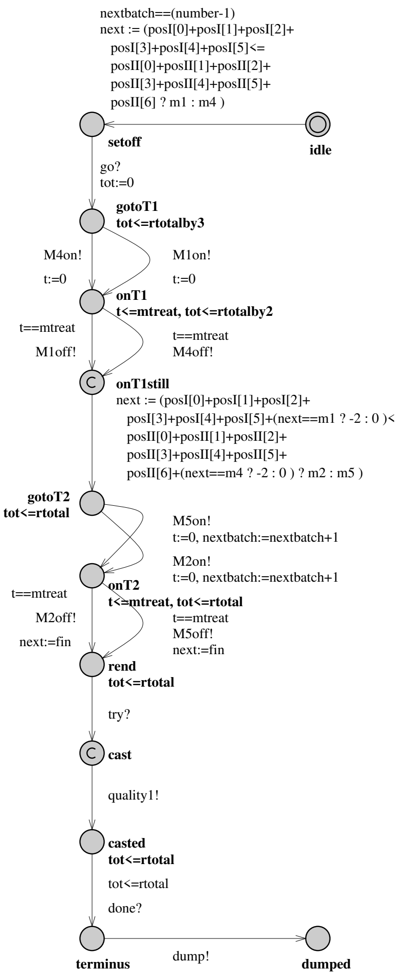

Figure 7. An example recipe automaton.

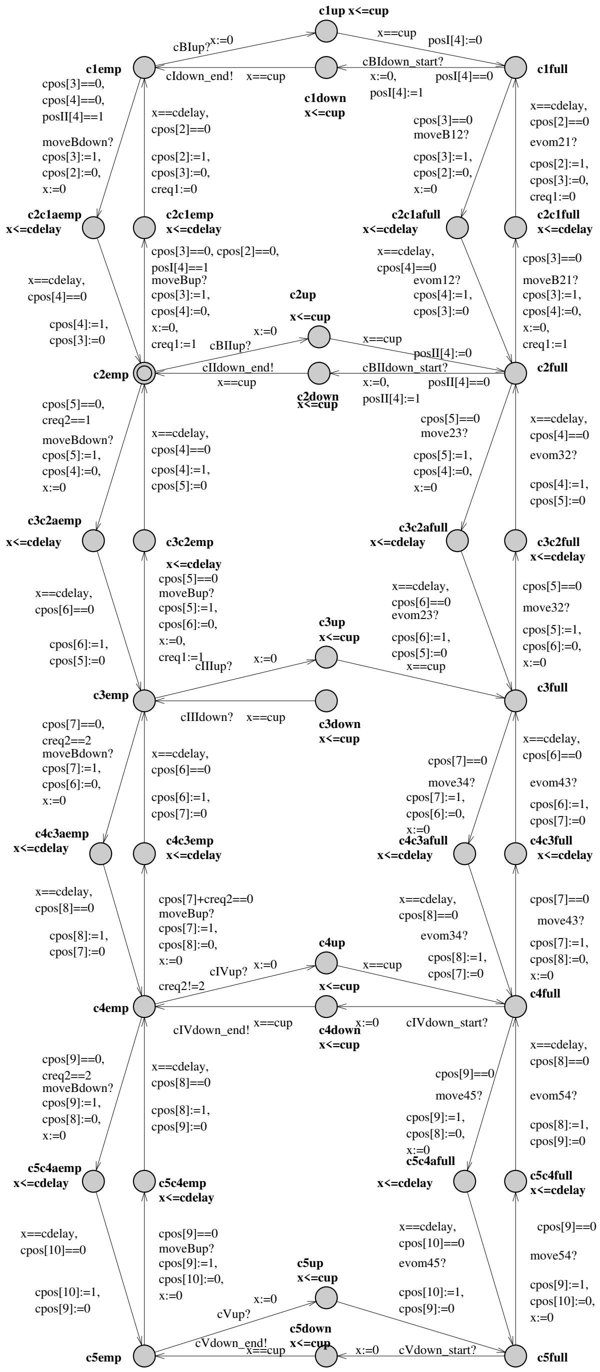

Figure 8. The lower crane.

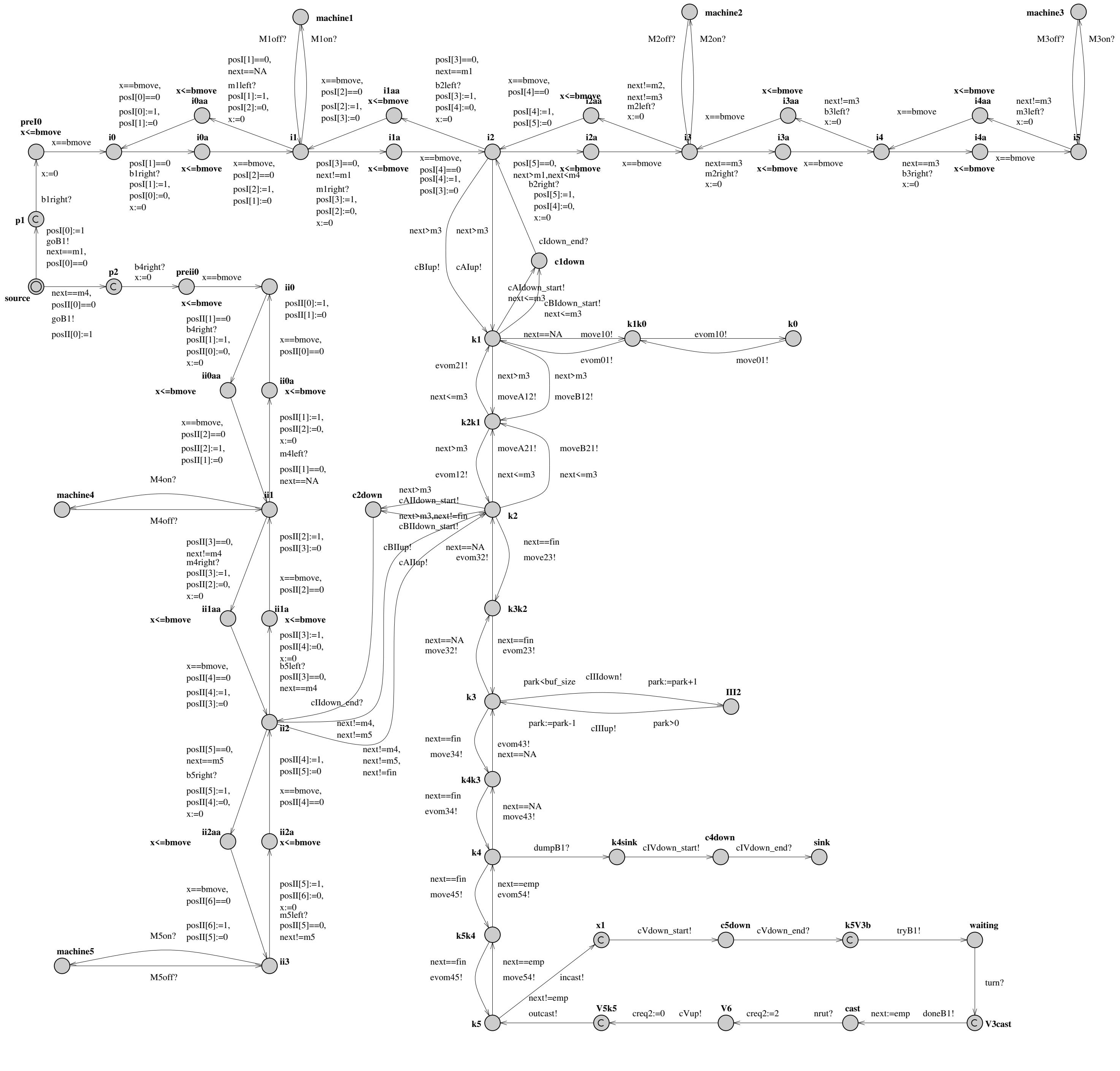

Figure 9. The batch automaton.