Production Cell Revisited

Dimitrie O. Păun Marsha Chechik * Bernd Biechele†

June 18, 1998

Abstract

This paper presents an analysis of the Production Cell system. We were able to model the system and verify most of its properties in Promela/SPIN. Our model is very close to the implementation level, and deriving code from it is trivial. In order to verify properties with SPIN's partial order reduction algorithms, we needed to ensure that all of our properties are closed under stuttering. We introduce the notion of logic edges and use them to show that properties of interest to us are closed under stuttering.

1 Introduction

In recent years model-checking has become a verification tool of choice for many projects. However, limitations of model-checking are well known - systems have to be finite, and abstractions have to be used to combat the state-space explosion. Although abstractions are essential in reasoning about complicated algorithms [6], they are not always natural or even feasible. Consider the following scenario: a customer specifies the environment at a certain level of abstraction. This might include timing, frequency of input sampling, etc. The goal of a system engineer then is to model and verify the system subject to the level of abstraction given by the customer. Mechanized checking is used to convince the customer that the model is correct. This process is quite different from the "designer-centered" point of view, where checking is used to verify correctness of the system, and we have a full control over the level of abstraction of the environment. Here, we only want to convince ourselves (i.e., not a customer) that our system is correct.

In our project, we worked with the Production Cell system [9]. This system has been specified in Promela/SPIN before by Thierry Cattel [2]. He described an approach to

*Contact address: Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4, Canada. Email: {dimi,chechik}@cs.toronto.edu.

†Contact address: Fakultät für Informatik, Universität Ulm, 89069 Ulm, Germany. Email: bernd@bach.informatik.uni-ulm.de.

specification by building models on different levels of abstraction - from a very abstract to more detailed ones through a sequence of refinement steps. Since SPIN cannot check equivalence between the levels of abstraction, Cattel used Concurrency Workbench [4] for this task. The goals of our work were quite different. We chose the Production Cell example to see if we can model non-trivial systems at a prescribed level of abstraction and check their properties without resorting to further abstraction. We were able to express in LTL and verify most of the properties described for the system in [9].

The rest of this paper is organized our as follows: Section 2 briefly describes the Production Cell case study. Section 3 describes the choices we made in modeling the system and suggests some modifications to SPIN. In Section 4, we discuss formalizing and proving safety and liveness properties about the system. In order to use SPIN's powerful partial order reduction algorithms, we needed to convince ourselves that the properties under analysis are closed under stuttering [8]. We describe our approach to proving closure under stuttering in Section 5. Section 6 summarizes our experience with the project.

2 The Production Cell System

The *Production Cell System* [9] is a case study initiated at the Forschungszentrum Informatik (FZI) to show the usefulness of formal methods for critical software systems and to show their applicability to real-world examples. In our work, we specified and verified a model of the Production Cell system in Promela/SPIN [7]. Below we provide a brief description of the specification of the system.

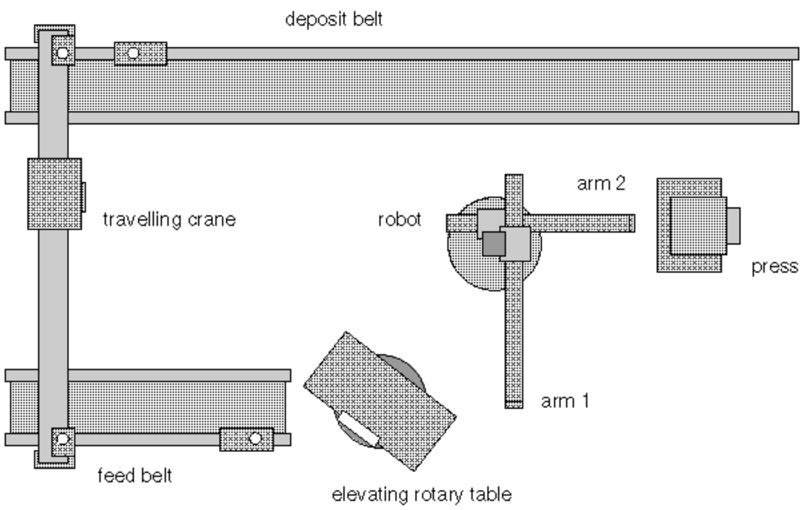

Figure 1: Top view of the Production Cell.

The production cell (see Figure 1) processes metal blanks which are conveyed to a press by a feed belt. A robot takes each blank from the feed belt and places it into the press. The robot arm withdraws from the press; the press processes the metal blank and opens again. Finally, the robot takes the forged metal plate out of the press and puts it on a deposit belt. To close the production cycle, the traveling crane picks up the forged blanks and puts them back onto the feed belt.[1] To increase throughput, the robot is equipped with two arms that can extend independently but have a fixed relative position.

3 Modeling the System

We designed the controller for the Production Cell so that it is a distributed, object-oriented system. Each hardware component is controlled by a process which has knowledge only about its immediate neighbours. A process has exclusive access to the component's sensors and actuators. The distributed nature of the controller allows for a high degree of parallelism between the processes while keeping the design simple.

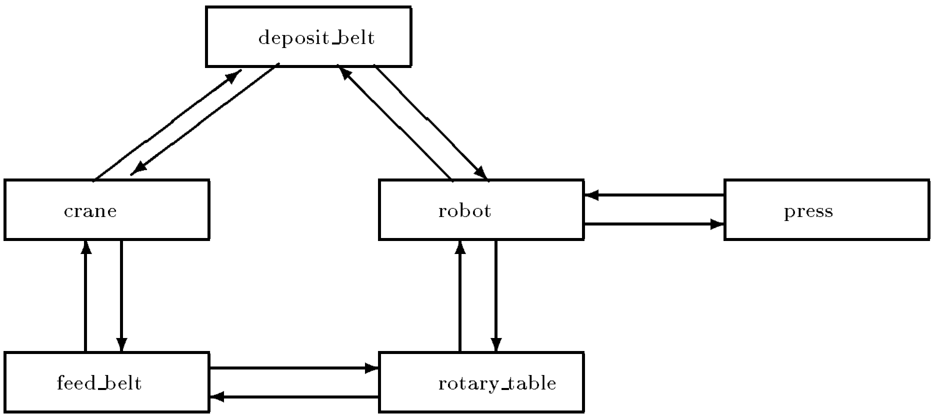

Figure 2: Architecture of The Controller.

The architecture of the controller is depicted in Figure 2. Each process is implemented as a Promela `proctype`; just one instance of each process is created. The processes exchange information through directed, synchronous communication channels and global variables. In order to model the system, we had to abstract from its reactive components and keep track of the state of hardware using global variables. We discuss these techniques below.

[1]In a "real" system, we would want to forge "new" blanks. However, the system specified in [9] has a closed production cycle to simulate a continuous stream of blanks. A closed cycle would typically be used to test the system.

3.1 Level of Abstraction

Abstraction is a powerful tool; properly used, it highlights the important characteristics of the problem and allows us to deal with the not-so-interesting details at a later stage. Unfortunately, SPIN provides no support for refinement, and thus one can not lower the abstraction level without potentially invalidating already proven properties.

Lack of refinement compels the user of the system to use one level of abstraction. The choice is normally constrained by two opposing needs: resource limitations force us to simplify the problem by using high levels of abstraction, while the desire for a correct implementation obliges us to use a level as close to the implementation as possible. Ideally, we would like to prove correct the *implementation*, because that is what the *customer* will ultimately use; the ease of modeling and the correctness of the model are side issues, relevant only to the *programmer*. In this study, our objective was to verify a model that is as close to the implementation as possible. The reasonable size of the problem, the state of the model checker SPIN, and the resources available on today's machines made this task feasible. The result is a detailed enough model that allows us an almost mechanical translation to the C code.

To keep the state space requirements for model-checking feasible, we had to carefully abstract from the time-dependent subsystems. On one hand, we were constrained to do this by SPIN's lack of support for modeling time. On the other hand, following the initial goal to create a model that is close to the implementation, we only abstracted from those parts of the system that could easily be refined into code separately without losing any confidence in the correctness of the implementation. So, in our case we abstracted from the *reactive components* of the system. All of these subcomponents fall in the category of feedback loop systems with the following form:

```
start(actuator)
wait_until(sensor <op> value)
stop(actuator)
```

where the action `stop(actuator)` is performed no later than a given time delay after the condition `sensor <op> value` becomes true. We assume no failures in the system and thus a finite completion time in the aforementioned components. To be able to reason about them separately, we also assume that they are controlled by a single process, and that the process blocks until the component completes its task. This restriction serializes some potentially parallel tasks. This has not been a problem in our case as the rest of the design captures the bulk of the possible parallelism present in the system.

So, under the above assumptions we can eliminate the reactive components by replacing them with statements that assign values to state variables. To ensure that no functionality of the model is dependent on these variables, they can only be *written to* inside the model. However, the state variables may be *used* in any temporal logic formulae. We ensure this by using a coding technique described in Section 3.2.

```
#ifdef HEAD_MGN
   bool head_mgn;
  #define SET_HEAD_MGN(s) head_mgn=s
#else
  #define SET_HEAD_MGN(s) skip
#endif
#define ACTIVATE_HEAD(s)                                \
  /*if s=true, insert code here to activate magnet*/    \
  SET_HEAD_MGN(s)                                       \
  /*if s=false, insert code here to deactivate magnet*/
```

Figure 3: An example macro for modeling hardware components.

3.2 Representing the Hardware State

To formalize and prove any of the required properties, we needed access to the state of the hardware. As mentioned above, we store this state in global variables. However, these variables do not reflect the real state of the hardware, and we want to ensure that the functionality of the model does not depend on these variables. Directly assigning variables in the model makes the code harder to read and maintain; the semantics of the hardware actions are lost and conversion to the C code is significantly more difficult. Moreover, the state space is enlarged unnecessarily as only a small subset of these variables are needed to prove any single property.

To overcome this problem, we developed a methodology that allows us to create models in which variables correctly reflect the state of the hardware without enlarging its size unnecessarily. We used macros like the one shown in Figure 3. In this example, we represent the hardware state of the magnet by the boolean variable `head_mgn`. The variable is declared only when the symbol `HEAD_MGN` is defined, eliminating unnecessary state-space enlargement. The macro `SET_HEAD_MGN` provides a consistent interface for writing to the variable, and is only used in the definition of `ACTIVATE_HEAD`. The later macro is used throughout the model to represent operations on hardware.

Similar macros are defined for all hardware actions. Aside from solving the aforementioned problems, such macros localize definition and usage of these variables in conjunction with the execution of the hardware action. This technique allows for simple checks[2] that the variables reflect the state of the hardware, and that the rest of the model does not depend on values of these variables. In fact, it is always safe to eliminate variables which are not read by the model nor the property under verification. Moreover, we believe that SPIN can be optimized to perform this reduction automatically, reducing the burden of the model designer.

[2]These checks are performed by simply searching the source code for the name of the variable; it should appear only in the macro definition and in the `never` claims.

4 Proving Properties

The original description of the system [9] contained a large number of properties concerning all aspects of its behaviour. We were able to translate many of these properties into LTL and verify them using the SPIN model-checker. The following is the summary of this work.

4.1 Timing Properties

The Production Cell description contained a number of *timing* properties - properties that require the system to react to the change in the environment within a specified amount of time. Typically, these properties were of the form

> *The system should react no later than t seconds from the moment sensor s changes state (or reaches value v) by turning actuator a on/off.*

These properties refer to the reactive behaviour, and we abstracted from the reactive components, as described in Section 3.1. Thus, we did not specify or verify these properties. How important is this limitation? At the level of abstraction we chose to model the system, these properties were not appropriate. Instead, we assumed that other verification methods could be used to ensure that the reactive components are implemented correctly. Since SPIN does not support direct verification of properties involving time, a different tool would have to be chosen to specify and verify these, e.g. [11].

4.2 Other Properties

The majority of the properties in the original description did not involve time. These included a number of safety and one liveness property. In order to translate the properties into linear-time logic (LTL) [10], we had to introduce a number of variables representing the state of hardware, as described in Section 3.2. In this section, we list several representative properties and describe the variables which appear in them. We present a complete list of properties in the Appendix. For more information about those and a description of all the system variables, please see [1].

Variables

`robot_pos` Describes the position of the robot; it may take the following values:

`POS_ARM1_TABLE`	Arm 1 is pointing to the rotary table;
`POS_ARM2_PRESS`	Arm 2 is pointing to the press;
`POS_ARM2_BELT`	Arm 2 is pointing to the deposit belt;
`POS_ARM1_PRESS`	Arm 1 is pointing to the press;
`POS_MOVING`	The robot is rotating.

`robot_ext[X]` Describes the state of the arm X; it may take the following values:

`EXT_RETRACTED`	The arm is retracted as far as possible;
`EXT_EXTENDING`	The arm is extending and its exact position is unknown;
`EXT_RETRACTING`	The arm is retracting and its exact position is unknown;
`EXT_INPRESS`	The arm is extended such that it is in the press;
`EXT_ABOVEBELT`	The arm is extended such that it is above the deposit belt. This setting makes sense only for arm 2;
`EXT_ABOVETABLE`	The arm is extended such that it is above the rotary table. This setting makes sense only for arm 1.

`robot_mgn[X]` Describes the state of the magnet of the arm X - `true` if the magnet is activated and `false` otherwise.

`forged[X]` Describes the state of the blank X - `true` if it has been forged and `false` otherwise. Initially, each blank is unforged. After going through the press, the blank is considered forged. It goes back to the unforged state when the traveling crane puts it onto the feed belt.

The Liveness Property

The liveness property of the Production Cell system,

> *Every blank introduced into the system via the feed belt will eventually be dropped by the crane on the feed belt again after having been forged.*

is formalized as

$$\Box((\mathit{forged[X]} \Rightarrow \Diamond\neg \mathit{forged[X]}) \wedge \ (\neg \mathit{forged[X]} \Rightarrow \Diamond \mathit{forged[X]}))$$

That is, a blank will infinitely toggle between its forged and unforged state. This property is checked for each blank in the system.

Safety Properties

The property

> *The magnet of arm 1 may only be deactivated if the arm points towards the press and the arm is extended such that it reaches the press.*

is represented as

$$\Box((robot\_mgn[0] \wedge \circ\neg robot\_mgn[0]) \Rightarrow$$
$$(robot\_pos = POS\_ARM1\_PRESS \wedge robot\_ext[0] = EXT\_INPRESS))$$

So, if the magnet on arm 1 is deactivated (i.e. the variable `robot_mgn[0]` changes its value from `true` to `false`), then arm 1 must be positioned inside the press.

Another property

The press may only close when no robot arm is positioned inside it.

is formalized as

$$\Box(press\_closed \Rightarrow$$
$$((arm1\_retracted \wedge arm2\_retracted) \vee$$
$$(robot\_pos = POS\_ARM1\_TABLE \wedge arm2\_retracted) \vee$$
$$(robot\_pos = POS\_ARM2\_BELT \wedge arm1\_retracted))),$$

where
$$arm1\_retracted = robot\_ext[0] = EXT\_RETRACTED$$
$$arm2\_retracted = robot\_ext[1] = EXT\_RETRACTED$$

This formula describes a slightly stronger property: the press may close only when both arms are retracted or one of them is pointing away from the press and the other one is retracted.

4.3 Verification Results

The Promela code in our model consists of about 300 lines. It includes 8 processes communicating through 15 channels and uses 19 variables to represent the hardware state. The system is designed so that up to 8 blanks can stay in the production cell concurrently. Due to memory constraints, we were able to prove all properties with a full state-space search for only up to 3 blanks present in the system.

In Table 1, we list the properties we were able to check, together with their verification time, memory used, and the number of computed transitions. Property numbers refer to the original description of the system [9]. We adopted the same numbering convention when listing the properties in the Appendix. The time measurements are approximate and are reported in seconds; the memory requirements reflect the real memory used, rounded to the nearest megabyte; the number of transitions is given in millions. Our tests were performed on a Sun Ultra Enterprise 3000 with 4 250MHz, 4MB cache UltraSPARC CPUs and 1GB RAM, using the SPIN version 2.9.5. The partial-ordering reduction algorithm was enabled, but no special memory compression options were used.

	Property													
	0	2a	2b	2c	3a	3b	3c	3d	3e	4a	4b	4c	4d	4e
Time(s)	56	18	15	15	22	16	18	13	54	27	20	29	29	16
Mem(MB)	80	59	55	56	66	57	61	54	93	65	60	65	66	56
Transitions(M)	1.4	0.4	0.3	0.3	0.5	0.4	0.4	0.3	1.3	0.6	0.5	0.7	0.6	0.4

Table 1: Resource requirements.

5 Closure Under Stuttering

To reduce the size of the state-space, SPIN uses a partial-order reduction technique which is applicable only to formulae that are closed under stuttering [8]. A formula is closed under stuttering (c.u.s) if its interpretation does not change when we repeat (stutter) some states. For example, $\Box a$ is c.u.s, whereas $\circ a$ is not.

Currently, one can easily ensure that a formula is c.u.s by avoiding the "next" operator [8]. Despite the nice mathematical properties of the resulting language, it is not expressive enough even for fairly simple properties, as we found out in our study. An algorithm which can identify a larger class of c.u.s formulae has been proposed in [5] but not yet implemented in SPIN. A complete procedure for recognizing c.u.s formulae, based on the ω-automata tableau-building algorithm, has also been developed and shown to be PSPACE-complete [12].

Unable to find the appropriate "next"-free LTL formalization for the desired properties and not having access to the automated procedures described in [5] and [12], we resorted to showing that these properties are c.u.s by hand. Our formulae are quite stylized: the "next" operator is used to identify *events* — that is, changes in the values of variables, e.g.

$$\begin{aligned} &\Box((\mathit{robot\_mgn[1]} \wedge \circ\neg \mathit{robot\_mgn[1]}) \Rightarrow \\ &\circ(\neg \mathit{new\_on\_deposit}\ \mathcal{U}\ (\neg \mathit{deposit\_sen} \wedge \circ\, \mathit{deposit\_sen}))) \end{aligned} \tag{1}$$

This formula says that when the robot drops an object on the deposit belt (there is a falling edge in *robot_mgn[1]*), a new one should not be deposited (*deposit_sen* should remain false) until the previous object has been cleared (there is a rising edge in *deposit_sen*).

We observed that stuttering cannot add or remove edges in propositional formulae which led us to develop a simple language and well-formedness criteria that allow us to easily identify a class of c.u.s formulae. This work is described in detail in [3]. We present some basic notions of this theory here.

Definition 5.1 *For a propositional formula a, $\uparrow a = \neg a \wedge \circ a$, $\downarrow a = a \wedge \circ\neg a$.*

We use $\uparrow a$ to indicate an *rising edge*, $\downarrow a$ to indicate a *falling edge*, and $\updownarrow a$ to indicate $\uparrow a$ or $\downarrow a$. The edges are related:

$$\uparrow \neg a = \downarrow a \tag{2}$$

$$\downarrow \neg a = \uparrow a \tag{3}$$

If a formula F is c.u.s, we write it as $\boxed{\mathrm{F}}$. So, if F does not contain a "next" operator, then $\boxed{\mathrm{F}}$ [8]. For arbitrary LTL formulae F, A and B, and a propositional formula a, the following relationships hold:

$$\boxed{\mathrm{F}} \Leftrightarrow \boxed{\neg\mathrm{F}} \tag{4}$$

$$\boxed{\mathrm{F}} \Rightarrow \boxed{\Box\mathrm{F}} \tag{5}$$

$$\boxed{\mathrm{F}} \Rightarrow \boxed{\Diamond\mathrm{F}} \tag{6}$$

$$\boxed{\mathrm{A}} \wedge \boxed{\mathrm{B}} \Rightarrow \boxed{\mathrm{A} \wedge \mathrm{B}} \tag{7}$$

$$\boxed{\mathrm{A}} \vee \boxed{\mathrm{B}} \Rightarrow \boxed{\mathrm{A} \vee \mathrm{B}} \tag{8}$$

$$\boxed{\mathrm{F}} \Rightarrow \boxed{\Box(\updownarrow\mathrm{a} \Rightarrow \mathrm{F})} \tag{9}$$

$$\boxed{\mathrm{F}} \Rightarrow \boxed{\Box(\updownarrow\mathrm{a} \Rightarrow \circ\, \mathrm{F})} \tag{10}$$

$$\boxed{\mathrm{F}} \Rightarrow \boxed{\mathrm{F}\ \mathcal{U} \updownarrow\mathrm{a}} \tag{11}$$

As remarked above, stuttering cannot add or remove an edge on a; it can only shift it relative to the start of the sequence of states. This observation is essential to understanding Equations 9–11. For example, Equation 9 is true because the "always" operator follows the shifting edge through the sequence of states; F is applied to the sequence from the edge onwards, and its interpretation is maintained due to its closure under stuttering. Equation 10 allows the explicit usage of the "next" operator. In this equation, an edge on a fixes two states: the one where a is false (true), and the next one where a is true (false). Thus, F can refer to either of these states and not change the interpretation of the resulting formula under stuttering. Property 1 described above is represented as

$$\Box(\downarrow \mathit{robot\_mgn[1]} \Rightarrow \circ\, (\neg \mathit{new\_on\_deposit}\ \mathcal{U}\ \uparrow \mathit{deposit\_sen}))$$

Equation 11 implies that $\boxed{\neg \mathit{new\_on\_deposit}\ \mathcal{U} \uparrow \mathit{deposit\_sen}}$, and thus the whole property is c.u.s via Equation 10.

Most properties encountered in the specification of the Production Cell, e.g. 3a-e, 4a-e, required the use of the "next" operator but were easily shown to be c.u.s using the aforementioned patterns.

6 Conclusion

This paper describes results of a case study in which we specified and verified properties of a Production Cell system. We chose the "customer"-centered point of view and modeled the system at a fairly low level of abstraction. We abstracted from the reactive components, replacing them by variables, and proposed a technique aimed to reduce the size of the model. We also described a simple language that allowed us to easily show that LTL formulae used in our study are closed under stuttering, even though some of them contained the "next" operator. We were able to formalize and verify the properties for the system containing 3 blanks, and deriving code from our model was easy. Our experience suggests that model-checking can easily be used to reason about implementations of non-trivial systems.

References

[1] Bernd Biechele and Dimitrie O. Păun. "A Case Study Production Cell with PROMELA/SPIN". In M. Chechik, editor, *Automated Verification: A Collection of Reports*. University of Toronto, Technical Report CSRG-374, 1998.

[2] Thierry Cattel. "Process Control Design Using SPIN". In *Proceedings of SPIN Workshop*, Montreal, Canada, 1995.

[3] M. Chechik and Dimitrie O. Păun. "Linear-Time Properties for Event-Based and State-Based Formalisms". (in preparation), June 1998.

[4] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems". *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[5] G. J. Holzmann and O. Kupferman. "Not Checking for Closure under Stuttering". In *Proceedings of SPIN Workshop*, 1996.

[6] Gerard Holzmann. Keynote address: "Designing Executable Abstractions". In *Proceedings of 2nd Workshop on Formal Methods in Software Practice*, March 1998.

[7] G.J. Holzmann. "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[8] Leslie Lamport. "The Temporal Logic of Actions". *ACM Transactions on Programming Languages and Systems*, 16:872–923, May 1994.

[9] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems. Case Study Production Cell.* Springer-Verlag, Berlin, 1995.

[10] Z. Manna and A. Pnueli. "Tools and Rules for the Practicing Verifier". Technical Report STAN-CS-90-1321, Department of Computer Science, Stanford University, 1990.

[11] J. Ostroff. "A Visual Toolset for the Design of Real-Time Discrete Event Systems". *IEEE Transactions on Control Systems Technology*, May 1997.

[12] Doron Peled, Thomas Wilke, and Pierre Wolper. "An Algorithmic Approach for Checking Closure Properties of ω-Regular Languages". In *Proceedings of CONCUR '96: 7th International Conference on Concurrency Theory*, August 1996.

A Properties of the Production Cell System

In this appendix, we list the properties we were able to formalize and verify, keeping the numbering convention introduced in [9]. For a complete list of state variables, please refer to [1].

0 (Liveness) Every blank introduced into the system via the feed belt will eventually be dropped by the crane on the feed belt again after having been forged.

$$\Box((\mathit{forged[X]} \Rightarrow \Diamond\neg \mathit{forged[X]}) \wedge (\neg \mathit{forged[X]} \Rightarrow \Diamond \mathit{forged[X]}))$$

2(a) The press may only close when no robot arm is positioned inside it.

$$\begin{aligned}&\Box(\mathit{press\_closed} \Rightarrow\\ &\quad((\mathit{arm1\_retracted} \wedge \mathit{arm2\_retracted})\vee\\ &\quad(\mathit{robot\_pos} = \mathit{POS\_ARM1\_TABLE} \wedge \mathit{arm2\_retracted})\vee\\ &\quad(\mathit{robot\_pos} = \mathit{POS\_ARM2\_BELT} \wedge \mathit{arm1\_retracted})))\end{aligned}$$

where

$$\begin{aligned}\mathit{arm1\_retracted} &= \mathit{robot\_ext}[0] = \mathit{EXT\_RETRACTED}\\ \mathit{arm2\_retracted} &= \mathit{robot\_ext}[1] = \mathit{EXT\_RETRACTED}\end{aligned}$$

2(b) A robot arm may only rotate in the proximity of the press if the arm is retracted or if the press is in its upper or lower position.

$$\begin{aligned}&\Box(\mathit{robot\_pos} = \mathit{POS\_MOVING} \Rightarrow\\ &\quad(\mathit{robot\_ext[0]} = \mathit{EXT\_RETRACTED} \wedge\\ &\qquad \mathit{robot\_ext[1]} = \mathit{EXT\_RETRACTED}))\end{aligned}$$

2(c) The traveling crane is not allowed to knock against a belt laterally.

$$\Box(\mathit{head\_hor} = \mathit{HEAD\_HORMV} \Rightarrow \mathit{head\_ver} = \mathit{HEAD\_UP})$$

3(a) The magnet of arm 1 may only be deactivated, if the arm points towards the press and the arm is extended such that it reaches the press.

$$\begin{aligned}&\Box((\mathit{robot\_mgn[0]} \wedge \circ\neg \mathit{robot\_mgn[0]}) \Rightarrow\\ &\quad(\mathit{robot\_pos} = \mathit{POS\_ARM1\_PRESS} \wedge\\ &\qquad \mathit{robot\_ext[0]} = \mathit{EXT\_INPRESS}))\end{aligned}$$

3(b) The magnet of arm 2 may only be deactivated if its position is above the deposit belt.

$$\Box((\mathit{robot\_mgn[1]} \wedge \circ\neg \mathit{robot\_mgn[1]}) \Rightarrow (\mathit{robot\_pos} = \mathit{POS\_ARM2\_BELT} \wedge \mathit{robot\_ext[1]} = \mathit{EXT\_ABOVEBELT}))$$

3(c) The magnet of the crane may only be deactivated if its magnet is above the feed belt and sufficiently close to it.

$$\Box((\mathit{head\_mgn} \wedge \circ\neg \mathit{head\_mgn}) \Rightarrow (\mathit{head\_ver} = \mathit{HEAD\_DOWN} \wedge \mathit{head\_hor} = \mathit{HEAD\_FEED}))$$

3(d) The feed belt may only convey a blank through its light barrier if the table is in the loading position.

$$\Box((\mathit{feed\_sen} \wedge \circ\neg \mathit{feed\_sen}) \Rightarrow \mathit{rotary\_loading})$$

3(e) The deposit belt must be stopped after a blank has passed the light barrier at its end and may only be started after the crane has picked up the blank.

$$\Box(((\mathit{deposit\_sen} \wedge \circ\neg \mathit{deposit\_sen}) \Rightarrow \circ\neg(\mathit{deposit\_eng}\, \mathcal{U}\, \mathit{deposit\_sen})) \wedge ((\mathit{deposit\_eng} \wedge \circ\neg \mathit{deposit\_eng}) \Rightarrow \circ(\neg \mathit{deposit\_eng}\, \mathcal{U}\, \mathit{pickup})))$$

where

$$\mathit{pickup} = (\mathit{head\_hor} = \mathit{HEAD\_DEPOSIT} \wedge \mathit{head\_ver} = \mathit{HEAD\_DOWN} \wedge \mathit{head\_mgn})$$

4(a) A new blank may only be put on the feed belt if sensor 14 confirms that the last one has arrived at the end of the feed belt.

$$\Box(\mathit{new\_on\_feed} \Rightarrow \circ(\neg \mathit{new\_on\_feed}\, \mathcal{U}\, (\neg \mathit{feed\_sen} \wedge \circ \mathit{feed\_sen})))$$

where

$$\mathit{new\_on\_feed} = (\mathit{head\_mgn} \wedge \circ\neg \mathit{head\_mgn}) \vee (\mathit{new\_ins} \wedge \circ\neg \mathit{new\_ins})$$

4(b) A new blank may only be put on the deposit belt if sensor 13 confirms that the last one has arrived at the end of the deposit belt.

$$\Box(\mathit{new\_on\_deposit} \Rightarrow \circ(\neg \mathit{new\_on\_deposit}\, \mathcal{U}\, (\neg \mathit{deposit\_sen} \wedge \circ \mathit{deposit\_sen})))$$

where

$$\mathit{new\_on\_deposit} = (\mathit{robot\_mgn[1]} \wedge \circ\neg \mathit{robot\_mgn[1]})$$

4(c) Do not put blanks on the table if it is already loaded.

$$\Box(\mathit{push\_on\_table} \Rightarrow \circ(\neg \mathit{push\_on\_table}\, \mathcal{U}\, \mathit{robot\_pickup}))$$

where

$$\begin{aligned} &\mathit{push\_on\_table} = (\mathit{feed\_sen} \wedge \circ \neg \mathit{feed\_sen}) \\ &\mathit{robot\_pickup} = (\neg \mathit{robot\_mgn[0]} \wedge \circ\, \mathit{robot\_mgn[0]}) \end{aligned}$$

4(d) Do not put blanks into the press if it is already loaded.

$$\Box(\mathit{drop\_in\_press} \Rightarrow \circ(\neg \mathit{drop\_in\_press}\, \mathcal{U}\, \mathit{pickup\_from\_press}))$$

where

$$\begin{aligned} &\mathit{drop\_in\_press} = (\mathit{robot\_mgn[0]} \wedge \circ \neg \mathit{robot\_mgn[0]}) \\ &\mathit{pickup\_from\_press} = (\neg \mathit{robot\_mgn[1]} \wedge \circ\, \mathit{robot\_mgn[1]}) \end{aligned}$$

4(e) Do not move the loaded robot arm 1 above the loaded table if the latter is in unloading position.

$$\begin{aligned} \Box(((&\mathit{robot\_mgn[0]} \wedge \neg \mathit{abovetbl}) \Rightarrow \circ \neg(\mathit{robot\_mgn[0]} \wedge \mathit{abovetbl})) \wedge \\ &\mathit{abovetbl} \Rightarrow (\mathit{rotary\_unloading}\, \mathcal{U}\, \neg \mathit{abovetbl})) \end{aligned}$$

where

$$\begin{aligned} \mathit{abovetbl} = (&\mathit{robot\_pos} = \mathit{POS\_ARM1\_TABLE} \wedge \\ &\mathit{robot\_ext[0]} = \mathit{EXT\_ABOVETABLE}) \end{aligned}$$