EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Verification and performance analysis of a paint factory

Baeten, M.C.

Award date:
2007

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. Mar. 2025

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Verification and Performance Analysis of a Paint Factory

By
Michelle Baeten

Supervisors:

dr. S.P. Luttik
N. Trčka
Department of Mathematics and Computing Science
Formal Methods Group
Technische Universiteit Eindhoven

March 2006 - March 2007

Abstract

This thesis researches in what way it is possible to perform model checking and performance analysis on a χ specification. As a case study, we studied a χ specification of a mock up model of a paint factory, constructed by the System Engineering section of the department of Mechanical Engineering.

χ is a specification language developed to model concurrent manufacturing systems. In this project, we create a new χ specification in χ 1.0. We will simulate this specification using a χ 1.0 simulator, to see if the behavior is as expected.

Then we use model checking techniques to verify the correctness of the specification. For model checking, we will use a tool called SPIN. To be able to use this tool, we will translate the specification to the language of SPIN called Promela. After translation, we will use SPIN to simulate the model and to check various properties.

After model checking we will do performance analysis. We want to know how well the specification performs and if changes we made have improved the performance. To do this, we will look at results we received from simulation, and use Markov chain analysis.

Finally, we investigate the possibility to use the paint factory model in Computer science education in high schools, and we describe a possible scenario for this.

Contents

1 Introduction 4
1.1 Background 4
1.2 Research question and objectives 4
1.3 Deliverables 5
1.4 Outline 5
1.5 Acknowledgements 5

2 The Paint Factory 6
2.1 Hardware 6
2.2 Control system 10

3 The χ specification 14
3.1 The language χ 14
3.2 Specification of the paint factory 15
3.2.1 The generator 15
3.2.2 The dispatcher 16
3.2.3 The resource controller 17
3.2.4 The hardware controller groups 18
3.3 The Sensors 21

4 Verifying Correctness 23
4.1 Simulation 23
4.2 Model checking 23
4.2.1 SPIN 24
4.3 Requirements 28
4.3.1 Requirements on the controllers 29
4.3.2 Requirements on the hardware 30

5 Performance Analysis 37
5.1 Simulation 37
5.2 Using mathematical techniques 38

6 The use of the paint factory in high Schools 44

7 Conclusions 46

8 Recommendations **48**

8.1 The specification language χ 48

8.2 Tools 48

8.3 Performance Analysis 49

8.4 The use of the paint factory in Computer Science education in high schools 49

A Original χ 0.8 specification **52**

B Promela specification **57**

C Adapted χ 1.0 specification **79**

D χ_σ specification **90**

1 Introduction

1.1 Background

The specification language χ has been developed about 20 years ago by a group lead by Prof. J.E. Rooda. It was originally developed to model concurrent manufacturing systems and to analyze these models by means of simulation. The χ language is still developing and changing. Nowadays, χ is becoming more and more a formal language, with possibilities for both formal performance analysis and verification. The project doing research in this area is called TIPSy [12]. This project is researching the use of several existing tools for model checking and performance analysis of χ specifications, and has created translation tools for χ to be able to use these tools.

For instance, one of the tools available for model checking is called SPIN. SPIN is an open-source software tool developed by Bell Labs. For more information about SPIN, see [4]. The input language of SPIN is Promela. To be able to use SPIN for model checking of χ specifications, the χ specification needs to be translated to Promela. Therefore, a translator tool has been developed by the TIPSy group. This translator tool can translate χ specifications to Promela.

Another useful tool is the CADP toolset. The CADP toolset is a toolbox for the design of communication protocols and distributed systems. Some of the tools in this toolkit are for doing performance analysis. CADP is developed by the VASY team at INRIA Rhone-Alpes. For further information about CADP, see [10]. To be able to use the CADP toolset, we use a tool that connects the χ specification to CADP. This tool has been developed by N. Trcka and uses the language χ_σ as input language. This language is developed to provide a formal framework for χ specifications, thus making it possible to do calculations like performance analysis. More information about χ_σ can be found in [11].

In this project, as a case study we will use the χ specification of a paint factory. This paint factory is a mock up model of a real paint factory, and has been constructed by the section of Systems Engineering of the faculty of Mechanical Engineering. The paint factory is constructed in 1998, and has 3 vessels containing 3 basic colors which can be mixed to yield a variety of colors, in different amounts and combinations. The first setup of the paint factory can be found in [1]. The paint factory is further described in Chapter 2.

We start from a specification written by members of the Systems Engineering group. Since 1998, several research projects have taken place regarding the paint factory. In 2000, M.H.M. van Duin presented the first detailed χ specification, in χ 0.6 [2]. In 2005, a new χ version was created in χ 0.8, by W.A.P van den Bremer [3].

1.2 Research question and objectives

In this project, we want to research a formal method for χ specifications to do model checking and performance analysis. We also want to test the usability of

several tools created for model checking and performance analysis. As a case study, we study the χ specification of a paint factory. The research objectives are the following:

- **Study the current χ specification. Find out what research has been done on the paint factory and the specification.**
- **Find out what tools exist for model checking and how they can be used for a χ model, and in particular for the paint factory specification.**
- **Through model checking, find out if there are errors in the χ specification, correct these, and optimize the specification**
- **Analyze the performance of the specification.**
- **Research the possibilities for use of the paint factory in Computer Science education in High Schools.**

1.3 Deliverables

The deliverable is in the first place a description of the results of the research described above. Furthermore, to be able to use the several tools described, we had to translate the χ specification into several other languages. Therefore, the various versions we made of the paint factory specifications are also deliverables.

1.4 Outline

In Chapter 2, we explain the paint factory hardware in the way it has been constructed by the Mechanical Engineering department. In Chapter 3, we describe the structure of the χ model that has been set up. Chapter 4 describes the steps we have taken in model checking and improving the specification. Chapter 5 explains the performance analysis we have done and the results of this. Chapter 6 investigates the possibility for using the paint factory in Computer Science education in High Schools. Finally, Chapters 7 and 8 contain conclusions and recommendations for this project.

1.5 Acknowledgements

First of all, I would like to thank my supervisor, Bas Luttik. He provided a lot of feedback and helped me when necessary. Secondly, I would like to thank Nikola Trcka, for all the help and feedback he gave me on SPIN and $\chi\sigma$.

2 The Paint Factory

In this chapter, we will describe how the paint factory works. First, we will describe the hardware of the paint factory. Secondly, we will describe the controller model that has been created for the paint factory.

The paint factory can be divided into a hardware and a controller part. The hierarchy is shown in Figure 1.

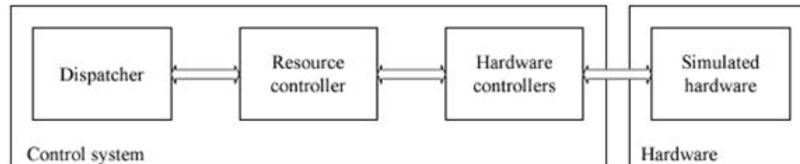
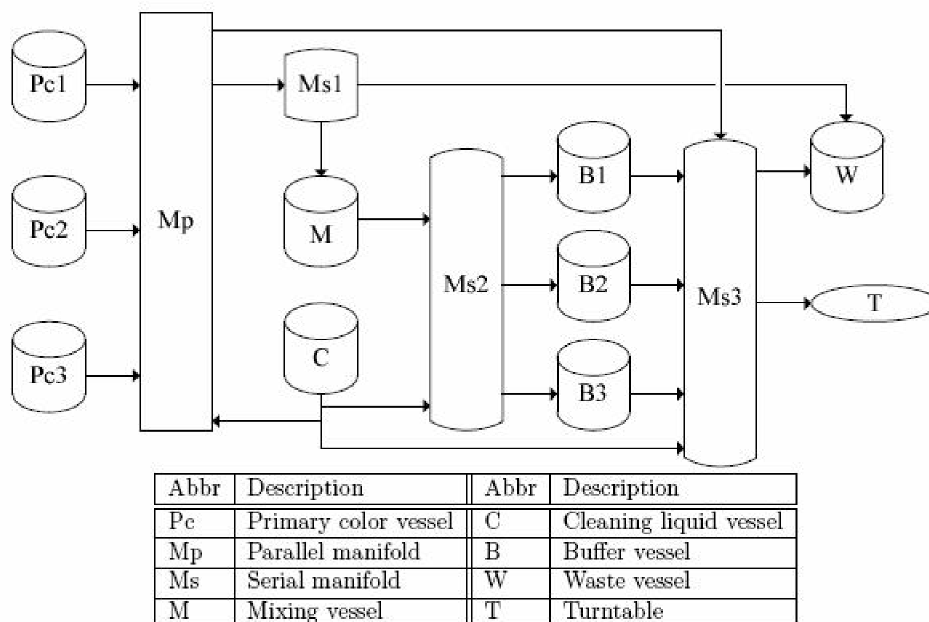


Figure 1: Hierarchy of the controller and hardware

2.1 Hardware

The paint factory uses three primary colors (red, blue and yellow) to mix every possible color. These primary colors are stored in three vessels (represented by Pc1-3 in Figure 2). Besides these vessels there exists one mixing vessel (represented by M in Figure 2), and three buffer vessels (B1-3) to store mixed paint. The paint is outputted to a turntable (T), a rotating platform with 12 cups to put the paint in. Furthermore, there exists a vessel containing cleaning fluid (C), and a waste disposal vessel (W). To get paint from one place to another, pipes, vessels and manifolds are used. Figure 2 shows these resources.



Abbr	Description	Abbr	Description
Pc	Primary color vessel	C	Cleaning liquid vessel
Mp	Parallel manifold	B	Buffer vessel
Ms	Serial manifold	W	Waste vessel
M	Mixing vessel	T	Turntable

Figure 2: Global presentation

The manifolds in Figure 2 are in fact clusters of valves, which connect input pipes with output pipes. One input pipe can be connected to one output pipe and vice versa. Two different kinds of manifolds can be distinguished: *parallel* and *sequential* manifolds. Parallel manifolds can establish multiple connections at the same time. In the paint factory, there is only one parallel manifold (Mp). This manifold has 4 input pipes and 2 output pipes, and can therefore establish 2 connections at the same time. The other manifolds (Ms1-3) are sequential manifolds. They can establish one connection at a time.

In Figure 3, the hardware setup is depicted in more detail. In the upper lefthand corner, we see the three primary color vessels and the vessel with cleaning liquid (C). The primary color vessels each contain one of the primary colors red, yellow and blue. The color of each vessel is fixed and cannot be changed. Underneath each vessel is a pump. The pump will pump the paint (or cleaning liquid in case of vessel C) through the pipes. The actual route of the paint is established by the valves. When using the pumps, the pump is turned on before valves are switched to pressurize the system; this gives better control of the flow than first switching valves before turning on the pump would [2].

There are two types of valves, both depicted in the lower left corner of Figure 3. An open/close valve has 1 input and 1 output pipe and can establish a connection between these pipes. A right/left valve has one input and 2 output pipes or vice versa, and can establish a connection between any one of the input and one of the output pipes.

The mixing vessel (M) can mix primary colors in any desired color. There is no stirrer present. The paint mixes as soon as it is put into the mixing vessel by turbulence. After mixing primary colors, the paint is output into one of the buffer vessels. Then the mixing vessel is cleaned and can be used to mix other colors.

In the upper righthand corner of Figure 3, the buffer motor is shown. The buffer motor can move between 4 positions; above each of the buffer vessels (shown as B1-3), and above the waste vessel (W). The 4 black triangles depict the 4 sensors at each of these positions. The buffer motor is controlled by the buffer motor controller. This controller can turn the motor on and off and can make it move either to the left or to the right. If for instance, one wants to put paint in buffer vessel 3, the buffer motor controller turns the motor on and makes it move to the right until sensor SB3 reports that it sees the motor. When we open valves 10 and 12 and turn on pump 5, the paint will flow from the mixer through valves number 10, 11 and 12, into buffer vessel 3.

Underneath each buffer vessel is a pump to pump paint from each of these vessels to either the waste vessel or the turntable.

In the lower right corner of Figure 3, the turntable is shown with the filler motor above. The filler motor can move between 2 positions (above the waste vessel and above the turntable) and is controlled by the filler motor controller. The black triangles SF1 and SF2 depict the sensors that can report a motor if it is at that position. When the valve of the filling motor is open (valve number 18),

the paint flows through the filler into either the waste vessel or the cup on the turntable that is in filling position.

The turntable motor, underneath the turntable T, can rotate the turntable in either direction, thus putting cups at the filling position. This motor is controlled by the turntable controller. There are 12 cups positioned at the turntable. The sensors ST1 and ST2 can tell if there is a cup at the filling position.

To get paint from the vessels to the output cups, the paint flow can follow several routes. If the color to be outputted is equal to one of the colors in the primary vessels, the paint will flow directly from the primary vessel to the turntable. If the desired color is not one of the primary colors, the right amounts of primary colors needed to mix the desired color will be put into the mixing vessel. After mixing the mixed paint will be put into one of the buffer vessels where it will be stored. Amounts of paint are measured by determining the time needed for the paint to flow from the primary vessel into the buffer vessel. The right amount of paint will be outputted to the turntable and the rest of the mixed paint will be saved for further use. If all the buffer vessels are in use and a new mixed color needs to be stored, one of the buffer vessels will be emptied and cleaned so the new color can be stored.

To let the paint flow, pumps are placed to pump the paint through the pipes. The pumps are shown in Figure 3 and are numbered 1 to 8. There are three pumps next to the primary color vessels to pump paint from these vessels through the pipes. Furthermore, there is a pump at the cleaning liquid vessel, one at the mixing vessel, and three at the buffer vessels.

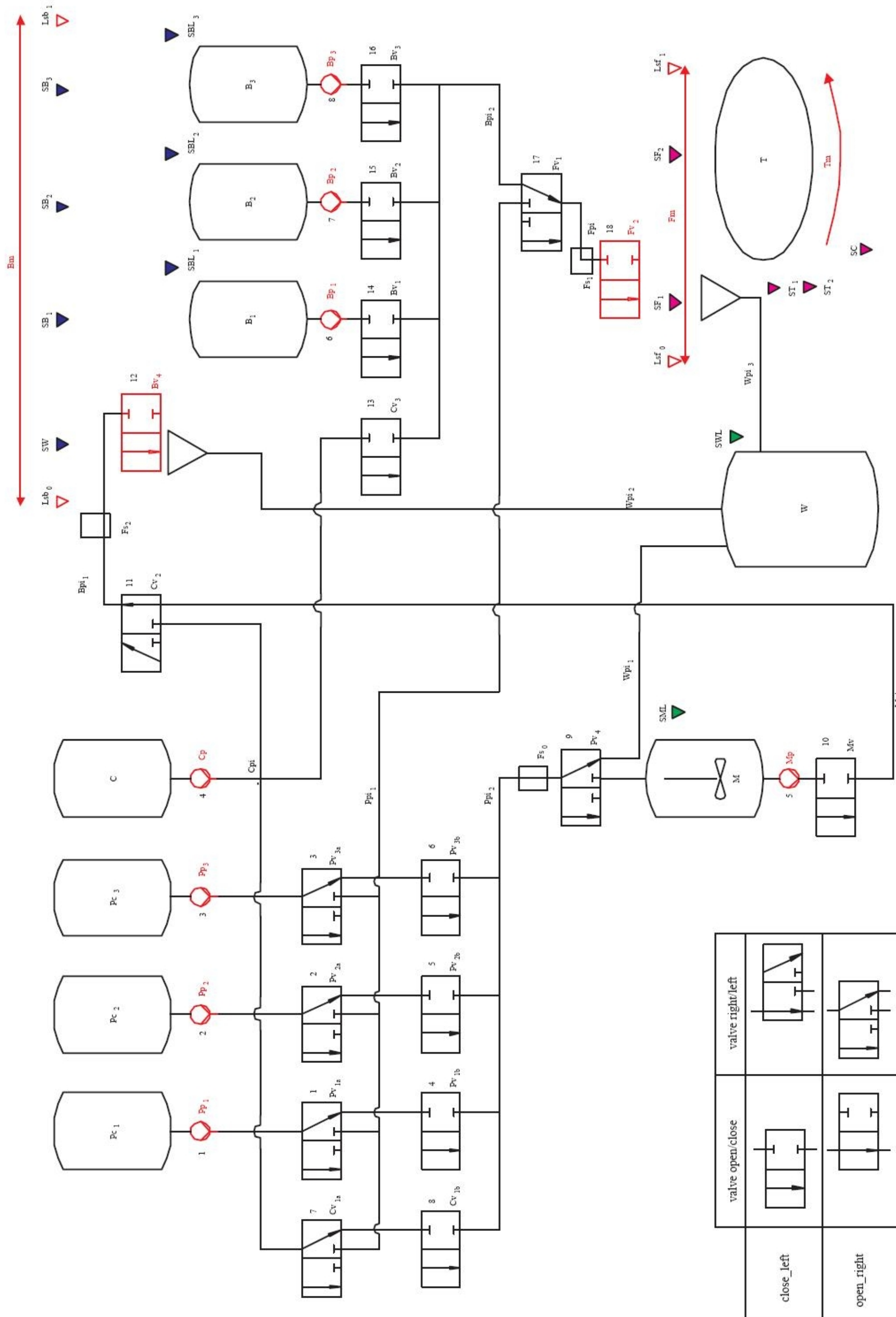


Figure 3: Hardware setup

2.2 Control system

The control system controls the hardware. The control system consists of a dispatcher, a resource controller and hardware controllers (see Figure 1)

The first controller model of the paint factory was made by M.H.M. van Duin (see [2]). First, a simulation model of the hardware was created, which then was controlled by this controller model. When this was working, the simulated hardware was replaced by real hardware.

In Figure 4, the logical model of the paint factory is shown. The logical model shows the behaviour of the hardware and the hardware controllers, as described in Section 2.1. The bottom layer of the model shows the simulated hardware. In the second layer, the hardware controllers are shown. The third layer consists of the resource controllers, and above that the dispatcher is shown. The uppermost layer shows the generator. Below, we will describe each entity in this model.

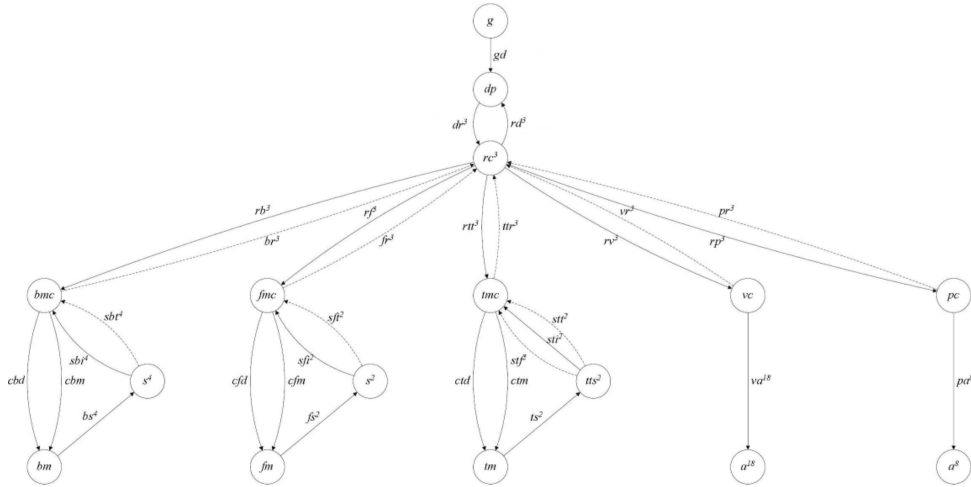


Figure 4: Logical model

g (generator): imports a list of orders from an input file. The generator sends all orders one by one to the dispatcher.

dp (dispatcher): keeps a list of available resources. The dispatcher receives the orders from the generator. Upon receiving an order, it checks whether the resources needed for the task are available. If this is the case, the task is sent to one of the three resource controllers. When for a task not all resources are available, the dispatcher waits until this is the case, and then sends the task to one of the resource controllers. Because there are three resource controllers, the dispatcher doesn't need to wait until the task is finished to send the next task. When an order is finished, the resource controller sends a message to the dispatcher and the used resources become available for the next order.

rc (resource controller): the resource controller communicates with all hardware controllers. Upon receiving the order from the dispatcher, it sends

the actions needed to be carried out to the corresponding hardware controllers, and then waits until it receives a message from the controllers, meaning that the action has been executed successfully. After all hardware commands are finished, it sends a message back to the dispatcher, saying that the order is finished.

In the paint factory, some tasks can be carried out in parallel with other orders. Because the resource controller carries out all actions one by one, more than one resource controller should be instantiated in order to achieve parallelism. M.H.M. van Duin researched in [2] how many orders can be processed in parallel. This turned out to be at most 3 orders. For this reason 3 resource controllers are instantiated.

bmc (buffer motor controller): controls the buffer motor. First, the bmc initializes the buffer motor. Because the initial position is unknown, the bmc first reads all sensors to see if the motor is in any of those positions. If this is not the case, the motor is turned on and sent to the rightmost position. When the sensor at this position reports the motor, it is turned off. After this initialization part, the bmc is ready to receive actions from the resource controller. After receiving a desired position, the controller sends the motor to this position. After receiving the signal from the corresponding sensor that the motor is in the desired position, the motor is turned off.

fmc (filling motor controller): controls the filling motor. The filling motor controller has the same functionality as the buffer motor controller.

tmc (turntable motor controller): controls the turntable motor. First, the tmc initializes the turntable. It turns on the motor in a clockwise direction, and waits until it receives a signal from sensor 0, the initialization sensor. After this, the turntable keeps going until sensor 1 is actuated, meaning that a cup is in fill position. After initialization, the resource controller is notified and the tmc can receive hardware commands. After receiving a command, the controller calculates which is the shortest path to that position, sets the direction and turns on the turntable motor. The controller keeps a counter denoting the number of cups that should pass before the right cup is at the filling position. When the counter is equal to 0, the motor is turned off. After this, the resource controller is notified of the succesfully executed command.

vc (valve controller): controls the valves. There are 18 valves. Depending on the valve, it can be either closed/right (the value is false) or open/left (the value is true). First, the valve controller initializes all valves to false. After initialization the vc sends a message to the resource controller saying it is ready to receive commands. When receiving a command, the corresponding valve is opened or closed.

pc (pump controller): controls the pumps. There are 8 pumps. The pump controller works in the same way as the valve controller, and is therefore not discussed here.

When a customer orders a number of colors to be outputted, this order is split in a number of tasks by the resource controller. We illustrate this with an example of a customer who orders a cup of orange paint (a mix of red and yellow). When this is the case, the following tasks have to be performed:

1. send paint from primary color 1 (red) to the mixer;
2. empty the track from primary color 1 and send the waste to the waste vessel;
3. send paint from primary color 2 (yellow) to the mixer;
4. empty the track from primary color 2 and send the waste to the waste vessel;
5. send paint from the mixer to buffer 1;
6. empty the mixer;
7. clean the mixer;
8. empty the track to buffer 1;
9. send paint from buffer 1 to the filler;
10. clean the track from the buffers to the filler.

In total, 30 possible tasks exist, that can be combined to constitute an order. All tasks are shown in Table 1.

Every task consists of a number of actions that are carried out by the motor controllers. For instance, the actions the paint factory performs when sending paint from primary color 1 to the mixer are the following:

- Turn on pump number 1.
- Switch valve number 9 to true
- Switch valve number 4 to true
- wait for 45 seconds for the paint to flow from the primary vessel to the mixer
- Switch valve number 4 back to false
- Switch valve number 9 back to false
- turn off pump number 1.

1.	send paint from the primary color 1 to the mixer
2.	send paint from the primary color 2 to the mixer
3.	send paint from the primary color 3 to the mixer
4.	send paint from the primary color 1 directly to the filler
5.	send paint from the primary color 2 directly to the filler
6.	send paint from the primary color 3 directly to the filler
7.	send paint from the mixer to buffer 1
8.	send paint from the mixer to buffer 2
9.	send paint from the mixer to buffer 3
10.	send paint from buffer 1 to the filler
11.	send paint from buffer 2 to the filler
12.	send paint from buffer 3 to the filler
13.	empty the track from primary color 1 and send the waste to the waste vessel next to the mixer
14.	empty the track from primary color 2 and send the waste to the waste vessel next to the mixer
15.	empty the track from primary color 3 and send the waste to the waste vessel next to the mixer
16.	empty the track from primary color 1 and send the waste to the waste vessel next to the filler
17.	empty the track from primary color 2 and send the waste to the waste vessel next to the filler
18.	empty the track from primary color 3 and send the waste to the waste vessel next to the filler
19.	empty the track to buffer 1
20.	empty the track to buffer 2
21.	empty the track to buffer 3
22.	empty the mixer
23.	clean the mixer
24.	clean buffer 1
25.	clean buffer 2
26.	clean buffer 3
27.	clean the track to the mixer
28.	clean the track to the buffers
29.	clean the track from the primary colors to the filler
30.	clean the track from the buffers to the filler

Table 1: Tasks

3 The χ specification

To model the control system of the paint factory, a specification needed to be created. As a specification language, χ is chosen. χ is a specification language especially designed to model concurrent manufacturing systems. It is thus designed to easily describe parallelism. In this chapter, we will first give an introduction to the language χ, and then we will describe the different parts of the paint factory specification in χ.

Research on the language χ for modelling, simulation and control of manufacturing systems was started around 20 years ago by Prof. J.E. Rooda. χ incorporates elements from process algebra, communicating sequential processes and functional programming.

The language χ is still developing. The first compiler was made for version χ 0.3. The version that is currently used is χ 1.0

3.1 The language χ

A χ specification is built around processes. A process has the following general form:

```
proc P() = |[ variable declarations :: statements ]|
```

The statements can consist of various constructions:

χ statement: S	
`skip`	skip statement
`x:= y`	assign statement
`x!y`	send the value of variable y over channel x
`x?y`	receive the value sent over channel x and store it in variable y
`S ; S`	sequential composition
`b -> S`	guarded command; S is executed if b is true
`S1	S2`	alternative composition. Which alternative is executed, is chosen non-deterministically
`*S`	loop statement
`delay x`	delay for x time units

Table 2: Actions

When using alternative composition, we can group alternatives together, thus making large alternative compositions shorter and easier to read. For example, the following statement:

```
 ( | , j <- 0..4 , (rd.j?yr; xr:= xr + yr))
```

can be translated to:

```
( rd.0?yr;  xr:= xr + yr
| rd.1?yr;  xr:= xr + yr
| rd.2?yr;  xr:= xr + yr
| rd.3?yr;  xr:= xr + yr)
```

3.2 Specification of the paint factory

The first controller model of the paint factory, made by M.H.M. van Duin, was made in an early version of χ, χ 0.6. The structure of this specification follows the logical model presented in the last Chapter. To do verification, besides the control system, the hardware also needed to be modeled in the χ specification. In 2005, W. Bremer converted this model to the newer version χ 0.8 and added some slight changes. This model can be found in Appendix A. In this project, we want to convert the model to the current version, χ 1.0. To convert this model, we need to know the differences between versions 0.8 and 1.0. We compared the documents [6] and [7] to find out what changed in the newer version. We found out that the translation is pretty straightforward. The changes we made were all syntax changes, and we could keep the structure of the original specification. Unfortunately, not all χ 1.0 language constructs are mentioned in [7]. We had to contact A.F. Hofkamp, the creator of the χ 1.0 C compiler, to find out the correct syntax of some data structures like channel bundling. At the end of this project, a new χ 1.0 reference manual came out which does contain all language constructs.

The χ 1.0 model can be found in Appendix C . Parts of this model are explained in this section.

3.2.1 The generator

The first action the paint factory performs is reading an order from a file and store this in a variable. Process G handles this part.

```
 proc G(chan gd!: (nat,{string},[(string,nat,nat)])) =
|[ var xs: [(nat,{string},[(string,nat,nat)])],
       x:(nat,{string},[(string,nat,nat)]) ::
   ??xs; (len(xs) > 0) *> (gd!hd(xs); xs:= tl(xs))
]|
```

The statement `??xs` denotes the reading of the order file and storing it in variable `xs`. While the length of the orderlist is bigger than zero, the first task (the head of the list, denoted by `hd(xs)`) is sent over channel `gd`. This task is then removed from the list by assigning the tail of the list (`tl(xs)` to variable xs.

pX	vessel with primary color X
c	vessel with cleaning liquid
m	mixing vessel
bY	buffer vessel Y
vm2m	pipes from valve matrix 2 to the mixing vessel
vm1vm3	pipes from valve matrix 1 to valve matrix 3
vm3	pipe within valve matrix 3
vm1b	pipes from valve matrix 1 to the buffer
vm3f	pipes from valve matrix 3 to the filler

Table 3: Resources

An order is of the following format:
`(order_id,{resources needed},[hardware commands])`. There are 30 possible orders, depicted in Table 1. All resources are shown in Table 3.

All possible hardware commands are shown in Table 4. An example of an order could be:

```
("p1wm", {"p1","vm2m"}, [("p",0,1), ("v",3,1), ("d",0,45),
("v",3,0), ("p",0,0)])
```

. The meaning of this order is: the order named `"p1wm"` (empty the track from primary color 1 and send the waste to the waste vessel next to the mixer) needs resources `"p1"` (primary vessel 1) and `"vm2m"` (pipes from valve matrix 2 to the mixing vessel) and performs the following actions:
- `("p",0,1)` ; turn on pump number 0;
- `("v",3,1)` ; switch valve number 3 to true;
- `("d",0,45)` ; wait for 45 seconds;
- `("v",3,0)` ; switch valve number 3 to false;
- `("p",0,0)` ; turn off pump number 0.

3.2.2 The dispatcher

When process G sends a task, process DP (the dispatcher) receives this task. The χ specification of process DP is:

```
proc DP(chan gd?: (nat,{string},[(string,nat,nat)]),
             dr!: 3 # (nat,{string},[(string,nat,nat)]),
             rd?: 3 # {nat}) =
|[ var x: (nat,{string},[(string,nat,nat)]), b: bool = true,
       xr: {string}, yr: {nat} ::
xr:= {"p1", "p2", "p3", "c", "m", "b1", "b2", "b3", "vm2m",
      "vm1vm3", "vm3", "vm1b", "vm3f"};
gd?x ;
  *(b -> gd?x; b:= false
```

```
    | ( | , j <- 0..2 , (not b and (x.1 sub xr)) -> dr.j!x ;
                          xr:= xr - x.1; b:= true)
    | ( | , j <- 0..2 , (rd.j?yr; xr:= xr + yr))
    )
]|
```

The set xr denotes the set of available resources. Upon receiving a task over channel gd and storing this in variable x, process DP checks whether the resources needed for the task are in the set of available resources xr. These resources are taken from the set and the task is sent to one of the three resource controllers (over channel dr). When for a task not all resources are available, the dispatcher waits until this is the case, and then sends the task to one of the resource controllers. Because there are three RC processes instantiated, process DP does not need to wait until the task is finished to send the next task. When an order is finished, RC sends a message to DP and process DP adds the resources to the set of available resources.

3.2.3 The resource controller

Process RC communicates with all hardware controllers. The χ specification of RC is:

```
proc RC(chan dr?: (nat,{string},[(string,nat,nat)]),
             rd!: {string}, rb!: nat, rf!: nat, rtt!: nat,
             rv!: (nat,nat), rp!: (nat,nat) , br?, fr?,
             ttr?, vr?, pr?: void ) =
|[var xt: (nat,{string},[(string,nat,nat)]),
      xs: [(string,nat,nat)], x: (string,nat,nat) ::
*(dr?xt; xs:= xt.2
 ;(len(xs) > 0) *> (x:=hd(xs); xs:= tl(xs)
                 ; (x.0 = "m"  and x.1 = 0 -> rb!x.2; br?
                   |x.0 = "m"  and x.1 = 1 -> rf!x.2; fr?
                   |x.0 = "mp" and x.1 = 1 -> rf!x.2
                   |x.0 = "mp" and x.1 = 2 -> rtt!x.2
                   |x.0 = "sm" and x.1 = 1 -> fr?
                   |x.0 = "sm" and x.1 = 2 -> ttr?
                   |x.0 = "v"              -> rv!(x.1,x.2); vr?
                   |x.0 = "p"              -> rp!(x.1,x.2); pr?
                   |x.0 = "d"              -> delay x.2
                   ))
        ; rd!xt.1
     )
 ]|
```

Upon receiving the order from process DP, it takes out the list of hardware commands (stored in `xt.2`) and stores this in variable `xs`. This list consists of tuples consisting of 3 values; the first one (`x.0`) denotes the hardware needed for the action, and tells the controller whether the action can take place in parallel with other actions or not. An `"m"` denotes one of the motors that needs to be moved, this action can however not take place in parallel with another action. An `"mp"` denotes a motor that can move in parallel with another action. When this parallel action is finished, an `"sm"` action is needed to let the resource controller know. Furthermore, a `"v"` denotes that one of the valves need to be switched, a `"p"` for a pump, and a `"d"` denotes a delay. The second value of the triple `x.1` denotes the exact resource (`0` for the buffer motor, `1` for the filler motor, and `2` for the turntable motor) and the third value defines the exact action. These values are shown in table 4. RC sends the actions one by one to the corresponding hardware controllers, and then waits until it receives a message from the controllers, meaning that the action has been executed successfully. After all hardware commands are finished, it sends a message back to DP, saying that the order is finished.

x.0	x.1	description
m	0	buffer motor (position 0: waste, 1: b1, 2: b2 and 3: b3)
m	1	filler motor (position 0: waste and 1: turntable)
mp	1	filler motor (in parallel with the turntable motor)
mp	2	turntable motor (in parallel with the filler motor) position n: cup n)
sm	1	synchronize filler motor
sm	2	synchronize turntable motor
v	X	valve X
p	Y	pump Y
d	Z	delay (no action is assigned, only the RC delays)

Table 4: Actions

3.2.4 The hardware controller groups

Three hardware controller groups exist: the buffer motor, filler motor and turntable motor controller group.

The buffer motor controller group

The buffer motor controller group consists of the buffer motor controller `BMC`, the buffer motor `BM` and two sensors `S` (see Figure 4).

The buffer motor can slide along a track between 4 possible positions (above each one of the three buffer vessels or above the waste vessel). At each position is a sensor. The χ specification of the buffer motor is:

```
 proc BM(chan cbd?: bool, cbm?: bool, bs!: 4 # bool, val rt: real)
= |[ var d: bool, m: bool, q,i: nat = (0, 0) :: bs.0!true ;
```

```
m:= false ;
 *(m ->          (delay rt;          (d       -> q:= q + 1
                                     |not d   -> q:= q - 1
                                     );
                   i:= 0;
                  (i < 4)*> (bs.i!(q = 20 * i); i:= i + 1 )
                 )
   |cbd?d
   |cbm?m
   )
 ]|
```

First, the buffer motor is initialized at position 0. In the original χ specification, there exists an initialization phase where every sensor is checked to see where exactly is the motor. If none of the sensors reports the motor this means the motor is somewhere in between the sensors. The motor is then moved until a sensor reports it. This initialization phase is added for real time behavior. In model checking and performance analisys, we are mainly interested in the long term behavior of the paint factory. If we could manually put all motors at a fixed position before we start using the paint factory, we could remove the initialization part of the specification. This makes the specification a lot shorter and easier to read, and does not change the long term behavior, because it is only an initial action.

The direction is received over channel `cbd` and stored in variable `d`. The on/off signal is received over channel `cbm` and stored in variable `m`. The position of the motor on the track is modelled by variable `q`. Between every 2 positions there are 20 steps to simulate real time behavior. The delay of `rt` is the time used to take one step. The main loop of BM is a repetitive selective waiting statement. The process tries to receive a direction `cbd?d` or an on/off signal `cbm?m`. When `m` = true, the motor will move, depending on the direction `d`. After adjusting the position `q`, the sensors are notified of this new position by sending a `true` value if the motor is at the position of one of the sensors.

The χ specification of the buffer motor controller is presented underneath.

```
 proc BMC(chan rb?: 3 # nat, br!: 3 # void, cbd!: bool, cbm!: bool
          , st?: 4 # void) =
 |[ var b: bool = false, k: nat, m: nat = 0 ::
   *( | , j <- 0..2 , rb.j?k
             ;(k = m    -> skip
              |k/= m    -> cbd!(k > m); cbm!true; st.k?
                         ; cbm!false; m:= k
              )
             ; br.j!
    )
 ]|
```

First process BMC receives a required position of the motor of one of the resource controllers and this is stored in variable `k`. When the required position is equal to the current position, the motor is already in place, and does not need to be moved. If this is not the case, the direction becomes true if `k > m` and false if `k < m` . The motor is turned on and the motor starts moving. The motor updates the sensors, and the target sensor `st.k` notifies the buffer motor controller when the motor is at the correct position, after which the motor is turned off, and the current position becomes the target position `m := k`. Finally, the resource controller is notified that the action is executed.

The filler motor controller group

The filler motor `fm` is similar to the buffer motor. The main difference is that `fm` slides along a track with two possible positions. Because of the similarity, the filler motor is not explained in detail.

The turntable motor controller group

The turntable is a rotating platform that can turn in both directions. It has 12 cups, and 10 steps between every cup to model realtime behaviour. The χ specification of the turntable motor is depicted below.

```
proc TM(chan ctd?: bool, ctm?: bool, ts!: bool, val rt: real) =
|[ var d: bool, m: bool, q: nat ::
 q:= 0;
 ts!true;
 m:= false;
 *(m -> delay rt;  (d      -> q:= (q + 1) mod 120
                   |not d  -> q:= (q + 119) mod 120
                   )
                   ; ts!q mod 10 = 0
  |ctd?d
  |ctm?m
  )
]|
```

Like for the other hardware controller groups, we removed the initialization phase and put the turntable at a fixed position, and notify the sensor that there is a cup at filling position (over channel `ts`)

The direction is received over channel `ctd` and stored in variable `d`. The on/off signal is received over channel `ctm` and stored in variable `m`. The position of the motor on the track is modelled by variable `q`. Between every 2 cups there are 10 steps to simulate real time behavior. There are thus 120 steps on the whole turntable. The delay of `rt` is the time used to take one step. The main loop of TM is a repetitive selective waiting statement. The process tries to receive a direction `ctd?d` or an on/off signal `ctm?m`. When `m` = true, the motor will move, depending on the direction `d`. After adjusting the position `q`, the sensors are notified of this new position by sending a `true` value if there is a cup present at the position of the sensor.

The χ specification of the turntable motor controller is presented next.

```
proc TMC(chan rtt?: 3 # nat, ttr!: 3 # void, ctd!: bool,
              ctm!: bool, st?: void) =
|[ var b: bool, k: nat, p,q: nat, m: nat = 0 ::
 *( | , j <- 0..2 , rtt.j?k;
             p:= (12 + k - m) mod 12;
               ( p = 0 -> skip
               | p /= 0 -> ctd!(p <= 6);
                 q:= p min 12 - p;
                 ctm!true;
                 (q>0) *> (sf?; st?; q:= q - 1);
                 ctm!false; m:= k
               )
             ; ttr.j!
  )
]|
```

First process TMC receives a required cup number of one of the resource controllers and this is stored in variable `k`. It then calculates the number of steps that need to be taken from the current position `m`. When the required cup number is equal to the current one, the turntable is already in place, and does not need to be moved. If this is not the case, the direction becomes true if `(p <= 6)` , thus if this is the fastest way to get to the desired cup, and false if `(p > 6)`. Next, the motor is switched on. Every time the sensor becomes true, variable `q` is diminished by one until `q = 0`. One would expect the statement `(q>0) *> (st?; q:= q - 1)` to control the stepping of the motor. The problem with this statement is that it can loop a few times during the time interval that the sensor is true. This can be solved by waiting until the motor has passed the sensor, before synchronizing again. This is done via channel `sf`, which sends a message when the sensor is false. When `q` $= 0$ the motor is turned off, and the current position becomes the target position `m := k`. Finally, the resource controller is notified that the action is executed.

3.3 The Sensors

After we created the χ 1.0 model, we had a closer look at the way the sensors are modeled in the paint factory specification. In the original χ specification, after the motor takes a step, it sends a message to all sensors to tell them whether it is located in front of this sensor or not. If the motor is in front of a sensor, after receiving this message from the motor, the sensor can send a message to the motor controller to notify the controller about the position of the motor. The way this is specified is however not really representative for the real paint factory. In the real paint factory, the sensor checks to see if it sees the motor, instead of the motor telling the sensor where it is located. We thus want to model the communication between the sensors, the motor and the

controller in a more realistic way. The most realistic would be to keep track of a global variable for each motor, denoting the current position of this motor. The processes simulating the motor can update this variable, and the sensor can read it to find out if the motor is at the position of one of the sensors. Unfortunately, in χ 1.0 it is not possible to keep global variables. This is why, instead of a global variable, we introduce an extra process that keeps track of the position of all motors. When the motor takes a step, it gives this process an update of the new position. At the same time, the sensors can check this process to find out the current position of the motor. This process looks like this:

```
proc POS(chan buf1?: nat,fil1?: nat ,tt1?: bool,tt!: bool,
buf!: 4 # bool, fil!: 2 # bool)= |[ var x,y: nat,z:bool ::
   *(buf1?x
    |fil1?y
    |tt1?z
    |(x = 0)  -> buf.0!true
    |(x = 20) -> buf.1!true
    |(x = 40) -> buf.2!true
    |(x = 60) -> buf.3!true
    |(y = 0)  -> fil.0!true
    |(y = 20) -> fil.1!true
    |(z = true) -> tt!z ; z := false
    )
]|
```

This process can either receive a new position from the buffer motor, filler motor or turntable motor, or send a message to one of the sensors if the motor is at the position of a sensor.

4 Verifying Correctness

Once we have a χ model, we want to use this to find out whether the paint factory behaves exactly the way we want, in any situation. To be able to say something about the behavior, we have first used a simulator to see whether it behaves as expected. However, by just simulating the model, we can never prove that it is a correct specification. We can do a lot of tests to see if the program does what it should do, but we can never be 100 % sure. To be able to say something about the correctness of a program, we will use *model checking*. Model checking is a method to algorithmically verify formal systems [9].

In this chapter, we will first describe the simulation we did, using a χ simulator. Secondly, we give an introduction to SPIN. After this, we will describe simulation with SPIN. Finally, we will describe the requirements we checked using the SPIN model checker and the results of this.

4.1 Simulation

The first step in verifying correctness, is simulation. With simulation, we can test to see if the χ model behaves as expected under normal circumstances. There are currently two tools available for the simulation of χ 1.0 specifications: the timed χ 1.0 simulator, a simulator that compiles the specification to an executable via C, and the timed χ Python simulator, a simulator that first compiles the specification to a Python file. For more information about χ simulators, see [5].

There are some crucial differences between the C and the Python simulator. Although they both accept the same input language, there are some data structures, like channel bundles and sets, that are implemented in the C simulator, but not in the Python simulator. This is why we did simulation of the paint factory model with the C simulator.

We did simulation with both a finite order list and an order generator, that generates random orders. We let the specification output the start and finish time of the several processes and values of the used variables to make sure all orders are carried out correctly.

During simulation with the χ simulator, we discovered no errors.

4.2 Model checking

To do model checking, we define properties that should hold for the system: the requirements. The next step is to check if these properties hold for every state the program can be in. To do this, we need to search all possible states of the program. Therefore, we can choose to generate a state space. A state space is the set of all possible states together with all possible transitions from one state to another. Now once we have all possible states, we can check whether the requirements hold for all these states. The tool we use for model

checking is called SPIN (see [4]). However, SPIN does not generate a whole state space prior to checking requirements. Instead, it runs through all the states and checks at the same time if the requirements hold. We use this tool because earlier research has been done about using SPIN for the verification of χ specifications (see [14]), and a translater tool has been developed to translate χ specifications to Promela, the input language of SPIN.

4.2.1 SPIN

SPIN can be used for the formal verification of distributed software systems. It can either simulate a specification, or verify the correctness of Linear Temporal Logic (LTL) formulae, and some standard properties like deadlock can be verified. SPIN is an acronym for Simple Promela Interpreter. The input language of SPIN is Promela. We also used an extension of SPIN, called DTSPIN. This extension makes it possible to use discrete timing with SPIN. This extension is created by D. Bosnacki (see [13]).

To be able to use this tool, we want to translate the χ specification to Promela with a translator tool designed by N. Trcka (see [14]). This tool was implemented by R. Meijer and can be found at [12]. SPIN can check properties of a Promela specification with a *never* claim. A *never* claim is used to specify behavior that should never occur. After each step SPIN executes, it checks if all *never* claims hold. SPIN gives an error if there is an execution possible for which one of the never claim holds. To create never claims, we use LTL formulae. SPIN translates LTL formulae into never claims.

The translation of the χ specification to Promela was unfortunately not very straightforward. The translator tool proved to be very limited, being only able to deal with simple χ specifications with basic datatypes like integers and booleans. Therefore, most of the translation has to be done by hand. To be able to make use of the translator tool, we took the following steps:

The first step was to write a χ specification similar to the paint factory model, but with the use of only boolean and integer variables, and without any complicated data structures like channel bundling. We thus removed all the data sent through the specification, and just kept the structure of processes and channels. This adapted specification is therefore not representative for the paint factory, but it could give us an idea about the structure of the Promela specification when we used the translator tool. Unfortunately, what we got after using the tool was a highly unreadable file with no breaks and a lot of errors. Errors were given for various structures the translator couldn't translate, like delays or numerical comparisons (i.e. less or equal).

After taking out the errors and adding structure to the text, we had the basics of a Promela specification. Now we had to translate the several data structures we use in the χ specification, like channel bundling, sets of data and guarded commands. With the help of a book on SPIN ([8]), we could translate the χ specification into a Promela specification with the same functionality. In Table

5, we will give a comparison of the χ to the SPIN syntax. The main difficulties we encountered in this process were:

1. In the dispatcher process, we use a set of resources. This set contains all the free resources. In Promela it is not possible to have a set datatype. Instead, we keep an array with as many elements as there are resources, and a *true* value for an available resource and a *false* value for an occupied one.

2. It is not possible to use channel bundling in Promela. Therefore, we had to separate the bundles and create separate channels.

3. In Promela, it is not possible to send sets or lists through a channel to other processes. Therefore, we cannot send the order in the format it is sent in the χ specification:
`(order_id,{resources needed},[hardware commands])`

   This order would be sent from process to process where every process would take out the data it needs.

   There are 30 possible order ID's, each consisting of a predefined set of resources and hardware commands. Because of this limited amount of possible orders, and the fixed resources and hardware commands, it is unnecessary to send the set of resources and hardware commands throughout the whole program. The only thing that is needed, in fact, is the order ID. Along the process, the program can calculate the needed resources and hardware commands.

   The change we made was to just send the order ID instead of the whole order. Thus, a number between 0 and 31 is sent from process to process. With this, we made the change from sending data through the system, to behaviour, executing a sequential number of actions.

   Because we only send the order ID, in the resource controller, all the corresponding hardware commands are assigned locally, and performed sequentially. In the original χ model, the hardware commands look like this: `[(string,nat,nat)]`, where the string and the first natural number would be an identifier for the hardware part, and the second number the desired action (see Table 4). The hardware commands would be taken from the list one by one, and the corresponding action would be taken. Because the resource controller now calculates all the necessary actions, it is unnnecessary to first generate the list of actions and then take all the commands one by one to carry them out. Instead, the resource controller carries out all the actions one by one, depending on the order ID.

The final Promela specification can be found in Appendix B.

During simulation with SPIN, we discovered an error in the specification. Because the behaviour is nondeterministic, in the current specification, the signal

χ statement	Promela statement
`skip`	`break`
`x:= y`	`x = y`
`x!y`	`x!y`
`x?y`	`x?y`
`S ; S`	`S ; S`

`b -> S`	`if`
	`:: b -> S`
	`fi`

`S1 | S2`	`if`
	`:: S1`
	`:: S2`
	`if`

`*S`	`do`
	`:: S`
	`od`

`b *> S`	`do`
	`:: b ; S`
	`:: not b -> break`
	`od`

`delay x`	`set(t,x); expire(t)`

Table 5: χ to SPIN translation

of the motor controller telling the motor to switch off could be missed. This would cause the motor to miss the position it should go to and just move on until it reached the end. The original specification of the filler motor in the χ model looks like this:

```
proc FM(chan cfd?: bool, cfm?: bool, fs!: 2 # bool, val rt: real)
= |[ var d: bool, m: bool, q,i: nat = (0,0)
:: fs.0!true
 ; m:= false
 ; *(m -> delay rt;  (d      -> q:= q + 1
                     |not d  -> q:= q - 1
                     )
                  ; i:= 0
                  ; (i < 2)*> (fs.i!q = 20 * i; i:= i + 1 )
   |cfd?d
   |cfm?m
   )
]|
```

We later found out that the reason we had this error with the Promela specification, was that the translation of the delay should have been a little different. The first translation of the FM process in SPIN looks like this:

```
proctype FM (  chan cfd, cfm, fs0, fs1)
    {  bool d
    ;  bool m
    ;  byte q
    ; timer t
    ; q = 0
    ; fs0 ! true , true
    ; m = false
    ; do
       :: (m == true) ; set(t,2);  expire(t)
        ; if
          :: d_step { d ; q = ( q + 1 ) }
          :: d_step { ( ! d ); q = ( q - 1 ) }
          fi
        ; fs0! (q == 0), true
        ; fs1! (q == 20), true
       :: cfd ? d , eval ( 2 - true )
       :: cfm ? m , eval ( 2 - true )
      od }
```

With this translation, in the `do` loop, the first alternative `m == true` is chosen. The SPIN simulator checks all alternatives, and can thus always choose the first one. However, if we model this differently, this changes. The second version of the filler motor process looks like this:

```
proctype FM (  chan cfd, cfm, fs0, fs1)
    {  bool d
    ;  bool m
    ;  byte q
    ; timer t
    ; q = 0
    ; fs0 ! true , true
    ; m = false
    ; do
      :: set(t,2)
         ; if
           :: expire(t) && (m == true)
              ; if
                :: d_step { d ; q = ( q + 1 ) }
                :: d_step { ( ! d ); q = ( q - 1 ) }
                fi
              ; fs0! (q == 0), true
              ; fs1! (q == 20), true
           :: cfd ? d , eval ( 2 - true )
           :: cfm ? m , eval ( 2 - true )
           fi
      od }
```

This way, we can make use of the way timing is implemented in DTSPIN. Because the delay is now part of the guard (together with `m == true`), SPIN will execute all possible other actions before executing the delay. Therefore, every time the loop is entered, the first alternative is only chosen if communication over channels `cfd` or `cfm` is not possible.

Another change we made in the Promela model, is to simplify the pump and valve processes. In the original specification, the actuator process A would simulate the behaviour of the pumps and the valves, controlled by either the pump or the valve controller. Instead of running 8 actuator processes to simulate the 8 pumps and 18 to simulate the 18 valves, we only run one process to simulate all the pumps, and one for all the valves. This does not change the specification behaviour, because the actuator process does not do anything else but receive signals from the hardware controller to turn on or off.

Again, we changed this in the Promela specification, and then adapted the χ specification accordingly. In χ, the actuator process looks like this:

```
proc A(chan ca?: bool) = |[ var k: bool :: *(ca?k) ]|
```

After these changes had been made, we could start model checking. To be able to do model checking, requirements had to be specified. We define these requirements in as LTL formulae.

Now to define requirements, we have to determine what is important in the model. The list of requirements is depicted in the following section.

4.3 Requirements

To define requirements, we use LTL formulae. In LTL, we can use the following operators (see [8]):

Unary Operators:

[] (the temporal operator *always*)
`<>` (the temporal operator *eventually*)

Binary Operators:

U (the temporal operator *strong until*)
`&&` (the boolean operator for *logical and*)
`->` (the boolean operator for *logical implication*)

4.3.1 Requirements on the controllers

First we defined requirements concerning the controller part of the paint factory. The requirements we checked are:

- The model does not contain a deadlock.
- When the buffermotor/fillingmotor/turntablemotor is in the right position, it will be turned off until the next desired position is received.
- If there is a finite order list, the end of this list will eventually be reached, thus all the orders will be carried out. Ofcourse we could not check this requirement for every possible finite order list, so we only checked this for a couple of order lists.
- A resource that is in use, will become available again within a finite amount of time
- When a resource is in use, it cannot be used by another order.

To define LTL formulae, global variables are used. Because in our specification only local variables exist, we have to introduce a set of global variables to use in the LTL formulae. We replace the local variables we want to use in the LTL formulae for global variables. These variables have the same functionality as the local variables, and are assigned locally by the processes they are used in. The global variables we introduced are the following:

- `bm`, `fm` and `tm` are boolean variables, denoting whether the buffer motor, filler motor and turntable motor are on or off. In the original χ specification, these variables are local variables named `m`.
- `posbk`, `posfk` and `postk` are integers. When the motor controllers receive a desired position, this value is stored in one of these variables. In the original χ specification, these variables are local variables named `k`.
- `posbq`, `posfq` and `postq` are integer values, denoting the current position of the buffer motor, filler motor and turntable motor. In the original χ specification, these variables are local variables named `q`.
- `lxs` is the order list. In the χ specification, the generator reads this list from a file, while in the Promela specification (because it is not possible to read from a file) it generates it itself. It stores the list in variable `lxs`.
- The list variable `lxr` keeps track of which resources are in use and which are not. When `lxr[x]` = false, this means that resource `x` is in use.

With these global variables, we constructed the following LTL formulae:

- When the buffermotor is in the right position, it will be turned off until the next desired position is received.

```
[] ((posbk == (posbq / 20)) ->
((posbk == (posbq / 20)) U (bm == false) )
&& (bm == false) ->
(bm == false U (posbk != (posbq / 20)))
)
```

- When the fillermotor is in the right position, it will be turned off until the next desired position is received.

```
[] ((posfk == (posfq / 20)) ->
((posfk == (posfq / 20)) U (fm == false))
 && (fm == false) ->
(fm == false U (posfk != (posfq / 20)))
```

- When the turntablemotor is in the right position, it will be turned off until the next desired position is received.

```
[] ((postk == (postq / 20)) ->
((postk == (postq / 20)) U (tm == false))
 && (tm == false) ->
(tm == false U (postk != (postq / 20)))
```

- The end of the order list will eventually be reached. (this requirement only applies when there is a finite order list.)

```
<> length(lxs) == 0
```

- A resource that is in use will become available again.

```
[]((lxr[x] == false)-> <> (lxr[x] == true ))
```

We checked these requirements with SPIN, and found out that they all hold.

4.3.2 Requirements on the hardware

Besides the proper functioning of the controllers, we want to know if the defined orders really accomplish their described tasks, thus we want to know if the hardware accomplishes all the described tasks. Therefore, we have to know more about the paint flow in the system. To be able to do this, we introduce a global variable in the Promela specification named `ptt[i]` that keeps track of the pipes, and knows what color is present at any location. We update this variable in the RC process. We define 5 clusters of pipes, which are shown in Table 6. In this table, we also show the actions that will have to have to take place for this variable to be updated. For instance, when pump number 1 is turned on (so the value is true) and valve number 1 is switched (so the value is

true), paint will flow into the pipes between the primary vessels and the filler. Therefore the value of `ptt[1]`, denoting the pipes between the primary vessels and the filler, will be set to 1, denoting the color in primary color vessel 1.

Variable	Pipes	Updated when
ptt[1]	pipes between the primary vessels and the filler	Pump 1 = true and valve 1 = true −> ptt[1] = 1 Pump 2 = true and valve 2 = true −> ptt[1] = 2 Pump 3 = true and valve 3 = true −> ptt[1] = 3 Pump 4 = true and valve 7 = true −> ptt[1] = 0
ptt[2]	Pipes between primary vessels and the mixer	Pump 1 = true and valve 4 = true −> ptt[2] = 1 Pump 2 = true and valve 5 = true −> ptt[2] = 2 Pump 3 = true and valve 6 = true −> ptt[2] = 3 Pump 4 = true and valve 8 = true −> ptt[2] = 0
ptt[3]	Pipes between the mixer and the buffer vessels	Pump 5 = true and valve 10 = true −> ptt[3] = mix
ptt[4]	Pipes between the buffer vessels and the filler	Pump 4 = true and valve 13 = true −> ptt[4] = 0 Pump 6 = true and valve 14 = true −> ptt[4] = buf1 Pump 7 = true and valve 15 = true −> ptt[4] = buf2 Pump 8 = true and valve 16 = true −> ptt[4] = buf3

Table 6: Clusters of pipes

We also keep track of the paint that is present in the buffer vessels and in the mixer, with variables `buf1, buf2, buf3` and `mix`. These variables are updated in process RC. When paint is put from one of the primary color vessels into the mixer, the new mixed color is calculated and assigned to variable `mix`. When paint is put from the mixer to one of the buffer vessels, the corresponding buffer variable is set to the value of `mix`. Furthermore, the variable `turnt[i]` is a list of boolean variables that keeps track of all the cups on the turntable and whether they are filled or not. When a cup gets filled, the boolean variable is switched. This way, we can find out whether a cup on the turntable can get filled twice. The colors that the buffers and the mixer can contain are depicted in Table 7.

Furthermore, we define a variable that keeps track of the start and end of orders, called `order[i]`. This variable is set to `true` at the start of an order in RC,

Color	Number
No color (Vessel is empty)	0
Primary color 1	1
Primary color 2	2
Primary color 3	3
A mix of primary color 1 and primary color 2	4
A mix of primary color 1 and primary color 3	5
A mix of primary color 2 and primary color 3	6
A mix of primary color 1, primary color 2 and primary color 3	7
Cleaning liquid	8

Table 7: Color numbering

and put back to `false` at the end of the order. This way, we can check the paint status at the beginning and the end of an order. Now we can describe requirements for each of the orders. The orders can be found in Table 1.

1. send paint from a primary color to the mixer (order numbers 1, 2 and 3). After finishing this order, the primary color should be present in the mixer. We will give the LTL formula for primary color 1, the formulae for the other primary colors are similar to this.

```
[](order[1] == true && (mix == 0 ||mix == 1) ->
<> (order[1] == false && mix == 1 ))
&&
[](order[1] == true && (mix == 2 ||mix == 4) ->
<> (order[1] == false && mix == 4 ))
&&
[](order[1] == true && (mix == 3 ||mix == 5) ->
<> (order[1] == false && mix == 5 ))
&&
[](order[1] == true && (mix == 6 ||mix == 7) ->
<> (order[1] == false && mix == 7 ))
```

2. send paint from a primary color vessel directly to the filler (order numbers 4, 5 and 6). After finishing this order, this primary color should be outputted into a cup on the turntable. We give the LTL formula for primary color 1 (order number 4), the formulae for the other primary colors are similar to this. In orders number 4, 5 and 6, a fixed cup is defined, cup number 1. We can see from these orders, that if they are carried out twice, that the paint will be outputted into the same cup twice.

```
[]((order[4] == true && turnt[1] == true ) ->
order[4] == true U turnt[1] == false ) &&
[]((order[4] == true && turnt[1] == false ) ->
order[4] == true U turnt[1] == true )
```

3. send paint from the mixer to one of the buffer vessels (order numbers 7, 8 and 9). After finishing this order, the paint that is in the mixer when the order starts, should be present in the buffer vessel. We give the LTL formula for buffer vessel 1, the formulae for the other buffer vessels are similar.

```
[](order[7] == true -> order[7] == true U buf1 == mix )
```

4. send paint from one of the buffer vessels to the filler (order numbers 10, 11, and 12). After finishing this order, the paint that is in the buffer vessel when the order starts, should be outputted to an empty cup on the turntable. Again, we give the LTL formula for buffer vessel 1.

```
[]((order[10] == true && turnt[2] == true ) ->
order[10] == true U (turnt[2] == false)) &&
[]((order[10] == true && turnt[2] == false ) ->
order[10] == true U (turnt[2] == true))
```

5. empty the track from one of the primary color vessels to the mixer and send the waste to the waste vessel (order numbers 13, 14 and 15). If we want to do this for primary color vessel 1, pump number 1 will be turned on, and paint will flow from the primary color vessel to the waste vessel, thus rinsing the pipes that might be contaminated with other colors of paint. After finishing this order, no pipes from primary color vessel 1 to the waste vessel are contaminated with other colors of paint. We give the LTL formula for primary vessel 1.

```
[]((order[13] == true) -> (order[13] == true U
ptt[2] == 1))
```

6. empty the track from the primary color vessels to the filler and send the waste to the waste vessel (order numbers 16, 17 and 18). If we want to do this for primary color vessel 1, pump number 1 will be turned on, and paint will flow from the primary color vessel to the waste vessel, thus emptying the pipes that might be contaminated with other colors of paint. After finishing this order, no pipes from primary color vessel 1 to the waste vessel are contaminated with other colors of paint. We give the LTL formula for primary vessel 1.

```
[]((order[16] == true) ->  (order[16] == true U
ptt[1] == 1))
```

7. empty the track to one of the buffer vessels (order numbers 19, 20 and 21). This order pumps paint from the particular buffer vessel to the waste vessel. If this order is finished, no pipes between the buffer vessel and the waste vessel are contaminated with other colors of paint. We give the LTL formula for buffer vessel 1.

```
[]((order[19] == true) -> (order[19] == true U
ptt[4] == buf1))
```

8. empty the mixer (order number 22). After finishing this order, the mixer is empty, and the waste is sent to the waste vessel.

```
[]((order[22] == true) ->  (order[22] == true U
mix == 0))
```

9. clean the mixer (clean the number 23). After finishing this order, cleaning liquid is in the mixer.

```
[]((order[23] == true) -> (order[23] == true U
mix == 8))
```

10. clean one of the buffer vessels (order numbers 24, 25 and 26). After finishing this order, cleaning liquid is put into a buffer vessel. We will give the LTL formula for buffer vessel 1.

```
[]((order[24] == true) ->  (order[24] == true U
buf1 == 8))
```

11. clean the track to the mixer (order number 27). After finishing this order, cleaning liquid is put through the pipes leading to the mixer, and put into the waste vessel. The track from the primary vessels to the mixer will be clean, and not contaminated with other colors of paint.

```
[]((order[27] == true) ->  (order[27] == false U
ptt[2] == 0))
```

12. clean the track to the buffers (order number 28). After finishing this order, cleaning liquid is put through the pipes leading to the buffer vessels, and put into the waste vessel. After finishing this order, no pipes leading from the cleaning liquid vessel to the buffers is contaminated with paint.

```
[]((order[28] == true) -> (order[28] == true U
ptt[5] == 0))
```

13. clean the track from the primary colors to the filler (order number 29). After finishing this order, cleaning liquid is put through the pipes leading from the primary color vessels to the filler, and put into the waste vessel. No pipes leading from the cleaning liquid vessel to the waste vessel is contaminated with paint.

```
[]((order[29] == true) -> (order[29] == true U
ptt[1] == 0))
```

14. clean the track from the buffers to the filler (order number 30). After finishing this order, cleaning liquid is put through the pipes leading from the buffer vessels to the filler, and put into the waste vessel. No pipes leading from the cleaning liquid vessel to the waste vessel is contaminated with paint.

    ```
    []((order[30] == true) -> (order[30] == true U
    ptt[4] == 0))
    ```

Besides checking these orders, we also check some general properties concerning the paint flow. These requirements are:

- When paint is outputted into a cup, this cup will not be used again to output paint in. Because we change the value of a turntable cup when we put in paint, we can check all cups on the turntable if they stay true once they are true.

  ```
  turnt[x] == true -> [] turnt[x] == true
  ```

  We found out that this requirement does not hold. Because in the χ model the cup number is fixed, when an order is carried out twice, the same cup number is filled.

- When there is paint in a buffer vessel, this vessel is emptied before putting a new color into the vessel; thus when putting paint into a buffer vessel, this vessel should be empty. The orders that put paint in the buffer vessels are orders 7, 8 and 9

  ```
  (order[7] == true -> buf1 == 0) &&
  (order[8] == true -> buf2 == 0) &&
  (order[9] == true -> buf3 == 0)
  ```

  Because this depends on the orders given, and not on the hardware of the paintfactory, it is hard to check whether this requirement is true or not. If we want to put paint in a buffer vessel, but this vessel already contains another color of paint, first one of the order numbers 19 to 21 (depending on the buffer vessel) should be carried out before putting new paint into the buffer vessel. However, even though we can pump paint from the buffer vessel to the waste vessel, there is no way to determine the paint level in the buffer vessels.

When checking these orders, we found some errors. Most of these were syntactical errors in the order list, for instance a wrong valve that is switched or turning a pump on instead of off. However, we also found some serious errors:

- In the predefined order, when outputting paint to the turntable, a fixed cup number is defined. This way, if we execute the same order a few

times in a row, the same cup number is used, and paint will be outputted to a cup that already contains paint. Instead of manually choosing the number of an empty cup, it would be better to keep track of which cups are full and which are empty, and let the turn table motor rotate to the next empty cup whenever paint is going to be outputted.

- When one of the buffers or the mixer is cleaned, cleaning liquid is put into these vessels. However, this cleaning liquid is not removed from the vessels. It seems logical to put the cleaning liquid into the vessel that needs to be cleaned, and then afterwards remove it and put it into the waste vessel. Because this is not the case, two orders are needed; one to put the cleaning liquid into the vessel, and one to empty the vessel. It would be better to make one order; one that cleans the buffer vessel and removes the cleaning liquid. This way, when making the order list, it is less likely to make a mistake by for instance put paint in the buffer vessel when there is still cleaning liquid inside.

- There is no order to empty the buffer vessels. Order numbers 19 to 21 pump the contents of the buffer vessel into the waste vessel, but there is no way to determine whether this is all the contents or just part of it. It would be useful to have some kind of measuring mechanism to determine the amount of paint in the buffer vessels, thus making it possible to change the amount of time for the pumps to pump paint out of the buffer vessel.

  A possible way of doing this is to keep a variable that stores the amount of paint in the buffer. This can be done by determining how much paint can flow through the pipes per time unit, and increasing the variable when paint is flowing into the buffer vessel. This way, when emptying the buffer, the pumps only need to be turned on to remove all the paint that is in the buffer, and can then be switched off. This way, time is saved so the paintfactory will be faster.

- A similar thing holds for the mixer. There exists an order for emptying the mixer, but the time to let the paint flow out of the mixer into the waste vessel is fixed. Therefore, it could happen that there is still paint left in the mixer after this order ends, unless the fixed time is long enough to empty the mixer, even if it is completely filled. A variable that keeps track of the amount of paint in the mixer would therefore be useful. This way, to empty the mixer, the pump would only need to pump paint just long enough to pump all the paint from the mixer to the buffer vessel. By doing this we can save time.

5 Performance Analysis

The next thing we want to do is performance analysis. We want to know how well the paint factory performs. Furthermore, performance analysis can give us information about the quantitative correctness of the specification, for instance about the occurance of nondeterminism.

There are a couple of measures we want to obtain. We want to know how long it takes to complete a list of orders. Furthermore, we want to know the average time it takes to complete one order, and the average amount of orders that can be completed per time unit (the throughput). We are also interested in the utilization of the motors. To get these measures, there are two methods we can use.

The first one is to look at results we get from simulation. We can simulate different orders, and let program output start and end times of each order. From this, we will get an idea how long it takes to complete orders, and we can calculate several performance measures.

The second method is a more mathematical approach. We want to generate a state space and a Markov chain, and then use mathematical techniques to calculate performance measures. The advantage of using these techniques over simulation is that this is a more formal approach, and this way we can abstract from deterministic delays. The disadvantage however is that in a Markov chain, the delays are exponential, while this does not correspond to the real situation. In reality, the delays will be between certain bounds.

Finally, we can compare the results of the simulation to the results of the mathematical approach.

5.1 Simulation

The tool we are using for simulation is the χ 1.0 C simulator. We add commands to the original χ 1.0 specification to make the simulator output the time when an order starts, and when an order finishes. We also add commands so the simulator gives a signal when one of the motors is turned on or off.

If there is no parallel behaviour, every order takes as long as the sum of all delays. If there is parallel behaviour however, the average running time per order reduces. Results for the running times from simulation are given in Table 8.

From this table, we can see that for instance order 21 and order 6 are carried out at the same time, and orders 1, 13 and 11 are as well. Thus we can clearly see that because orders are carried out in parallel, more orders can be carried out per time unit. If we let the simulator run for a long time, we get an average time per order of 19.25 seconds. From this, we can calculate a throughput of 0.052 orders per second.

To calculate the utilization of the motors, we let the simulator output times the motors are turned on and off. From this we can calculate the amount of time

Order number	Begin time	End time	Running time	Average time
26	0	51	51	51
7	51	100	49	50
21	100	145	45	48,3
6	100	146	46	36,5
1	145	190	45	38
13	146	191	45	31,8
11	146	193	47	27,6

Table 8: Running times from simulation

the motors are turned on. We found the following utilizations:

Motor	**Utilization**
Buffer motor	2.7 %
Filler motor	0.57 %
Turntable motor	0.39 %

Table 9: Utilization of the motors

From this table, we can see that the utilization of the motors is very low. What can we conclude from this? Because for every order, the flow of paint through the pipes takes 45 seconds, and moving the motors from one position to another only takes 2 seconds, this is not a surprising number. Furthermore, a motor only moves when the target position is different from the current position. As the filler motor only has 2 possible positions, the chance that the filler motor is already at the correct position is 0.5.

From these utilization results, we can conclude that buying faster motors will not improve the performance.

5.2 Performance analysis with mathematical techniques

The next step is to analyze the specification with the help of mathematical methods. The first step in doing this is to generate a state space and a Markov chain.

To do this, several tools exist. One of these tools is a Python state space generator, being developed for use with the χ 1.0 language. This is a state space generator based on the Python χ simulator. We want to use this tool, because it uses χ 1.0 as an input language. This is very convenient because we would not have to translate the paint factory specification. However, when this project was being carried out, the state space generator was not finished. Because this took too much time, we decided not to wait for the Python state space generator and only use the χ_σ tool to generate a state space and Markov chain.

The χ_σ tool is a tool developed to do verification and performance analysis on χ specifications. This tool, developed by V. Bos and J.J.T. Kleijn and

extended by N. Trcka, can generate a state space or a Markov chain by using the CADP [10] toolset. To be able to use this tool, we have to translate the χ 1.0 specification to another version of χ, called χ_σ. This language is developed to provide a formal framework for χ specifications, thus making it possible to do calculations like performance analysis.

We translated the χ 1.0 specification to χ_σ with the help of [11]. The translation was pretty straightforward. [11] provides a translation scheme for translation from χ 1.0 to χ_σ. The χ_σ specification can be found in Appendix D.

We then tried to generate a state space and Markov chain with this tool from the translated χ 1.0 specification. Because the tool is quite slow and uses a lot of memory, it was impossible to use this tool for the paint factory specification. When we tried to generate a state space or markov chain on a computer with 4 GB of memory, after running for about 1 hour it gave an error saying there is not enough memory. To overcome this problem, we simplified the specification. We took out the processes regarding the pipes and valves. Because there are no delays in these processes, this should not influence the time it takes to complete the orders. Only orders which involved moving one of the motors remained. We also changed the working of the generator. Instead of reading an input file, we let the generator send a fixed sequence of orders. These changes didn't fix the memory problem, however. Only after taking out all resource controllers except one, we could generate a state space and Markov chain. However, by taking out two of the resource controllers, it is not possible to have parallel behavior, and all orders will be carried out one by one. The Markov chain and state space generated are thus not representative for the paint factory.

When a new server with more memory (100 GB) arrived at the faculty, this gave new possibilities for generating a Markov chain and state space. We tried to generate a state space with this new server, however, this was still unsuccessful. Our last try on doing performance analysis was to generate a Markov chain and state space with a finite amount of orders. After sending the fixed sequence of orders, the generator would just stop sending orders. A deadlock would thus occur. This way, there is a finite amount of actions and the state space would be smaller, thus making it easier to generate a Markov chain. We tried this with a predefined list of 3 orders (order numbers 26, 10 and 25). Because these orders only use the buffer motor and filler motor, we could abstract from the turntable motor. We chose these orders because they use the buffer motor and filler motor. With these orders, we were able to generate a state space and a Markov chain.

The first thing we did was generating the state space. When generating a state space, the tool generates every possible state, with all possible actions and communications. We can choose to show all actions, or hide certain actions to get a clearer view at the state space. To get a clear view at the running time, we chose to generate a state space, but to hide all actions and communications, and to minimize it modulo branching bisimulation. What we get is a graph showing all delays. Because this graph is minimized modulo branching bisimulation, the graph also shows whether there occurs nondeterminism or not. From this

graph, we can calculate the running time in case of discrete delays. This state space can be found in Table 5. The state space starts at the bottom with the *start* transition. There is a transition to another state for every delay. Because there are only 3 orders to perform, we can see the d-loop in the last state, indicating the deadlock. From this state space, we can clearly see that it would take 90.8 seconds to perform these three orders.

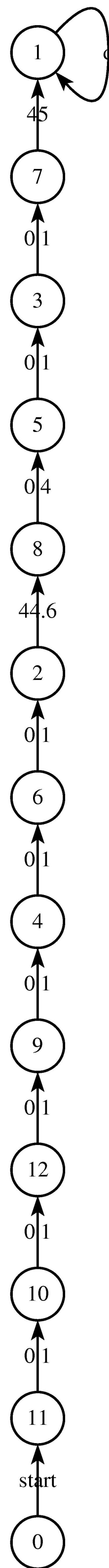


Figure 5: State space

The next thing we did was generate a Markov chain from the same specification. The generated Markov chain is depicted in Figure 6. In this Markov chain, we can see all time steps as exponential rates. Because the order list is finite, this Markov chain is a finite one, and thus has an absorbing state. This makes it not very useful to calculate measures like throughput and utilization, because we don't have a steady state distribution, and in the long run we will always end up in the final state. What is an interesting measure we can calculate, is the mean time until absorption in the final state, thus the time to complete all orders.

We define τ_a as the time to reach the absorbing state from t = 0. The mean time until absorption is thus ($E[\tau_a]$). This is calculated with the following formula (see [15]):

$$E[\tau_a] = \pi_0 * (-Q')^{-1} * \mathbf{1},$$

where π_0 is the row vector with the initial probabilities for the starting state, so this is a 1 and the rest of the row 0's. Q' is the transition rate matrix without the row and column of the absorbing state:

Q = $[q_{ij}]$ where q_{ij} = the transition rate from state i to state j
and

$$q_{ii} = -\sum_{j,j\neq i} q_{ij}$$

Finally, **1** is a column vector of all 1's.

Through this formula, we found out that the mean time until absorbtion is 101.851 seconds.

The reason why this number is different from the number we got from the state space, is that in a markov chain the delays are exponential, while in a state space they are deterministic. In the state space, when there are 2 possible delays, the smallest one will always be chosen. However, in the Markov chain, the delays are exponential. A delay d only means that d is the expected value, in fact any of the rates (the rate is 1/d) can be chosen. The bigger the rate (so the smaller d) the bigger the chance this rate will be chosen.

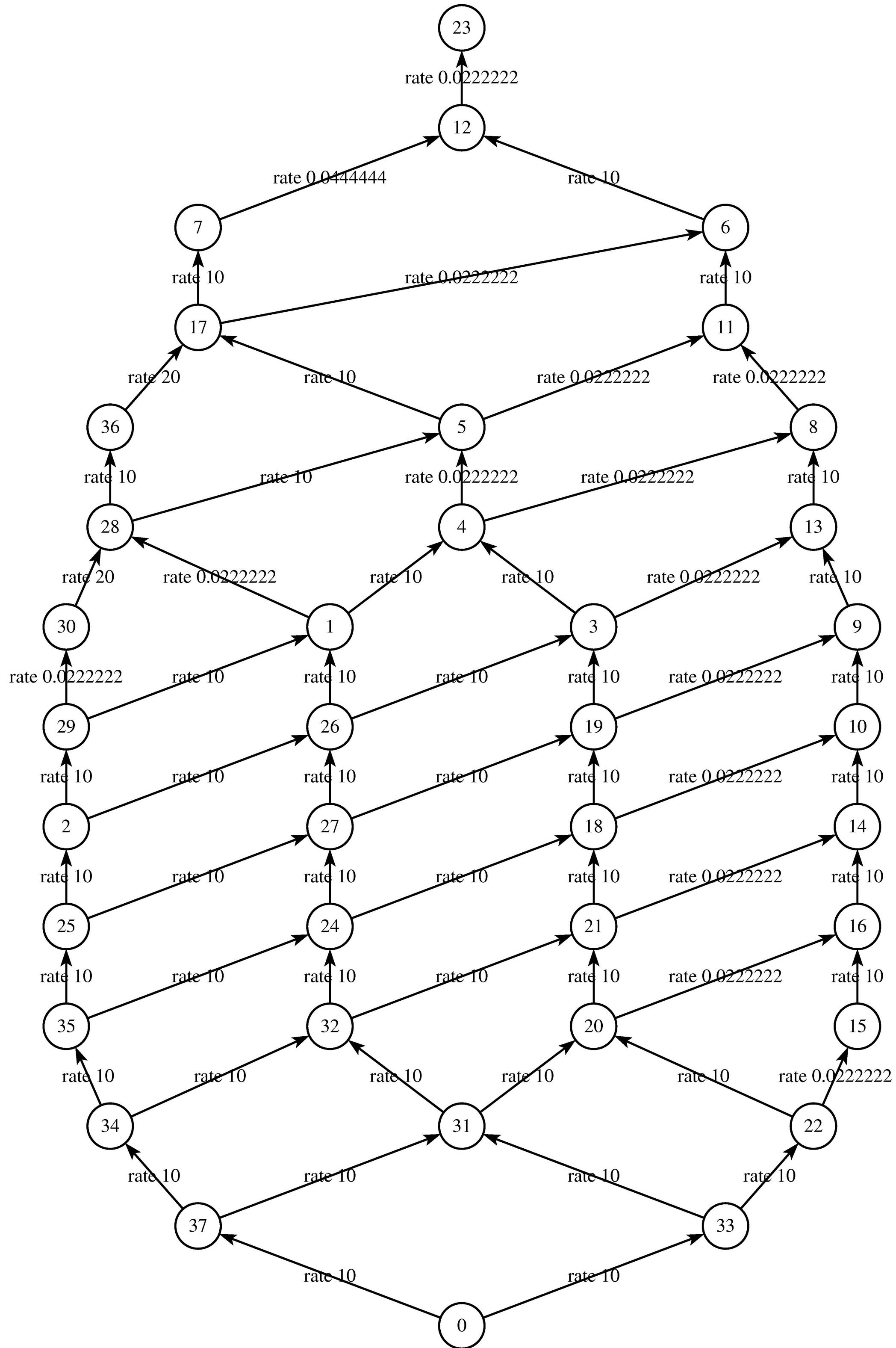


Figure 6: Markov chain

6 The use of the paint factory in high Schools

Computer science education at high schools is still very much in development. In 1998, the subject was introduced in high schools as an optional subject, but the methods and books used in this subject are very limited. In this chapter, we will investigate the possible use of the paint factory for use in computer science education, and describe a possible scenario for doing this.

The paint factory is easy to understand and tangible; the effect of actions is immediately visible. The paint factory would thus be very suitable to use as an introduction to programming, specification and scheduling. These principles are very important in computer science. Unfortunately, in the current program in high schools, the attention given to these principles is very limited. Therefore, we present a way to introduce this in an easy and fun way.

The first step in using the paint factory would be to make a virtual version of the paint factory, where students can see it as a whole but where they can also explore each part separately. A good way to let them get a feeling for how the paint factory works, is make each part clickable, enabling the students to click on several parts of the paint factory. This way, they can get to know the different parts of the paint factory. A nice feature would be to enable them to control parts of the factory by clicking on them, for instance by turning on or off pumps, switching valves and move motors. This way, students can get a feeling for the paint factory and find out what exactly happens when changes are made.

Secondly, students should be challenged to think about designing a simple program for the paint factory. By giving them a simple task (for example output one cup of green paint and clean the system afterwards), students have to think about designing an optimal and efficient program that will bring the desired result. Instead of teaching them the syntax and semantics of χ or another specification language, a graphical user interface would be easier to understand and to use. I would suggest an environment where students can design a program through the use of several 'building blocks' where each block denotes an action, for instance open or close a valve, turn on a pump or move a motor. These blocks can be put in sequences, an order can be created in this way. Sequences of blocks can be saved, and stored in a newly created block. These blocks can be reused in other programs.

Once students are able to design simple orders (for instance filling one cup on the turntable and clean the system afterwards), more complicated orders can be designed, thus forcing students to think about parallelism and scheduling. They can be challenged to design the fastest program. This can be done as an assigment for small groups of students.

Furthermore, the paint factory can also be used to introduce students to formal methods. To do this, a tool that can check properties of the created programs is necessary. After students created a program for the paint factory, they should specify properties that should always hold. They can for instance define a property that a resource can only be used for one order at the same time. With

the tool, these properties can be checked. If the property does not hold, an example program can be given to show what can go wrong. Finally, the created program can be adapted if necessary.

This way, students are introduced to basic programming, scheduling problems and the verification of programs, and challenged to think about this without needing to know anything about programming or specification languages.

7 Conclusions

The main objective for this project was to analyze and improve a model for the paint factory. We have studied the specification, and run different tests on them. We came to a couple of conclusions in this project.

The first objective of this project was to study the current χ specification. To do this, first we translated it to the newest version of χ, χ 1.0. What we learnt in doing this is that there is insufficient information about the syntax of the different χ versions, and the differences between them. There are also different compilers/simulators, that accept different data structures. The first conclusion is that more documentation about the χ language and the available tools is required.

The second objective is to find out what tools exist for model checking and how they can be used for the paint factory specification. There are several tools available for model checking. The tool we used to do this is called SPIN. SPIN is a very useful tool with a lot of possibilities. Because SPIN does not accept χ specifications we had to translate the paint factory specification to a language called Promela. Once this was done, SPIN proved to be a very suitable tool. It is very easy to simulate programs and to verify properties the program should have. With the SPIN model checker, we found a couple of errors in the χ specification, and we were able to correct them. Therefore, the conclusion we can take is that SPIN is a suitable tool to use in model checking χ specifications.

Regarding performance analysis, the results were not as we hoped. It seems impossible to generate a state space or Markov chain for a reasonably large specification. We can conclude that with the available tools for performance analysis for χ specifications, it is very hard to generate a Markov chain and thus do performance analysis.

We also reached a conclusion about tools in general:

There are a lot of tools available to analyze specifications. These tools can be very useful to use for χ specifications, for instance to do model checking or performance analisys. However, these tools all use a different input language. To be able to use these tools, translator tools have been developed.

However, the translators that currently exist are insufficient to be used on complicated χ specification like the paintfactory one. Therefore, we had to do most translation by hand. This is a dangerous operation because we risk introducing mistakes.

The conclusion we reach is that there are suitable tools for analyzing specifications, but the translator tools that have been developed to be able to use these tools with χ specifications are insufficient.

The last objective was to research possibilities for use of the paint factory in Computer Science education in high schools. We researched this, and think that the paint factory is very suitable for use in Computer Science education, because it is easy to understand and tangible. We described a possible scenario

for using the paint factory. Therefore, the last conclusion we make is that the paint factory is very usable in Computer Science education in high schools.

8 Recommendations

In this section, we will give recommendations about various things we came across in this project.

8.1 The specification language χ

The first step in this project was the translation of the current χ specification to the newest version of χ, χ 1.0. In doing this, we found out that the information about the syntax of χ 1.0 is very limited. The first recommendation would thus be to provide more information about the χ language, both about syntax and semantics. But, during this project, a χ reference manual came out. This manual came out in February 2007, and describes the syntax and static semantics of all available χ 1.0 language constructs. However, some information about the differences between 2 versions of χ, like a translation scheme, would be very convenient.

The Chi 1.0 Language Reference Manual describes the syntax and the static semantics of all available Chi 1.0 language constructs in a compact manner. It is set up as a reference manual, intended for users that know the language but need a more precise and complete description of a construct.

There are currently 2 compilers/simulators for χ 1.0; one based on the language C, and one using the language Python. Although both of these simulators accept χ 1.0 specifications, the Python simulator does not accept all data structures. This is however not documented. It is up to the user of the simulator to contact the creator of this simulator to find out what data structures are accepted, and which ones are not.

8.2 Tools

The biggest problem we had when using the several tools is that their input language is different from χ. Tools like SPIN are very useful for model checking, but we have to translate the specification to another language before we can do model checking. We risk introducing errors when translating, or removing errors that were present in the original specification. Furthermore, some datastructures in one language are not accepted in another, thus risking losing functionality in the translated specification.

My recommendation is to extend the current tools that accept specifications in χ 1.0 and can use other toolkits, like SPIN, CDAP and UPPAAL to measure performance and verify several properties. We thus want to have a translator tool that can handle big specifications like the χ one. This way, a lot of time is saved on translation and the risk of making errors with manual translation is avoided. A translator tool like this is a necessity, especially if we want to further develop these measuring tools for use in industry.

8.3 Performance Analysis

The current tools for doing performance analysis on χ specifications, do not work on large and complicated specifications. There are more efficient tools for generating a state space or a Markov chain, like the μCRL toolkit (see [16]). If this tool can be used to generate a state space for χ specifications, this can be helpful for use with large specifications. To be able to use this toolkit, we need a translator tool from χ to μCRL. The recommendation in this is thus either the development of a more efficient tool that can deal with complicated specification and is able to generate a state space or a Markov chain from a χ specification, or a translator tool that can translate χ specifications to the input language of an existing tool (like μCRL) that can generate the state space or a Markov chain for large and complicated specifications.

8.4 The use of the paint factory in Computer Science education in high schools

The paint factory could be easily used in high schools. However, to be able to use this, a piece of software needs to be designed, as described in Chapter 6. The recommendation in this is to create this software and test this on high school students.

References

[1] K. Eijsvogels
Introductie en ontwerp van de verffabriek
Systems Engineering group, Mechanical engineering, Eindhoven, 1998

[2] M.H.M. van Duin
From simulation using χ *to implementation using VxWorks. A case: The paint factory*
SE 420225

[3] W.A.P. van den Bremer
Design of a Supervisory Controller for the SE-Paint Factory
SE 420429

[4] Spin - Formal verification
`http://spinroot.com`

[5] chi [SeWiKi - Systems Engineering]
`http://se.wtb.tue.nl/sewiki/chi/`

[6] J. Vervoort and J.E. Rooda
Learning χ *0.8*
Department of Mechanical engineering, 2004

[7] J. Vervoort and J.E. Rooda
Learning χ *1.0*
Department of Mechanical engineering, 2006

[8] Gerard J. Holzmann
The Spin Model Checker, Primer and Reference Manual
2004, Addison-Wesley

[9] Model Checking
`http://en.wikipedia.org/wiki/Model_checking`

[10] CADP Home page
`http://www.inrialpes.fr/vasy/cadp.html`

[11] V. Bos and J.J.T. Kleijn
Formal specification and analysis of industrial systems
Computer science, Eindhoven, 2002

[12] TIPSy Project
`http://homepages.cwi.nl/~wijs/TIPSy/`

[13] Discrete-time Promela and Spin
`http://www.win.tue.nl/~dragan/DTSpin/`

[14] N. Trcka,
Verifying χ Models of Industrial Systems with Spin
CS-Report 05-12, Department of mathematics and computer science, Eindhoven, 2005

[15] Andrea Bobbio
Continuous-time Markov chains
`http://www.mfn.unipmn.it/~bobbio/BISS/ctmcfig.ps`

[16] μCRL
`http://homepages.cwi.nl/~mcrl/index.html`

A Original χ 0.8 specification

```
proc G(chan gd!: order, f?: file) = |[ var xs: [order], x: order ::
f?xs
 ; (len(xs) > 0) *> (x:=hd(xs); gd!x; xs:= tl(xs))
]|

proc DP(chan gd?: order, dr! 3*(order), rd?: 3*(rsrc)) = |[ var x: order,
             b: bool = true, xr: rsrc =
{"pc1", "pc2", "pc3", "c", "1", "b1", "b2", "b3", "ppi2", "fpi", "bpi"},
yr: rsrc ::
  *(b;gd?x   -> b:= false
    | ( | , j <- 0..2 , not b and x.1 sub xr; dr.j      ->
       xr:= xr - x.1; b:= true)
    | ( | , j <- 0..2 , true                ; rd.j?yr   ->
       xr:= xr + yr)
    )
]|


proc RC(chan dr?: order, rd!: rsrc, rb!: bmorder, rf!: fmorder
        , rt!: ttmorder, rv!: vorder, rp!: porder
        , br?, fr?, ttr?, vr?, pr?: void
        ) =
|[var xt: order, xs: [act], x: act ::vr?; pr?; br?; fr? ttr?
 ;  *(dr?xt; xs:= xt.2
        ; (len(xs) > 0) *> (x:=hd(xs); xs:= tl(xs)
                         ; (x.0 = "m"  -> (x.1 = 0 -> rb!x.2; br?
                                          |x.1 = 1 -> rf!x.2; fr?
                                          )
                           |x.0 = "mp" -> (x.1 = 1 -> rf!x.2
                                          |x.1 = 2 -> rt!x.2
                                          )
                           |x.0 = "sm" -> (x.1 = 1 -> fr?
                                          |x.1 = 2 -> ttr?
                                          )
                           |x.0 = "v"  -> rv!(x.1,x.2); vr?
                           |x.0 = "p"  -> rp!(x.1,x.2); pr?
                           |x.0 = "d"  -> delay x.2
                           ))
        ; rd!xt.1
     )
 ]|

 proc BMC(chan rb?: (bmorder)^3, br!: (void)^3, cbd!: dir, cbm!: m_on
          , st? (void)^4, si? (bool)^4
```

```
        ) =
|[ var b: bool = false, k: bmorder, i: nat = 0, m: nat
:: (i<4 and not b) *> (si.i?b; i:= i+1)
 ; (b      -> m:= i - 1
   |not b  -> cbd!true; cbm!true
              ; st.3? m:= 3; cbm!false
   )
 ; i:= 0; (i < 3) *> (br.i!; i:= i + 1)
 ;*( | , j <- 0..2 , rb.j?k
           ;(k = m    -> skip
            |k/= m    -> cbd!k > m; cbm!true
                        ; st.k?; cbm!false; m:= k
            )
           ; br.j!

   )
]|

proc BM(chan cbd?: dir, cbm?: m_on, bs!: (bool)^4, val rt: real) =
|[ var d: dir, m: m_on, u: ->nat = uniform(20,81), i, q: nat
:: q:= sample u
 ; i:= 0
 ; (i < 4)*> (bs.i!q = 20 * (i + 1); i:= i + 1 )
 ; m:= false
 *(m -> delta rt;  (d      -> q:= q + 1
                   |not d  -> q:= q - 1
                   )
                ; i:= 0
                ; (i < 4)*> (bs.i!q = 20 * (i + 1); i:= i + 1 )
  |cbd?d -> skip
  |cbm?m -> skip
  )
]|

proc FMC(chan rf?: (fmorder)^3, fr! (void)^3, cfd!: dir, cfm!: m_on
        , st?: (void)^2, si?: (bool)^2 ) =
|[ var b: bool = false, k: fmorder, i: nat = 0, m: nat
:: ( i < 2 and not b)*> (si.i?b; i:= i + 1)
 ; (b      -> m:= i - 1
   |not b  -> cfd!true; cfm!true; st.1?; m:= 1; cfm!false
   )
 ; i:= 0; (i < 3) *> (fr.i!; i:= i + 1)
 ;*( | , j <- 0..2 , rf.j?k
           ;(k = m    -> skip
            |k/= m    -> cfd!k > m; cfm!true
                       ; st.k?; cfm!false; m:= k
            )
```

```
             ; br.j!
   )
]|

 proc FM(chan cfd?: dir, cfm?: m_on, fs!: (bool)^2, val rt: real) =
 |[ var d: dir, m: m_on, u: ->nat = uniform(20,41), i, q: nat
 :: q:= sample u
  ; i:= 0
  ; (i < 2)*> (fs.i!q = 20 * (i + 1); i:= i + 1 )
  ; m:= false
  *(m -> delta rt;  (d      -> q:= q + 1
                    |not d  -> q:= q - 1
                    )
                  ; i:= 0
                  ; (i < 4)*> (bs.i!q = 20 * (i + 1); i:= i + 1 )
   |cfd?d -> skip
   |cfm?m -> skip
   )
 ]|

 proc TMC(chan rtt?: (ttmorder)^3, ttr!: (void)^3, ctd!dir, ctm!: m_on
         ,sf?, st?: (void)^2, si? (bool)^2) =
 |[ b: bool, k: ttmorder, i, m, p q: nat
 :: si.0?b
  ; ctd!true; ctm!true
  ; (b      -> st.1?
    |not b  -> st.0?; st.1?
    )
  ;ctm!false; m:= 0
  ;i:= 0; (i<3)*>(ttr.i!; i:= i + 1)
  *( | , j <- 0..2 , rtt.j?k
              ; p:= (12 + k - m) mod 12
              ; (p = 0 -> skip
                | p /= 0 -> ctd!p <= 6
                ; q:= min(p,12 - p)
                ; ctm!true; (q>0) *> (sf.1?; st.1?; q:= q - 1)
                ; ctm!false; m:= k
                )
              ; ttr.j!
   )
 ]|

 proc TM(chan ctd?: dir, ctm?: m_on, ts!: (bool)^2, val rt: real) =
 |[ var d: dir, m: m_on, u: ->nat = uniform(0,120), q: nat
 :: q:= sample u
  ; ts.0!q = 5
  ; m:= false
```

```
  *(m -> delta rt;  (d      -> q:= (q + 1) mod 120
                    |not d  -> q:= (q + 119) mod 120
                    )
                 ; ts.0!q = 5; ts.1!q mod 10 = 0
   |ctd?d -> skip
   |ctm?m -> skip
   )
 ]|

 proc VC(chan rv? (vorder)^3, vr!: (void)^3, cv!: (nat)^8 =
 |[ var i: nat = 0, x: vorder
 :: (i < 18) *> (cv.i!0; i:= i + 1 )
  ; i:= 0; ( i < 3) *> ( vr.i!; i:= i + 1)
  ; *( | , j <- 0..2 , rv.j?x
                ; cv.(x.0)!x.1; vr.j!
     )
 ]|

  proc PC(chan rp? (porder)^3, pr!: (void)^3, cp!: (nat)^8 =
 |[ var i: nat = 0, x: porder
 :: (i < 8) *> (cp.i!0; i:= i + 1 )
  ; i:= 0; ( i < 3) *> ( pr.i!; i:= i + 1)
  ; *( | , j <- 0..2 , rp.j?x
                ; cp.(x.0)!x.1; pr.j!
     )
 ]|

proc A(chan ca?: nat) = |[ var k: nat :: *(ca?k) ]|

proc S(chan senin?: bool, st!: void, sini!: bool) = |[ var b: bool
:: senin?b
 ; *(true   -> senin?b
    |b      -> st!
    |true;  -> sini!b
    )
]|

proc TTS(chan senin?: bool, sf!, st!: void, sini!: bool) = |[ var b:
bool :: senin?b
 ; *(true   -> senin?b
    |not b  -> sf!
    |b      -> st!
    |true;  -> sini!b
    )
]|

model FACTORY()= |[ chan   gd: order
```

```
, dr: (order)^3, rd: (rsrc)^3
, rb: (bmorder)^3, rf: (fmorder)^3, rtt: (ttmorder)^3
, rv: (vorder)^3, rp: (porder)^3
, br: (void)^3, fr: (void)^3, ttr: (void)^3
, vr: (void)^3, pr: (void)^3
, bs, sbi: (bool)^4, sbt: (void)^4, cbd: dir, cbm: m_on
, fs, sfi: (bool)^2, sft: (void)^2, cfd: dir, cfm: m_on
, ts, sti: (bool)^2, stf, stt: (void)^2, ctd: dir, ctm: m_on
, va: (nat)^18, pa: (nat)^8
:: G(gd, filiein("bestand.txt"))
|| DP(gd, dr, rd) || *( | , j <- 0..2 , RC(dr.j, rd.j, rb.j,
rf.j, rtt.j, rv.j, rp.j, br.j, fr.j, ttr.j, vr.j, pr.j))
|| BMC(rb, br, cbd, cbm, sbt, sbi)
|| BM(cbd, cbm, bs, 0.1)
|| *( |, j <- 0..3 , S(bs.j, sbt.j, sbi.j))
|| FMC(rf, fr, cfd, cfm, sft, sfi)
|| FM(cf, cfm, fs, 0.1)
|| *( | , j <- 0..1: S(fs.j, sft.j, sfi.j))
|| TMC(rtt, ttr, ctd, ctm, stf, stt, sti)
|| TM(ctd, ctm,ts, 0.1)
|| *( | , j <- 0..1: TTS(ts.j, stf.j, stt.j, sti.j))
|| VC(rv, vr, va)
|| *( | , j <- 0..18: A(va.j))
|| PC(rp, pr, pa)
|| *( | , j <- 0..8: A(pa.j)) ]|
```

B Promela specification

```
#include "dtime.h"

/*the following global variables have been defined to use in the
 *defined LTL formulae used for model checking.
 */

 byte legelijst;
 byte posbq;
 byte posfq;
 byte postq;
 byte posbk;
 byte posfk;
 byte postk;
 bool lxr[14];
 bool bm;
 bool fm;
 bool tm;
 int ptt[5];
 int mix;
 int buf1;
 int buf2;
 int buf3;
 int ltt[12];
 bool order[31];
 bool turnt[11];

/* the following definitions have been defined to make use of lists.
 *The add(x,lst) function adds a value x to list lst, hd(lst) gives
 *the head (first value) of the list, tail(lst) gives everything but
 *the first value(the tail), and length(lst) gives the number of
 *values in list lst.
 */

 #define add(x,lst) \
                           lst.l!x;\
                           if\
                            :: len(lst.l) == 1 -> lst.head = x\
                            :: else\
                           fi;\
```

```
 #define hd(lst) (lst.head)

 #define tail(lst) d_step{ lst.l?_;\
                    if\
                     :: len(lst.l) > 0 -> lst.l?<lst.head>\
                     :: else\
                    fi;\
                   }
  #define length(lst) (len(lst.l))




active proctype FACTORY (  ) {     chan gd   = [ 0 ] of { byte , int
}
                                ;  chan dr0  = [ 0 ] of { byte , int }
                                ;  chan dr1  = [ 0 ] of { byte , int }
                                ;  chan dr2  = [ 0 ] of { byte , int }
                                ;  chan rd0  = [ 0 ] of { byte , int }
                                ;  chan rd1  = [ 0 ] of { byte , int }
                                ;  chan rd2  = [ 0 ] of { byte , int }
                                ;  chan rb0  = [ 0 ] of { byte , int }
                                ;  chan rb1  = [ 0 ] of { byte , int }
                                ;  chan rb2  = [ 0 ] of { byte , int }
                                ;  chan rf0  = [ 0 ] of { byte , int }
                                ;  chan rf1  = [ 0 ] of { byte , int }
                                ;  chan rf2  = [ 0 ] of { byte , int }
                                ;  chan rtt0 = [ 0 ] of { byte , int }
                                ;  chan rtt1 = [ 0 ] of { byte , int }
                                ;  chan rtt2 = [ 0 ] of { byte , int }
                                ;  chan rv00  = [ 0 ] of { byte , int }
                                ;  chan rv01  = [ 0 ] of { byte , int }
                                ;  chan rv10  = [ 0 ] of { byte , int }
                                ;  chan rv11  = [ 0 ] of { byte , int }
                                ;  chan rv20  = [ 0 ] of { byte , int }
                                ;  chan rv21  = [ 0 ] of { byte , int }
                                ;  chan rp00  = [ 0 ] of { byte , int }
                                ;  chan rp01  = [ 0 ] of { byte , int }
                                ;  chan rp10  = [ 0 ] of { byte , int }
                                ;  chan rp11  = [ 0 ] of { byte , int }
                                ;  chan rp20  = [ 0 ] of { byte , int }
                                ;  chan rp21  = [ 0 ] of { byte , int }
                                ;  chan bs0  = [ 0 ] of { bool , int }
                                ;  chan bs1  = [ 0 ] of { bool , int }
                                ;  chan bs2  = [ 0 ] of { bool , int }
                                ;  chan bs3  = [ 0 ] of { bool , int }
                                ;  chan br0  = [ 0 ] of { bool , int }
```

```
                                ;  chan br1  = [ 0 ] of { bool , int }
                                ;  chan br2  = [ 0 ] of { bool , int }
                                ;  chan fr0  = [ 0 ] of { bool , int }
                                ;  chan fr1  = [ 0 ] of { bool , int }
                                ;  chan fr2  = [ 0 ] of { bool , int }
                                ;  chan ttr0 = [ 0 ] of { bool , int }
                                ;  chan ttr1 = [ 0 ] of { bool , int }
                                ;  chan ttr2 = [ 0 ] of { bool , int }
                                ;  chan vr0  = [ 0 ] of { bool , int }
                                ;  chan vr1  = [ 0 ] of { bool , int }
                                ;  chan vr2  = [ 0 ] of { bool , int }
                                ;  chan pr0  = [ 0 ] of { bool , int }
                                ;  chan pr1  = [ 0 ] of { bool , int }
                                ;  chan pr2  = [ 0 ] of { bool , int }
                                ;  chan sbt0 = [ 0 ] of { byte , int }
                                ;  chan sbt1 = [ 0 ] of { byte , int }
                                ;  chan sbt2 = [ 0 ] of { byte , int }
                                ;  chan sbt3 = [ 0 ] of { byte , int }
                                ;  chan cbd = [ 0 ] of { bool , int }
                                ;  chan cbm = [ 0 ] of { bool , int }
                                ;  chan fs0  = [ 0 ] of { bool , int }
                                ;  chan fs1  = [ 0 ] of { bool , int }
                                ;  chan sft0 = [ 0 ] of { byte , int }
                                ;  chan sft1 = [ 0 ] of { byte , int }
                                ;  chan cfd = [ 0 ] of { bool , int }
                                ;  chan cfm = [ 0 ] of { bool , int }
                                ;  chan ts  = [ 0 ] of { bool , int }
                                ;  chan stt = [ 0 ] of { byte , int }
                                ;  chan ctd = [ 0 ] of { bool , int }
                                ;  chan ctm = [ 0 ] of { bool , int }
                                ;  chan va  = [ 0 ] of { byte , int }
                                ;  chan pa  = [ 0 ] of { byte , int }
; atomic { run G ( gd )
; run DP ( gd , dr0 , dr1 , dr2 , rd0 ,
         rd1 , rd2 )
; run RC ( dr0 , rd0 , rb0 , rf0 , rtt0 ,
          rv00, rv01 , rp00, rp01 , br0 , fr0 ,
          ttr0 , vr0 , pr0 )
; run RC ( dr1 , rd1 , rb1 , rf1 , rtt1 ,
          rv10, rv11 , rp10, rp11 , br1 , fr1 ,
          ttr1 , vr1 , pr1 )
; run RC ( dr2 , rd2 , rb2 , rf2 , rtt2 ,
          rv20, rv21 , rp20, rp21 , br2 , fr2 ,
          ttr2 , vr2 , pr2 )
; run BMC ( rb0 , rb1 , rb2 , br0 , br1 ,
           br2 , cbd , cbm , sbt0 , sbt1 , sbt2 ,
           sbt3 )
```

```
 ; run BM ( cbd , cbm , bs0 , bs1 , bs2 ,
           bs3 )
 ; run S ( bs0 , sbt0 , 0 )
 ; run S ( bs1 , sbt1 , 1)
 ; run S ( bs2 , sbt2 , 2)
 ; run S ( bs3 , sbt3 , 3)
 ; run FMC ( rf0 , rf1 , rf2 , fr0 , fr1 ,
            fr2 , cfd , cfm , sft0 , sft1 )
 ; run FM ( cfd , cfm , fs0 , fs1)
 ; run S ( fs0 , sft0 , 0 )
 ; run S ( fs1 , sft1 , 1 )
 ; run TMC ( rtt0 , rtt1 , rtt2 , ttr0 ,
            ttr1 , ttr2 , ctd , ctm , stt )
 ; run TM ( ctd , ctm , ts)
 ; run S ( ts , stt , 0 )
 ; run VC ( rv00, rv01 , rv10, rv11 , rv20,
           rv21 , vr0 , vr1 , vr2 , va )
 ; run A ( va )
 ; run PC ( rp00, rp01 , rp10, rp11 , rp20,
           rp21 , pr0 , pr1 , pr2 , pa )
 ; run A ( pa ) }}


/* Process G creates the initial order list. it sends the order
 * numbers one by one to the dispatcher. Then it nondeterministically
 * chooses a number between 1 and 30 to add to the order list.
 * Finally it checks whether the list is empty, if this is the case,
 * the process ends.
 */
 proctype G (
chan gd ) {
                        listint lxs;
                        d_step {add(10,lxs)};
                        do
                        :: gd ! hd(lxs), (length(lxs) > 0); tail(lxs)
                        do
                            :: add(1,lxs); break
                            :: add(2,lxs); break
                            :: add(3,lxs); break
                            :: add(4,lxs); break
                            :: add(5,lxs); break
                            :: add(6,lxs); break
                            :: add(7,lxs); break
                            :: add(8,lxs); break
                            :: add(9,lxs); break
                            :: add(10,lxs); break
                            :: add(11,lxs); break
```

```
                    :: add(12,lxs); break
                    :: add(13,lxs); break
                    :: add(14,lxs); break
                    :: add(15,lxs); break
                    :: add(16,lxs); break
                    :: add(17,lxs); break
                    :: add(18,lxs); break
                    :: add(19,lxs); break
                    :: add(20,lxs); break
                    :: add(21,lxs); break
                    :: add(22,lxs); break
                    :: add(23,lxs); break
                    :: add(24,lxs); break
                    :: add(25,lxs); break
                    :: add(26,lxs); break
                    :: add(27,lxs); break
                    :: add(28,lxs); break
                    :: add(29,lxs); break
                    :: add(30,lxs); break
                    od
                :: length(lxs) == 0 -> legelijst = 1; break
                 od}



proctype DP (  chan gd, dr0, dr1, dr2, rd0, rd1, rd2 ) {
        byte x
        ; bool b
        ; byte y
        ; byte z
        ; byte p
        ; y = 0
        ; z = 0
        ; p = 0
        ; b = true
        ; bool lxr[14]
        /* To initialize list lxr, we have to assign each value separately.
         * Promela unfortunately does not have a way to assign all values
         * of a list in a single step.
         */
         ; d_step{ lxr[0] = true ; lxr[1] = true ; lxr[2] = true ;
         lxr[3] = true ; lxr[4] = true ; lxr[5] = true ; lxr[6] = true ;
         lxr[7] = true ; lxr[8] = true ; lxr[9] = true ; lxr[10] = true ;
         lxr[11] = true ; lxr[12] = true ; lxr[13] = true }
            ; do
            :: gd ? x , eval ( 2 - b )
```

```
; b = false
    ; if
    :: d_step{x == 1 ; y = 1 ; z = 9 ; p = 5}
    :: d_step{x == 2 ; y = 2 ; z = 9  ; p = 5}
    ::  d_step{x == 3 ; y = 3 ; z = 9  ; p = 5 }
    ::  d_step{x == 4 ; y = 1 ; z = 10 ; p = 13 }
    ::  d_step{x == 5 ; y = 2 ; z = 10 ; p = 13 }
    ::  d_step{x == 6 ; y = 3 ; z = 10 ; p = 13}
    ::  d_step{x == 7 ; y = 6 ; z = 5  ; p = 12 }
    ::  d_step{x == 8 ; y = 7 ; z = 5  ; p = 12 }
    ::  d_step{x == 9 ; y = 8 ; z = 5  ; p = 12 }
    ::  d_step{x == 10 ; y = 6 ; z = 11  ; p = 13 }
    ::  d_step{x == 11 ; y = 7 ; z = 11  ; p = 13 }
    ::  d_step{x == 12 ; y = 8 ; z = 11  ; p = 13 }
    ::  d_step{x == 13 ; y = 1 ; z = 9  ; p = 0 }
    ::  d_step{x == 14 ; y = 2 ; z = 9  ; p = 0 }
    ::  d_step{x == 15 ; y = 3 ; z = 9  ; p = 0 }
    ::  d_step{x == 16 ; y = 1 ; z = 10  ; p = 0 }
    ::  d_step{x == 17 ; y = 2 ; z = 10  ; p = 0 }
    ::  d_step{x == 18 ; y = 3 ; z = 10  ; p = 0 }
    ::  d_step{x == 19 ; y = 6 ; z = 11  ; p = 13 }
    ::  d_step{x == 20 ; y = 7 ; z = 11  ; p = 13 }
    ::  d_step{x == 21 ; y = 8 ; z = 11  ; p = 13 }
    ::  d_step{x == 22 ; y = 5 ; z = 12  ; p = 0 }
    ::  d_step{x == 23 ; y = 4 ; z = 9  ; p = 5 }
    ::  d_step{x == 24 ; y = 6 ; z = 4  ; p = 12 }
    ::  d_step{x == 25 ; y = 7 ; z = 4  ; p = 12 }
    ::  d_step{x == 26 ; y = 8 ; z = 4  ; p = 12 }
    ::  d_step{x == 27 ; y = 4 ; z = 9  ; p = 0 }
    ::  d_step{x == 28 ; y = 4 ; z = 12  ; p = 0 }
    ::  d_step{x == 29 ; y = 4 ; z = 10  ; p = 13 }
    ::  d_step{x == 30 ; y = 4 ; z = 11  ; p = 13 }
    fi ;
   /* Because channel bundling is not possible in Promela,
    * we need to define separate channels. Therefore,
    * the specification becomes a lot bigger because we
    * cannot group assignments together.
    * In the chi specification, the following 18 lines of
    * code are described as the following:
    * | ( | , j <- 0..2 , (not b and (xr.y = true) and
    * (xr.z = true) and (xr.p = true) ) ->
    * dr.j!x ; xr.y := false ; xr.z := false ; xr.p := false ;
    * b:= true ; xr.0 := true)
    * In Promela, we need to define all actions for each
    * j <- 0..2 seperately, with 3 channels dr0, dr1 and
    * dr2 instead of one channel bundle dr.j.
    */
```

```
:: atomic{ dr0 ! x , ( ( ( ( ! b ) && ( lxr[y] == true ) ) &&
(lxr[z] == true ) ) && ( lxr[p] == true ) )
; lxr[y] = false
; lxr[z] = false
; lxr[p] = false
; b = true
; lxr[0] = true }
:: atomic{ dr1 ! x , ( ( ( ( ! b ) && ( lxr[y] == true ) ) &&
( lxr[z] == true ) ) && ( lxr[p] == true ) )
; lxr[y] = false
; lxr[z] = false
; lxr[p] = false
; b = true
; lxr[0] = true }
:: atomic{ dr2 ! x , ( ( ( ( ! b ) && ( lxr[y] == true ) ) &&
( lxr[z] == true ) ) && ( lxr[p] == true ) )
; lxr[y] = false
; lxr[z] = false
; lxr[p] = false
; b = true
; lxr[0] = true }
:: rd0 ? q , eval ( 2 - true );
if
    ::  d_step{q == 1 ; y = 1 ; z = 9 ; p = 5}
    ::  d_step{q == 2 ; y = 2 ; z = 9  ; p = 5}
    ::  d_step{q == 3 ; y = 3 ; z = 9  ; p = 5 }
    ::  d_step{q == 4 ; y = 1 ; z = 10 ; p = 13 }
    ::  d_step{q == 5 ; y = 2 ; z = 10 ; p = 13 }
    ::  d_step{q == 6 ; y = 3 ; z = 10 ; p = 13}
    ::  d_step{q == 7 ; y = 6 ; z = 5  ; p = 12 }
    ::  d_step{q == 8 ; y = 7 ; z = 5  ; p = 12 }
    ::  d_step{q == 9 ; y = 8 ; z = 5  ; p = 12 }
    ::  d_step{q == 10 ; y = 6 ; z = 11  ; p = 13 }
    ::  d_step{q == 11 ; y = 7 ; z = 11  ; p = 13 }
    ::  d_step{q == 12 ; y = 8 ; z = 11  ; p = 13 }
    ::  d_step{q == 13 ; y = 1 ; z = 9  ; p = 0 }
    ::  d_step{q == 14 ; y = 2 ; z = 9  ; p = 0 }
    ::  d_step{q == 15 ; y = 3 ; z = 9  ; p = 0 }
    ::  d_step{q == 16 ; y = 1 ; z = 10  ; p = 0 }
    ::  d_step{q == 17 ; y = 2 ; z = 10  ; p = 0 }
    ::  d_step{q == 18 ; y = 3 ; z = 10  ; p = 0 }
    ::  d_step{q == 19 ; y = 6 ; z = 11  ; p = 13 }
    ::  d_step{q == 20 ; y = 7 ; z = 11  ; p = 13 }
    ::  d_step{q == 21 ; y = 8 ; z = 11  ; p = 13 }
    ::  d_step{q == 22 ; y = 5 ; z = 12  ; p = 0 }
    ::  d_step{q == 23 ; y = 4 ; z = 9  ; p = 5 }
    ::  d_step{q == 24 ; y = 6 ; z = 4  ; p = 12 }
```

```
        ::  d_step{q == 25 ; y = 7 ; z = 4  ; p = 12 }
        ::  d_step{q == 26 ; y = 8 ; z = 4  ; p = 12 }
        ::  d_step{q == 27 ; y = 4 ; z = 9  ; p = 0 }
        ::  d_step{q == 28 ; y = 4 ; z = 12  ; p = 0 }
        ::  d_step{q == 29 ; y = 4 ; z = 10  ; p = 13 }
        ::  d_step{q == 30 ; y = 4 ; z = 11  ; p = 13 }
    fi
    ; lxr[y] = true
    ; lxr[z] = true
    ; lxr[p] = true
    :: rd1 ? q , eval ( 2 - true ) ;
    if
        ::  d_step{q == 1 ; y = 1 ; z = 9 ; p = 5}
        ::  d_step{q == 2 ; y = 2 ; z = 9  ; p = 5}
        ::  d_step{q == 3 ; y = 3 ; z = 9  ; p = 5 }
        ::  d_step{q == 4 ; y = 1 ; z = 10 ; p = 13 }
        ::  d_step{q == 5 ; y = 2 ; z = 10 ; p = 13 }
        ::  d_step{q == 6 ; y = 3 ; z = 10 ; p = 13}
        ::  d_step{q == 7 ; y = 6 ; z = 5  ; p = 12 }
        ::  d_step{q == 8 ; y = 7 ; z = 5  ; p = 12 }
        ::  d_step{q == 9 ; y = 8 ; z = 5  ; p = 12 }
        ::  d_step{q == 10 ; y = 6 ; z = 11  ; p = 13 }
        ::  d_step{q == 11 ; y = 7 ; z = 11  ; p = 13 }
        ::  d_step{q == 12 ; y = 8 ; z = 11  ; p = 13 }
        ::  d_step{q == 13 ; y = 1 ; z = 9  ; p = 0 }
        ::  d_step{q == 14 ; y = 2 ; z = 9  ; p = 0 }
        ::  d_step{q == 15 ; y = 3 ; z = 9  ; p = 0 }
        ::  d_step{q == 16 ; y = 1 ; z = 10  ; p = 0 }
        ::  d_step{q == 17 ; y = 2 ; z = 10  ; p = 0 }
        ::  d_step{q == 18 ; y = 3 ; z = 10  ; p = 0 }
        ::  d_step{q == 19 ; y = 6 ; z = 11  ; p = 13 }
        ::  d_step{q == 20 ; y = 7 ; z = 11  ; p = 13 }
        ::  d_step{q == 21 ; y = 8 ; z = 11  ; p = 13 }
        ::  d_step{q == 22 ; y = 5 ; z = 12  ; p = 0 }
        ::  d_step{q == 23 ; y = 4 ; z = 9  ; p = 5 }
        ::  d_step{q == 24 ; y = 6 ; z = 4  ; p = 12 }
        ::  d_step{q == 25 ; y = 7 ; z = 4  ; p = 12 }
        ::  d_step{q == 26 ; y = 8 ; z = 4  ; p = 12 }
        ::  d_step{q == 27 ; y = 4 ; z = 9  ; p = 0 }
        ::  d_step{q == 28 ; y = 4 ; z = 12  ; p = 0 }
        ::  d_step{q == 29 ; y = 4 ; z = 10  ; p = 13 }
        ::  d_step{q == 30 ; y = 4 ; z = 11  ; p = 13 }
    fi
    ; lxr[y] = true
    ; lxr[z] = true
    ; lxr[p] = true
    :: rd2 ? q , eval ( 2 - true );
```

```
            if
                ::  d_step{q == 1 ; y = 1 ; z = 9 ; p = 5}
                ::  d_step{q == 2 ; y = 2 ; z = 9  ; p = 5}
                ::  d_step{q == 3 ; y = 3 ; z = 9  ; p = 5 }
                ::  d_step{q == 4 ; y = 1 ; z = 10 ; p = 13 }
                ::  d_step{q == 5 ; y = 2 ; z = 10 ; p = 13 }
                ::  d_step{q == 6 ; y = 3 ; z = 10 ; p = 13}
                ::  d_step{q == 7 ; y = 6 ; z = 5  ; p = 12 }
                ::  d_step{q == 8 ; y = 7 ; z = 5  ; p = 12 }
                ::  d_step{q == 9 ; y = 8 ; z = 5  ; p = 12 }
                ::  d_step{q == 10 ; y = 6 ; z = 11  ; p = 13 }
                ::  d_step{q == 11 ; y = 7 ; z = 11  ; p = 13 }
                ::  d_step{q == 12 ; y = 8 ; z = 11  ; p = 13 }
                ::  d_step{q == 13 ; y = 1 ; z = 9  ; p = 0 }
                ::  d_step{q == 14 ; y = 2 ; z = 9  ; p = 0 }
                ::  d_step{q == 15 ; y = 3 ; z = 9  ; p = 0 }
                ::  d_step{q == 16 ; y = 1 ; z = 10  ; p = 0 }
                ::  d_step{q == 17 ; y = 2 ; z = 10  ; p = 0 }
                ::  d_step{q == 18 ; y = 3 ; z = 10  ; p = 0 }
                ::  d_step{q == 19 ; y = 6 ; z = 11  ; p = 13 }
                ::  d_step{q == 20 ; y = 7 ; z = 11  ; p = 13 }
                ::  d_step{q == 21 ; y = 8 ; z = 11  ; p = 13 }
                ::  d_step{q == 22 ; y = 5 ; z = 12  ; p = 0 }
                ::  d_step{q == 23 ; y = 4 ; z = 9  ; p = 5 }
                ::  d_step{q == 24 ; y = 6 ; z = 4  ; p = 12 }
                ::  d_step{q == 25 ; y = 7 ; z = 4  ; p = 12 }
                ::  d_step{q == 26 ; y = 8 ; z = 4  ; p = 12 }
                ::  d_step{q == 27 ; y = 4 ; z = 9  ; p = 0 }
                ::  d_step{q == 28 ; y = 4 ; z = 12  ; p = 0 }
                ::  d_step{q == 29 ; y = 4 ; z = 10  ; p = 13 }
                ::  d_step{q == 30 ; y = 4 ; z = 11  ; p = 13 }
            fi
            ; lxr[y] = true
            ; lxr[z] = true
            ; lxr[p] = true od }

proctype RC (  chan dr, rd, rb ,rf, rtt , rv0, rv1 ,rp0, rp1 , br ,
fr , ttr , vr , pr ) {
       byte xt;
      bool z;
      turnt[0] = 0 ; turnt[1] = 0; turnt[2] = 0; turnt[3] = 0; turnt[4] = 0;
      turnt[5] = 0; turnt[6] = 0; turnt[7] = 0; turnt[8] = 0; turnt[9] = 0;
      turnt[10] = 0 ; turnt[11] = 0
       do
     :: dr ? xt , eval ( 2 - true ) ;
     if
```

```
:: xt == 1 ->order[1] = true; rp0!1,true; rp1!1,true ;
pr?z, eval(2 - true) ; rv0!9,true ; rv1!1,true ;
vr?z, eval(2 - true) ; rv0!4,true ; rv1!1,true ;
vr?z, eval(2 - true) ; ptt[2] = 1;
if
:: mix == 0 -> mix = 1
:: mix == 1 -> mix = 1
:: mix == 2 -> mix = 4
:: mix == 3 -> mix = 5
:: mix == 4 -> mix = 4
:: mix == 5 -> mix = 5
:: mix == 6 -> mix = 7
:: mix == 7 -> mix = 7
fi ;
rv0!4,true ;rv1!0,true ;vr?z, eval(2 - true) ; rv0!9,true ; rv1!0,true
; vr?z, eval(2 - true); rp0!1,true ; rp1!0,true ; pr?z, eval(2 - true)
; order[1] = false

:: xt == 2 ->order[2] = true; rp0!2,true; rp1!1,true ;
pr?z, eval(2 - true) ; rv0!9,true; rv1!1,true;
vr?z, eval(2 - true) ; rv0!5,true ; rv1!1,true ;
vr?z, eval(2 - true) ; ptt[2] = 2;
if
:: mix == 0 -> mix = 2
:: mix == 1 -> mix = 4
:: mix == 2 -> mix = 2
:: mix == 3 -> mix = 6
:: mix == 4 -> mix = 4
:: mix == 5 -> mix = 7
:: mix == 6 -> mix = 6
:: mix == 7 -> mix = 7
fi ;
rv0!5,true ;rv1!0,true ;vr?z, eval(2 - true) ; rv0!9,true ; rv1!0,true
; vr?z, eval(2 - true); rp0!2,true ; rp1!0,true ; pr?z, eval(2 -
true) ; order[2] = false

:: xt == 3 ->order[3] = true; rp0!3,true; rp1!1,true ;
pr?z, eval(2 - true) ; rv0!9,true ; rv1!1,true ;
vr?z, eval(2 - true) ; rv0!6,true ; rv1!1,true ;
vr?z, eval(2 - true) ; ptt[2] = 3;
if
:: mix == 0 -> mix = 3
:: mix == 1 -> mix = 5
:: mix == 2 -> mix = 6
:: mix == 3 -> mix = 3
:: mix == 4 -> mix = 7
:: mix == 5 -> mix = 5
```

```
:: mix == 6 -> mix = 6
:: mix == 7 -> mix = 7
fi ;
rv0!6,true ;rv1!0,true ;vr?z, eval(2 - true) ; rv0!9,true ; rv1!0,true
; vr?z, eval(2 - true); rp0!3,true ; rp1!0,true ; pr?z, eval(2 -
true) ; order[3] = false

:: xt == 4 -> order[4] = true; rf!0,true ; rtt!1,true;
fr?z, eval(2 - true); ttr?z, eval(2 - true); rp0!1,true;
rp1!1,true; pr?z, eval(2 - true); rv0!1,true; rv1!1,true;
vr?z, eval(2 - true); rv0!18,true; rv1!1,true; vr?z, eval(2 - true);
rv0!17,true; rv1!1,true; vr?z, eval(2 -true);
       if
       :: turnt[1] == false -> turnt[1] = true
       :: turnt[1] == true -> turnt[1] = false
       fi ; ptt[1] = 1;
rv0!17,true; rv1!0,true; vr?z, eval(2 - true); rv0!18,true; rv1!0,true;
vr?z, eval(2 - true); rv0!1,true; rv1!0,true; vr?z, eval(2 - true);
rp0!1,true; rp1!0,true; pr?z, eval(2 - true); order[4] = false

:: xt == 5 -> order[5] = true; rf!0,true ; rtt!1,true;
fr?z, eval(2 - true); ttr?z, eval(2 - true); rp0!2,true;
rp1!1,true; pr?z, eval(2 - true); rv0!2,true;rv1!1,true;
vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true); rv0!17,true; rv1!1,true; vr?z, eval(2 - true);
       if
       :: turnt[1] == false -> turnt[1] = true
       :: turnt[1] == true -> turnt[1] = false
       fi ; ptt[1] = 2;
rv0!17,true; rv1!0,true; vr?z, eval(2 - true) ; rv0!18,true; rv1!0,true;
vr?z, eval(2 - true); rv0!2,true; rv1!0,true; vr?z, eval(2 - true);
rp0!2,true; rp1!0,true; pr?z, eval(2 - true); order[5] = false

:: xt == 6 -> order[6] = true; rf!0,true ; rtt!1,true;
fr?z, eval(2 - true); ttr?z, eval(2 - true); rp0!3,true;
rp1!1,true; pr?z, eval(2 - true); rv0!3,true;rv1!1,true;
vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true); rv0!17,true; rv1!1,true; vr?z, eval(2 - true);
       if
       :: turnt[1] == false -> turnt[1] = true
       :: turnt[1] == true -> turnt[1] = false
       fi ; ptt[1] = 3;
rv0!17,true; rv1!0,true; vr?z, eval(2 - true) ; rv0!18,true; rv1!0,true;
vr?z, eval(2 - true); rv0!3,true; rv1!0,true; vr?z, eval(2 - true);
rp0!3,true; rp1!0,true; pr?z, eval(2 - true); order[6] = true

:: xt == 7 -> order[7] = true; rb!1,true ; br?z, eval(2 - true);
```

```
rp0!5,true; rp1!1,true; pr?z, eval(2 - true); rv0!10,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true); ptt[3] = mix; buf1 = mix; rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); rv0!4,true; rv1!0,true; vr?z, eval(2 - true);
rp0!4,true; rp1!0,true; pr?z, eval(2 - true); order[7] = false

:: xt == 8 -> order[8] = true; rb!2,true ; br?z, eval(2 - true);
rp0!5,true; rp1!1,true; pr?z, eval(2 - true); rv0!10,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true); ptt[3] = mix; buf2 = mix; rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); rv0!4,true; rv1!0,true; vr?z, eval(2 - true);
rp0!4,true; rp1!0,true; pr?z, eval(2 - true); order[9] = false

:: xt == 9 -> order[9] = true; rb!3,true ; br?z, eval(2 - true);
rp0!5,true; rp1!1,true; pr?z, eval(2 - true); rv0!10,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true);  ptt[3] = mix; buf3 = mix; rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); rv0!4,true; rv1!0,true; vr?z, eval(2 - true);
rp0!4,true; rp1!0,true; pr?z, eval(2 - true); order[9] = false

 :: xt == 10 -> order[10] = true; rf!1,true ; rtt!2,true ;
 fr?z, eval(2 - true); ttr?z, eval(2 - true); rp0!6,true;
 rp1!1,true; pr?z, eval(2 - true); rv0!14,true; rv1!1,true;
 vr?z, eval(2 - true); rv0!18,true;rv1!1,true;
 vr?z, eval(2 - true);
       if
       :: turnt[2] == false -> turnt[2] = true
       :: turnt[2] == true -> turnt[2] = false
       fi ; ptt[4] = buf1 ;
  rv0!18,true; rv1!0,true;
 vr?z, eval(2 - true); rv0!14,true; rv1!0,true;
 vr?z, eval(2 - true); rp0!6,true; rp1!0,true;
 pr?z, eval(2 - true); order[10] = false

 :: xt == 11 -> order[11] = true; rf!1,true ; rtt!2,true ;
 fr?z, eval(2 - true); ttr?z, eval(2 - true); rp0!7,true;
 rp1!1,true; pr?z, eval(2 - true); rv0!15,true; rv1!1,true;
 vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
 vr?z, eval(2 - true);
       if
       :: turnt[2] == false -> turnt[2] = true
       :: turnt[2] == true -> turnt[2] = false
       fi ; ptt[4] = buf2 ;
 rv0!18,true; rv1!0,true;
 vr?z, eval(2 - true); rv0!15,true; rv1!0,true;
 vr?z, eval(2 - true); rp0!7,true; rp1!0,true;
 pr?z, eval(2 - true); order[11] = false
```

```
 :: xt == 12 -> order[12] = true; rf!1,true ; rtt!2,true ;
 fr?z, eval(2 - true); ttr?z, eval(2 - true); rp0!8,true;
 rp1!1,true; pr?z, eval(2 - true); rv0!16,true; rv1!1,true;
 vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
 vr?z, eval(2 - true);
       if
       :: turnt[2] == false -> turnt[2] = true
       :: turnt[2] == true -> turnt[2] = false
       fi ; ptt[4] = buf3 ;
 rv0!18,true; rv1!0,true;
 vr?z, eval(2 - true); rv0!16,true; rv1!0,true;
 vr?z, eval(2 - true); rp0!8,true; rp1!0,true;
 pr?z, eval(2 - true); order[12] =
 false

:: xt == 13 -> order[13] = true; rp0!1,true; rp1!1,true;
pr?z, eval(2 - true); rv0!4,true; rv1!1,true;
vr?z, eval(2 - true); ptt[2] = 1; rv0!4,true;
rv1!0,true; vr?z, eval(2 - true); rp0!1,true; rp1!0,true;
pr?z, eval(2 - true); order[13] = false

:: xt == 14 -> order[14] = true; rp0!2,true; rp1!1,true;
pr?z, eval(2 - true); rv0!5,true; rv1!1,true;
vr?z, eval(2 - true); ptt[2] = 2;  rv0!5,true;
rv1!0,true; vr?z, eval(2 - true); rp0!2,true; rp1!0,true;
pr?z, eval(2 - true); order[14] = false

:: xt == 15 -> order[15] = true; rp0!3,true; rp1!1,true;
pr?z, eval(2 - true); rv0!6,true; rv1!1,true;
vr?z, eval(2 - true); ptt[2] = 3; rv0!6,true;
rv1!0,true; vr?z, eval(2 - true); rp0!3,true; rp1!0,true;
pr?z, eval(2 - true); order[15] = false

:: xt == 16 -> order[16] = true; rf!0,true;
fr?z, eval(2 - true); rp0!1,true; rp1!1,true;
pr?z, eval(2 - true); rv0!1,true; rv1!1,true;
vr?z, eval(2 - true); rv0!17,true; rv1!1,true;
vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[1] = 1;
rv0!18,true; rv1!0,true; vr?z, eval(2 - true); rv0!17,true;
rv1!0,true;vr?z, eval(2 - true) ; rv0!1,true; rv1!0,true;
vr?z, eval(2 - true); rp0!1,true; rp1!0,true; pr?z, eval(2 -
true); order[16] = false

:: xt == 17 -> order[17] = true; rf!0,true; fr?z, eval(2 - true);
```

```
rp0!2,true; rp1!1,true; pr?z, eval(2 - true); rv0!2,true;
rv1!1,true; vr?z, eval(2 - true); rv0!17,true; rv1!1,true;
vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[1] = 2;
rv0!18,true; rv1!0,true; vr?z, eval(2 - true); rv0!17,true;
rv1!0,true;vr?z, eval(2 - true) ; rv0!2,true; rv1!0,true;
vr?z, eval(2 - true); rp0!2,true; rp1!0,true; pr?z, eval(2 -
true); order[17] = false

:: xt == 18 -> order[18] = true; rf!0,true; fr?z, eval(2 - true);
rp0!3,true; rp1!1,true; pr?z, eval(2 - true); rv0!3,true;
rv1!1,true; vr?z, eval(2 - true); rv0!17,true; rv1!1,true;
vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[1] = 3;
rv0!18,true; rv1!0,true; vr?z, eval(2 - true); rv0!17,true;
rv1!0,true;vr?z, eval(2 - true) ; rv0!3,true; rv1!0,true;
vr?z, eval(2 - true); rp0!3,true; rp1!0,true; pr?z, eval(2 -
true); order[18] = false

:: xt == 19 -> order[19] = true; rf!0,true; fr?z, eval(2 - true);
rp0!6,true; rp1!1,true; pr?z, eval(2 - true); rv0!14,true;
rv1!1,true; vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[4] = buf1;
rv0!18,true; rv1!0,true; vr?z, eval(2 - true); rv0!14,true;
rv1!0,true;vr?z, eval(2 - true); rp0!6,true; rp1!0,true;
pr?z, eval(2 - true); order[19] = false

:: xt == 20 -> order[20] = true; rf!0,true; fr?z, eval(2 - true);
rp0!7,true; rp1!1,true; pr?z, eval(2 - true); rv0!15,true;
rv1!1,true; vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[4] = buf2;
rv0!18,true; rv1!0,true; vr?z, eval(2 - true); rv0!15,true;
rv1!0,true;vr?z, eval(2 - true); rp0!7,true; rp1!0,true;
pr?z, eval(2 - true); order[20] = false

:: xt == 21 -> order[21] = true; rf!0,true; fr?z, eval(2 - true);
rp0!8,true; rp1!1,true; pr?z, eval(2 - true); rv0!16,true;
rv1!1,true; vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[4] = buf3;
rv0!18,true; rv1!0,true; vr?z, eval(2 - true); rv0!16,true;
rv1!0,true;vr?z, eval(2 - true); rp0!8,true; rp1!0,true;
pr?z, eval(2 - true); order[21] = false
```

```
:: xt == 22 -> order[22] = true; rb!0,true; br?z, eval(2 - true);
rp0!5,true; rp1!1,true; pr?z, eval(2 - true); rv0!10,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true); rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); ptt[3] = mix; rv0!10,true; rv1!0,true;
vr?z, eval(2 - true); rp0!5,true; rp1!0,true;
pr?z, eval(2 - true); order[22] = false

:: xt == 23 -> order[23] = true; rp0!4,true; rp1!1,true;
pr?z, eval(2 - true); rv0!9,true; rv1!1,true;
vr?z, eval(2 - true); rv0!8,true; rv1!1,true; vr?z, eval(2 - true);
mix = 0;   rv0!8,true; rv1!0,true;
vr?z, eval(2 - true); rv0!9,true; rv1!0,true; vr?z, eval(2 - true);
rp0!4,true; rp1!0,true; pr?z, eval(2 - true); order[23] = false

:: xt == 24 -> order[24] = true; rb!1,true; br?z, eval(2 - true);
rp0!4,true; rp1!1,true; pr?z, eval(2 - true); rv0!11,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true); rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); rv0!11,true; rv1!0,true;
vr?z, eval(2 - true); rp0!4,true; rp1!0,true;
pr?z, eval(2 - true); order[24] = false

:: xt == 25 -> order[25] = true; rb!2,true; br?z, eval(2 - true);
rp0!4,true; rp1!1,true; pr?z, eval(2 - true); rv0!11,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true); rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); rv0!11,true; rv1!0,true;
vr?z, eval(2 - true); rp0!4,true; rp1!0,true;
pr?z, eval(2 - true); order[25] = false

:: xt == 26 -> order[26] = true; rb!3,true; br?z, eval(2 - true);
rp0!4,true; rp1!1,true; pr?z, eval(2 - true); rv0!11,true;
rv1!1,true; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
vr?z, eval(2 - true); rv0!12,true; rv1!0,true;
vr?z, eval(2 - true); rv0!11,true; rv1!0,true;
vr?z, eval(2 - true); rp0!4,true; rp1!0,true;
pr?z, eval(2 - true); order[26] = false

:: xt == 27 -> order[27] = true; rp0!4,true; rp1!1,true;
pr?z, eval(2 - true); rv0!8,true; rv1!1,true;
vr?z, eval(2 - true);
ptt[2] = 0; rv0!8,true;
rv1!0,true; vr?z, eval(2 - true); rp0!4,true; rp1!0,true;
pr?z, eval(2 - true); order[27] = false
```

```
     :: xt == 28 -> order[28] = true; rb!0,true; br?z, eval(2 - true);
     rp0!4,true; rp1!1,true; pr?z, eval(2 - true); rv0!11,true;
     rv1!1,true ; vr?z, eval(2 - true); rv0!12,true; rv1!1,true;
     vr?z, eval(2 - true); rv0!12,true; rv1!0,true;
     vr?z, eval(2 - true); rv0!11,true; rv1!0,true;
     vr?z, eval(2 - true); rp0!4,true; rp1!0,true;
     pr?z, eval(2 - true); order[28] = false

     :: xt == 29 -> order[29] = true; rf!0,true; fr?z, eval(2 - true);
     rp0!4,true; rp1!1,true; pr?z, eval(2 - true); rv0!7,true;
     rv1!1,true; vr?z, eval(2 - true); rv0!17,true; rv1!1,true;
     vr?z, eval(2 - true); rv0!18,true; rv1!1,true;
     vr?z, eval(2 - true);
     ptt[1] = 0;
     rv0!18,true;
     rv1!1,true; vr?z, eval(2 - true); rv0!17,true; rv1!1,true;
     vr?z, eval(2 - true) ; rv0!7,true; rv1!1,true;
     vr?z, eval(2 - true); rp0!4,true; rp1!0,true;
     pr?z, eval(2 - true); order[29] = false

     :: xt == 30 -> order[30] = true; rf!0,true; fr?z, eval(2 - true);
     rp0!4,true; rv1!1,true; pr?z, eval(2 - true); rv0!13,true;
     rv1!1,true; vr?z, eval(2 - true); rv0!16,true; rv1!1,true;
     vr?z, eval(2 - true);
     ptt[4] = 0;
     rv0!16,true; rv1!1,true; vr?z, eval(2 - true); rv0!13,true;
     rv1!1,true;vr?z, eval(2 - true)  ; rp0!4,true; rp1!0,true;
     pr?z, eval(2 - true); order[30] = false fi

    ; rd ! xt , true od
}

proctype BMC (  chan rb0 , rb1 , rb2 , br0, br1, br2, cbd, cbm, st0,
st1, st2, st3 ) {
        byte m
     ;  bool y
     ; m = 0
     ; do
        :: rb0 ? posbk , eval ( 2 - true )
        ; if
            :: ( posbk == m )
            :: cbd ! ( posbk > m ) , ( posbk != m )
            ; cbm ! true , true
            ; if
                :: (posbk == 0); st0 ? y , eval ( 2 - true )
                :: (posbk == 1); st1 ? y , eval ( 2 - true)
                :: (posbk == 2); st2 ? y , eval ( 2 - true )
```

```
                :: (posbk == 3); st3 ? y , eval ( 2 - true )
              fi
            ; cbm ! false , true
            ; m = posbk
          fi
        ; br0 ! true , true
        :: rb1 ? posbk , eval ( 2 - true )
        ; if
            :: ( posbk == m )
            :: cbd ! ( posbk > m ) , ( posbk != m )
            ; cbm ! true , true
            ; if
                :: (posbk == 0) ; st0 ? y , eval ( 2 - true )
                :: (posbk == 1) ; st1 ? y , eval ( 2 - true)
                :: (posbk == 2) ; st2 ? y , eval ( 2 - true )
                :: (posbk == 3) ; st3 ? y , eval ( 2 - true )
              fi
            ; cbm ! false , true
            ; m = posbk
          fi
        ; br1 ! true, true
        :: rb2 ? posbk , eval ( 2 - true )
        ; if
            :: ( posbk == m )
            :: cbd ! ( posbk > m ) , ( posbk != m )
            ; cbm ! true , true
            ; if
                :: (posbk == 0); st0 ? y , eval ( 2 - true )
                :: (posbk == 1); st1 ? y , eval ( 2 - true)
                :: (posbk == 2); st2 ? y , eval ( 2 - true )
                :: (posbk == 3); st3 ? y , eval ( 2 - true )
              fi
            ; cbm ! false , true
            ; m = posbk
          fi
        ; br2 ! true, true
     od }

proctype BM (  chan cbd, cbm , bs0 , bs1 , bs2 , bs3)
    { timer  t
    ; bool d
    ; posbq = 0
    ; bs0 ! true , true
    ; bs1 ! false , true
    ; bs2 ! false, true
    ; bs3 ! false , true
    ; bm = false
```

```
    ; do
      :: set(t,2)
         ; if
            ::  expire(t) && (bm == true)
                ;  if
                     :: d_step { d ; posbq = ( posbq + 1 ) }
                     :: d_step{ ( ! d ) ; posbq = ( posbq - 1 ) }
                   fi
                ; bs0 ! (posbq == 0 ) , true
                ; bs1 ! (posbq == 20), true
                ; bs2 ! (posbq == 40), true
                ; bs3 ! (posbq == 60) , true
            :: cbd ? d , eval ( 2 - true )
            :: cbm ? bm , eval ( 2 - true )
          fi
      od }


proctype FMC ( chan rf0, rf1 , rf2, fr0, fr1, fr2, cfd, cfm, st0,
st1 )
    {  byte m
    ;  bool y
    ; m = 0
    ; do
       :: rf0 ? posfk , eval ( 2 - true )
          ; if
             :: ( posfk == m )
             :: cfd ! ( posfk > m ) , ( posfk != m )
              ; cfm ! true , true
              ; if
                 :: ( posfk == 0) ; st0 ? y , eval ( 2 - true )
                 :: (posfk == 1)  ; st1 ? y , eval ( 2 - true )
                fi
              ; cfm ! false , true
              ; m = posfk
            fi
          ; fr0 ! true , true
       :: rf1 ? posfk , eval ( 2 - true )
         ; if
             :: ( posfk == m )
             :: cfd ! ( posfk > m ) , ( posfk != m )
              ; cfm ! true , true
              ; if
                 :: ( posfk == 0) ; st0 ? y , eval ( 2 - true )
                 :: (posfk == 1)  ; st1 ? y , eval ( 2 - true )
                fi
              ; cfm ! false , true
```

```
              ; m = posfk
           fi
         ; fr1 ! true , true
      :: rf2 ? posfk , eval ( 2 - true )
        ; if
            :: ( posfk == m )
            :: cfd ! ( posfk > m ) , ( posfk != m )
             ; cfm ! true , true
             ; if
                :: ( posfk == 0) ; st0 ? y , eval ( 2 - true )
                :: (posfk == 1)  ; st1 ? y , eval ( 2 - true )
               fi
            ; cfm ! false , true
            ; m = posfk
         fi
       ; fr2 ! true , true
    od }


proctype FM (  chan cfd, cfm, fs0, fs1)
    {  bool d
    ; timer t
    ; fm = 0
    ; posfq = 0
    ; fs0 ! true , true
    ; fs1 ! false , true
    ; fm = false
    ; do
       :: set(t,2)
         ; if
             ::  expire(t) && (fm == true)
               ; if
                  :: d_step { d ; posfq = ( posfq + 1 ) }
                  :: d_step { ( ! d ); posfq = ( posfq - 1 ) }
                 fi
               ;  fs0! (posfq == 0), true
               ; fs1! (posfq == 20), true
             :: cfd ? d , eval ( 2 - true )
             :: cfm ? fm , eval ( 2 - true )
           fi
    od }


proctype TMC (  chan rtt0, rtt1, rtt2, ttr0, ttr1, ttr2, ctd, ctm, st )
{       byte p
     ;  byte q
     ;  byte m
```

```
; bool y
; m = 0
; do
   :: rtt0 ? postk , eval ( 2 - true )
     ; p = ( ( ( 12 + postk ) - m )  %  12 )
     ; if
        :: ( p == 0 )
        :: ctd ! ( p <= 6 ) , ( p != 0 )
          ; if
             :: (p < 12 - p) ; q = p
             :: (p >= 12 - p) ; q = 12 - p
            fi
          ; ctm ! true , true
          ; do
             :: ( q > 0 ); st ? y , eval ( 2 - true )
               ; q = ( q - 1 )
             :: ! ( ( q > 0 ) ) -> break
            od
          ; ctm ! false , true
          ; m = postk
       fi
     ; ttr0 ! true , true
   :: rtt1 ? postk , eval ( 2 - true )
     ; p = ( ( ( 12 + postk ) - m )  %  12 )
     ; if
        :: ( p == 0 )
        :: ctd ! ( p <= 6 ) , ( p != 0 )
          ; if
             :: (p < 12 - p) ; q = p
             :: (p >= 12 - p) ; q = 12 - p
            fi
          ; ctm ! true , true
          ; do
             :: ( q > 0 ); st ? y , eval ( 2 - true )
               ; q = ( q - 1 )
             :: ! ( ( q > 0 ) ) od
          ; ctm ! false , true
          ; m = postk
       fi
     ; ttr1 ! true , true
   :: rtt2 ? postk , eval ( 2 - true )
     ; p = ( ( ( 12 + postk ) - m ) % 12 )
     ; if
       :: ( p == 0 )
       :: ctd ! ( p <= 6 ) , ( p != 0 )
         ; if
             :: (p < 12 - p) ; q = p
```

```
                :: (p >= 12 - p) ; q = 12 - p
              fi
            ; ctm ! true , true
            ; do
               :: ( q > 0 ); st ? y , eval ( 2 - true )
                 ; q = ( q - 1 )
               :: ! ( ( q > 0 ) )
              od
            ; ctm ! false , true
            ; m = postk
          fi
        ; ttr2 ! true , true
      od }



proctype TM (  chan ctd, ctm , ts)
    {  bool d
    ; timer t
    ; postq = 0
    ; ts ! true , true
    ; tm = false
    ; do
       :: set(t,2)
         ; if
            ::  expire(t) && (tm == true)
             ; if
                :: d_step { d ; postq = ( ( postq + 1 ) %  120 ) }
                :: d_step { ( ! d ) ; postq = ( ( postq + 119 ) %  120 ) }
               fi
             ; ts!(postq % 10 == 0) , true
            :: ctd ? d , eval ( 2 - true )
            :: ctm ? tm , eval ( 2 - true )
          fi
      od }


proctype VC (  chan rv00, rv01, rv10, rv11, rv20, rv21 , vr0 , vr1 ,
vr2 , cv )
    { byte x
    ; byte y
    ; do
        :: rv00 ? x , eval ( 2 - true ) ; rv01 ? y , eval (2 - true)
          ; cv ! y , true
          ; vr0 ! true , true
        :: rv10 ? x , eval ( 2 - true ) ; rv11 ? y, eval ( 2 - true )
          ; cv ! y , true
```

```
          ; vr1 ! true , true
        :: rv20 ? x , eval ( 2 - true ) ; rv21 ? y , eval ( 2 - true )
          ; cv ! y , true
          ; vr2 ! true , true
      od }

proctype PC (  chan rp00, rp01 , rp10, rp11 , rp20, rp21 , pr0 , pr1
, pr2 , cp )
    { byte x
    ; byte y
    ; do
        :: rp00 ? x , eval ( 2 - true ) ; rp01 ? y , eval (2 - true)
          ; cp ! y , true
          ; pr0 ! true , true
        :: rp10 ? x , eval ( 2 - true ) ; rp11 ? y, eval ( 2 - true )
          ; cp ! y , true
          ; pr1 ! true , true
        :: rp20 ? x , eval ( 2 - true ) ; rp21 ? y , eval ( 2 - true )
          ; cp ! y , true
          ; pr2 ! true , true
      od }

proctype A (  chan ca )
    {  byte k
    ; do
       :: ca ? k , eval ( 2 - true )
      od }

proctype S (  chan senin , st )
    {  bool b
    ; if
        :: senin ? b , eval ( 2 - true )
      fi
    ; do
        :: senin ? b , eval ( 2 - true )
        :: st ! true , (b == true)
      od }
```

C Adapted χ 1.0 specification

```
from standardlib import *

// There are 30 possible orders. The orders are numbered 1 to 30 and are
// the following:
//1. send paint from the primary color 1 to the mixer
//2. send paint from the primary color 2 to the mixer
//3. send paint from the primary color 3 to the mixer
//4. send paint from the primary color 1 directly to the filler
//5. send paint from the primary color 2 directly to the filler
//6. send paint from the primary color 3 directly to the filler
//7. send paint from the mixer to buffer 1
//8. send paint from the mixer to buffer 2
//9. send paint from the mixer to buffer 3
//10. send paint from buffer 1 to the filler
//11. send paint from buffer 2 to the filler
//12. send paint from buffer 3 to the filler
//13. empty the track from primary color 1 and send the waste to the waste
//    vessel next to the mixer
//14. empty the track from primary color 2 and send the waste to the waste
//    vessel next to the mixer
//15. empty the track from primary color 3 and send the waste to the waste
//    vessel next to the mixer
//16. empty the track from primary color 1 and send the waste to the waste
//    vessel next to the filler
//17. empty the track from primary color 2 and send the waste to the waste
//    vessel next to the filler
//18. empty the track from primary color 3 and send the waste to the waste
//    vessel next to the filler
//19. empty the track to buffer 1
//20. empty the track to buffer 2
//21. empty the track to buffer 3
//22. empty the mixer
//23. clean the mixer
//24. clean buffer 1
//25. clean buffer 2
//26. clean buffer 3
//27. clean the track to the mixer
//28. clean the track to the buffers
//29. clean the track from the primary colors to the filler
//30. clean the track from the buffers to the filler

//Process G generates a number
// between 1 and 30 and sends it over channel gd to process DP.
 proc G(chan gd!: nat) = |[ var u: -> nat
```

```
= uniform(1,30):: (gd!(sample u)) ]|

//Process DP checks whether needed resources are available and
//assignes these to the
//order. There are 13 possible resources which are numbered the
//following way:
//1. vessel with primary color 1
//2. vessel with primary color 2
//3. vessel with primary color 3
//4. vessel with cleaning liquid
//5. mixing vessel
//6. buffer vessel 1
//7. buffer vessel 2
//8. buffer vessel 3
//9. pipes from valve matrix 2 to the mixing vessel
//10. pipes from valve matrix 1 to valve matrix 3
//11. pipe within valve matrix 3
//12. pipes from valve matrix 1 to the buffer
//13. pipes from valve matrix 3 to the filler

proc DP(chan gd?: (nat), dr!: 3 # (nat), rd?: 3 # nat) = |[ var x:
nat, q: nat, b: bool = false, xr: {nat}
 = { 1,2,3,4,5,6,7,8,9,10,11,12,13}, y: {nat} , z: nat = 0, p: nat = 0 ::
   gd?x  // DP receives the order number from process G
; !!time, "\t Order ", x, " received \n" ; (x = 1  -> y := {1,9,5}
  | x = 2  -> y := {2,9,5}
  | x = 3  -> y := {3,9,5}
  | x = 4  -> y := {1,10,13}
  | x = 5  -> y := {2,10,13}
  | x = 6  -> y := {3,10,13}
  | x = 7  -> y := {6,5,12}
  | x = 8  -> y := {7,5,12}
  | x = 9  -> y := {8,5,12}
  | x = 10 -> y := {6,11,13}
  | x = 11 -> y := {7,11,13}
  | x = 12 -> y := {8,11,13}
  | x = 13 -> y := {1,9}
  | x = 14 -> y := {2,9}
  | x = 15 -> y := {3,9}
  | x = 16 -> y := {1,10}
  | x = 17 -> y := {2,10}
  | x = 18 -> y := {3,10}
  | x = 19 -> y := {6,11,13}
  | x = 20 -> y := {7,11,13}
  | x = 21 -> y := {8,11,13}
  | x = 22 -> y := {5,12}
  | x = 23 -> y := {4,9,5}
```

```
  | x = 24 -> y := {6,4,12}
  | x = 25 -> y := {7,4,12}
  | x = 26 -> y := {8,4,12}
  | x = 27 -> y := {4,9}
  | x = 28 -> y := {4,12}
  | x = 29 -> y := {4,10,13}
  | x = 30 -> y := {4,11,13})




  ; *(b -> gd?x; b:= false
; !!time, "\t Order ", x, " received \n" ; (x = 1  -> y := {1,9,5}
  | x = 2  -> y := {2,9,5}
  | x = 3  -> y := {3,9,5}
  | x = 4  -> y := {1,10,13}
  | x = 5  -> y := {2,10,13}
  | x = 6  -> y := {3,10,13}
  | x = 7  -> y := {6,5,12}
  | x = 8  -> y := {7,5,12}
  | x = 9  -> y := {8,5,12}
  | x = 10 -> y := {6,11,13}
  | x = 11 -> y := {7,11,13}
  | x = 12 -> y := {8,11,13}
  | x = 13 -> y := {1,9}
  | x = 14 -> y := {2,9}
  | x = 15 -> y := {3,9}
  | x = 16 -> y := {1,10}
  | x = 17 -> y := {2,10}
  | x = 18 -> y := {3,10}
  | x = 19 -> y := {6,11,13}
  | x = 20 -> y := {7,11,13}
  | x = 21 -> y := {8,11,13}
  | x = 22 -> y := {5,12}
  | x = 23 -> y := {4,9,5}
  | x = 24 -> y := {6,4,12}
  | x = 25 -> y := {7,4,12}
  | x = 26 -> y := {8,4,12}
  | x = 27 -> y := {4,9}
  | x = 28 -> y := {4,12}
  | x = 29 -> y := {4,10,13}
  | x = 30 -> y := {4,11,13})

  // Each order has a predefined set of needed resources.
  // The numbers of the needed resources is assigned to variable y.

    | ( | , j <- 0..2 , (not b and (y sub xr)) ->
```

```
    dr.j!x ; xr:= xr - y ; b:= true)
// The set of available resources is stored in variable xr. The process
// checks whether the needed resources are still available. If this is the
// case, the order number is sent over channel dr to process RC and the
// resources are subtracted from the set.
    | ( | , j <- 0..2 , (rd.j?q;
  (q = 1  -> y := {1,9,5}
  | q = 2  -> y := {2,9,5}
  | q = 3  -> y := {3,9,5}
  | q = 4  -> y := {1,10,13}
  | q = 5  -> y := {2,10,13}
  | q = 6  -> y := {3,10,13}
  | q = 7  -> y := {6,5,12}
  | q = 8  -> y := {7,5,12}
  | q = 9  -> y := {8,5,12}
  | q = 10 -> y := {6,11,13}
  | q = 11 -> y := {7,11,13}
  | q = 12 -> y := {8,11,13}
  | q = 13 -> y := {1,9,0}
  | q = 14 -> y := {2,9,0}
  | q = 15 -> y := {3,9,0}
  | q = 16 -> y := {1,10,0}
  | q = 17 -> y := {2,10,0}
  | q = 18 -> y := {3,10,0}
  | q = 19 -> y := {6,11,13}
  | q = 20 -> y := {7,11,13}
  | q = 21 -> y := {8,11,13}
  | q = 22 -> y := {5,12,0}
  | q = 23 -> y := {4,9,5}
  | q = 24 -> y := {6,4,12}
  | q = 25 -> y := {7,4,12}
  | q = 26 -> y := {8,4,12}
  | q = 27 -> y := {4,9,0}
  | q = 28 -> y := {4,12,0}
  | q = 29 -> y := {4,10,13}
  | q = 30 -> y := {4,11,13})

  ; xr:= xr + y))
// The receipt of the number q from process RC means that the order with this
// order number is completed. The corresponding resources are added to set xr
// so they can be used for another order.
    )
]|


// Every order has a predefined set of actions. Upon receiving an order
// number, the RC process sends these actions one by one to the
```

```
// corresponding controllers.
proc RC(chan dr?: (nat), rd!: nat, rb!: nat, rf!: nat
        , rtt!: nat, rv!: (nat,bool), rp!: (nat,bool)
        , br?, fr?, ttr?, vr?, pr?: void
        ) =
|[var xt: nat, xs: [(nat,nat,nat)], x: (nat,nat,nat) :: *(dr?xt;
 !!time, "\t Order ", xt, " in process \n"
 ; ( xt = 1  -> rp!(0,true) ; pr? ; rv!(8,true) ; vr? ; rv!(3,true); vr? ;
    delay 45 ; rv!(3,false); vr?; rv!(8,false); vr?; rp!(0,false); pr?
  | xt = 2  ->  rp!(1,true) ; pr? ; rv!(8,true) ; vr? ; rv!(5,true); vr? ;
    delay 45 ; rv!(5,false); vr?; rv!(8,false); vr?; rp!(1,false); pr?
  | xt = 3  -> rp!(2,true) ; pr? ; rv!(8,true) ; vr? ; rv!(7,true); vr? ;
    delay 45 ; rv!(7,false); vr?; rv!(8,false); vr?; rp!(2,false); pr?
  | xt = 4  -> rf!0 ; rtt!1; fr?; ttr?; rp!(0,true); pr?; rv!(2,true); vr?;
    rv!(16,true); vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?
  ; rv!(16,false); vr?; rv!(2,false); vr?; rp!(0,false); pr?
  | xt = 5  -> rf!0 ; rtt!1; fr?; ttr?; rp!(1,true); pr?; rv!(4,true); vr?;
    rv!(16,true); vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?
  ; rv!(16,false); vr?; rv!(4,false); vr?; rp!(1,false); pr?
  | xt = 6  -> rf!0 ; rtt!1; fr?; ttr?; rp!(2,true); pr?; rv!(6,true); vr?;
    rv!(16,true); vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?
  ; rv!(16,false); vr?; rv!(6,false); vr?; rp!(2,false); pr?
  | xt = 7  -> rb!1 ; br?; rp!(4,true); pr?; rv!(9,true); vr?; rv!(11,true);
    vr?;
    delay 45; rv!(11,false); vr?; rv!(9,false); vr?; rp!(4,false); pr?
  | xt = 8  -> rb!2 ; br?; rp!(4,true); pr?; rv!(9,true); vr?; rv!(11,true);
    vr?;
    delay 45; rv!(11,false); vr?; rv!(9,false); vr?; rp!(4,false); pr?
  | xt = 9  -> rb!3 ; br?; rp!(4,true); pr?; rv!(9,true); vr?; rv!(11,true);
    vr?;
    delay 45; rv!(11,false); vr?; rv!(9,false); vr?; rp!(4,false); pr?
  | xt = 10 -> rf!1 ; rtt!2 ; fr?; ttr?; rp!(5,true); pr?; rv!(13,true); vr?;
    rv!(17,true); vr?; delay 45; rv!(17,false); vr?; rv!(13,false); vr?;
    rp!(5,false); pr?
  | xt = 11 -> rf!1 ; rtt!2 ; fr?; ttr?; rp!(6,true); pr?; rv!(14,true); vr?;
    rv!(17,true); vr?; delay 45; rv!(17,false); vr?; rv!(14,false); vr?;
    rp!(6,false); pr?
  | xt = 12 -> rf!1 ; rtt!2 ; fr?; ttr?; rp!(7,true); pr?; rv!(15,true);
    vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?; rv!(15,false);
    vr?; rp!(7,false); pr?
  | xt = 13 -> rp!(0,true); pr?; rv!(3,true); vr?; delay 45; rv!(3,false);
    vr?; rp!(0,false); pr?
  | xt = 14 -> rp!(1,true); pr?; rv!(5,true); vr?; delay 45; rv!(5,false);
    vr?; rp!(1,false); pr?
  | xt = 15 -> rp!(2,true); pr?; rv!(7,true); vr?; delay 45; rv!(7,false);
    vr?; rp!(2,false); pr?
  | xt = 16 -> rf!0; fr?; rp!(0,true); pr?; rv!(2,true); vr?; rv!(16,true);
```

```
  vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?; rv!(16,false);vr?
; rv!(2,false); vr?; rp!(0,false); pr?
| xt = 17 -> rf!0; fr?; rp!(1,true); pr?; rv!(4,true); vr?; rv!(16,true);
  vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?; rv!(16,false);vr?
; rv!(4,false); vr?; rp!(1,false); pr?
| xt = 18 -> rf!0; fr?; rp!(2,true); pr?; rv!(6,true); vr?; rv!(16,true);
  vr?; rv!(17,true); vr?; delay 45; rv!(17,false); vr?; rv!(16,false);vr?
; rv!(6,false); vr?; rp!(2,false); pr?
| xt = 19 -> rf!0; fr?; rp!(5,true); pr?; rv!(13,true); vr?; rv!(17,true);
  vr?;
  delay 45; rv!(17,false); vr?; rv!(13,false);vr?; rp!(5,false); pr?
| xt = 20 -> rf!0; fr?; rp!(6,true); pr?; rv!(14,true); vr?; rv!(17,true);
  vr?;
  delay 45; rv!(17,false); vr?; rv!(14,false);vr?; rp!(6,false); pr?
| xt = 21 -> rf!0; fr?; rp!(7,true); pr?; rv!(15,true); vr?; rv!(17,true);
  vr?;
  delay 45; rv!(17,false); vr?; rv!(15,false);vr?; rp!(7,false); pr?
| xt = 22 -> rb!0; br?; rp!(4,true); pr?; rv!(9,true) ; vr?; rv!(11,true);
  vr?;
  delay 45; rv!(11,false); vr?; rv!(9,false); vr?; rp!(4,false); pr?
| xt = 23 -> rp!(3,true); pr?; rv!(8,true); vr?; rv!(1,true); vr? ;
  delay 45;
  rv!(1,false); vr?; rv!(8,false); vr?; rp!(3,false); pr?
| xt = 24 -> rb!1; br?; rp!(3,true); pr?; rv!(10,true); vr?; rv!(11,true);
  vr?;
  delay 45; rv!(11,true); vr?; rv!(10,true); vr?; rp!(3,false); pr?
| xt = 25 -> rb!2; br?; rp!(3,true); pr?; rv!(10,true); vr?; rv!(11,true);
  vr?;
  delay 45; rv!(11,true); vr?; rv!(10,true); vr?; rp!(3,false); pr?
| xt = 26 -> rb!3; br?; rp!(3,true); pr?; rv!(10,true); vr?; rv!(11,true);
  vr?;
  delay 45; rv!(11,true); vr?; rv!(10,true); vr?; rp!(3,false); pr?
| xt = 27 -> rp!(3,true); pr?; rv!(1,true); vr?; delay 45; rv!(1,false);
  vr?; rp!(3,false); pr?
| xt = 28 -> rb!0; br?; rp!(3,true); pr?; rv!(10,true) ; vr?; rv!(11,true);
  vr?;
  delay 45; rv!(11,false); vr?; rv!(10,false); vr?; rp!(3,false); pr?
| xt = 29 -> rf!0; fr?; rp!(3,true); pr?; rv!(0,true); vr?; rv!(16,true);
  vr?;
  rv!(17,true); vr?; delay 45; rv!(17,true); vr?; rv!(16,true);vr? ;
  rv!(0,true); vr?; rp!(3,false); pr?
| xt = 30 -> rf!0; fr?; rp!(3,true); pr?; rv!(12,true); vr?; rv!(17,true);
  vr?;
  delay 45; rv!(17,true); vr?; rv!(12,true);vr?
; rp!(3,false); pr?  )

; !!time, "Order ", xt, " finished \n"
```

```
        ; rd!xt
     )
]|

proc BMC(chan rb?: 3 # nat, br!: 3 # void, cbd!: bool, cbm!: bool
         , st?: 4 # nat) =
|[ var b: bool = false, k: nat, m: nat = 0
:: *( | , j <- 0..2 , rb.j?k
// The number that processor BMC receives is the desired position
// for the buffer motor.
            ;(k = m    -> skip
// if the motor is already at the desired position, it does not need
// to be moved.
             |k/= m    -> cbd!(k > m); cbm!true; st.k?
// if the motor is not at the desired position, the moving direction is
// determined and the motor is turned on.
                       ; cbm!false; m:= k
// if the sensor of the required position reports the motor, the motor is
// rurned off.
             )
            ; br.j!
// a message is sent back to process RC, reporting that the action is done.

   )
]|


proc BM(chan cbd?: bool, cbm?: bool, bs!: 4 # bool, val rt: real) =
|[ var d: bool, m: bool, q,i: nat = (0, 0)
::bs.0!true
// initialization of the buffer motor. The motor is initially placed at
// position 0. This is reported to the sensors.
; m:= false
; *(m  -> (delay rt;             (d      -> q:= q + 1
                                |not d  -> q:= q - 1
                                 )
// q is the current position of the motor. depending on the desired direction
// (variable d) the motor moves either a position to the left or to the right.
                ; i:= 0
                ; (i < 4)*> (bs.i!(q = 20 * i);  i:= i + 1 ))
// a sensor is placed between every 20 positions. If the motor is in one of
// these positions, a message is sent to the corresponding sensor.
 |cbd?d
 |cbm?m
  )
]|
```

```
 proc FMC(chan rf?: 3 # nat, fr!: 3 # void, cfd!: bool, cfm!: bool
         , st?: 2 # nat ) =
 |[ var b: bool = false, k: nat, m: nat = 0
 :: *( | , j <- 0..2 , rf.j?k
 // The number that process FMC receives is the desired position
 // for the buffer motor.
            ;(k = m    -> skip
 // if the motor is already at the desired position, it does not need
 // to be moved.
             |k/= m    -> cfd!(k > m); cfm!true; st.k?
                        ; cfm!false; m:= k
 // if the sensor of the required position reports the motor, the motor is
 // rurned off.
             )
            ; fr.j!
 // a message is sent back to process RC, reporting that the action is done.
   )
]|


 proc FM(chan cfd?: bool, cfm?: bool, fs!: 2 # bool, val rt: real) =
 |[ var d: bool, m: bool, q,i: nat = (0,0)
 :: fs.0!true; fs.1!false
 // initialization of the buffer motor. The motor is initially placed at
 // position 0. this is reported to the sensors.
  ; m:= false
  ; *(m -> (delay rt;      (d      -> q:= q + 1
                           |not d  -> q:= q - 1
                           )
 // q is the current position of the motor. depending on the desired direction
 // (variable d) the motor moves either a position to the left or to the right.
                ; i:= 0
                ; (i < 2)*> (fs.i!q = 20 * i; i:= i + 1 ))
 // a sensor is placed between every 20 positions. If the motor is in one of
 // these positions, a message is sent to the corresponding sensor.
    |cfd?d
    |cfm?m
   )
 ]|


// The turntable is a rotating disk with 12 cups to output paint.
 proc TMC(chan rtt?: 3 # nat, ttr!: 3 # void, ctd!: bool, ctm!: bool
          , st?,sf?: nat) =
 |[ var b: bool, k: nat, p,q: nat, m: nat = 0
```

```
:: *( | , j <- 0..2 , rtt.j?k
// The number that process TMC receives is the desired position
// for the buffer motor.
             ; p:= (12 + k - m) mod 12
             ; (p = 0 -> skip
// if the turntable is already in the desired position, it does not need to
// be moved.
               | p /= 0 -> ctd!(p <= 6)
               ; q:= p min 12 - p
// if the turntable is not in the correct position, the shortest path to the
// desired position is determined and the direction is set.
               ; ctm!true; (q>0) *> (sf?; st?; q:= q - 1)
// the turntable is moved the desired number of positions.
               ; ctm!false; m:= k
               )
             ; ttr.j!
  )
]|


proc TM(chan ctd?: bool, ctm?: bool, ts!: bool, val rt: real) =
|[ var d: bool, m: bool, q: nat
:: q:= 0
 ; ts!true
 ; m:= false
 ; *(m -> (delay rt;   (d      -> q:= (q + 1) mod 120
                       |not d  -> q:= (q + 119) mod 120
                       )
// q is the current position of the motor. depending on the desired direction
// (variable d) the motor moves either a position to the left or to the right.
                  ; ts!(q mod 10 = 0))
// a sensor is placed at every cup. between every cup there are 10 steps. If
// the motor is at the position of a cup, a message is sent to the
// corresponding sensor.
  |ctd?d
  |ctm?m
  )
]|

proc VC(chan rv?: 3 # (nat,bool), vr!: 3 # void, cv!: 18 # bool) =
|[ var j: nat = 0, x: (nat,bool)
:: *( | , j <- 0..2 , rv.j?x

// the valve controller receives the number of the valve that needs to be
// changed and whether it should open (true) or close (false)
             ; cv.(x.0)!x.1; vr.j!
// the process sends the boolean value to the corresponding sensor, and then
```

```
 // sends a message back to the resource controller.
     )
 ]|

  proc PC(chan rp?: 3 # (nat,bool), pr!: 3 # void, cp!: 8 # bool) =
 |[ var i: nat = 0, x: (nat,bool)
 :: *( | , j <- 0..2 , rp.j?x

 // the pump controller receives the number of the pump that needs to be
 // changed and whether it should open (true) or close (false)
               ; cp.(x.0)!x.1; pr.j!
 // the process sends the boolean value to the corresponding sensor, and
 // then sends a message back to the resource controller.
     )
 ]|

proc A(chan ca?: bool) = |[ var k: bool :: *(ca?k) ]|
// this process models the valves and pumps

proc S(chan senin?: bool, st!: nat) = |[ var b: bool ::
senin?b
 ; *(true   -> senin?b
    |b      -> st!
    )
// process S models the sensors. It checks if there is a motor at
// the sensor and then sends its a message to the corresponding
// controller.
]|

proc TS(chan senin?: bool, st!: void,sf!: void) = |[ var b: bool ::
senin?b
 ; *(true   -> senin?b
    |b      -> st!
    |not b  -> sf!
    )
// process S models the turntable sensors. It checks if there is a motor
// at the sensor and then sends its a message to the corresponding
// controller. To make sure the turntable does not rotate too far,
// the sensor has an extra channel sf! that can be used to send a message

]|

model FACTORY()=
        |[ chan   gd: (nat)
        , dr: 3 # (nat), rd: 3 # nat
        , rb: 3 # nat, rf: 3 # nat, rtt: 3 # nat
        , rv: 3 # (nat,bool), rp: 3 # (nat,bool)
```

```
        , br: 3 # void, fr: 3 # void, ttr: 3 # void
        , vr: 3 # void, pr: 3 # void
        , bs: 4 # bool, sbt: 4 # nat, cbd: bool, cbm: bool
        , fs: 2 # bool, sft: 2 # nat, cfd: bool, cfm: bool
        , ts: bool, stt: nat, ctd: bool, ctm: bool
        , va: 18 # bool, pa: 8 # bool
    :: G(gd)
    || DP(gd, dr, rd)
    || ( || , j <- 0..2 , RC(dr.j, rd.j,rb.j, rf.j, rtt.j, rv.j
                         , rp.j, br.j, fr.j, ttr.j, vr.j, pr.j))
// there are 3 resources controllers working parallel.
    || BMC(rb, br, cbd, cbm, sbt)
    || BM(cbd, cbm, bs, 0.1)
    || ( || , j <- 0..3 , S(bs.j, sbt.j))
    || FMC(rf, fr, cfd, cfm, sft)
    || FM(cfd, cfm, fs, 0.1)
    || ( || , j <- 0..1 , S(fs.j, sft.j))
    || TMC(rtt, ttr, ctd, ctm, stt)
    || TM(ctd, ctm, ts, 0.1)
    || S(ts, stt)
    || VC(rv, vr, va)
    || ( || , j <- 0..17 , A(va.j))
// there are 18 valves
    || PC(rp, pr, pa)
    || ( || , j <- 0..7 , A(pa.j))
// there are 8 pumps
    ]|
```

D χ_σ specification

```
G(gd: chan) = |[xs: nat = 10 |
  (gd!26; gd!10; gd!25)

  ]|

DP(gd, dr0,dr1,dr2,  rd0,rd1,rd2: chan) = |[ x,q: nat, b: bool = true,
   xr1:bool = true, xr2:bool = true,xr3:bool = true,xr4:bool = true,
   xr5:bool = true,xr6:bool = true,xr7:bool = true ,xr8:bool = true,
   xr9:bool = true,xr10:bool = true,xr11:bool = true,xr12:bool = true,
   xr13: bool = true |
 ((b :-> (gd?x; b:= false)
    | (not b and x = 10 and xr6 = true and xr11 = true and xr13 = true :->
      (dr0!x | dr1!x | dr2!x) ; xr6 := false; xr11:= false; xr13:= false ;
       b:= true)
    | (not b and x = 16 and xr1 = true and xr10 = true                 :->
      (dr0!x | dr1!x | dr2!x) ; xr1 := false; xr10 := false; b:= true)
    | (not b and x = 25 and xr7 = true and xr4 = true and xr12 = true  :->
      (dr0!x | dr1!x | dr2!x) ; xr7 := false  ; xr4 := false; xr12 := false;
      b:= true)
    | (not b and x = 26 and xr8 = true and xr4 = true and xr12 = true  :->
      (dr0!x | dr1!x | dr2!x) ; xr8 := false  ; xr4 := false; xr12 := false;
      b:= true)
    | (not b and x = 28 and xr4 = true and xr12 = true                 :->
      (dr0!x | dr1!x | dr2!x) ; xr4 := false  ; xr12 := false; b:= true)
    | (rd0?q; (q = 10 :->  xr6 := true; xr11 := true; xr13 := true
              |q = 16 :->  xr1 := true; xr10 := true
              |q = 25 :->  xr7 := true; xr4  := true; xr12 := true
              |q = 26 :->  xr8 := true; xr4  := true; xr12 := true
              |q = 28 :->  xr4 := true; xr12 := true))
    | (rd1?q; (q = 10 :->  xr6 := true; xr11 := true; xr13 := true
              |q = 16 :->  xr1 := true; xr10 := true
              |q = 25 :->  xr7 := true; xr4  := true; xr12 := true
              |q = 26 :->  xr8 := true; xr4  := true; xr12 := true
              |q = 28 :->  xr4 := true; xr12 := true))
    | (rd2?q; (q = 10 :->  xr6 := true; xr11 := true; xr13 := true
              |q = 16 :->  xr1 := true; xr10 := true
              |q = 25 :->  xr7 := true; xr4  := true; xr12 := true
              |q = 26 :->  xr8 := true; xr4  := true; xr12 := true
              |q = 28 :->  xr4 := true; xr12 := true)))

 )*
; deadlock
```

```
]|

RC(dr, rd, rf, fr,rb,br: chan
        ) =
|[xt: nat, z:bool | (dr?xt
 ; (  xt = 10 :-> rf!1 ; fr?z ;  delay 45
     |xt = 16 :-> rf!0 ; fr?z ; delay 45
     |xt = 25 :-> rb!2 ; br?z ;delay 45
     |xt = 26 :-> rb!3 ; br?z ; delay 45
     |xt = 28 :-> rb!0 ; br?z ; delay 45   )
        ; rd!xt
     )*
; deadlock
 ]|



BMC(rb0,rb1,rb2, br0, br1,br2,cbd, cbm, st0,st1,st2,st3: chan) =
 |[ b: bool = false, c:bool, k: nat, m: nat = 0, x,y,z,a: bool
 | (rb0?k
             ;(k = m    :-> skip
              |k/= m    :->  cbd!(k > m); cbm!true;
                             (k = 0 :-> st0?x
                             |k = 1 :-> st1?x
                             |k = 2 :-> st2?x
                             |k = 3 :-> st3?x)
                             ; cbm!false; m := k
              )
             ; br0!c
  |rb1?k
             ;(k = m    :-> skip
              |k/= m    :->  cbd!(k > m); cbm!true;
                             (k = 0 :-> st0?x
                             |k = 1 :-> st1?x
                             |k = 2 :-> st2?x
                             |k = 3 :-> st3?x)
                             ; cbm!false; m := k
              )
             ; br1!c
  |rb2?k
             ;(k = m    :-> skip
              |k/= m    :->  cbd!(k > m); cbm!true;
                             (k = 0 :-> st0?x
                             |k = 1 :-> st1?x
                             |k = 2 :-> st2?x
                             |k = 3 :-> st3?x)
                             ; cbm!false; m := k
```

```
             )
            ; br2!c


    )*
; deadlock
 ]|


BM(cbd, cbm, bs0,bs1,bs2,bs3:chan, rt: real) =
 |[ d: bool, m: bool, q: nat = 0,i: nat = 0
 |bs0!true
 ; m:= false;
 (m  :-> delay rt;                (d      :->  q:= q + 1
                                 |not d  :->  q:= q - 1
                                  )
                ; bs0!(q = 0)
                ; bs1!(q = 2)
                ; bs2!(q = 4)
                ; bs3!(q = 6)
 |cbd?d
 |cbm?m
   )*
; deadlock
 ]|


FMC(rf0, rf1,rf2,fr0,fr1,fr2, cfd, cfm, st0,st1 :chan ) =
 |[ b: bool = false,c:bool = true, k: nat, m: nat = 0, x:bool,y:bool
 | (rf0?k
            ;(k = m    :-> skip
             |k/= m    :-> cfd!(k > m); cfm!true;
                          (k = 0 :-> st0?x
                          |k = 1 :-> st1?x)
                        ; cfm!false; m:=k
             )
            ; fr0!c
 |rf1?k
            ;(k = m    :-> skip
             |k/= m    :-> cfd!(k > m); cfm!true;
                          (k = 0 :-> st0?x
                          |k = 1 :-> st1?x)
                        ; cfm!false; m:=k
             )
            ; fr1!c
 |rf2?k
            ;(k = m    :-> skip
```

```
               |k/= m    :-> cfd!(k > m); cfm!true;
                             (k = 0 :-> st0?x
                             |k = 1 :-> st1?x)
                           ; cfm!false; m:=k
               )
              ; fr2!c

   )*
; deadlock
]|


FM(cfd, cfm, fs0,fs1: chan, rt: real) =
 |[ d: bool, m: bool = false, q: nat = 0
 | fs0!true
  ; (m :->   delay rt ; (d      :-> q:= q + 1
                       | d = false  :-> q:= q - 1
                       )
                   ; fs0!(q = 0)
                   ; fs1!(q = 2)
   |cfd?d
   |cfm?m)*
; deadlock
 ]|




S(senin, st: chan) = |[ b: bool |senin?b
 ; (true      :-> senin?b
    | b    :-> st!b
    )*
; deadlock
]|



 sys()= hide aa(*,*):ca(*,*,*):mtset
( mp ( enc sa(*,*):ra(*,*):mtset (
 |[ ~gd, ~dr0, ~dr1,~dr2,~rd0, ~rd1,~rd2,~rf0, ~rf1,~rf2,~br0,~br1,~br2,
 ~fr0,~fr1,~fr2,~rb0,~rb1,~rb2, ~cbd,~cbm,~bs0,~bs1,~bs2,~bs3,
 ~sb0,~sb1,~sb2,~sb3, ~cfd,~cfm,~sf0,~sf1, ~fs0,~fs1,~pr0,~pr1,~pr2,
 ~rp00,~rp10,~rp20,~rp01,~rp11,~rp21,~cp,~rv00,~rv10,~rv20,
 ~rv01,~rv11,~rv21,~vr0,~vr1,~vr2,~cv, ~rtt0,~rtt1,~rtt2,~ttr0,~ttr1,~ttr2,
 ~ctd,~ctm,~stt,~ts |
                        G(~gd)
                        || DP(~gd,~dr0,~dr1,~dr2,~rd0,~rd1,~rd2)
```

```
|| RC(~dr0, ~rd0,  ~rf0,   ~fr0,~rb0,~br0)
|| RC(~dr1, ~rd1,  ~rf1,   ~fr1,~rb1,~br1)
|| RC(~dr2, ~rd2,  ~rf2,   ~fr2,~rb2,~br2)
|| BMC(~rb0, ~rb1,~rb2,~br0, ~br1,~br2,~cbd, ~cbm,
       ~sb0,~sb1,~sb2,~sb3)
|| BM(~cbd, ~cbm, ~bs0,~bs1,~bs2,~bs3, 0.1)
|| S(~bs0, ~sb0)
|| S(~bs1, ~sb1)
|| S(~bs2, ~sb2)
|| S(~bs3, ~sb3)
|| FMC(~rf0, ~rf1,~rf2,~fr0, ~fr1,~fr2,~cfd, ~cfm,
       ~sf0,~sf1)
|| FM(~cfd, ~cfm, ~fs0,~fs1, 0.1)
|| S(~fs0, ~sf0)
|| S(~fs1, ~sf1)]|)))
```