Deliverable 5

The Demonstrator

Esprit Project 27439 - HIDE

High-level Indegrated Design Environment for Dependability

Gy. Csertán, M. Dal Cin, G. Huszerl, J. Jávorsky, K. Kosmidis, A. Pataricza, Cs. Szász

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Conzorcio Pisa Ricerche-Pisa Dependable Computing Centre (PDCC)
Technical University of Budapest (TUB)

HIDE/D5/TUB/1/v2

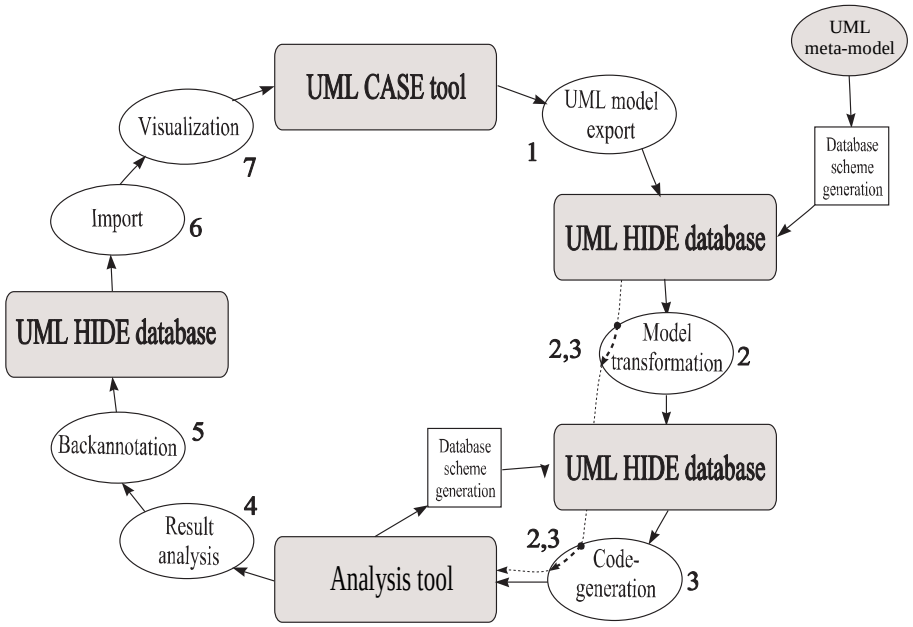

Figure 1: Analysis steps of an UML model

1 The HIDE core technology

In this section the HIDE core technology is presented according to the definition elaborated in work phase 2. In phase 2 many new features have to be implemented which are missing in the first phase, like fault-tolerant component library, fault-injection engine, back-annotation. If we propose for the next phase the technology used in the first, then we must investigate it's capability to handle the new features.

The HIDE repository must enable the following features:

- Storage of UML models designed in a CASE tool
- Storage of the generated models for target tools
- Storage and execution of the transformation library
- Storage of fault tolerant components (conform the UML meta-model)

Phase 1 showed that the implementation of the HIDE repository on the basis of a relational database system is useful not only for rapid prototyping but it is also a very universal approach, adaptable for other projects as well, when different tools must be integrated.

There is a wide range of usable programming languages for implementing the features of the HIDE environment as most languages have a standard interface for handling relational databases. In spite of variety of choosable programming language, the most effective one is the standard internal programing language of the relational databases, the PL/SQL.

2 Transformation from the UML tools to the HIDE database

As in the previous sections already presented, the starting point of implementing any model-transformation is to have the UML model in the HIDE repository. In the Section X the nearly-automatic generation of a UML database scheme was presented. In the following section the generated database scheme is presented, then the implementation of the export feature from CASE tool to the HIDE repository.

2.1 The database structure

The DDL file (see Deliverable 3. for detail), which holds the statements for creating the database scheme corresponding to the UML meta-model, can be used on most relation database systems. The database scheme contains 125 database tables, each table storing one aspect of the UML model. For example a table will store the instances of generalizations between classes, while others the properties of an attribute.

The most important concept in the database scheme is that the IDs identifying an object in an UML model can hold any ID used and given by an arbitrary CASE tool. Thus, it can hold the UML model exported from various CASE tools, while preparing the IDs given by the host tool. This is very important when after the formal analysis the results must be visualized in the tool. Without the ability of connecting the results to one or more particular objects in the original UML model, the results cannot be interpreted and used. Therefore, it is supposed that the IDs are preserved all the way through the model-transformation and the analysis.

2.2 Transformation to the database

In the demonstrator, as a front-end UML CASE tool, the Innovator was integrated. Its internal script language is the Tcl/Tk. This standard script language is extended with functions that can read and write the repository holding the UML model. In order to enable on-line connection to the database holding the HIDE repository it was extended, so that in Windows platform it can connect to various databases through ODBC interface.

As part of the demonstrator the export function was implemented which can export the content of the repository in a text format or directly into the database. In the first case the exported file must be executed inside the database (for inserting the exported data into the database), while in the second case the data exported from the repository is inserted into the database during the execution of the script.

The export script exports only the logic aspect of the UML model, i.e.. the visual properties are ignored, because the analysis tools cannot utilize these information.

The script can be executed using a menu, integrated into Innovator (Figure 2). This way, after designing a UML model in the Innovator, the contents can be exported by selecting a menu item.

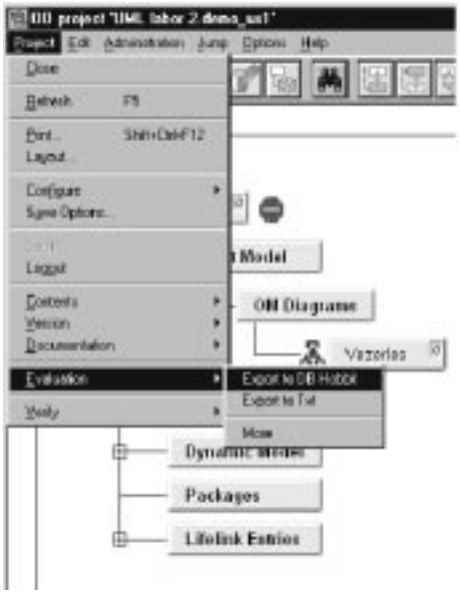

Figure 2: Extending the Innovator with export feature

3 Transformation generation toolbox and its application to selected transformations

In this section two model transformation implementations and their integration into the host UML CASE tool is presented. In the demonstrator example not every possibility was used to make the integration of the analysis procedure as user-friendly as possible. The execution of the transformation procedure, which makes the conversion between the UML model and the target model, can be initialized inside the CASE tool, however the execution of the analysis tool with the generated input file and the result analysis must be done by the user, which means that he must understand the results generated by the analysis tool. The elimination of this leakage will be the task of the second work phase as shown in Figure 1 (steps 4-7).

3.1 Transformation from UML to Spin[1]

3.1.1 Introduction

In accordance with our previous consideration the process of transformation to Promela starts with the transformation of the statechart into an intermediate form, on the basis of which then the transformation will occur according to the stated rules. This intermediate form is the EHA (Extended Hierarchical Automaton).

The very same consideration holds for our transformation, i.e. an EHA database is created on the basis of the generated UML database, with the aid of which EHA database the rules will then be implemented.

[1]For a short description of the Spin see Deliverable 3.

It is important to stress that the EHA database is independent of SPIN, i.e. the structure within the EHA is one equivalent to the statechart. Thus, if needed and if the conversion rules are properly modeled the mass of data can be converted to the input language of other tools.

The programming language used is PL/SQL.

3.1.2 The structure of the EHA

Four major tables constitute the structure of the EHA database, the content of which is generated from the UML database. These are:

- State,
- Seqaut,
- Transition,
- Trlabel.

These tables describe the EHA as follows. Table State contains the states of EHA and table Seqaut includes the different sequential automatons, while Table Transition comprises the transitions defining the automaton they have been moved to and also the corresponding source and target states. Finally, table Trlabel is the one already given at the definition of the EHA, including all transitions with source and target states, the events that triger these transitions and also their possible guards, etc.

Three additional tables are generated for the transformation, yet these are only auxiliary tables needed for programming technique considerations such as decreasing procedure time, etc.

In the followings the EHA database, the fields of the different major tables and the meaning of these fields are discussed in detail.

3.1.3 The State Table

The fields of the table are :

- ID :character
- Seqaut: character
- IsBasic: numeric
- IsInitial: numeric
- StName: character
- Pos: numeric

Field ID is the same as field ID in the State table of UML database. Field Seqaut indicates the sequential automaton of the Extended Hierarchical Automaton, in which the given state is found. The value of IsBasic is either 1 or 0. Value 1 means that the given state is a basic state in the EHA, while value 0 means that the given state is not a basic state. Field IsInitial is very similar to IsBasic, it indicates whether the given state is a initial state of the EHA or not.

Field StName indicates the name the user has attributed to the given state in the original statechart, while Field Pos simply contains serial numbers to states starting from 0. It is needed for later designation of states in the generated Promela code as:

$$S < \text{serialnumber} >< \_ >< \text{statename} >$$

Both fields ID and Pos have different values for each state, the primary key is field ID, while the foreign key is field ID in the following Seqaut table corresponding to field SeqAut in table State.

3.1.4 The Seqaut Table

This table contains the following fieldsl:

- ID: character
- IsRoot: numeric
- ParState: character

Field ID figures the identifiers of sequentiel automatons that were defined during the production of table State. According to the EHA field IsRoot is 1 for the root automaton, i.e. at the top hierachy automaton, while it is 0 for any other automatons.

Field ParState contains the parent state of a given automaton, i.e. the state, from which the EHA derives. Obviously, this field is void for the root automaton, while it is filled in for all the other automatons. The sate is identified by the ID field of table State.

In this table the primary key is ID.

3.1.5 The Transition Table

The fields of the table are:

- ID: character
- Transition: character
- SeqAut: character
- Target: character
- Source: character

- Label: character

Field ID is equivalent to the ID field in the transition table of the UML database, while field Transition is an identifier generated by us and with the following syntax:

$$t < \text{serialnumber} >$$

Field Label is generated alike, it identifies the label belonging to the transition and its structure is:

$$l < serialnumber >$$

Field SeqAut contains the identifier of a sequential automaton, since the target and source states of a transition have to be in the same sequential automaton in the EHA. Therefore, field Seqaut contains the automaton, in which the transition is found. Field Target and Source comprise state identifiers (corresponding to field ID of table State). These states are those, in which the transition arrives or from which it starts. These fields are not the original ones, because in the statechart there may be interlevel transitions, and it was proposed in the theoretical bases that in this case the transition has to be moved to that level of the extended hierarhical automaton where it can be handled as a simple transition.

Fields ID, Transition and Label have all different values for each state, the primary key is field ID, while the foreign key is field Label, which corresponds to field LabelID in the following Trlabel table.

3.1.6 The Trlabel Table

The fields of the table are:

- LabelID: character
- TransitionID: character
- Target: character
- Source: character
- Event: character
- EvName: character
- EvPos: numeric
- Action: character
- Guard: character

Fields LabelID and TransitionID correspond to fields Transition and Label of the Transition table. Fields Target and Source contain the original target state and source state of the transition, i.e. the states between which the transitions really occur. These states figure in field Transition of Transition table of the UML database.

Field Event contains the identifier of the event, which triggers the transition, this identifier corresponds to the event identifier of the UML database. However, field EvName contains the original name of the event given by the user. If a transition is not be triggered by an event then this field contains the string "pseudoevent". Field EvPos comprises a serial number, all particular event names have particular serial numbers. This fact does not mean that all of the transitions have their own serial numbers because the same event can trigger more transitions.

An action generated by a transition figures in field Action if an action belongs to this transition. In case there is no generated action the field is void. Similar considerations are valid for field Guard: if a transition has a guard then it figures in this field, if not then the field is void.

Both fields LabelID and TrasitionID have different values for different states in the table. The primary key to the table is LabelID, while the foreign key is TransitionID corresponding to field Transition of table Transition.

3.1.7 The Possibility of Back Annotation

During the generation of the Promela code variable names generated by us are used in order to help the better understanding of the code, since for the time being the verification of SPIN is a manual process. However, in the EHA database original UML identifiers figure as well, therefore following the automatic SPIN verification it will be easy to back-trace the results up to the UML database and then to do it from there back up to the model. This all means that there is no theoretical hindrance to the future realisation of back-annotation.

3.1.8 From Statechart Diagrams to Kripke Structures Sketch of an Algorithm

In the following an algorithm for building the Kripke Structure resulting by applying the operational semantics of UML Statechart Diagrams defined defined in Deliverable 2 to a statechart is sketched. The description of the algorithm is rather informal, given in a kind of pseudo-pascal notation, where, on the other hand, set notation is freely used.

The *input* to the algorithm is a extended hierarchical automaton H and its initial status $(\mathcal{C}_0, \mathcal{E}_0)$. It uses relations $A \uparrow P :: (\mathcal{C}, \{e\}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$ and $(Sel\ \mathcal{E}\ e\ \mathcal{E}')$, defined in Deliverable 2, as boolean functions. Also function *join* on event queues, defined in Deliverable 2, is used. The termination condition is reached when there are no more (new) statuses to analyze. At this point, the *output* of the alrorithm is given by the values of the variables St and Tr holding respectively the set of statuses and the set of step-transitions of the resulting Kripke Structure (the initial status is that given as an input). The algorithm is given in Fig 3.

Types

EHA (* to represent extended hierarchical automata *)

State (* to represent states *)

Event (* to represent events *)

Queue = **queue_of** Event

Config = **set_of** State

Status (* to represent a status as a pair
(configuration, event queue) *)

S-Transition (* to represent a STEP-transition as a pair of statuses *)

Variables

H : EHA

S : **set_of** Status (* Stores the set of currently generated statuses *)

St : **set_of** Status (* Stores the set of currently analyzed statuses *)

Tr : **set_of** S-Transition (* Stores the set of currently generated transitions *)

$\mathcal{C}, \mathcal{C}'$: Config

e : Event

$\mathcal{E}, \mathcal{E}', \mathcal{E}''$: Queue

Initialization

H := (* the input extended hierarchical automaton *);

S := $\{(\mathcal{C}_0, \mathcal{E}_0)\}$;

St:= $\emptyset$;

Tr:= $\emptyset$;

repeat

begin

select any element from S and assign it to $(\mathcal{C}, \mathcal{E})$;

S := S $\setminus\{(\mathcal{C}, \mathcal{E})\}$;

St := St $\cup\{(\mathcal{C}, \mathcal{E})\}$;

for $(e, \mathcal{E}'')$ **in** $\{(x, Y) \mid (Sel\ \mathcal{E}\ e\ Y)\}$ **do**

if $H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}')$ for some set of transitions L, and status $(\mathcal{C}', \mathcal{E}')$

then **begin**

S:= S $\cup(\{(\mathcal{C}', join\mathcal{E}'\ \mathcal{E}'')\} \setminus$ St);

Tr:= Tr $\cup\{((\mathcal{C}, \mathcal{E}), (\mathcal{C}', join\mathcal{E}'\ \mathcal{E}''))\}$

end

end

until S $\neq \emptyset$

Figure 3: KS algorithm sketch

3.1.9 The Transformation

The transformation has been carried out according to the previously-defined rules.

3.2 Extending the CASE tools with HIDE engine

As the CASE tool used in the demonstrator is the Innovator, the extension mechanism has the following steps:

1. Extending the Innovator with the *Export* function, which will export a UML model into the HIDE repository. This must be done only once, and used for all transformations.
2. Extending the Innovator with a Tcl script, which will execute the transformation procedure (written in PL/SQL and stored by the database) inside the database. This function is responsible for transforming the UML model stored in the database to the meta-model of the target analysis tool. It must also generate the input file of the target tool.
3. Optionally: Initiating the analysis with the target tool with the generated file as input. This is not always possible, as the analysis tools used in Phase 1. can only be executed in UNIX platform.

The Innovator extensibility interface supports user defined menus to be added, which call some predefined Tcl scripts. These scripts can connect to the internal repository of the Innovator, as well as, – this is implemented only on the Windows platform in Phase 1.– to the HIDE repository.

4 The fault injections engine

In work Phase 1 no fault injection engine was implemented, since model extensions and transformations that rely on the fault injection engine were not selected for implementation.

5 The back-annotation mechanism

In work Phase 1. no back-annotation mechanisms were implemented, but the implemented transformations try to propagate as many information about the original UML model as possible. This is necessary, when we want to understand, moreover, to automatically analyze the results generated by the analysis tool. The easiest way for enabling back-annotation is by preserving the internal IDs given by UML CASE tool. This way, we can go back directly to the element in the original UML model in the host CASE tool.

In the second work phase when implementing the transformations, additional effort must be addressed for preserving the information (at least the IDs of the elements) needed for back-annotation.

6 Production Cell as Demonstrator Example

6.1 Introduction

The production cell is a benchmark [Claus Lewerentz and Thomas Lindner, editors. Formal Development of Reactive Systems, volume 891 of Lecture Notes in Computer Science. Springer-Verlag, January 1995] in modelling embedded systems. The model the of production cell in UML is manageable. Thus, experienced in modelling real systems, we can provide evaluates such throughput. We also developed formal methods to verify the specification of the production cell.

6.2 The production cell

The production cell processes metal blanks which are conveyed to a press by a feed belt. A robot takes each blank from the rotary table and places it into one of the two presses. The press forges the blanks and the robot takes the metal plate from the press and puts it on a deposit belt. A worker puts the blanks on the feed belt and take the forged blanks from the robot. The production cell should tolerate the failure of one of the presses.

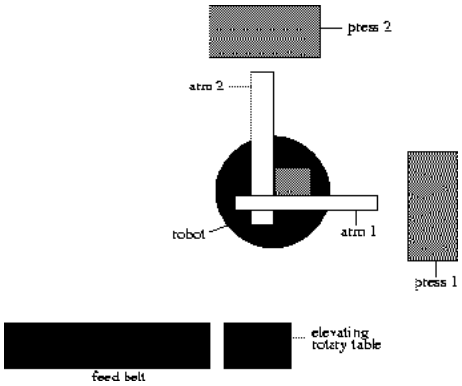

Figure 4: Schematic diagram of the production cell

6.2.1 Architecture of the cell

Figure 1 shows a top view of the production cell. On the bottom left the feed belt is shown which conveys the blanks to an elevating rotary table. This table has to be between the feed belt and the robot to bring the blanks into the right position so that the robot can pick them up. To increase the utilization of the presses, the robot is fitted with two arms - one always used for loading (arm 1), the other one (arm 2) for unloading the presses.

The system is controlled by means of actuators using information received from sensors. The system has a set of actuators, for rotating the robot base, picking up metal plates with each robot arm and other similar tasks. There is also a set of sensors, providing the control program with information about the state of the system. They return discrete values, like "press

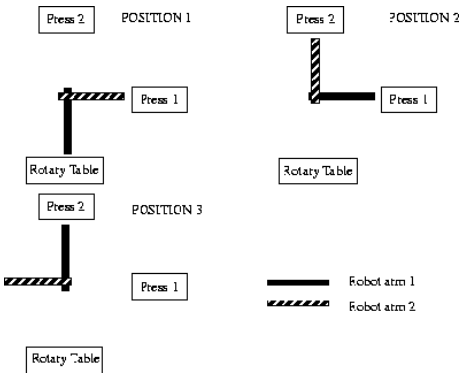

Figure 5: Positions of the robot

is open for loading" or "a blank arrived at the end of the feed belt". The production cell can be modelled as a faulty set of finite automata.

Feed Belt The feed belt conveys the blanks to an elevating rotary table. The belt is powered by an electrical motor. On the belt there is space for two blanks. There are two light barries at the start and the end of the belt. The first sensor indicates if there is a blank on the first position of the blank. The second sensor is active if a blank leaves the belt.

Rotary Table The rotary table is between the feed belt and the robot. The task of the rotary table is to load the blank from the feed belt, to turn into a right position for the robot and to lift the blank up in such a way that the robot arm can be loaded. The feed belt and the robot arm are in different vertical positions. For loading the arm we have to lift the blank up. The rotation movement is necessary because we can't rotate the picker arm.

Robot The robot's task is to pick up the metal blanks from the rotary table, put them in one the presses to be forged and taking the forged plates from the presses. To raise up the efficiency the robot has two robot arms which build a 90 degree angle. Every arm has his own task. Robot arm 1 always picks up the blanks and loads the presses. Robot arm 2 always unloads the presses.

Presses The task of the two presses is to forge the metal blanks. The press is loaded by robot arm 1. After forged the metal blank the press is unloaded by robot arm 2. Our model contains 2 presses to tolerate the failure of one of the presses.

6.2.2 Movement sequence of the robot

To simplify our model the robot can take one of the three position depicted in Figure 2. At the beginning the robot arm 1 is over the rotary table and ready to pick up a metal blank. The robot can now load the press 1 by taking position 2 or press 2 by taking position 3. For unloading the presses by robot arm the robot enters either position 1 (for press 1) or position 2 (for press 2).

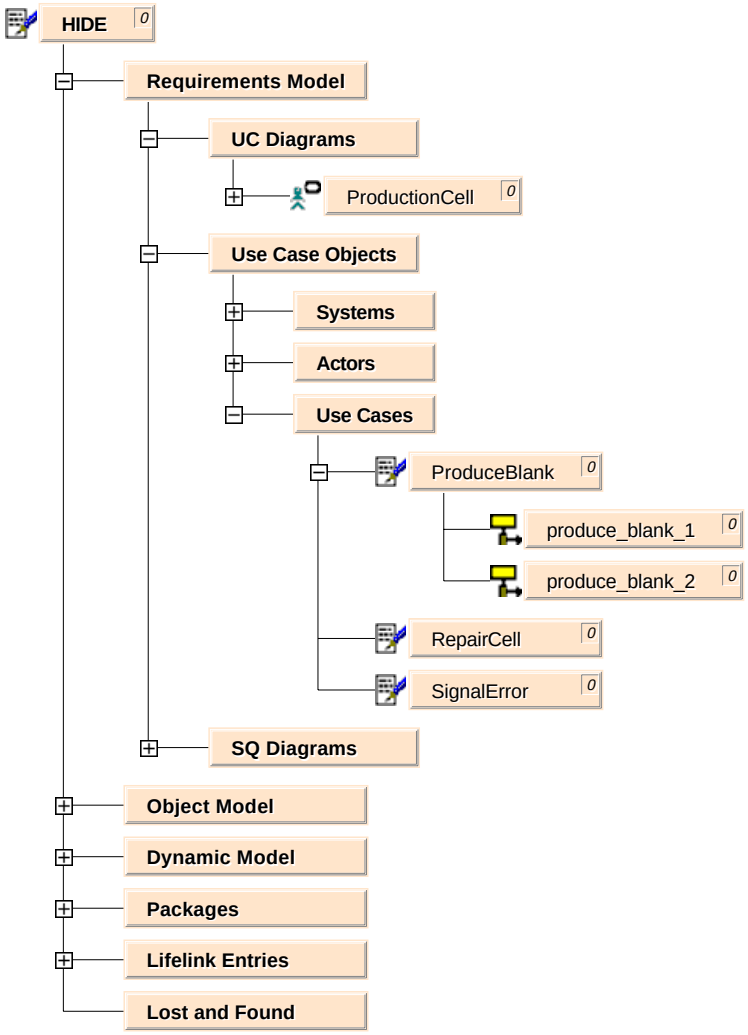

Figure 6: The requirement model of the production cell

6.3 The structural diagrams

6.3.1 The requirement model

The requirement model is given in Figure 6. It contains the use case diagrams, the use case objects, and the sequence diagrams. The use case objects are the objects that are defined in the use case diagrams: actors, systems, and use cases. To each use case sequence diagrams can be assigned, but further sequence diagrams can also be defined, and they remain unassigned.

Figure 7 depicts the use case diagram of the production cell. The elements of the use case diagram are (see also Figure 6) the use case objects:

Worker The actor of the system. In the normal functionality the worker is responsible for feeding the production cell with unprocessed metal blanks and for removing the forged metal blanks for the system. The relationship is bidirectional between worker and system. In an extended functionality that is not implemented yet the worker can repair the crashed components of the cell. The relationship is from the worker to the system, although it can

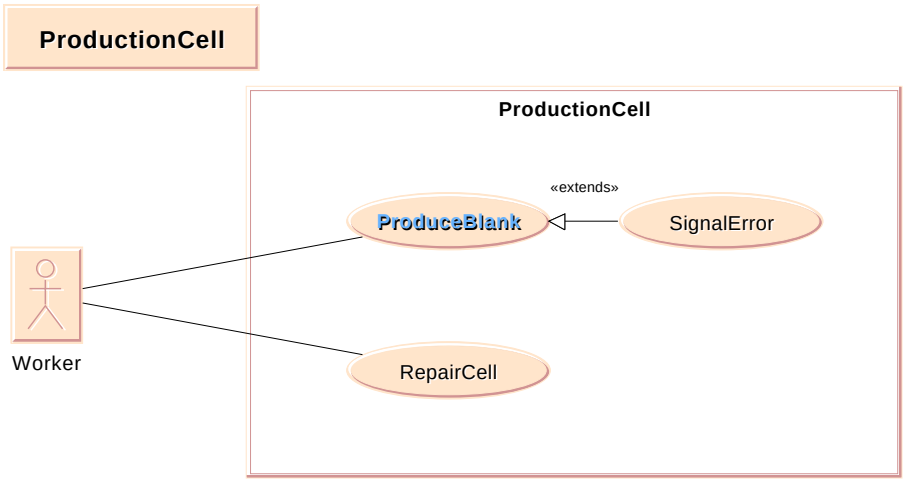

Figure 7: The use case view of the production cell

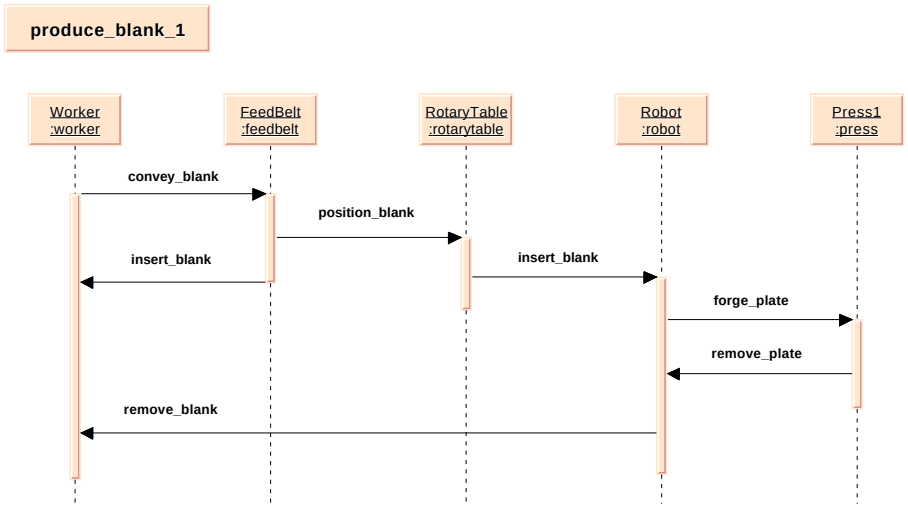

Figure 8: Sequence of using Press1 for producing a blank

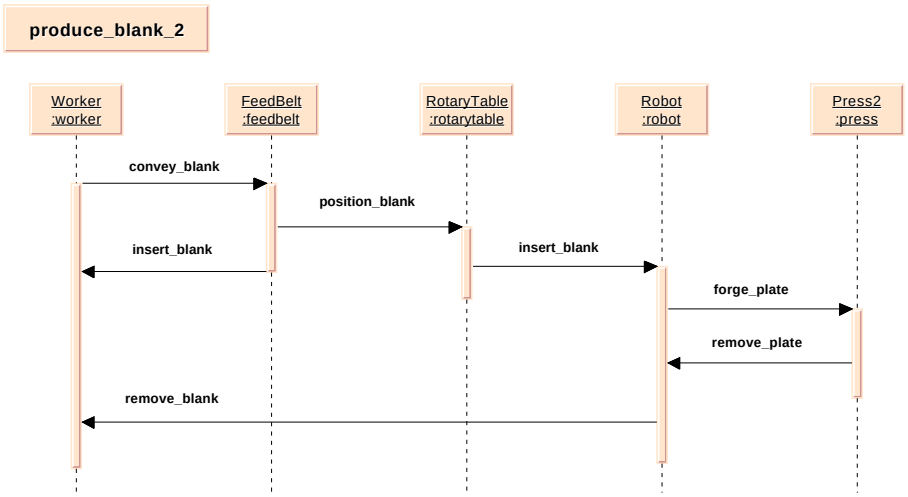

Figure 9: Sequence of using Press2 for producing a blank

not be represented graphically.

ProductionCell The system itself. Its functionalities are expressed by the three use cases: ProduceBlank, SignalError, RepairCell. It is in connection with its environment, with the Worker.

ProduceBlank The main use case of the system. It describes the primary function of the system when it is used to produce forged metal blanks. In this use case an unprocessed blank is taken from the worker forged by one of the presses and then the forged blank is given back to the worker.

SignalAlarm This use case extends the ProduceBlank use case by signaling a failure, e.g. the crash of a given system components. (This use case is not implemented yet.)

RepairCell This use case represents the functionality of the system in which the worker is able to repair a crashed component and to set the system back to its fault-free state. (This use case is not implemented yet.)

The implemented use case is the ProduceBlank use case. The two most important sequences of this use case are depicted in the sequence diagrams in Figure 8 and Figure 9. The only difference between the two sequence diagrams is that in the one Press1 is used while in the other Press2 is used. Therefore only the first sequence diagram is described in detail.

The worker places a blank onto the feed belt that interprets it as a convey_blank message. The feed belt conveys the blank to the rotary table and pushes the blank onto the table. The table interprets it as a position_blank message. The feed belt signals the worker that a new blank can be inserted by an insert_blank message. The table positions the blank for the robot by turning right and rising to the level of arm1. Upon reaching the correct position the table sends an insert_blank message to the robot. The robot places the blank into the press1 and starts the forging process by sending a forge_blank message. When forging is finished the press signals it to the robot by a remove_plate message. The robot removes the blank from the press

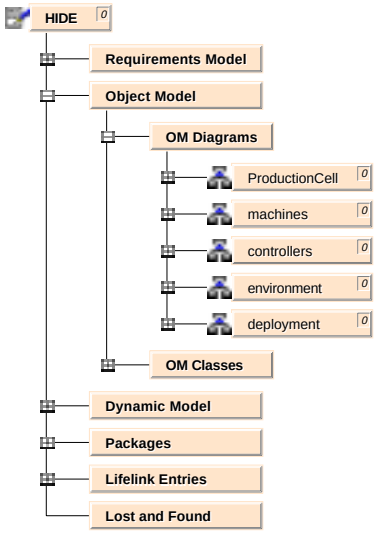

Figure 10: The object model of the production cell

and gives a remove_blank signal to the worker who should remove the blank from the arm of the robot.

6.3.2 The object model

The object model describes the object view of the system. It is a set of graphs that are made of nodes; the classes and objects of the system and arcs; the relationship among the classes and objects. Figure 10 shows the object model of the system. It consists of two parts: the object model diagrams and the object model classes. The latter one is only a listing of the classes used in the former ones. In the object model of the production cell five object diagrams can be found:

ProductionCell This diagram describes the objects of the production cell and their relationship.

machines This diagram describes the classes and the derived objects of the machines.

controllers This diagram describes the classes and the derived objects of the controllers for the machines.

environment This diagram describes the environment of the production cell.

deployment This diagram is not a real class diagram its only function is to be a container for the deployment diagrams.

Figure 11 shows the objects that describes the hardware parts of the machines of the production cell. The classes are: conveyor_belt_hw, table_hw, robot_hw, and press_hw. From the classes objects are derived by instanciation. The two presses Press1HW and Press2HW have the same parent, while the other objects do not share a same parent.

Each object has a single method the name of which is equivalent with that of the object. The methods describe the hardware functionality of the machine, e.g. the sensors and the motors of the feed belt or the motors, sensors, or coils around the electromagnet of the robot.

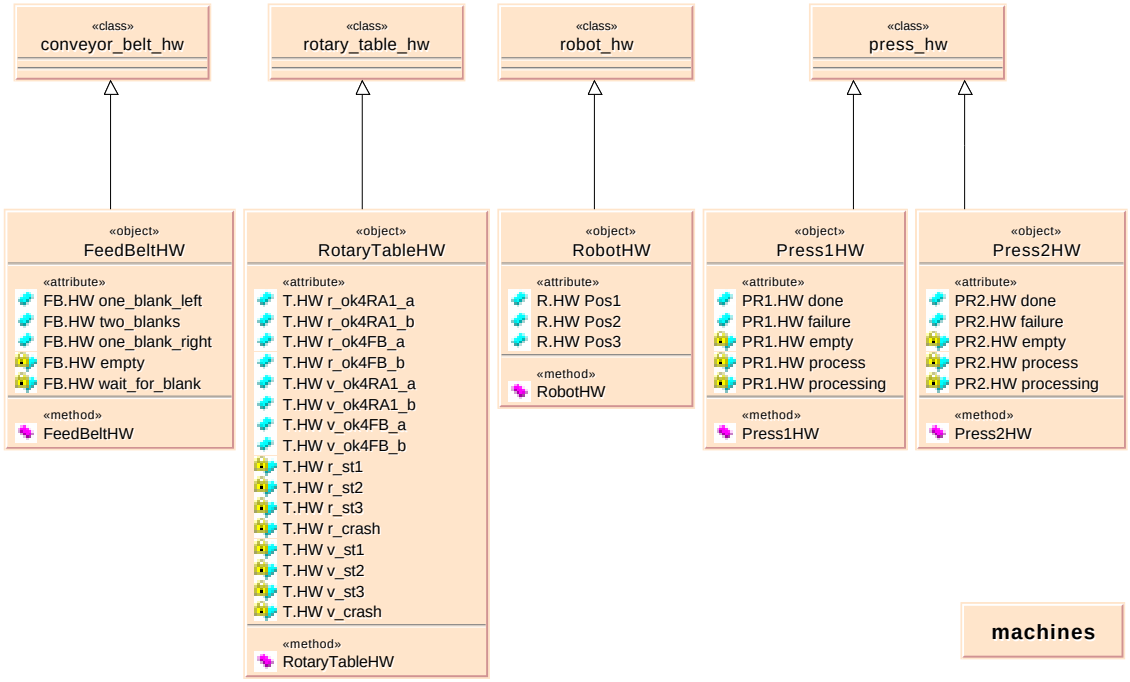

Figure 11: Class diagram of the machines

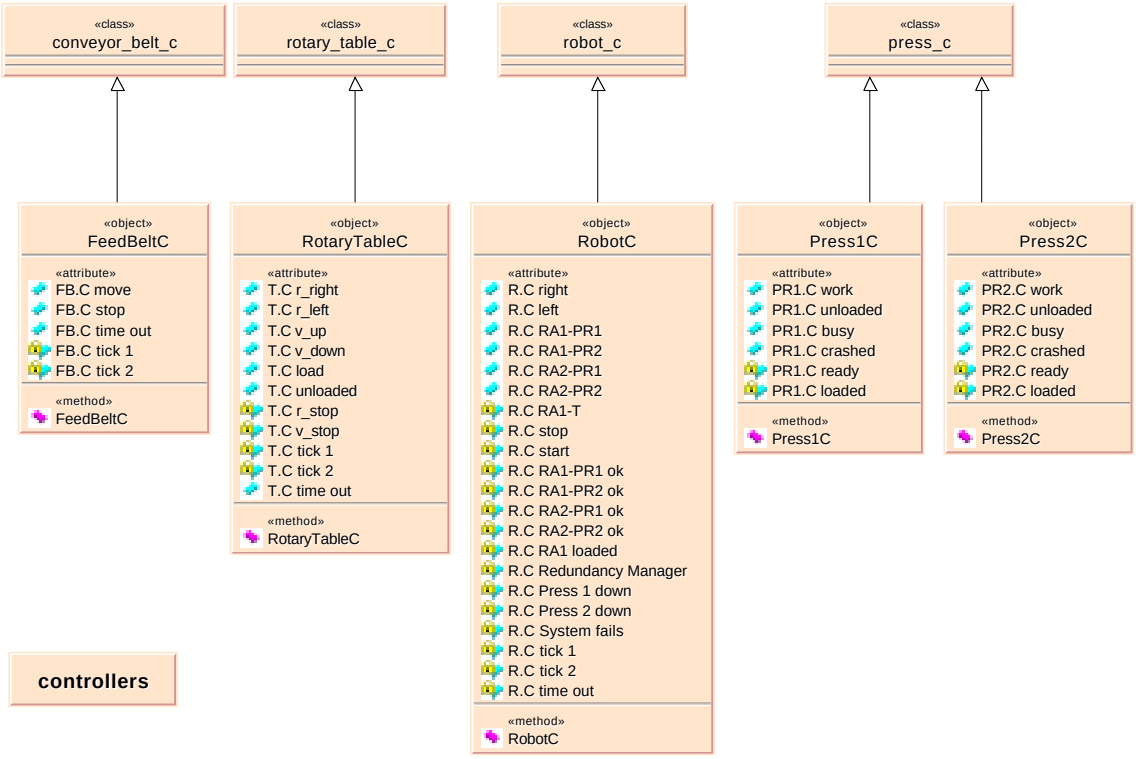

Figure 12: Class diagram of the controllers

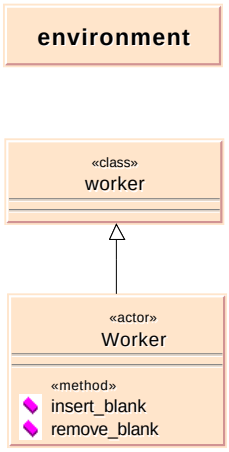

Figure 13: Class diagram of the environment

The attributes of the objects can be divided into two parts. Public attributes describe the signal and actuator signals of the objects. They are public since other objects should see it. The private attributes are used to identify the internal states of the object.

Figure 12 shows the objects that describe the controller part of the machines of the production cell. The classes are: conveyor_belt_c, table_c, robot_c, and press_c. Instanciation is done similarly to the hardware part and methods and attributes can be explained in the same way.

In Figure 13 the environment of the production cell is described. The environment is the Worker that is an object. It is instanciated from the class worker. The two methods of the worker describe the insertion and removal of blanks onto the feed belt and from the robot arm.

Finally the object diagram ProductionCell (shown in Figure 14) describes the relationship among the different object of the production cell. The Worker is in connection with the FeedBeltHW that signals the Worker that a new blank can be inserted and with the RobotHW that signals the Worker that the forged blank can be removed.

The object ProductionCell is a container object for the system. It contains the parts FeedBelt, RotaryTable, Robot, Press1, and Press2 that denote the objects of the production cell. The containment is described by the aggregation relation. Each of the above objects contains two parts, the object that represent the hardware of the machine and the controller that describes the controller functionality of the machine. Again the aggregation relation is used to express this relation.

As an example the feed belt is composed of two parts FeedBeltHW and FeedBeltC. They are in relation controls where the hardware has the role machine and the controller has the role controls. This is expressed by the association relation between the objects.

Synchronization of the machines occurs via the synchronization of the corresponding controllers. One machine can not start working without receiving the starting signal from the other machine. It is represented by a dependency relationship. For example the feed belt loads the rotary table that can start its work then. To carry on the work the rotary table should be unloaded by the robot. Similarly the presses should be loaded and unloaded by the robot.

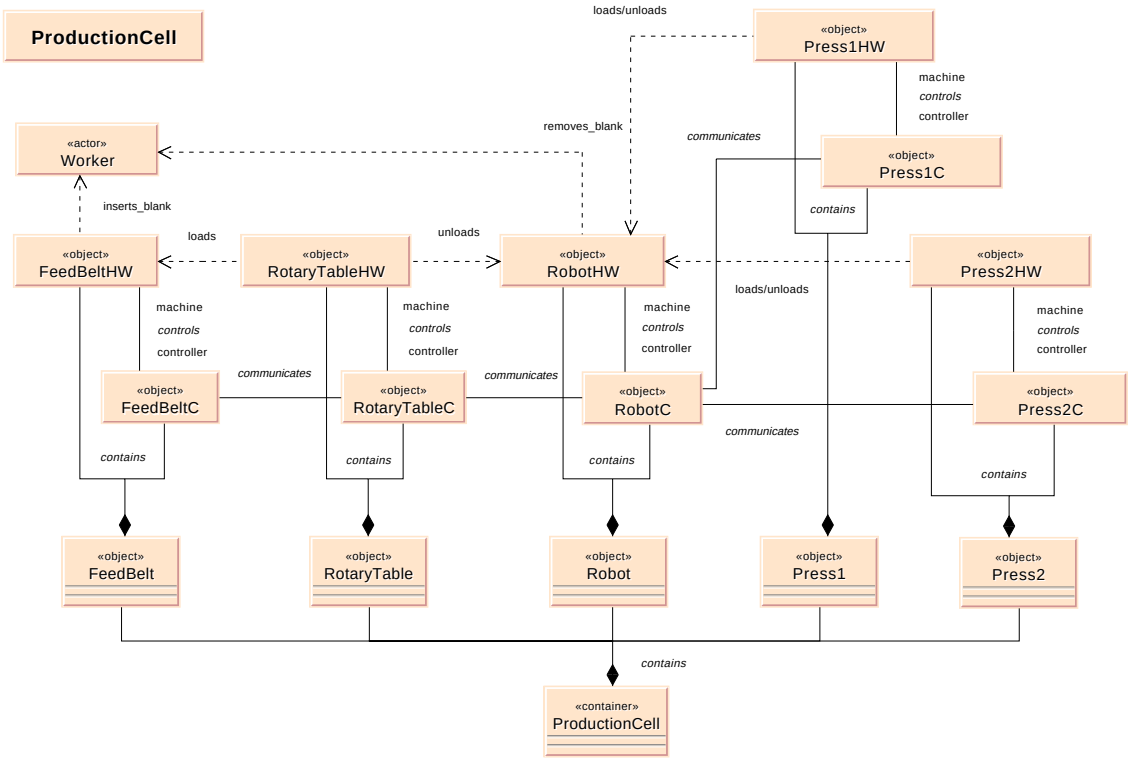

Figure 14: The objects of the production cell and their relationship

While the robot is unloaded by the worker.

6.4 The package model

The package model of the production cell is relatively simple, since the system does not contain to many objects. Figure 15 shows the package model that consists of three packages and a package diagram. Additional elements of the package model are: the deployment diagrams and the object model diagram ProductionCell. They are linked to the model as assigned objects.

Figure 16 shows the package diagram. The packages of the system are:

machines This package includes the nodes of the system that represent the physical machines. Elements of this package are: machine_feed_belt, machine_rotary_table, machine_robot, machine_press1, machine_press2.

controllers This package contains the nodes of the system that represent the physical controllers that can be used in the system. Elements of the package are: controller_pc, controller_sps.

functionalities The last package contains components of the system that describe the systems functionalities. Elements of the package are: FeedBeltC, FeedBeltHW, RotaryTableC, RotaryTableWH, RobotC, RobotHW, Press1C, Press1HW, Press2C, Press2HW.

Dashed lines in the figure represents the dependencies between the packages. In our model two such dependencies exist. They express that the functionality of the system depends on the

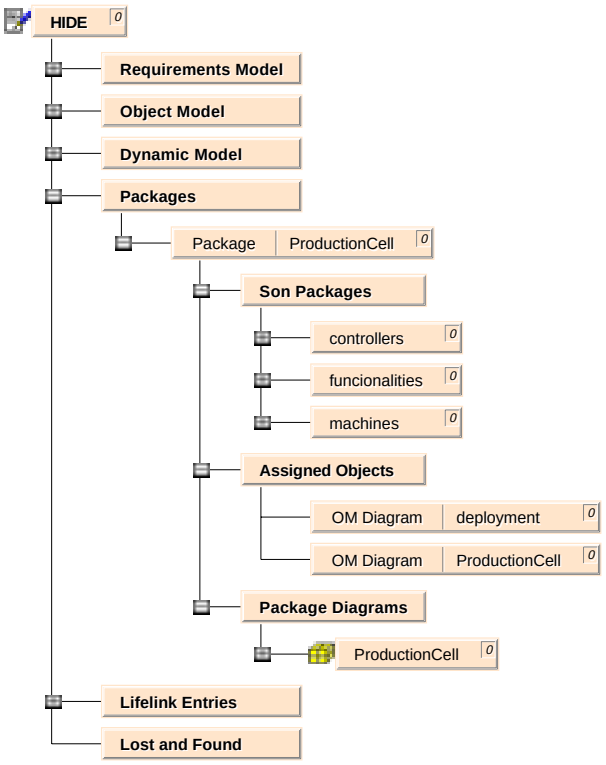

Figure 15: Package model of the production cell

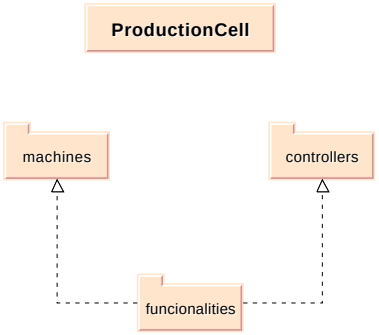

Figure 16: Package diagram of the production cell

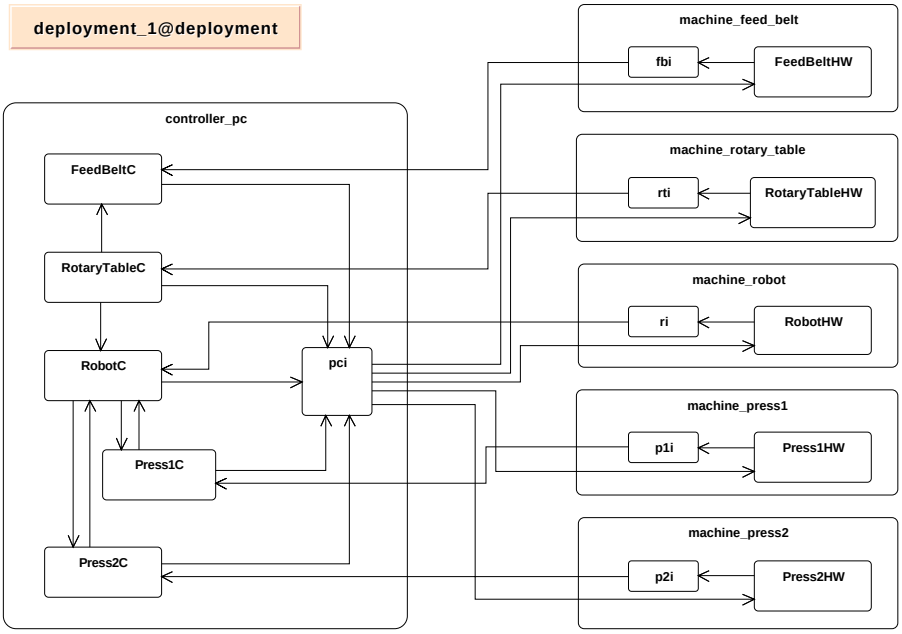

Figure 17: Production cell with centralized controller

function of the machine and controller nodes.

6.4.1 The deployment model

The deployment model consists of the deployment diagrams of the system. The deployment diagram describes a possible architecture of the system; a possible assignment of the components to the nodes.

Figure 17 depicts the first deployment diagram of the production cell. Controlling functionalities are centralized; each control component is mapped to the same node. This node is a controller PC. Communication between components occurs through the interface of the node. (The interfaces are the small rectangles in the figure.)

In Figure 18 the second deployment diagram of the system is shown. It describes a distributed version of the controller; each control component is mapped to a different SPS controller node. Similarly to the centralized system, the components communicate via the interfaces of the nodes.

6.5 The dynamic diagrams

The behaviour of the modelled system can be described by a statechart which contains all the sub-statecharts of the components. Every component of the system consists of two sub-statecharts. One describes the behaviour of the controller of the software and the other that of the hardware. Only communication between hardware- and controller-sub-statecharts of the same component is allowed. The controller sub-statecharts of different components can also communicate together as shown in picture x.

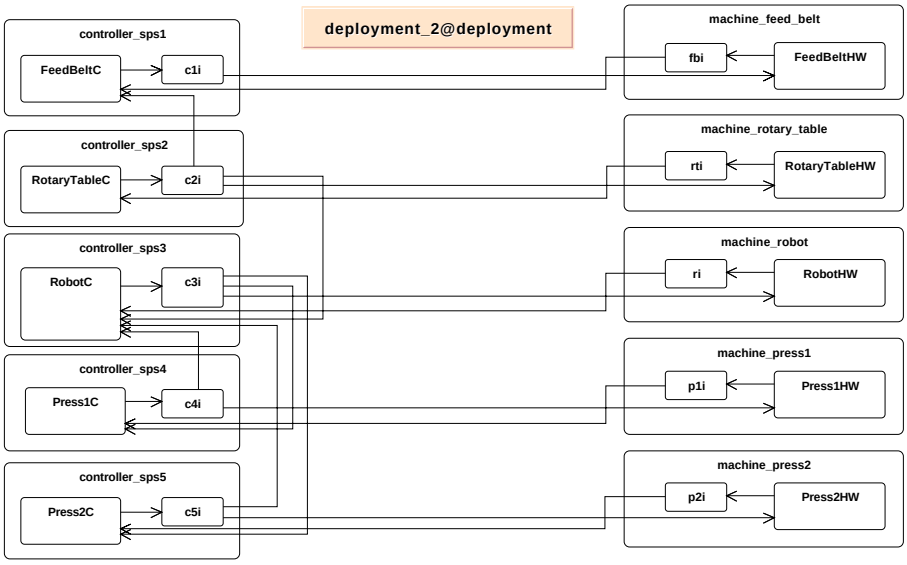

Figure 18: Production cell with distributed controller

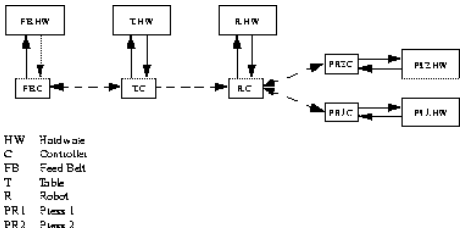

Figure 19: Controller and components

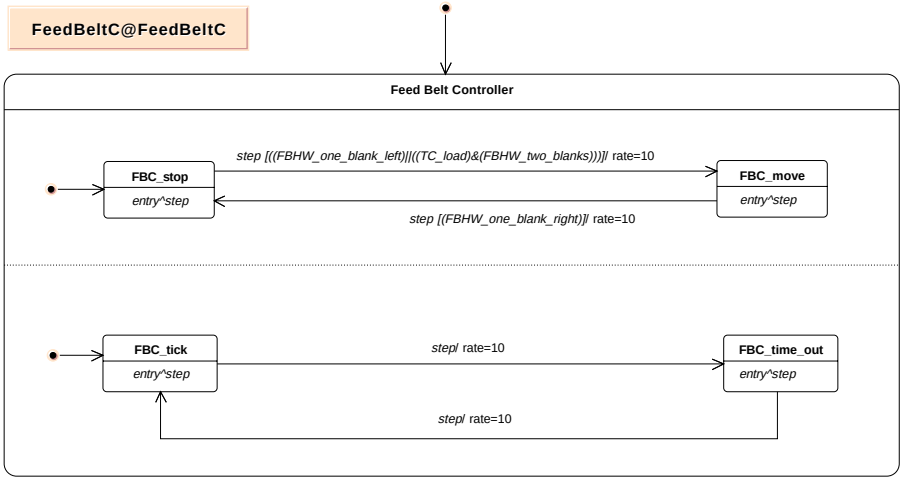

Figure 20: State chart of the feed belt controller

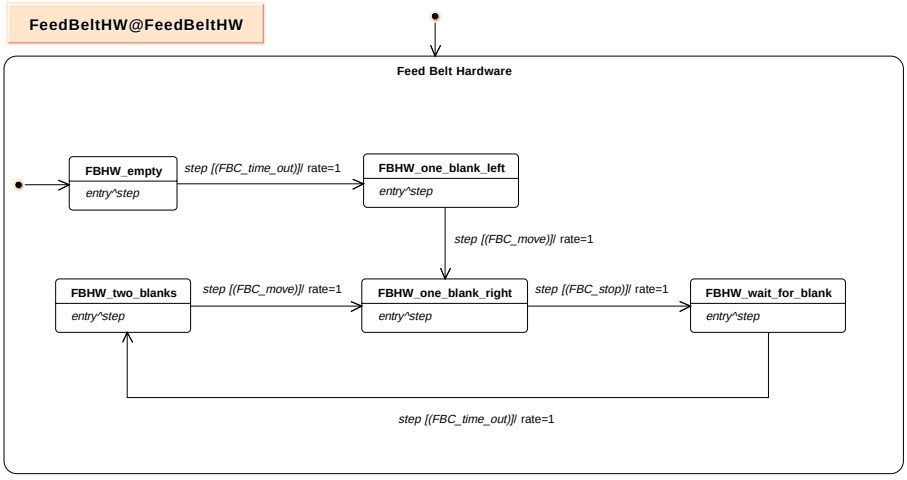

Figure 21: State chart of the feed belt hardware

Some states of the hardware sub-statecharts represent sensors in the real system while some states in the controller sub-statechart represent the actuators. Through setting of the actuators (entering the state) a state change is caused in the hardware sub-statechart and vice versa. The guard of the transition is verified and if it evaluates to true the state changes. The guards are boolean expressions composed from state predicates of other sub-statecharts.

6.5.1 Feed belt

The behaviour of the feed belt is described by the statecharts FeedBeltC and FeedBeltHW. The first one models the controller of the software while the second one is modelling the hardware.

The feed belt is empty at the beginning (FBHW_empty). The worker puts a blank on the feed belt and the statechart goes in the state FBHW_one_blank_left. The controller verifies the guard and starts the electrical motor through entering the state FBC_move. If the blank attains the end of the belt the motor is stopped (FBC_stop). The worker puts another blank on the

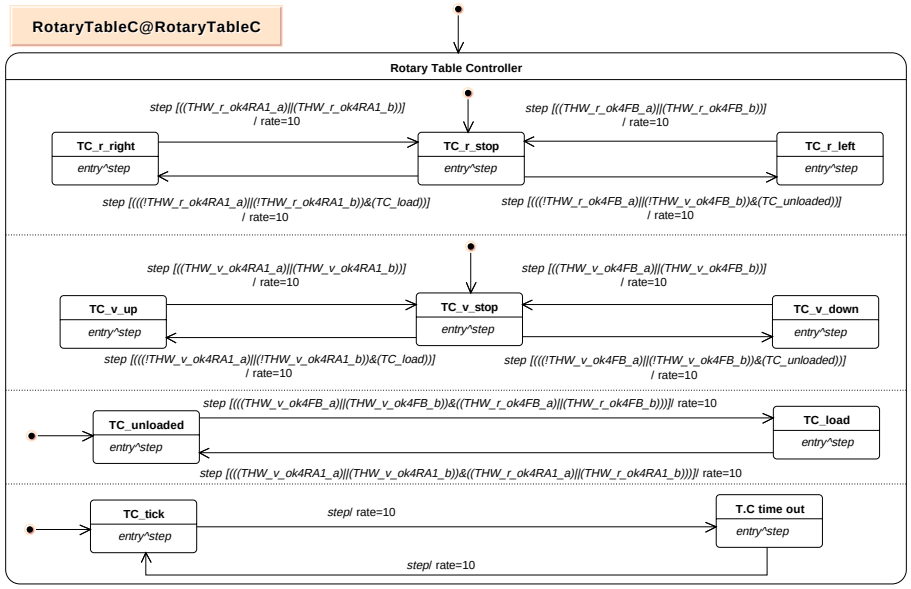

Figure 22: State chart of the rotary table controller

feed belt (FBHW_two_blanks). If the feed belt is able to load the rotary table the motor starts again. Therefore a communication between the controllers of the feed belt and the rotary table is necessary. The rotary table must be in the right position for loading the blank. After loading the rotary table the worker has to put another blank on the belt and the system waits to load the rotary table again.

6.5.2 Rotary table

RotaryTableC and RotaryTableHW describe the behaviour of the rotary table. If the rotary table is empty it moves down and turns right to load a blank from the feed belt. These movements are described by sub-statecharts in the controller- and the hardware-statechart. After loading the blank it moves up and turns left to be in the right position for the robot. Thus a communication between the rotary table, the robot (RC_RA1_loaded) and the feed belt (FBC_can_load) is necessary.

6.5.3 Robot

The robot is the most complex component of the modelled system. The statecharts RobotC and RobotHW describe the behaviour of the robot. To simplify our model we allow the robot to be in one of the three positions depicted in Figure 2. In the first position RA1 is over the rotary table and RA2 is in front of press 1. In position two RA1 is in front of press1 and RA2 is in front of press 2. In the last position RA1 is in front of press 2. These positions are the states of the RobotHW statechart.

Because of the model's comprehensibility we don't consider the loading/unloading and the extend/retract process of the robot arm.

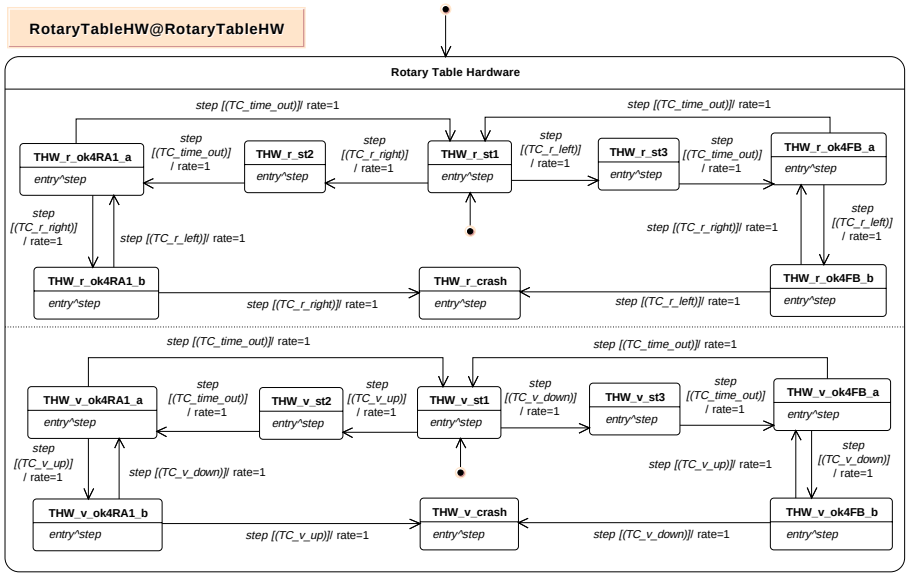

Figure 23: State chart of the rotary table hardware

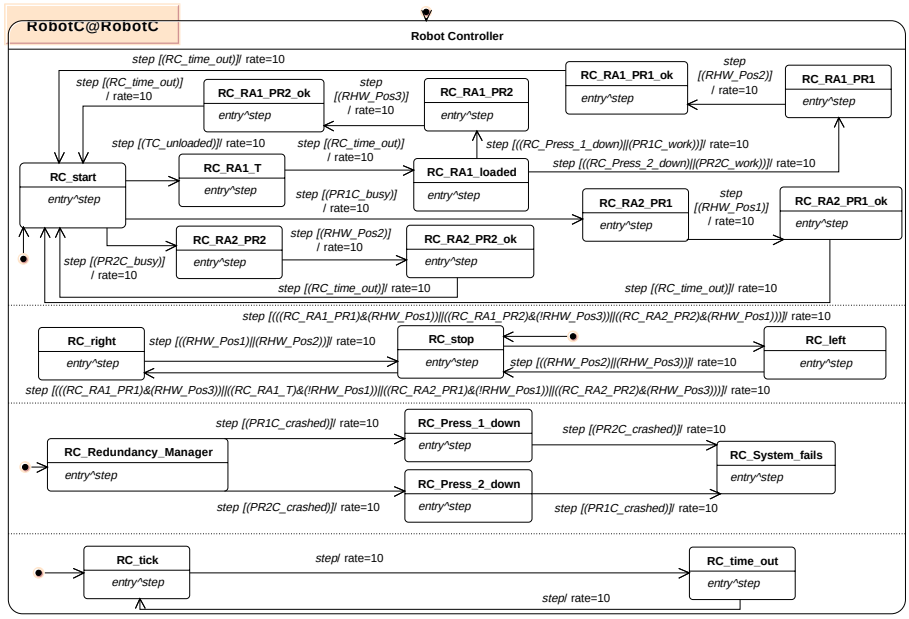

Figure 24: State chart of the robot controller

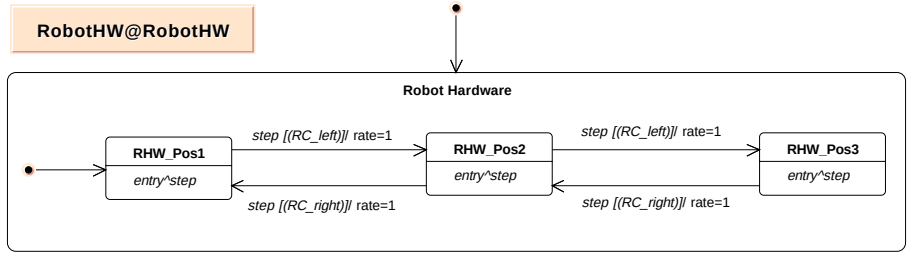

Figure 25: State chart of the robot hardware

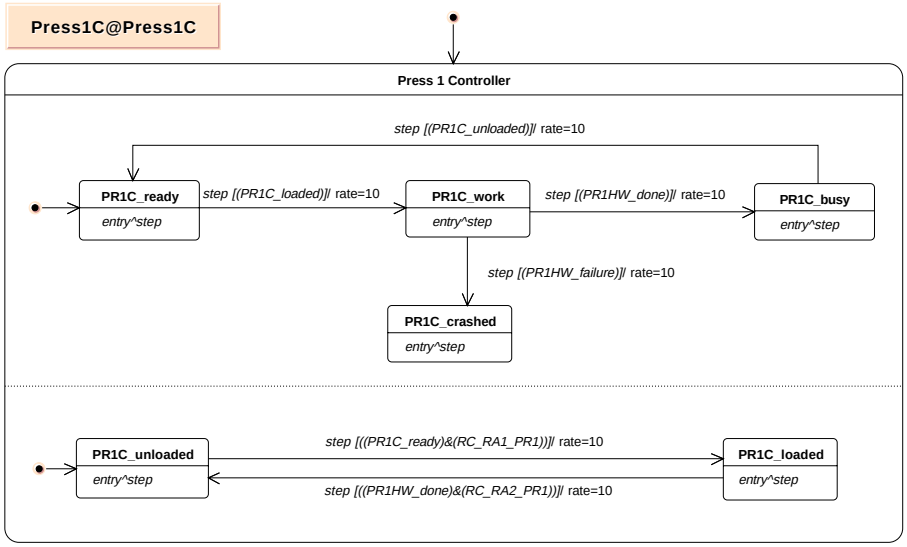

Figure 26: State chart of the press1 controller

The RobotC statechart consists of four sub-statecharts. We have a clock for setting time-outs and a sub-statechart for activating the movements of the robots (RC_stop, RC_left, RC_right). The most interesting sub-statecharts are these for the redundancy manager and the motion manager.

The motion manager indicates if the robot picks up a blank from the rotary table or if he loads/unloads a press. The statechart is non deterministic in the case that one press has forged a blank and waits to be unloaded by the RA2 and at the same time the rotary table is in the right position to be unloaded by RA1. It is not specified whether if the robot unloads the rotary table or the press first.

The redundancy manager provides important information for the motion manager. If a press is crashed the redundancy manager enters the state RC_Press_1_down or RC_Press_2_down. These states are parts of the guards of the motion manager transitions and the robot doesn't try to load or to unload the crashed press avoiding a dead-lock.

The controller of the robot communicates with the presses and the rotary table.

6.5.4 Presses

The dynamic behaviour of the presses is described by the statecharts Press1C, Press1HW for press 1 and Press2C, Press2HW for press 2. Press1C (Press2C) is the statechart for the controller of the software. Press1HW (Press2HW) is the statechart for the hardware.

If we consider the Press1C statechart we can see that there is a communication between the controllers of the press and the robot. This is necessary for loading/unloading the press and we have to check if the robot is in the right position for loading the press through the guard ((PR1C_ready)&(RC_RA1_PR1_ok)).

The other sub-statechart shows the current state of the press to the other components of the system (e.g. PR1C_work means that press 1 is working, PR1C_busy means that the blank

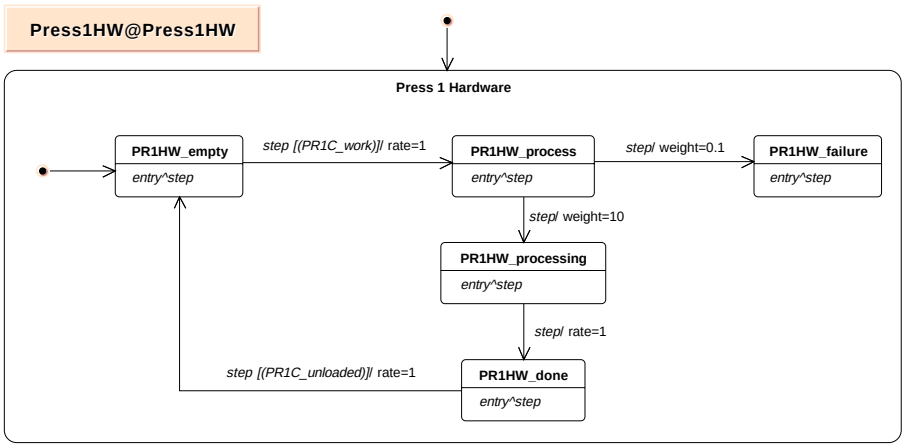

Figure 27: State chart of the press1 hardware

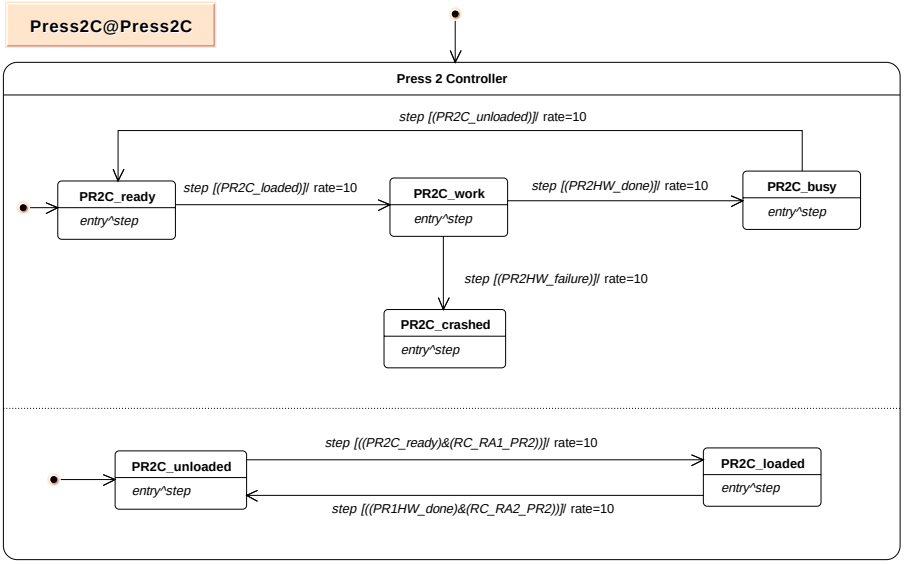

Figure 28: State chart of the press2 controller

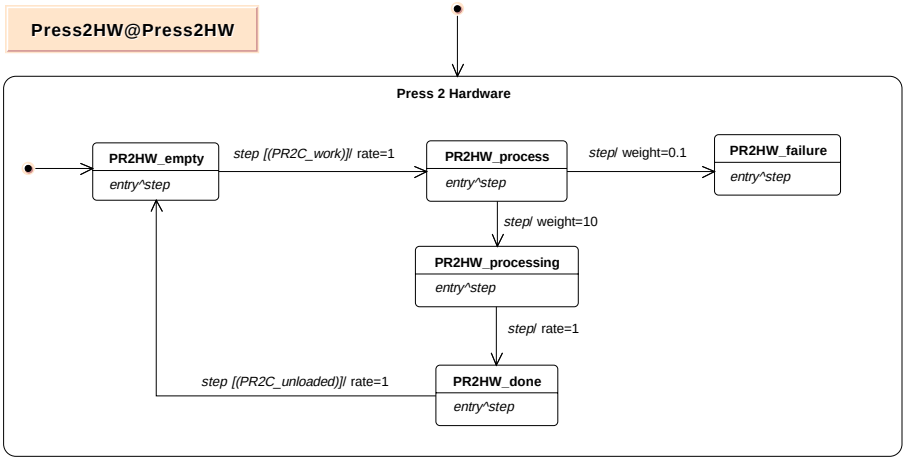

Figure 29: State chart of the press2 hardware

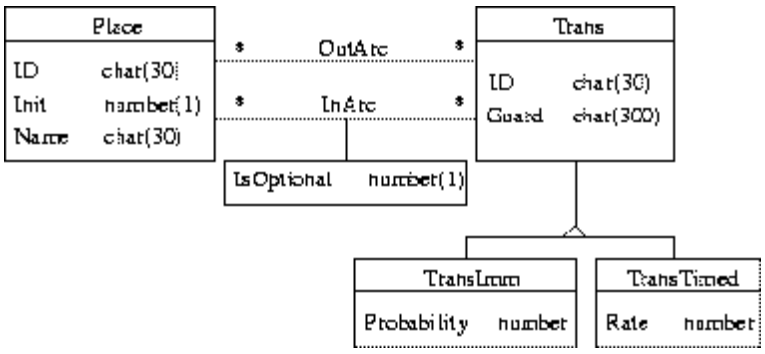

Figure 30: SQL model of the Petri net

is forged and the press is waiting to be unloaded, PR1C_crashed shows the crash of the press). Attention should be paid to the following states of the sub-statechart modelling the hardware. After entering the state PR1HW_process there are two possibilities. The press works correct and forges the blank (PR1HW_processing) or the press crashes (PR1HW_failure). We specify a weights (i.e. branching probabilities) for both transitions.

6.6 Implementation of the transformations

6.6.1 General

We transform several parts of the UML models to Petri-nets (PNs) to analyse numerically these parts with a Petri-net tool. To be more exact, first we transform the UML model described in a SQL database to PNs described in an other SQL database, then we generate a tool-specific description of the second database. Several descriptions in several description languages can be generated for several PN tools.

The input of the transformation is the UML model described in a SQL data model as exported by the tool Innovator. The SQL data model of the (output) Petri-net is:

This model of PNs is tool-independent, however it supposes a PN-tool knowing immediate and timed transitions, guarded transitions and an arbitrary technique to make arcs optional. The PN-tool has to give to immediate transitions a higher priority than to timed ones.

6.6.2 Statecharts to Petri-nets

First of all the transformation eliminates all of the states containing diagrams or other states. We transform in this phase only a restricted class of Statecharts (the so-called Guarded Statecharts), so all of these states contain no information. All of the base states (containing no diagrams or states) are transformed to PN places. The semantic of the PN model is that a place (corresponding to a state of the Statechart) has a token if and only if the given state is active. The Places corresponding to states with initial transitions leading to them have Init=1, other Places have Init=0. The ID and the Name of these Places corresponds to the ID and the name of the states of the Statechart (SC), respectively.

In a second step we transform all of the transitions between base states of the SC to transitions of the PN. Every transition (with its guard) is a Trans.

The original UML model contains no timing information, but we need rates and probabilities of timed and immediate transitions. The modeller has to assign these parameters while creating the UML model (only timed transitions can have guards). It would be a good way to label the transitions with these parameters but the tool Innovator makes it not possible yet. In the future we count on this feature of the tool, but now we have assigned these parameters as "actions" of the transitions. This temporary solution violates the UML rules but it is necessary for handling timing information. When changing the tool the program code of the transformation can be changed appropriately, it has no effect on the transformation itself.

The timed transitions have an exponential firing function with the given parameter. The firing function can be some of others too (e.g. PANDA accepts following functions: exponential, erlang-k, gamma, weibull, normal, lognormal, beta, triangular, deterministic, hyperexponential, k-stage hyperexponential, uniform and cox). In this phase we have used only exponential functions. The immediate transitions have firing "probabilities". We call them weights, because they can have values over 1. In general we do not need to restrict the weights to the interval (0,1), they assign only relative recurrence.

The timing parameter of each transition is examined. If the transformation has no assigned timing parameter, then the default parameter describes an immediate transition with weight 1. Timed transitions are described as TransTimeds, immediate ones as TransImms. To each transition of the SC correspond an InArc and an OutArc they are created also.

In order to be able to model communication failures of embedded systems in the future, the guards are processed once more. The guards of the Guarded SCs contain statements over states being active or not. In our modelling technique these guards implement implicit communication by checking actor or sensor states. In our failure model actor and sensor signal can be corrupted, that is a guard can see a state being active as inactive and vice versa. For this purpose we duplicate the places corresponding to public states, one place for modelling whether the state is active or not and another place for modelling whether the state seems to be active or not. When duplicating places the according InArcs and OutArcs are also duplicated. The duplicate of the InArcs are optional.

There are 4 cases for a duplicated place:

1. Both of the places are empty (the according state of the SC is inactive and the guards of the transitions see it so) The transitions "after" the places can not fire.
2. Both of the places have tokens (the according state of the SC is active and the guards of the transitions see it as such). The transactions "after" the places can fire. When firing they take both tokens from the places.
3. Only the original place has a token (the according state of the SC is active, but the guards of the transitions see it as it were inactive, e.g. the sensor is passive, but there is a sensor signal sent). This case can only occur if the token of the duplicate place was removed by an error (e.g. by a transition modelling failures)

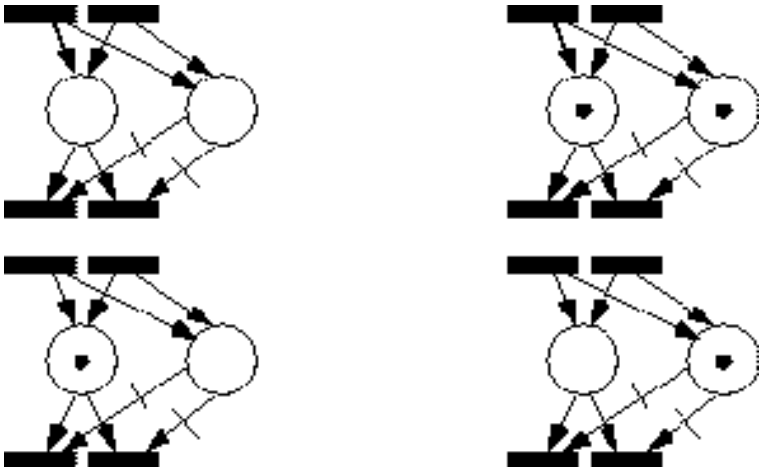

Figure 31: The Petri net model of a state chart

Figure 32: The Petri net model of a sequence diagram

The transactions "after" the places can fire, because if the duplicate place has no token, then there is no InArc from it to the transitions. (These arc are optional, that is they are only there if there is a token in the duplicate place.) When firing the transitions get the token from the original place.

4. Only the duplicate place has token (the according state of the SC is inactive, but the guards of the transitions see it as it were active, e.g. the sensor is active, but there is no sensor signal sent) This case can only ensue if the token of the duplicate place was produced by an error (e.g. by a transition modelling failures).
   The transactions "after" the places can not fire, because there are always InArcs from the original place to the transitions. Shortly the transitions of a component of the model can fire if and only if the according original places have tokens. Discrepancies in the state of the original and duplicate states can only be ensued by extra constructions modelling failures.

6.6.3 Sequence Diagrams to Petri-nets

First of all we transform each messages to a place with Init 0, a timed transition with the firing rate of the message, an immediate transition with weight 1, and two arcs: an OutArc leading from the timed transition to the place and an InArc leading from the place to the immediate transition.

The messages are ordered by the "Precedence" relation of the UML model, so we can take the first (an then always the next) transition. We always keep count of the "last" place of each

object. Each object of the Sequence Diagram (SqD) is transformed to a place with Init 1, and initially they are the last places of their objects.

Then we take the first message and its sender and receiver objects. We make an InArc leading from the last place of the sender to the timed transition of the message and another from the last place of the receiver to the immediate transition of the message. We create two new places and make an OutArc from the timed transition to the one (it become the last place of the sender) an another OutArc from the immediate transition to the other place (it become the last place of the receiver). We connect the other messages in an order defined by the "Precedence" relation.

In the PN model of the SqD there are no guards and no optional InArcs. The communication failures can be modelled the same way as in the SCs. The tokens of the places of the messages can be removed with a given rate or injected with a given weight externally.

6.6.4 CSPL

The CSPL [K. S. Trivedi: SPNP User's Manual, Duke University, 1998] is the input language of the PN tool PANDA. It is a C-like language for describing Stochastic Petri-nets. By the help of C-functions It allow to describe very complex structures. The main element of the language we used:

`place("<p_name>")` Definition of a place with the ID ¡p_name¿.

`init("<p_name>", <#tokens>)` Definition of the initial marking of the giving place. ¡#tokens¿ is an integer.

`trans("<t_name>")` Definition of a transition with the ID ¡t_name¿.

`enabling("<t_name>", <function name>)` Linking a previously described function to a given C-transition. The transition can only fire if the function returns TRUE.

`iarc("<t_name>", "<p_name>")` Definition of an inarc leading from a given place to given transition.

`viarc("<t_name>", "<p_name>", <function name>)` Definition of an inarc with variable multiplicity leading from a given place to given transition. The multiplicity of the arc is computed by the help of the given C-function described previously.

`oarc("<t_name>", "<p_name>")` Definition of an outarc leading from a given transition to a given place.

`probval("<t_name>", <weight>)` Definition of the weight of an immediate transition. ¡weight¿ is a real.

`rateval("<t_name>", <param>)` Definition of the rate of an (exponential) timed transition. ¡param¿ is a real. In the C-functions (enabling functions of transitions and multiplicity functions of arcs) we used the following predefined function:

`mark(<p_name>)` Actual number of tokens in a given place.

6.6.5 Quantitative Results

We performed some measurement with the transformed model, the two transformations described above, the PN-tool PANDA and the available computers, which provided very useful experiences.

The transformed SC model ("Production Cell") has 133 places, 98 transitions (15 unguarded and 83 guarded), 155 outarcs and 155 inarcs (inclusive 57 optional ones). This PN can reach 374.544 states. (The product of the number of states of the several components of the SC is over 197 075 Millions, but the parallel components are very strongly coupled by the guards.) Computing and storing the reachability graph of this (not very complex) UML model needs a special computer environment. Actually we used a central parallel computer of the FAU (Convex: 8 HP PA-RISC 7200 processors, 895 MB memory, 4 GB disk space, 2 GFLOP/s computing power), but even a little more realistic model would cause problems in computing. In the future we have, most probably, to concentrate our quantitative analysis on components of the SCs.

With PANDA

- we can detect absorbing states of the system (states, which can be entered but can not be exited),
- we can determine the number of the reachable states of the system
- we can determine the expected number of firing of a given transition of the SC (from the beginning till arbitrary time points),
- the expected time the system spend in a given state of the SC (from the beginning till arbitrary time points), etc.

The number of firing of a given transition is the throughput of the (sub)system and the time spent in a given state can help in detecting bottlenecks in the system.

Usually the transformed SqDs are small (Compare: the SqD "produce_blank_1". It has 25 places, 14 transitions, 20 outarcs, 20 inarcs and 12 reachable states). They can be analysed without any problem. We can determine whether a scenario can work or not (deadlocks), and we can determine the probability, that the scenario ends up in a given time. For example, the scenario of the above SqD ends up earlier than time X with the probability Y (Practically, this is the cumulative distribution function of the time it take to process one blank).

6.7 Model verification

The dynamic behavior of a model described by state charts and designed in UML can be automatically verified using the SPIN verification tool. The validity and the process of this verification will be shown in the example of the production cell.

A file written in Promela, i.e. in the verification language of SPIN can automatically be generated out of the dynamic model.

The state chart describing the dynamic behavior of the system is the Production Cell. Within that there are the following parallel sub-state charts: FeedBeltC, FeedBeltHW, RotaryTableC, RotaryTableHW, RobotC, RobotHW, Press1C, Press1HW, Press2C, and Press2HW.

6.7.1 The SPIN code

The following Promela code is automatically generated out of the above-described model. The skeleton of the code is presented the appendix, the whole code being too large for full presentation.

6.7.2 The Result of the Verification with SPIN

The model can either be simulated or verified by SPIN. First, having chosen verification, SPIN will roam the state space of the model up to the previously given steps. The process stops if a deadlock or a recurrent cycle is found in the state space. In our given example the reason for the halt was a recurrent cycle resulting in the following output:

```
(Spin Version 3.2.3 -- 1 August 1998)
+ Partial Order Reduction

Full statespace search for:
never-claim - (not selected)
assertion violations - (disabled by -A flag)
cycle checks - (disabled by -DSAFETY)
invalid endstates +

State-vector 512 byte, depth reached 19749, errors: 0
200 states, stored
1 states, matched
201 transitions (= stored+matched)
19524 atomic
steps hash conflicts: 0 (resolved)
(max size 2^21 states)

34.189 memory usage (Mbyte)

unreached in proctype queue
line 46, state 6, ''-end-''
(1 of 6 states)
unreached in proctype STEP
line 11610, state 216, ''(1)''
line 11612, state 220, ''(1)''
line 11619, state 225, ''Fire26 = 1''

...
```

As seen in the results, SPIN has found an invalid endstate and several code parts in the code describing the model that have not been treated during running. Based on this information we know that there is some kind of fault in the model, therefore we need a simulation in order to find it.

6.7.3 The Result of the Simulation with SPIN

Using the simulation option of SPIN the state transitions in the model can be tracked down and the reason for not having a satisfactory resultant state will be found. Omitting only the sate transitions in the result of the simulation the following is obtained:

```
396: [S52_T_C_v_stop = 0] 397: [S53_T_C_v_down = 1]
411: [S49_T_C_r_stop = 0] 412: [S50_T_C_r_left = 1]
423: [S45_T_C_tick_1 = 0] 424: [S46_T_C_tick_2 = 1]
459: [S22_FB_C_tick_1 = 0] 460: [S23_FB_C_tick_2 = 1]
606: [S71_R_C_start = 0] 607: [S72_R_C_RA1_T = 1]
642: [S57_R_C_tick_1 = 0] 643: [S58_R_C_tick_2 = 1]
1309: [S46_T_C_tick_2 = 0] 1310: [S47_T_C_time_out = 1]
1345: [S23_FB_C_tick_2 = 0] 1346: [S24_FB_C_time_out = 1]
1438: [S89_T_HW_v_st1 = 0] 1439: [S84_T_HW_v_st3 = 1]
1456: [S92_T_HW_r_st1 = 0] 1457: [S0_T_HW_r_st3 = 1]
1531: [S58_R_C_tick_2 = 0] 1532: [S59_R_C_time_out = 1]
2148: [S84_T_HW_v_st3 = 0] 2149: [S86_T_HW_v_ok4FB_a = 1]
2199: [S47_T_C_time_out = 0] 2200: [S45_T_C_tick_1 = 1]
2235: [S24_FB_C_time_out = 0] 2236: [S22_FB_C_tick_1 = 1]
2262: [S39_FB_HW_empty = 0] 2263: [S40_FB_HW_one_blank_left = 1]
2349: [S0_T_HW_r_st3 = 0] 2350: [S3_T_HW_r_ok4FB_a = 1]
2388: [S72_R_C_RA1_T = 0] 2389: [S81_R_C_RA1_loaded = 1]
2424: [S59_R_C_time_out = 0] 2425: [S57_R_C_tick_1 = 1]
3034: [S3_T_HW_r_ok4FB_a = 0] 3035: [S1_T_HW_r_ok4FB_b = 1]
3073: [S53_T_C_v_down = 0] 3074: [S52_T_C_v_stop = 1]
3088: [S50_T_C_r_left = 0] 3089: [S49_T_C_r_stop = 1]
3094: [S45_T_C_tick_1 = 0] 3095: [S46_T_C_tick_2 = 1]
3106: [S54_T_C_unloaded = 0] 3107: [S55_T_C_load = 1]
3133: [S22_FB_C_tick_1 = 0] 3134: [S23_FB_C_tick_2 = 1]
3145: [S6_FB_C_stop = 0] 3146: [S7_FB_C_move = 1]
3217: [S86_T_HW_v_ok4FB_a = 0] 3218: [S87_T_HW_v_ok4FB_b = 1]
3319: [S57_R_C_tick_1 = 0] 3320: [S58_R_C_tick_2 = 1]
3970: [S52_T_C_v_stop = 0] 3971: [S51_T_C_v_up = 1]
3985: [S49_T_C_r_stop = 0] 3986: [S48_T_C_r_right = 1]
3997: [S46_T_C_tick_2 = 0] 3998: [S47_T_C_time_out = 1]
4033: [S23_FB_C_tick_2 = 0] 4034: [S24_FB_C_time_out = 1]
4066: [S40_FB_HW_one_blank_left = 0] 4067: [S42_FB_HW_one_blank_right = 1]
4216: [S58_R_C_tick_2 = 0] 4217: [S59_R_C_time_out = 1]
4883: [S47_T_C_time_out = 0] 4884: [S45_T_C_tick_1 = 1]
4919: [S24_FB_C_time_out = 0] 4920: [S22_FB_C_tick_1 = 1]
4943: [S7_FB_C_move = 0] 4944: [S6_FB_C_stop = 1]
5003: [S87_T_HW_v_ok4FB_b = 0] 5004: [S86_T_HW_v_ok4FB_a = 1]
5042: [S1_T_HW_r_ok4FB_b = 0] 5043: [S3_T_HW_r_ok4FB_a = 1]
5108: [S59_R_C_time_out = 0] 5109: [S57_R_C_tick_1 = 1]
5762: [S45_T_C_tick_1 = 0] 5763: [S46_T_C_tick_2 = 1]
5798: [S22_FB_C_tick_1 = 0] 5799: [S23_FB_C_tick_2 = 1]
5840: [S42_FB_HW_one_blank_right = 0] 5841: [S43_FB_HW_wait_for_blank = 1]
5981: [S57_R_C_tick_1 = 0] 5982: [S58_R_C_tick_2 = 1]
6637: [S46_T_C_tick_2 = 0] 6638: [S47_T_C_time_out = 1]
6673: [S23_FB_C_tick_2 = 0] 6674: [S24_FB_C_time_out = 1]
6853: [S58_R_C_tick_2 = 0] 6854: [S59_R_C_time_out = 1]
7453: [S3_T_HW_r_ok4FB_a = 0] 7454: [S92_T_HW_r_st1 = 1]
7516: [S47_T_C_time_out = 0] 7517: [S45_T_C_tick_1 = 1]
7552: [S24_FB_C_time_out = 0] 7553: [S22_FB_C_tick_1 = 1]
7591: [S43_FB_HW_wait_for_blank = 0] 7592: [S41_FB_HW_two_blanks = 1]
7627: [S86_T_HW_v_ok4FB_a = 0] 7628: [S89_T_HW_v_st1 = 1]
```

7738: [S59_R_C_time_out = 0] 7739: [S57_R_C_tick_1 = 1]
8396: [S45_T_C_tick_1 = 0] 8397: [S46_T_C_tick_2 = 1]
8432: [S22_FB_C_tick_1 = 0] 8433: [S23_FB_C_tick_2 = 1]
8444: [S6_FB_C_stop = 0] 8445: [S7_FB_C_move = 1]
8534: [S89_T_HW_v_st1 = 0] 8535: [S85_T_HW_v_st2 = 1]
8546: [S92_T_HW_r_st1 = 0] 8547: [S93_T_HW_r_st2 = 1]
8621: [S57_R_C_tick_1 = 0] 8622: [S58_R_C_tick_2 = 1]
9285: [S46_T_C_tick_2 = 0] 9286: [S47_T_C_time_out = 1]
9321: [S23_FB_C_tick_2 = 0] 9322: [S24_FB_C_time_out = 1]
9357: [S41_FB_HW_two_blanks = 0] 9358: [S42_FB_HW_one_blank_right = 1]
9504: [S58_R_C_tick_2 = 0] 9505: [S59_R_C_time_out = 1]
10119: [S85_T_HW_v_st2 = 0] 10120: [S90_T_HW_v_ok4RA1_a = 1]
10167: [S47_T_C_time_out = 0] 10168: [S45_T_C_tick_1 = 1]
10203: [S24_FB_C_time_out = 0] 10204: [S22_FB_C_tick_1 = 1]
10227: [S7_FB_C_move = 0] 10228: [S6_FB_C_stop = 1]
10314: [S93_T_HW_r_st2 = 0] 10315: [S4_T_HW_r_ok4RA1_a = 1]
10389: [S59_R_C_time_out = 0] 10390: [S57_R_C_tick_1 = 1]
11004: [S4_T_HW_r_ok4RA1_a = 0] 11005: [S2_T_HW_r_ok4RA1_b = 1]
11028: [S51_T_C_v_up = 0] 11029: [S52_T_C_v_stop = 1]
11043: [S48_T_C_r_right = 0] 11044: [S49_T_C_r_stop = 1]
11058: [S45_T_C_tick_1 = 0] 11059: [S46_T_C_tick_2 = 1]
11073: [S55_T_C_load = 0] 11074: [S54_T_C_unloaded = 1]
11097: [S22_FB_C_tick_1 = 0] 11098: [S23_FB_C_tick_2 = 1]
11139: [S42_FB_HW_one_blank_right = 0] 11140: [S43_FB_HW_wait_for_blank = 1]
11205: [S90_T_HW_v_ok4RA1_a = 0] 11206: [S88_T_HW_v_ok4RA1_b = 1]
11283: [S57_R_C_tick_1 = 0] 11284: [S58_R_C_tick_2 = 1]
11930: [S52_T_C_v_stop = 0] 11931: [S53_T_C_v_down = 1]
11945: [S49_T_C_r_stop = 0] 11946: [S50_T_C_r_left = 1]
11960: [S46_T_C_tick_2 = 0] 11961: [S47_T_C_time_out = 1]
11996: [S23_FB_C_tick_2 = 0] 11997: [S24_FB_C_time_out = 1]
12176: [S58_R_C_tick_2 = 0] 12177: [S59_R_C_time_out = 1]
12779: [S2_T_HW_r_ok4RA1_b = 0] 12780: [S4_T_HW_r_ok4RA1_a = 1]
12845: [S47_T_C_time_out = 0] 12846: [S45_T_C_tick_1 = 1]
12881: [S24_FB_C_time_out = 0] 12882: [S22_FB_C_tick_1 = 1]
12920: [S43_FB_HW_wait_for_blank = 0] 12921: [S41_FB_HW_two_blanks = 1]
12968: [S88_T_HW_v_ok4RA1_b = 0] 12969: [S90_T_HW_v_ok4RA1_a = 1]
13067: [S59_R_C_time_out = 0] 13068: [S57_R_C_tick_1 = 1]
13720: [S45_T_C_tick_1 = 0] 13721: [S46_T_C_tick_2 = 1]
13756: [S22_FB_C_tick_1 = 0] 13757: [S23_FB_C_tick_2 = 1]
13936: [S57_R_C_tick_1 = 0] 13937: [S58_R_C_tick_2 = 1]
14588: [S46_T_C_tick_2 = 0] 14589: [S47_T_C_time_out = 1]
14624: [S23_FB_C_tick_2 = 0] 14625: [S24_FB_C_time_out = 1]
14804: [S58_R_C_tick_2 = 0] 14805: [S59_R_C_time_out = 1]
15408: [S4_T_HW_r_ok4RA1_a = 0] 15409: [S92_T_HW_r_st1 = 1]
15465: [S47_T_C_time_out = 0] 15466: [S45_T_C_tick_1 = 1]
15501: [S24_FB_C_time_out = 0] 15502: [S22_FB_C_tick_1 = 1]
15597: [S90_T_HW_v_ok4RA1_a = 0] 15598: [S89_T_HW_v_st1 = 1]
15684: [S59_R_C_time_out = 0] 15685: [S57_R_C_tick_1 = 1]
16339: [S45_T_C_tick_1 = 0] 16340: [S46_T_C_tick_2 = 1]
16375: [S22_FB_C_tick_1 = 0] 16376: [S23_FB_C_tick_2 = 1]
16471: [S89_T_HW_v_st1 = 0] 16472: [S84_T_HW_v_st3 = 1]
16489: [S92_T_HW_r_st1 = 0] 16490: [S0_T_HW_r_st3 = 1]
16561: [S57_R_C_tick_1 = 0] 16562: [S58_R_C_tick_2 = 1]
17220: [S46_T_C_tick_2 = 0] 17221: [S47_T_C_time_out = 1]
17256: [S23_FB_C_tick_2 = 0] 17257: [S24_FB_C_time_out = 1]
17436: [S58_R_C_tick_2 = 0] 17437: [S59_R_C_time_out = 1]

The state transitions in the original UML Statecharts show that they occur cyclically in the

following sub-statecharts:

- Feed Belt Controller, Rotary Table Controller and Robot Controller tick1 - tick2 - timeout,
- Rotary Table Controller unloaded - load,
- Rotary Table Controller v-stop - v-down - v-stop - v-up - v-stop and r-stop - r-left - r-stop - r-right - r-stop,
- Rotary Table Hardware r_st1 - r_st3 - r_ok4FB_a - r_ok4FB_b - r_ok4FB_a - r_st1 and v_st1 - v_st3 - v_ok4FB_a - v_ok4FB_b - v_ok4FB_a - v_st1,
- Feed Belt Controller stop - move,
- In Feed Belt Hardware after having reached state one_blank_right: one_blank_right - wait_for_blank - two_blanks.

Starting from state start the Robot Controller reaches state RA1 and from there state RA1 loaded.

What can be stated altogether is that none of the presses react, however the feed belt starts operating and the rotary table turns to the appropriate position. The robot arm senses as if holding a sheet, while in reality it holds nothing, which plate it cannot then put down from the feed belt onto the rotary table. Therefore, the system is in a quasi dead-lock.

A Promela code

```
/* Bits for states */
bit S0_T_HW_r_st3, S1_T_HW_r_ok4FB_b,
S2_T_HW_r_ok4RA1_b, S3_T_HW_r_ok4FB_a, S4_T_HW_r_ok4RA1_a,
S5_Feed_Belt_Controller, S6_FB_C_stop, S7_FB_C_move,
S8_Press_1_Controller, S9_PR1_C_ready, S10_PR1_C_work,
S11_PR1_C_busy, S12_Presse_2_Controller, S13_PR1_C_crashed,
S14_PR1_C_unloaded, S15_PR1_C_loaded, S16_PR2_C_unloaded,
S17_PR2_C_loaded, S18_PR2_C_ready, S19_PR2_C_work,
S20_PR2_C_busy, S21_PR2_C_crashed, S22_FB_C_tick_1,
S23_FB_C_tick_2, S24_FB_C_time_out, S25_Production_Cell,
S26_PR2_HW_empty, S27_PR2_HW_processing, S28_PR2_HW_failure,
S29_PR2_HW_done, S30_PR2_HW_process, S31_Press_2_Hardware,
S32_Press_1_Hardware, S33_PR1_HW_failure, S34_PR1_HW_process,
S35_PR1_HW_processing, S36_PR1_HW_done, S37_PR1_HW_empty,
S38_Feed_Belt_Hardware, S39_FB_HW_empty,
S40_FB_HW_one_blank_left, S41_FB_HW_two_blanks,
S42_FB_HW_one_blank_right, S43_FB_HW_wait_for_blank,
S44_Rotary_Table_Controller, S45_T_C_tick_1, S46_T_C_tick_2,
S47_T_C_time_out, S48_T_C_r_right, S49_T_C_r_stop,
S50_T_C_r_left, S51_T_C_v_up, S52_T_C_v_stop,
S53_T_C_v_down, S54_T_C_unloaded, S55_T_C_load,
S56_Robot_Controller, S57_R_C_tick_1, S58_R_C_tick_2,
S59_R_C_time_out, S60_R_C_Redundancy_Manager,
S61_R_C_Press_1_down, S62_R_C_Press_2_down,
S63_R_C_System_fails, S64_R_C_right, S65_R_C_stop,
S66_R_C_left, S67_Robot_Hardware, S68_R_HW_Pos1, S69_R_HW_Pos2,
S70_R_HW_Pos3, S71_R_C_start, S72_R_C_RA1_T, S73_R_C_RA2_PR2,
S74_R_C_RA2_PR1, S75_R_C_RA1_PR2, S76_R_C_RA1_PR1,
S77_R_C_RA2_PR2_ok, S78_R_C_RA2_PR1_ok, S79_R_C_RA1_PR1_ok,
S80_R_C_RA1_PR2_ok, S81_R_C_RA1_loaded,
```

```
S82_Rotary_Table_Hardware, S83_T_HW_v_crash, S84_T_HW_v_st3,
S85_T_HW_v_st2, S86_T_HW_v_ok4FB_a, S87_T_HW_v_ok4FB_b,
S88_T_HW_v_ok4RA1_b, S89_T_HW_v_st1, S90_T_HW_v_ok4RA1_a,
S91_T_HW_r_crash, S92_T_HW_r_st1, S93_T_HW_r_st2;

/* Bits for transition priorities */

bit P0, P1, P2, P3, P95, P96, P97;

/* Bits for transition selections */

bit Sel0, Sel1, Sel2, Sel3, Sel95, Sel96, Sel97;

/* Bits for transition firing */

bit Fire0, Fire1, Fire2, Fire3, Fire95, Fire96, Fire97;

/* Channels */

chan to_stmachine = [0] of {int}
chan to_queue = [104] of {int}

/* Definition of priorities */

#define SM(x,y) (0)

/* Definitions of events */

#define e1_step 1

/* Event variable */

int Event;

proctype queue()
{
int e;
do
::
to_queue?e;
to_stmachine!e;
od
}

proctype STEP()
{
int e;
do
::
to_stmachine?e;
Event=e;
atomic
{
...
/* Automaton 12 */

/* Begin Automaton 12 progress*/
/* Transition 86 */
```

```
Sel86 =
/* Progress 3 */
S6_FB_C_stop & (Event==e1_step) &
((S40_FB_HW_one_blank_left)||%
((S55_T_C_load)&(S41_FB_HW_two_blanks)))

/* End Progress 3 */
&
/* Progress 4 */
!(P0 & SM(86,0))& !(P1 & SM(86,1))& !(P2 & SM(86,2))& !(P3 & SM(86,3))&
!(P4 & SM(86,4))& !(P5 & SM(86,5))& !(P96 & SM(86,96))& !(P97 & SM(86,97))
/* End Progress 4 */
&
/* Progress 5 */
/* End Progress 5 */
/* Progress 6 */
1
/* End Progress 6 */
/* End Transition 86 */
;
/* Transition 92 */
Sel92 =
/* Progress 3 */
S7_FB_C_move & (Event==e1_step) & (S42_FB_HW_one_blank_right)
/* End Progress 3 */
&
/* Progress 4 */
!(P0 & SM(92,0))& !(P1 & SM(92,1))& !(P96 & SM(92,96))& !(P97 & SM(92,97))
/* End Progress 4 */
&
/* Progress 5 */
/* End Progress 5 */
/* Progress 6 */
1
/* End Progress 6 */
/* End Transition 92 */
;
/* End Automaton 12 progress*/
/* Begin Automaton 12 composition */
/* Updating priorities with active transitions */
P86 = S6_FB_C_stop & (Event==e1_step) & ((S40_FB_HW_one_blank_left)||
((S55_T_C_load)&(S41_FB_HW_two_blanks)));
P92 = S7_FB_C_move & (Event==e1_step) & (S42_FB_HW_one_blank_right);
/* End Automaton 12 composition */
/* End Automaton 12 */
...
/* Selecting transition to fire */
/* Automaton 3 */
if
:: (Sel0 || Sel1 || Sel2 || Sel3 || Sel4 || Sel5 || Sel6 || Sel7 || Sel8
|| Sel9 || Sel10 || Sel11 || Sel12 || Sel13 || Sel14 || Sel15 || Sel16
|| Sel17 || Sel18 || Sel19 || Sel20 || Sel21 || Sel22 || Sel23 || Sel24
|| Sel25 || Sel26 || Sel27 || Sel28 || Sel29 || Sel30 || Sel31 || Sel32
|| Sel33 || Sel34 || Sel35 || Sel36 || Sel37 || Sel38 || Sel39 || Sel40
|| Sel41 || Sel42 || Sel43 || Sel44 || Sel45 || Sel46 || Sel47 || Sel48
|| Sel49 || Sel50 || Sel51 || Sel52 || Sel53 || Sel54 || Sel55 || Sel56
|| Sel57 || Sel58 || Sel59 || Sel60 || Sel61 || Sel62 || Sel63 || Sel64
|| Sel65 || Sel66 || Sel67 || Sel68 || Sel69 || Sel70 || Sel71 || Sel72
```

```
|| Sel73 || Sel74 || Sel75 || Sel76 || Sel77 || Sel78 || Sel79 || Sel80
|| Sel81 || Sel82 || Sel83 || Sel84 || Sel85 || Sel86 || Sel87 || Sel88
|| Sel89 || Sel90 || Sel91 || Sel92 || Sel93 || Sel94 || Sel95 || Sel96
|| Sel97 || 0) ->
/* Automaton 0 */
if
:: (Sel83 || Sel84 || Sel85 || Sel86 || Sel92 || 0) ->
/* Automaton 12 */
if
:: Sel86 -> Fire86=1;
:: Sel92 -> Fire92=1;
:: else -> skip;
fi;
/* Automaton 19 */
if
:: Sel83 -> Fire83=1;
:: Sel84 -> Fire84=1;
:: Sel85 -> Fire85=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 1 */
if
:: (Sel24 || Sel25 || Sel26 || Sel27 || Sel28 || Sel29 || 0) ->
/* Automaton 14 */
if
:: Sel26 -> Fire26=1;
:: Sel27 -> Fire27=1;
:: Sel28 -> Fire28=1;
:: Sel29 -> Fire29=1;
:: else -> skip;
fi;
/* Automaton 17 */
if
:: Sel24 -> Fire24=1;
:: Sel25 -> Fire25=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 10 */
if
:: (Sel36 || Sel37 || Sel38 || Sel39 || Sel40 || Sel41 || Sel42 || Sel43
|| Sel44 || Sel45 || Sel46 || Sel47 || Sel48 || Sel49 || Sel50 || Sel51
|| Sel52 || Sel53 || Sel54 || Sel55 || Sel56 || Sel57 || Sel58 || Sel59
|| 0) ->
/* Automaton 46 */
if
:: Sel46 -> Fire46=1;
:: Sel47 -> Fire47=1;
:: Sel48 -> Fire48=1;
:: Sel55 -> Fire55=1;
:: Sel50 -> Fire50=1;
:: Sel56 -> Fire56=1;
:: Sel49 -> Fire49=1;
:: Sel51 -> Fire51=1;
:: Sel52 -> Fire52=1;
```

```
:: Sel53 -> Fire53=1;
:: Sel54 -> Fire54=1;
:: Sel57 -> Fire57=1;
:: else -> skip;
fi;
/* Automaton 47 */
if
:: Sel43 -> Fire43=1;
:: Sel44 -> Fire44=1;
:: Sel45 -> Fire45=1;
:: Sel36 -> Fire36=1;
:: Sel39 -> Fire39=1;
:: Sel42 -> Fire42=1;
:: Sel58 -> Fire58=1;
:: Sel59 -> Fire59=1;
:: Sel37 -> Fire37=1;
:: Sel38 -> Fire38=1;
:: Sel40 -> Fire40=1;
:: Sel41 -> Fire41=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 2 */
if
:: (Sel77 || Sel78 || Sel79 || Sel80 || Sel81 || Sel82 || 0) ->
/* Automaton 16 */
if
:: Sel79 -> Fire79=1;
:: Sel80 -> Fire80=1;
:: Sel81 -> Fire81=1;
:: Sel82 -> Fire82=1;
:: else -> skip;
fi;
/* Automaton 18 */
if
:: Sel77 -> Fire77=1;
:: Sel78 -> Fire78=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 4 */
if
:: (Sel30 || Sel31 || Sel32 || Sel33 || Sel34 || 0) ->
/* Automaton 26 */
if
:: Sel32 -> Fire32=1;
:: Sel30 -> Fire30=1;
:: Sel31 -> Fire31=1;
:: Sel33 -> Fire33=1;
:: Sel34 -> Fire34=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 5 */
if
```

```
:: (Sel87 || Sel88 || Sel89 || Sel90 || Sel91 || 0) ->
/* Automaton 27 */
if
:: Sel87 -> Fire87=1;
:: Sel91 -> Fire91=1;
:: Sel88 -> Fire88=1;
:: Sel89 -> Fire89=1;
:: Sel90 -> Fire90=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 6 */
if
:: (Sel93 || Sel94 || Sel95 || Sel96 || Sel97 || 0) ->
/* Automaton 28 */
if
:: Sel94 -> Fire94=1;
:: Sel97 -> Fire97=1;
:: Sel93 -> Fire93=1;
:: Sel95 -> Fire95=1;
:: Sel96 -> Fire96=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 7 */
if
:: (Sel64 || Sel65 || Sel66 || Sel67 || Sel68 || Sel69 || Sel70 || Sel71
|| Sel72 || Sel73 || Sel74 || Sel75 || Sel76 || 0) ->
 /* Automaton 30 */
if
:: Sel70 -> Fire70=1;
:: Sel71 -> Fire71=1;
:: Sel68 -> Fire68=1;
:: Sel69 -> Fire69=1;
:: else -> skip;
fi;
/* Automaton 31 */
if
:: Sel73 -> Fire73=1;
:: Sel72 -> Fire72=1;
:: Sel74 -> Fire74=1;
:: else -> skip;
fi;
/* Automaton 32 */
if
:: Sel75 -> Fire75=1;
:: Sel76 -> Fire76=1;
:: else -> skip;
fi;
/* Automaton 33 */
if
:: Sel64 -> Fire64=1;
:: Sel65 -> Fire65=1;
:: Sel66 -> Fire66=1;
:: Sel67 -> Fire67=1;
:: else -> skip;
```

```
fi;
:: else -> skip;
fi;
/* Automaton 8 */
if
:: (Sel0 || Sel1 || Sel2 || Sel3 || Sel4 || Sel5 || Sel6 || Sel7 || Sel8
|| Sel9 || Sel10 || Sel11 || Sel12 || Sel13 || Sel14 || Sel15 || Sel16
|| Sel17 || Sel18 || Sel19 || Sel20 || Sel60 || Sel61 || Sel62 || Sel63
|| 0) ->
/* Automaton 35 */
if
:: Sel16 -> Fire16=1;
:: Sel6 -> Fire6=1;
:: Sel7 -> Fire7=1;
:: Sel8 -> Fire8=1;
:: Sel9 -> Fire9=1;
:: Sel10 -> Fire10=1;
:: Sel11 -> Fire11=1;
:: Sel12 -> Fire12=1;
:: Sel13 -> Fire13=1;
:: Sel14 -> Fire14=1;
:: Sel15 -> Fire15=1;
:: Sel17 -> Fire17=1;
:: Sel4 -> Fire4=1;
:: Sel5 -> Fire5=1;
:: else -> skip;
fi;
/* Automaton 36 */
if
:: Sel19 -> Fire19=1;
:: Sel18 -> Fire18=1;
:: Sel20 -> Fire20=1;
:: else -> skip;
fi;
/* Automaton 37 */
if
:: Sel61 -> Fire61=1;
:: Sel60 -> Fire60=1;
:: Sel62 -> Fire62=1;
:: Sel63 -> Fire63=1;
:: else -> skip;
fi;
/* Automaton 39 */
if
:: Sel0 -> Fire0=1;
:: Sel3 -> Fire3=1;
:: Sel1 -> Fire1=1;
:: Sel2 -> Fire2=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
/* Automaton 9 */
if
:: (Sel21 || Sel22 || Sel23 || Sel35 || 0) ->
/* Automaton 38 */
if
:: Sel35 -> Fire35=1;
```

```
:: Sel23 -> Fire23=1;
:: Sel21 -> Fire21=1;
:: Sel22 -> Fire22=1;
:: else -> skip;
fi;
:: else -> skip;
fi;
:: else -> skip;
fi;

/* Firing selected transitions */
if :: Fire52-> S2_T_HW_r_ok4RA1_b=0; S91_T_HW_r_crash=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire53-> S2_T_HW_r_ok4RA1_b=0; S4_T_HW_r_ok4RA1_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire54-> S3_T_HW_r_ok4FB_a=0; S92_T_HW_r_st1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire55-> S3_T_HW_r_ok4FB_a=0; S1_T_HW_r_ok4FB_b=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire56-> S4_T_HW_r_ok4RA1_a=0; S92_T_HW_r_st1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire57-> S4_T_HW_r_ok4RA1_a=0; S2_T_HW_r_ok4RA1_b=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire58-> S84_T_HW_v_st3=0; S86_T_HW_v_ok4FB_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire59-> S85_T_HW_v_st2=0; S90_T_HW_v_ok4RA1_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire60-> S60_R_C_Redundancy_Manager=0; S61_R_C_Press_1_down=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire61-> S60_R_C_Redundancy_Manager=0; S62_R_C_Press_2_down=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire62-> S61_R_C_Press_1_down=0; S63_R_C_System_fails=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire63-> S62_R_C_Press_2_down=0; S63_R_C_System_fails=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire64-> S51_T_C_v_up=0; S52_T_C_v_stop=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire65-> S52_T_C_v_stop=0; S53_T_C_v_down=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire66-> S52_T_C_v_stop=0; S51_T_C_v_up=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire67-> S53_T_C_v_down=0; S52_T_C_v_stop=1; to_queue!e1_step; skip
:: else->skip
```

```
fi;
if :: Fire68-> S48_T_C_r_right=0; S49_T_C_r_stop=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire69-> S49_T_C_r_stop=0; S50_T_C_r_left=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire70-> S49_T_C_r_stop=0; S48_T_C_r_right=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire71-> S50_T_C_r_left=0; S49_T_C_r_stop=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire72-> S45_T_C_tick_1=0; S46_T_C_tick_2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire73-> S46_T_C_tick_2=0; S47_T_C_time_out=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire74-> S47_T_C_time_out=0; S45_T_C_tick_1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire75-> S54_T_C_unloaded=0; S55_T_C_load=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire76-> S55_T_C_load=0; S54_T_C_unloaded=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire77-> S16_PR2_C_unloaded=0; S17_PR2_C_loaded=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire78-> S17_PR2_C_loaded=0; S16_PR2_C_unloaded=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire79-> S18_PR2_C_ready=0; S19_PR2_C_work=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire80-> S19_PR2_C_work=0; S20_PR2_C_busy=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire81-> S19_PR2_C_work=0; S21_PR2_C_crashed=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire82-> S20_PR2_C_busy=0; S18_PR2_C_ready=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire83-> S22_FB_C_tick_1=0; S23_FB_C_tick_2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire84-> S23_FB_C_tick_2=0; S24_FB_C_time_out=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire85-> S24_FB_C_time_out=0; S22_FB_C_tick_1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire86-> S6_FB_C_stop=0; S7_FB_C_move=1; to_queue!e1_step; skip
:: else->skip
fi;
```

```
if :: Fire87-> S34_PR1_HW_process=0; S35_PR1_HW_processing=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire88-> S34_PR1_HW_process=0; S33_PR1_HW_failure=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire89-> S35_PR1_HW_processing=0; S36_PR1_HW_done=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire90-> S36_PR1_HW_done=0; S37_PR1_HW_empty=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire91-> S37_PR1_HW_empty=0; S34_PR1_HW_process=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire92-> S7_FB_C_move=0; S6_FB_C_stop=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire93-> S39_FB_HW_empty=0; S40_FB_HW_one_blank_left=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire94-> S40_FB_HW_one_blank_left=0; S42_FB_HW_one_blank_right=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire95-> S41_FB_HW_two_blanks=0; S42_FB_HW_one_blank_right=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire96-> S42_FB_HW_one_blank_right=0; S43_FB_HW_wait_for_blank=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire97-> S43_FB_HW_wait_for_blank=0; S41_FB_HW_two_blanks=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire26-> S9_PR1_C_ready=0; S10_PR1_C_work=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire27-> S10_PR1_C_work=0; S13_PR1_C_crashed=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire28-> S10_PR1_C_work=0; S11_PR1_C_busy=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire29-> S11_PR1_C_busy=0; S9_PR1_C_ready=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire30-> S26_PR2_HW_empty=0; S30_PR2_HW_process=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire31-> S27_PR2_HW_processing=0; S29_PR2_HW_done=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire32-> S29_PR2_HW_done=0; S26_PR2_HW_empty=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire33-> S30_PR2_HW_process=0; S28_PR2_HW_failure=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire34-> S30_PR2_HW_process=0; S27_PR2_HW_processing=1; to_queue!e1_step; skip
```

```
:: else->skip
fi;
if :: Fire35-> S68_R_HW_Pos1=0; S69_R_HW_Pos2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire36-> S86_T_HW_v_ok4FB_a=0; S89_T_HW_v_st1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire37-> S86_T_HW_v_ok4FB_a=0; S87_T_HW_v_ok4FB_b=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire38-> S87_T_HW_v_ok4FB_b=0; S83_T_HW_v_crash=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire39-> S87_T_HW_v_ok4FB_b=0; S86_T_HW_v_ok4FB_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire40-> S88_T_HW_v_ok4RA1_b=0; S90_T_HW_v_ok4RA1_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire41-> S88_T_HW_v_ok4RA1_b=0; S83_T_HW_v_crash=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire42-> S89_T_HW_v_st1=0; S84_T_HW_v_st3=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire43-> S89_T_HW_v_st1=0; S85_T_HW_v_st2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire44-> S90_T_HW_v_ok4RA1_a=0; S89_T_HW_v_st1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire45-> S90_T_HW_v_ok4RA1_a=0; S88_T_HW_v_ok4RA1_b=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire46-> S92_T_HW_r_st1=0; S93_T_HW_r_st2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire47-> S92_T_HW_r_st1=0; S0_T_HW_r_st3=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire48-> S93_T_HW_r_st2=0; S4_T_HW_r_ok4RA1_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire49-> S0_T_HW_r_st3=0; S3_T_HW_r_ok4FB_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire50-> S1_T_HW_r_ok4FB_b=0; S91_T_HW_r_crash=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire51-> S1_T_HW_r_ok4FB_b=0; S3_T_HW_r_ok4FB_a=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire0-> S65_R_C_stop=0; S66_R_C_left=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire1-> S65_R_C_stop=0; S64_R_C_right=1; to_queue!e1_step; skip
:: else->skip
```

```
fi;
if :: Fire2-> S66_R_C_left=0; S65_R_C_stop=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire3-> S64_R_C_right=0; S65_R_C_stop=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire4-> S81_R_C_RA1_loaded=0; S76_R_C_RA1_PR1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire5-> S81_R_C_RA1_loaded=0; S75_R_C_RA1_PR2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire6-> S71_R_C_start=0; S74_R_C_RA2_PR1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire7-> S71_R_C_start=0; S72_R_C_RA1_T=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire8-> S71_R_C_start=0; S73_R_C_RA2_PR2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire9-> S72_R_C_RA1_T=0; S81_R_C_RA1_loaded=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire10-> S73_R_C_RA2_PR2=0; S77_R_C_RA2_PR2_ok=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire11-> S74_R_C_RA2_PR1=0; S78_R_C_RA2_PR1_ok=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire12-> S75_R_C_RA1_PR2=0; S80_R_C_RA1_PR2_ok=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire13-> S76_R_C_RA1_PR1=0; S79_R_C_RA1_PR1_ok=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire14-> S77_R_C_RA2_PR2_ok=0; S71_R_C_start=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire15-> S78_R_C_RA2_PR1_ok=0; S71_R_C_start=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire16-> S79_R_C_RA1_PR1_ok=0; S71_R_C_start=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire17-> S80_R_C_RA1_PR2_ok=0; S71_R_C_start=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire18-> S57_R_C_tick_1=0; S58_R_C_tick_2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire19-> S58_R_C_tick_2=0; S59_R_C_time_out=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire20-> S59_R_C_time_out=0; S57_R_C_tick_1=1; to_queue!e1_step; skip
:: else->skip
fi;
```

```
if :: Fire21-> S69_R_HW_Pos2=0; S68_R_HW_Pos1=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire22-> S69_R_HW_Pos2=0; S70_R_HW_Pos3=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire23-> S70_R_HW_Pos3=0; S69_R_HW_Pos2=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire24-> S14_PR1_C_unloaded=0; S15_PR1_C_loaded=1; to_queue!e1_step; skip
:: else->skip
fi;
if :: Fire25-> S15_PR1_C_loaded=0; S14_PR1_C_unloaded=1; to_queue!e1_step; skip
:: else->skip
fi;

do :: (len(to_queue)>1) -> to_queue?e;
:: (len(to_queue)$<$2) -> break; od;

/* Cleaninig up structures */
P0=0; P1=0; P2=0; P3=0; P4=0; P5=0; P6=0; P7=0; P8=0; P9=0; P10=0; P11=0;
P12=0; P13=0; P14=0; P15=0; P16=0;...
Sel0=0; Sel1=0; Sel2=0; Sel3=0; Sel4=0; Sel5=0; Sel6=0; Sel7=0; Sel8=0;
Sel9=0; Sel10=0; Sel11=0; Sel12=0;...
Fire0=0; Fire1=0; Fire2=0; Fire3=0; Fire4=0; Fire5=0; Fire6=0; Fire7=0;
Fire8=0; Fire9=0; Fire10=0; Fire11=0;...
{
od
}

init
{
/* Initial configuration */
S0_T_HW_r_st3=0;
S1_T_HW_r_ok4FB_b=0; S2_T_HW_r_ok4RA1_b=0;
S3_T_HW_r_ok4FB_a=0; S4_T_HW_r_ok4RA1_a=0;
S5_Feed_Belt_Controller=1; S6_FB_C_stop=1; S7_FB_C_move=0;
S8_Press_1_Controller=1; S9_PR1_C_ready=1; S10_PR1_C_work=0;
S11_PR1_C_busy=0; S12_Presse_2_Controller=1; S13_PR1_C_crashed=0;
S14_PR1_C_unloaded=1; S15_PR1_C_loaded=0; S16_PR2_C_unloaded=1;
S17_PR2_C_loaded=0; S18_PR2_C_ready=1; S19_PR2_C_work=0;
S20_PR2_C_busy=0; S21_PR2_C_crashed=0; S22_FB_C_tick_1=1;
S23_FB_C_tick_2=0; S24_FB_C_time_out=0; S26_PR2_HW_empty=1;
S27_PR2_HW_processing=0; S28_PR2_HW_failure=0; S29_PR2_HW_done=0;
S30_PR2_HW_process=0; S31_Press_2_Hardware=1;
S32_Press_1_Hardware=1; S33_PR1_HW_failure=0; S34_PR1_HW_process=0;
S35_PR1_HW_processing=0; S36_PR1_HW_done=0; S37_PR1_HW_empty=1;
S38_Feed_Belt_Hardware=1; S39_FB_HW_empty=1;
S40_FB_HW_one_blank_left=0; S41_FB_HW_two_blanks=0;
S42_FB_HW_one_blank_right=0; S43_FB_HW_wait_for_blank=0;
S44_Rotary_Table_Controller=1; S45_T_C_tick_1=1;
S46_T_C_tick_2=0; S47_T_C_time_out=0; S48_T_C_r_right=0;
S49_T_C_r_stop=1; S50_T_C_r_left=0; S51_T_C_v_up=0;
S52_T_C_v_stop=1; S53_T_C_v_down=0; S54_T_C_unloaded=1;
S55_T_C_load=0; S56_Robot_Controller=1; S57_R_C_tick_1=1;
S58_R_C_tick_2=0; S59_R_C_time_out=0;
S60_R_C_Redundancy_Manager=1; S61_R_C_Press_1_down=0;
S62_R_C_Press_2_down=0; S63_R_C_System_fails=0; S64_R_C_right=0;
```

```
S65_R_C_stop=1; S66_R_C_left=0; S67_Robot_Hardware=1;
S68_R_HW_Pos1=1; S69_R_HW_Pos2=0; S70_R_HW_Pos3=0;
S71_R_C_start=1; S72_R_C_RA1_T=0; S73_R_C_RA2_PR2=0;
S74_R_C_RA2_PR1=0; S75_R_C_RA1_PR2=0; S76_R_C_RA1_PR1=0;
S77_R_C_RA2_PR2_ok=0; S78_R_C_RA2_PR1_ok=0;
S79_R_C_RA1_PR1_ok=0; S80_R_C_RA1_PR2_ok=0;
S81_R_C_RA1_loaded=0; S82_Rotary_Table_Hardware=1;
S83_T_HW_v_crash=0; S84_T_HW_v_st3=0; S85_T_HW_v_st2=0;
S86_T_HW_v_ok4FB_a=0; S87_T_HW_v_ok4FB_b=0;
S88_T_HW_v_ok4RA1_b=0; S89_T_HW_v_st1=1;
S90_T_HW_v_ok4RA1_a=0; S91_T_HW_r_crash=0; S92_T_HW_r_st1=1;
S93_T_HW_r_st2=0; S25_Production_Cell=1;

to_queue!e1_step;

/* Start process */
atomic {run queue(); run STEP()}
}
```