T U M

I N S T I T U T F Ü R I N F O R M A T I K

Seamless Modeling of an Automation Example Using the SPES Methodology

Sebastian Eder, Andreas Vogelsang, Martin Feilkas

TUM-I11
Mai 11

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-I11-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Contents

1 Acknowledgment

This work was funded by the German Federal Ministry of Education and Research (BMBF), grant "SPES2020, 01IS08045A".

2 Introduction

Similar to other domains, software within systems of the automation domain is getting more complex and crucial to the success of such systems. Modeling approaches—like the SPES modeling approach—that try to facilitate the engineering of automation systems have to cope with a variety of challenges:

Tracing of dependencies. The complexity that arises from current automation systems can be mastered by different views onto the system that hide information that is not necessary in a certain development step or for a certain aspect. However, design decisions and entities of different views onto a system may depend on each other. This results in the necessity to ensure the consistency of the data of the different views. This is especially challenging if changes in a certain view also affect other views.

Integration of different engineering disciplines. A variety of engineering disciplines are involved in the realization of an automation system. Engineers from the fields of software, electronics, material sciences, and others all have their own models and methods that are specialized for a certain purpose. However, there are design decisions and information that affect more than one engineering discipline. A modeling approach has to offer discipline independent system models as well as views onto these models that support the different disciplines.

Scalability. Complexity not only arises from several dependencies between the objects within an automation system but also from their number. Current automation systems consists of a great number of functions and devices. Models and especially tools that support a modeling approach have to be aware of this number and must support the developers to stay on top of things.

Process integration. The development process of automation systems in practice currently consists of different steps. Engineering relevant steps are, beneath others, the *Bid Preparation*, the *Basic Engineering*, the *Detailed Engineering*, and the *Commissioning*. Each step has different requirements that a modeling approach has to fulfill with respect to the given development phase. The Bid Preparation for example is a rather short iteration at the beginning of a project with the aim to launch a bid depending on a rough estimation of the major cost items (e.g., Equipment, Engineering, and Project Management). This estimation is already crucial, since on the one hand the resulting bid is binding but on the other hand competitors might get the project due to an inappropriate bid.

This case study shows the SPES modeling approach using the cylinder head production example by Siemens. From the requirements, which are not modeled here, we develop a functional, logical and technical architecture. We focus on the modeling approach, not on the concrete behavior of the system. Furthermore, we trace functions from the functional architecture over

components of the logical architecture to control units of the technical architecture. This case study might provide a basis for a discussion of the SPES modeling approach with respect to the given challenges from the automation domain.

In this work, functions are denoted as $^{\bigcirc}$*function name*, components as $^{\square}$*component name*, control units as $^{\diamond}$*control unit name*, and channels (internal and external) as $^{\rightarrow}$*channel name*.

3 The system to develop

The main goal of the complete system is to produce cylinder heads. This means raw material is fed into the production cell that is illustrated in Figure 1 and finished cylinder heads are delivered by the production cell. In order to produce cylinder heads, the material has to pass several stations: From the supply band that transports raw material into the cell, the workpieces have to be placed consecutively at the milling station, grinding station, measuring station and assembly station, before they are put onto the delivery band that transports finished workpieces out of the cell.

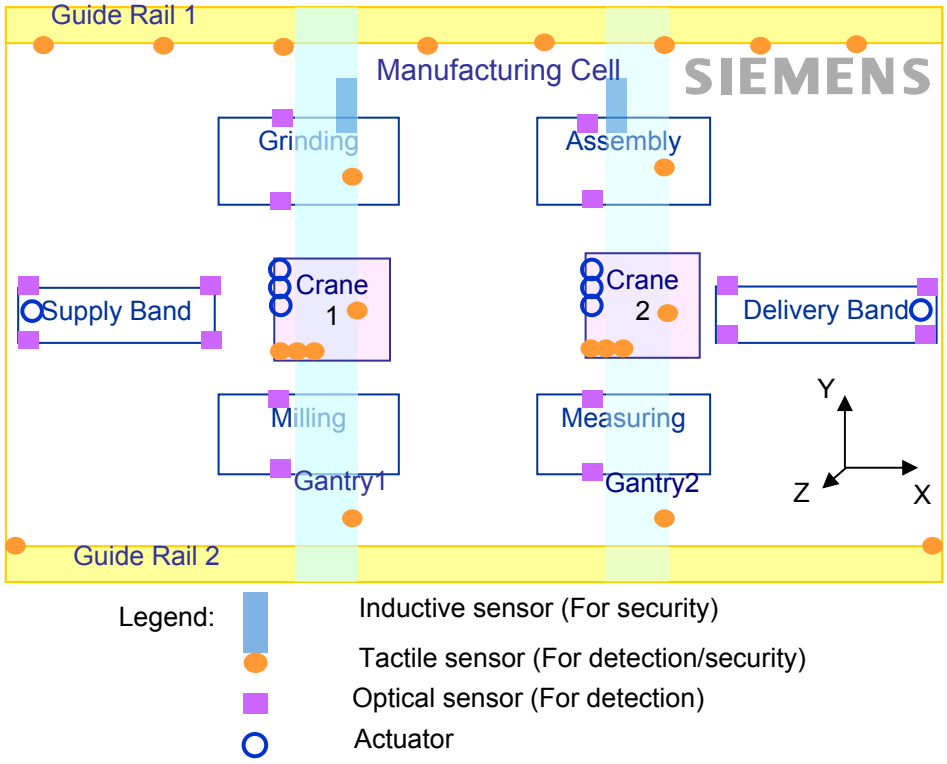

Figure 1: Geometrical view on the cylinder head production cell [Den07]

The requirements documents do not mention how the production itself has to be realized. This is why the system that will be developed in this work realizes the transport of the workpieces between the stations and the bands. Therefore, the system boundary is not defined by the boundaries of the physical production cell, which would mean, that the input of the system is raw material and the output are cylinder heads. In fact, the system boundary of the system under development is defined by the sensors and actuators of the production cell that

control the transportation of the material. That means the system receives sensor data from the production cell and sends data to the actuators.

A detailed list of all sensors of the system can be found in Section A.3.1.

4 Functional View

4.1 General Methodology

The functional architecture structures functionality that is observable by the environment of the system. This is why this artifact is the connecting link between the requirements documents and the system design. The system's functionality is therefore broken down into sub-functions that are responsible for a subset of output values and rely only on a subset of input values of the system. Thus, a sub-function's behaviour can be considered as projection of the entire system behaviour. Additionally, dependencies between those functions are illustrated.

Dependencies arise, if a function's output does not just depend on the specific function input values from the environment of the system but also on additional input values that are processed by another function. For example, a function that realizes a certain crane job by mapping sensor data to crane actions might also be dependent on inputs that indicate whether the system is switched on or off. In case these inputs are processed by another function there is a dependency between the two functions. In this approach, these dependencies between functions are abstracted by so called *modes*, which influence a function's behaviour. Functions are assumed to be able to enter different modes similar to state machines. A function that alters a mode can propagate the value of the mode over so called *mode channels* to other functions that depend on the mode. This way, functional specifications can be written down independently, allowing a modular development, while dependencies between functions are made explicit by the mode channels.

To gain a functional architecture, the high level functionality of the system is decomposed into functions of finer granularity. Functions that are more concrete can then be specified formally. These specifications can be used for verification, testing or validation. Functions at a lower level of granularity are projections of the behavior of the high-level function and the main goal of the decomposition is to gain a modular architecture of the complete system. The decomposition stops, if the leaf nodes of the decomposition tree are simple enough to be specified.

The behavior of functions can be specified partially. That means, that not every input sequence to the system has to be mapped to an output sequence. For example, a specification could cover only the good cases of system inputs. Concrete behavior can be specified with the help of several techniques: Message sequence charts, state machines, activity diagrams, tabular specifications or declarative techniques are viable alternatives. In this example, we used a tabular specification technique that is explained in detail in Appendix A. The specified behavior is useful for verification and validation of the functionality a system offers. Furthermore, the presence of all functional requirements can be ensured, since functional features are modeled comprehensively in this view.

4.2 Application to the Case Study

In this example, the high level functionality is the control of the transport of workpieces between several stations in a cylinder head production cell. So, this functionality is decomposed, as

Figure 2 shows. Solid lines represent sub-function relationships, whereas dashed lines represent mode channels indicating dependent functions. The result is just one possible decomposition. In this section, we explain how and why we developed this tree.

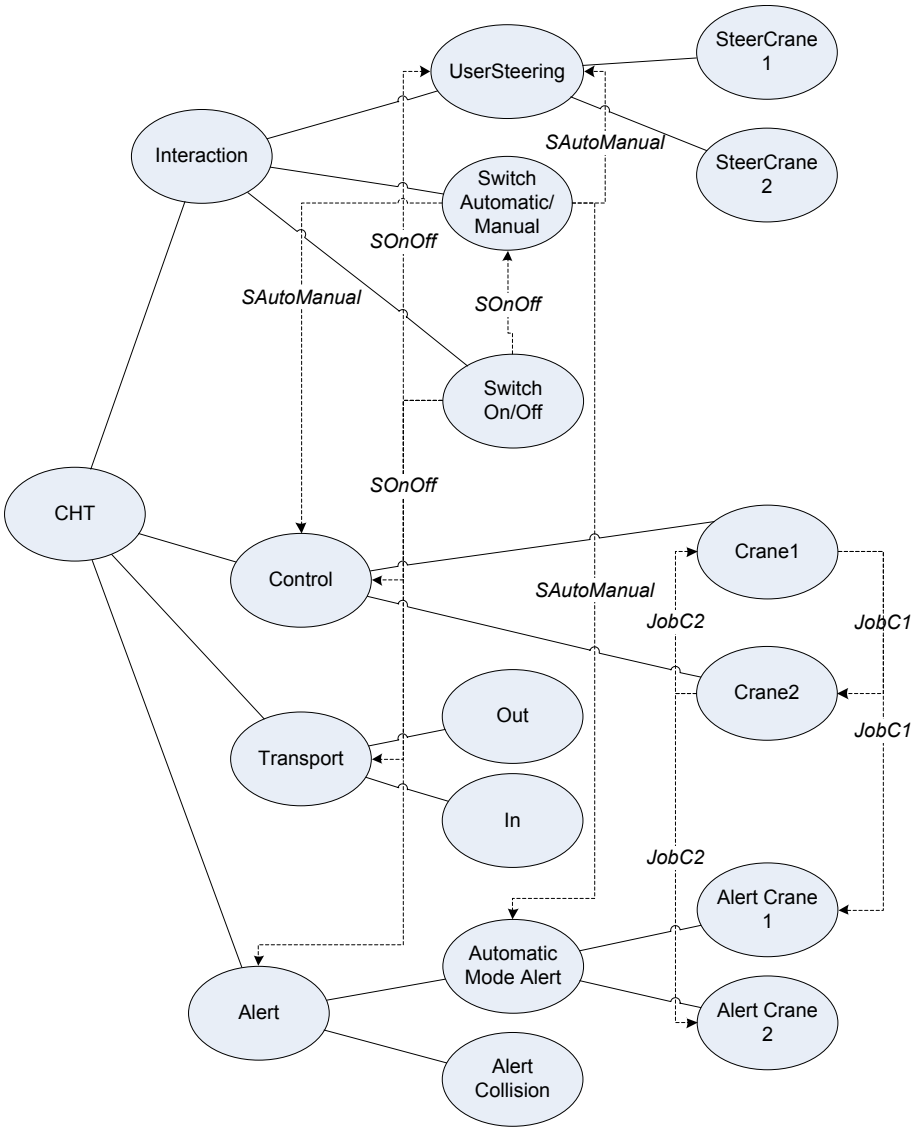

Figure 2: Functional architecture of the cylinder head transportation system

According to the requirements documents [Köh06], the main functions of the system are:

- *Transport*: Transport of workpieces to and from the production cell
- *Interaction*: User interaction
- *Control*: Automatic control of the transport of workpieces between stations
- *Alert*: Alerting system

These functions are decomposed further as shown in the next sections according to [Den07]. For a detailed overview of the syntactic interfaces of the functions and behavior specifications, see Appendix A.

4.2.1 Transport

Transporting the workpieces to and from the cylinder head production cell consists of two functions: Transporting the workpiece into the cell ($\bigcirc$*In*) and transporting the workpieces from the cell ($\bigcirc$*Out*). Those functions are not decomposed any further, because they already are quite simple. But not decomposing $\bigcirc$*Transport* any further would lead to a function that had to realize the supply and delivery band. But those jobs have very different concerns and thus $\bigcirc$*Transport* is decomposed.

4.2.2 Interaction

The user interaction consists of three functions: $\bigcirc$*Switch On/Off* for switching the system on and off, $\bigcirc$*Switch Automatic/Manual* for switching between automatic and manual transportation and production mode, and $\bigcirc$*User Steering* for manually steering the transport system between stations according to the requirements [Köh06]. The last function can be decomposed even further with the knowledge of having two cranes [Den07]. Therefore, the user can steer the first crane by using $\bigcirc$*Steer Crane 1* and the second crane by using $\bigcirc$*Steer Crane 2*.

4.2.3 Control

The automatic control of the system can also be decomposed further with the knowledge about having two cranes. So the automatic control of the first crane is specified by $\bigcirc$*Crane1* and the automatic control of the second crane is specified by $\bigcirc$*Crane2*. This decision is made in the requirements document [Den07], because there, different (disjoint) transport areas are assigned to both cranes (the first crane moves workpieces from the supply band to the measuring station, then the second crane takes over). Thus, the cranes realize different functionality.

4.2.4 Alert

The alerting system notifies the user in case of errors. As some errors only arise in automatic mode, the $\bigcirc$*Alert* is decomposed into $\bigcirc$*Alert Collision*, which signals a collision, and $\bigcirc$*Automatic Mode Alert*, which signals errors in case of spontaneous appearing or disappearing workpieces. Again, this function is decomposed into two functions: one for every crane ($\bigcirc$*Alert Crane1* and $\bigcirc$*Alert Crane2*), because of the different stations the cranes travel to. An error that relates to the first crane can be an error that does not influence crane 2.

4.2.5 Mode dependencies

Every function depends on the system to be switched on. This is why every function but $\bigcirc$*Switch On/Off* itself and $\bigcirc$*Switch Automatic/Manual* depend on it. This is similar for the automatic and manual mode. To capture these dependencies, the mode channels $^{\rightarrow}$*SOnOff* and $^{\rightarrow}$*SAutoManual* are included in the functional architecture. $\bigcirc$*Switch Automatic/Manual* does not depend on $\bigcirc$*Switch On/Off*, because the user should be able to switch between the modes even if the system is turned off.

$^{\bigcirc}$*Automatic Mode Alert* only signals an alarm, if the system is in automatic mode. Furthermore, the automatic control of the transportation within the production cell ($^{\bigcirc}$*Control*) is only active, if the system is in automatic mode. In this case, $^{\bigcirc}$*User Steering* is inactive. This is why these functions depend on $^{\bigcirc}$*Switch Automatic/Manual*. $^{\bigcirc}$*Transport* does not depend on $^{\bigcirc}$*Switch Automatic/Manual*, because these bands always transport workpieces automatically.

In order to realize the automatic control for the transport within the production cell and to avoid collisions, each crane has to know what the other crane intends to do. Therefore, each crane depends on the current job of the other crane (mode channels $^{\rightarrow}$*Job1* and $^{\rightarrow}$*Job2*). The alerting system also has to be aware of the job, each crane is assigned to in order to detect failures in the procedure (that arise because of (dis)appearing workpieces).

5 Logical View

5.1 General Methodology

The logical architecture is the connecting link between the functional architecture, which resides in the problem domain to the solution domain. In contrast to the functional architecture, the system is treated as a white box. The logical architecture refines the functional architecture by introducing logical components that realize certain functions, and communication channels between the logical components. The behavior of components is specified totally, that means that every possible input sequence to the system has to be considered by the logical view. This view also decouples functional and logical aspects from the technical implementation of the system under development.

A function can be realized by just one component or by many. One component can implement more than one function. Thus, there is an *m:n-mapping* between functions and components. Like in the functional architecture, behavior can also be specified for the logical architecture. State machines for every component can be found in Section B.

The concrete structure of a logical perspective is motivated by functional as well as non functional requirements. Especially non functional requirements slip into the architecture, because here the basis for features like maintainability, extensibility or adaptability, just to name a few, is formed. This is also the reason why design patterns emerge in this view.

Verification, simulation, testing, and the generation of test cases and code are supported by the logical architecture, because the behavior of components is totally and formally specified .

Alternatives for specifying logical architectures are SysML (Internal Block Diagrams), UML (Composite Structure Diagrams), Heterogeneous Rich Components or AutoFocus3 (Structure Diagrams). This work's figures illustrating components originate from the tool AutoFocus3. In these figures, three different colors are used: Red, blue and yellow. Red components are decomposed further into components of finer granularity. Blue and yellow components are not decomposed any further but a behavior specification is assigned to them. The behavior of blue components is strongly causal and the behavior of yellow components is weakly causal. For a detailed description of strong and weak causality see [BS01].

Concrete behavior can be specified with the help of state machines, tabular specifications, SysML state charts or UML state charts. In this work, we used state charts in the tool AutoFocus3 in order to get a prototype that we could simulate in the tool and to generate code.

5.2 Application to the Case Study

5.2.1 Overview

Figure 3 shows the hierarchical decomposition of the logical components of the cylinderhead transportation system. These components and the corresponding communication channels form the logical architecture.

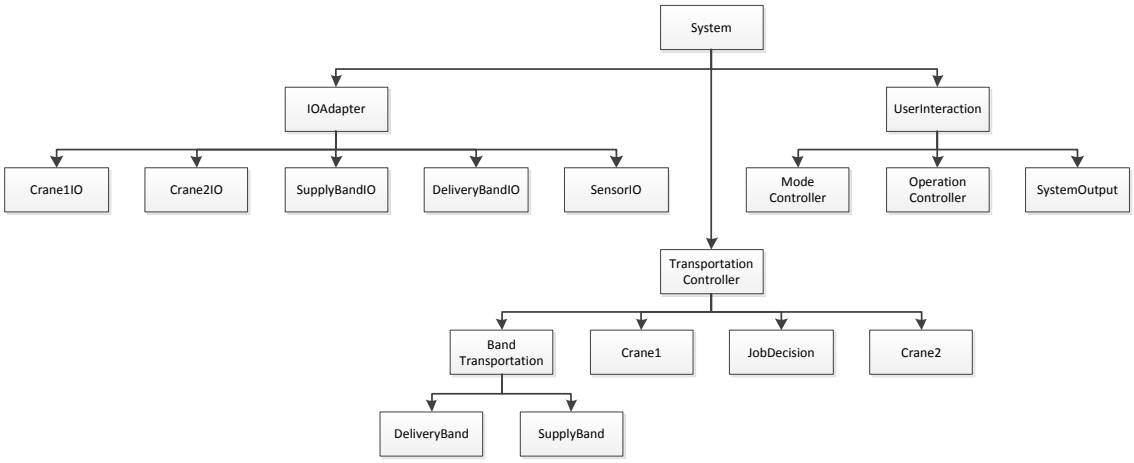

Figure 3: Hierarchical decomposition of the logical components

5.2.2 System level

The system consists of three top-level components as illustrated in Figure 4: $\square$*IOAdapter*, $\square$*TransportationController* and $\square$*UserInteraction*. $\square$*IOAdapter* is the interface between the control system and the production cell. It transforms data into a readable format for the $\square$*TransportationController* and the production cell. The channel $\rightarrow$*Sensor* contains all sensor data. The different sensors could be modeled with a channel each. But this would only complicate the model. The $\rightarrow$*Crane*-channels are used to send signals to cranes and the $\rightarrow$*Band*-channels are used to send signals to the bands. The subdivision in those components is chosen to gain separation of interaction, control and signal conversion. This is similar to the MVC pattern.

5.2.3 User interaction

The component $\square$*UserInteraction* realizes the function $\bigcirc$*Interaction* partially. Its only input channel is $\rightarrow$*UserInput* that is used by the user to send data into the system. The channel $\rightarrow$*SystemOutput* is used to send feedback to the user. $\rightarrow$*OperationOn* and $\rightarrow$*AutoMode* signals the user inputs to the $\square$*TransportationController*.

$\square$*UserInteraction* contains two components for interaction as illustrated in Figure 5: The component $\square$*OperationController* for switching the system on and off (realizing the function $\bigcirc$*Switch On/Off*) and $\square$*ModeController* for switching between manual and automatic production mode (realizing the function $\bigcirc$*Switch Automatic/Manual*). This rather canonical partition

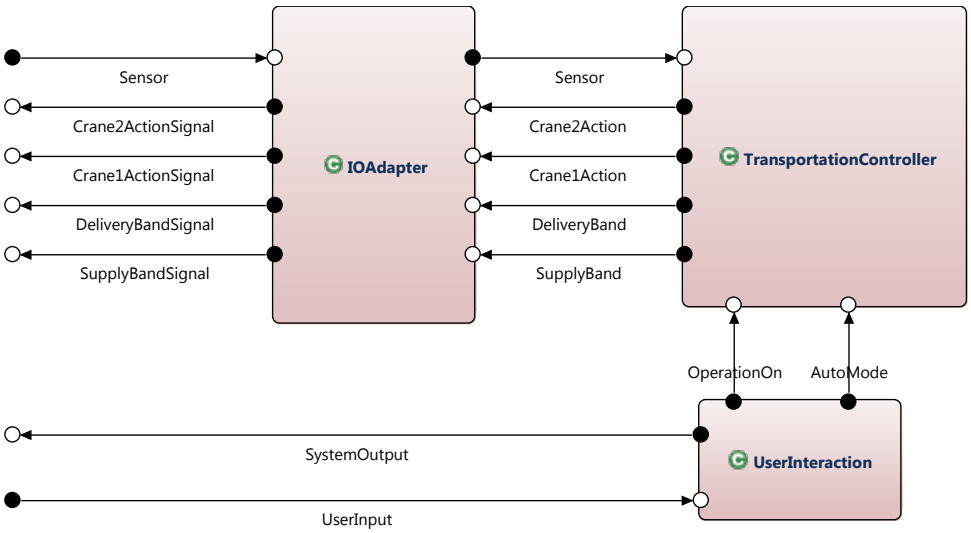

Figure 4: Logical system overview

of concerns follows the subdivision in the functional architecture. The outputs of these components are led into □*SystemOutput*, where the outputs of the switching components are merged onto one channel that is used to give feedback to the user. The switching component send their signals to the component □*TransportationController*.

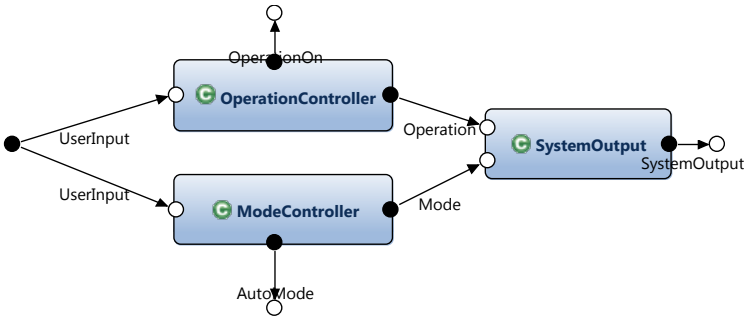

Figure 5: User interaction component

5.2.4 IOAdapter

This component translates signals from a representation that is understood by logical components into a representation that is suitable for the physical production cell. Thus, □*IOAdapter* realizes a part of all functions that are sending signals to the production cell. Figure 6 shows the decomposition of □*IOAdapter*.

Since □*IOAdapter* simply realizes some signal conversions the behavior is specified by weakly causal functional specifications. For each channel that is used to send signals from the logical component □*TransportationController* to the production cell, there is a component that translates the corresponding signals.

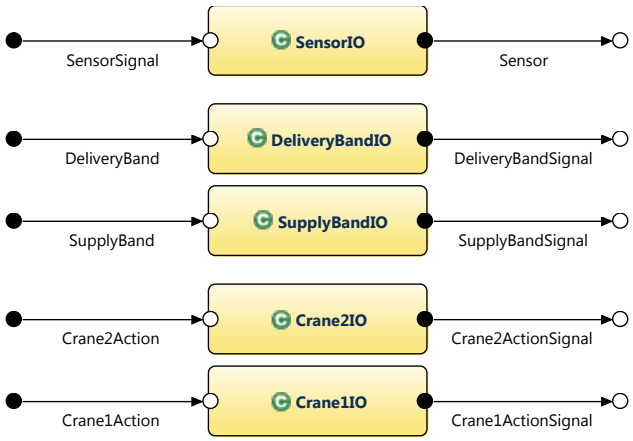

Figure 6: IOAdapter component

5.2.5 TransportationController

The functions ○ *Transport* and ○ *Control* are realized by both □ *TransportationController* and □ *IOAdapter*. This is why □ *TransportationController* is decomposed into four components as illustrated in Figure 7. The three components □ *JobDecision*, □ *Crane1* and □ *Crane2* partially realize the two functions ○ *Crane1* and ○ *Crane2* (the other part is realized by □ *IOAdapter*). Transportation to and from the production cell is realized by □ *BandTransportation* (and □ *IOAdapter*, of course). According to the received data it sends control signals to the supply or delivery band.

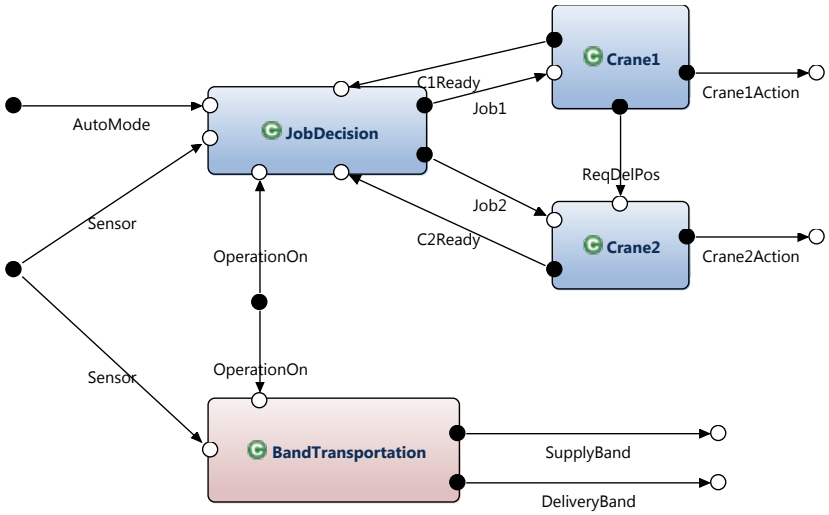

Figure 7: □ *TransportationController*

In the functional architecture, both cranes send the action they intend to perform to the other crane in order to avoid collisions. So, each crane decides by itself which job should be done next. The difficulty to implement this behavior becomes apparent in the functional

behavior specification of ○*Crane1* and ○*Crane2* in Section A.6. In the logical architecture, this decision is not made distributed by the cranes, but by the component □*JobDecision*. This component decides which job to do next and sends it to the corresponding crane. The decision is based on the sensor data received from the production cell and on the current state of both of the cranes (whether they are ready or busy). But, in order to avoid deadlocks, □*Crane1* must be able to send □*Crane2* to the delivery band. Therefore, the channel →*ReqDelPos* is introduced. In order to execute the desired jobs, □*Crane1* and □*Crane2* are explicitly modeled. They receive jobs and send the according sequence of control signals to the production cell.

5.2.6 Band transportation

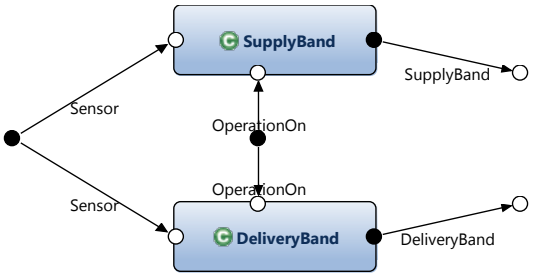

Figure 8: Band transportation component (□*BandTransportation*)

□*BandTransportation* is decomposed into a component for every band as shown in Figure 8. Those components realize the functions ○*In* and ○*Out*. Based on sensor values and on whether the system is switched on, □*SupplyBand* and □*DeliveryBand* turn the corresponding band on or off.

5.2.7 Unimplemented functions

The function ○*Alert* is not realized by the introduced logical architecture because of the prototypical character of this work. Furthermore, ○*User Steering* is not implemented.

6 Technical View

6.1 General Methodology

The technical architecture models the platform used to execute the system. The main entities are therefore ECUs (Electronic Control Units), bus systems as well as actuators and sensors. This network represents the hardware topology the logical components can be allocated to. The technical view is furthermore concerned with mapping logical components to tasks. These model can then be used for analyses to verify if real-time requirements are fulfilled by a given scheduling, message catalogs etc.

6.2 Application to the Case Study

6.2.1 Overview

Figure 9 shows the hierarchical decomposition of the technical control units of the cylinder-head transportation system. These control units and the corresponding bus system form the technical architecture.

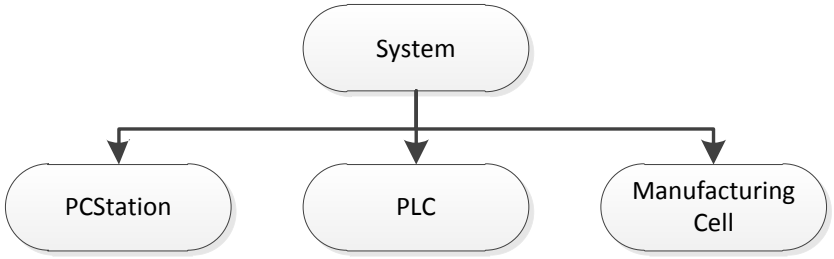

Figure 9: Hierarchical decomposition of the technical components

6.2.2 System level

The components of the logical architecture are deployed onto the three control units shown in Figure 10. In this example, the control units communicate using a bus system. The technical choice which bus system is modeled is arbitrary in this case.

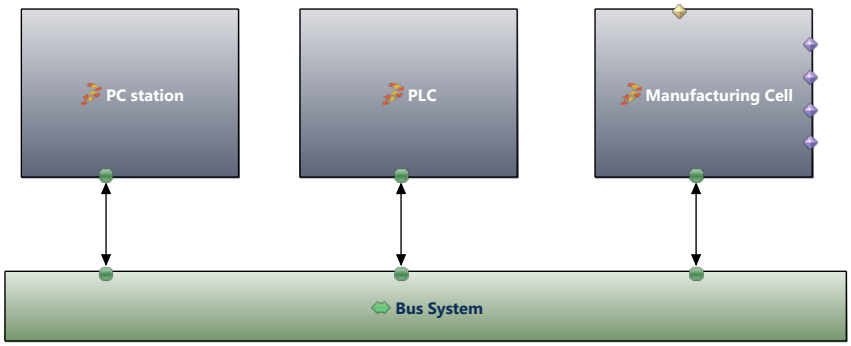

Figure 10: Technical architecture

The deployment of the different components onto the control units is shown in Table 1.

With this final allocation of logical components to control units we gain a traceability of requirements over functions, logical components to technical control units and bus messages. The deployment of component □ *UserInterface* on the ◇ *PC-Station* for example allows to trace the function ○ *Interaction* to this control unit, meaning this function is entirely realized by one control unit. In contrast, the function ○ *Crane1* is realized by the logical components

Component		Control unit
$^{\square}$*UserInteraction*	$\rightarrow$	$^{\diamond}$*PC-Station*
$^{\square}$*TransportationController*	$\rightarrow$	$^{\diamond}$*PLC*
$^{\square}$*IOAdapter*	$\rightarrow$	$^{\diamond}$*Manufacturing Cell*

Table 1: Deployment assignments

$^{\square}$*JobDecision*, $^{\square}$*Crane1* and $^{\square}$*IOAdapter*, which themselves are deployed onto control units $^{\diamond}$*PLC* and $^{\diamond}$*ManufacturingCell* connected by a bus.

7 Conclusion

In this work, the cylinder head transportation control was modeled. A functional, logical, and technical architecture was discussed. A tracing of functions to technical components was possible.

Furthermore, this model serves as basis for a common understanding of the presented concepts in the automation domain within the SPES project. Especially the content and aim of the functional and the logical perspective are clarified and comprehensively modeled.

Still some open questions remain open according to the integration of the approach into existing processes of the automation domain as well as to the applicability of large scale projects.

Process Integration. Since the presented modeling methods do not prescribe a certain process it is still not obvious how the introduced artifacts can be integrated according to the process that was introduced in Section 2. Especially in the bid preparation phase the models must provide a first estimation of major cost items within a short period of time. Therefore, a pure top-down approach may not be appropriate. The functional, logical and technical architecture should rather be developed in parallel. This way, major cost items concerning technical devices, software complexity and also organizational issues are considered. As a next step a detailed study on the integration of the introduced models and artifacts into a given development process should be carried out.

Application to large scale projects. Although the presented example is a realistic system that covers a variety of characteristical problems of the automation domain it is still a rather small example. Large automation systems nowadays consists of hundreds of sensors, actuators, and computing units and often realize a variety of functions. This constitutes challenges for the organization and the visualization of the presented models. It also entails the need for an appropriate tool support that has to be customized to cope with large scale projects and large sets of data.

References

[Bro10] Manfred Broy. Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, 2010.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[Den07] Kristian Dencovski. Dokumentation zum CT SE 5 - Anlagendemonstrator. Technical report, Siemens AG, 2007.

[Köh06] Adrian Köhnlein. Lastenheft – Fischertechnik-Modell. Technical report, Siemens AG, 2006.

[Vog11] Andreas Vogelsang. Cylinder Head Transportation AutoFocus 3 Model. cylinder-head_transportation.af3, 2011.

A Syntactic interfaces and behavior specifications for the functional architecture

In the functional architecture, functions of the system will be described by their syntactic interface and their behavior specification In this work, we use table specifications [Bro10] to specify the interface behavior. Syntactic interfaces and table specifications are explained in the next sections. Note: There are several different techniques for specifying interface behavior. With the help of table specifications or state machines, operational behavior is defined. But also declarative specifications like interface assertions are possible for specifying the interface behavior of a function.

A.1 Syntactic interfaces

Syntactic interfaces are used to describe a function's input and output channels with their data types. An example can be found in Figure 11. There, the syntactic interface of a function $\bigcirc AAnd$ with five input and two output channels is illustrated. All channels are of the type $\mathbb{B}$. The channels printed in italics denote internal mode channels. This does not change the semantics of the input or output on that channel, it just helps to understand what the function does. A syntactic interface does not specify behavior in any way.

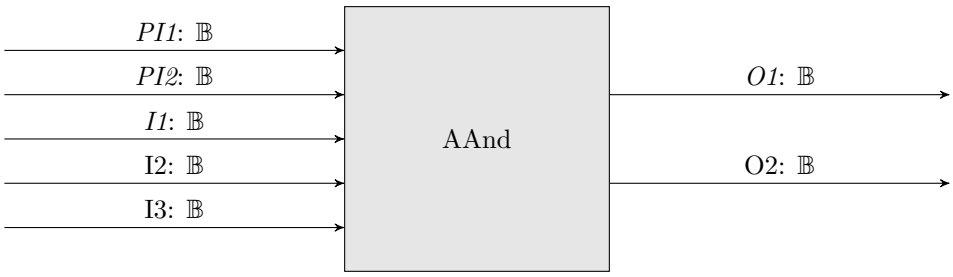

Figure 11: Syntactic interface of $\bigcirc AAnd$

A more detailed and formal description of syntactic interfaces can be found in [Bro10].

A.2 Table specifications

This specification technique allows to specify possibly partial behavior in a simple and clear way. Table specifications will be explained with the following example:

The function $\bigcirc AAnd$ has five input channels: PI1, PI2, I1, I2, and I3. The output channels are O1 and O2. The behavior is only specified for PI1 = *true* and PI2 = *false*. Thus, the second and third line of the table specification can be seen as a precondition for the specification. If those preconditions are not valid, no behavior is specified, which means arbitrary behavior. In case the preconditions are valid, the function sends the value, it received on channel I1 on the channel O1. Furthermore, it writes I2 $\wedge$ I3 to O2, if I1 = *true*. If I1 = *false*, the output on O2 is not restricted. In order to keep table specifications short, we introduce the Else-operator. It expands to all possibly combinations in the columns it spans, that are not mentioned in other lines that match the other columns of the line. In this case, it expands to (*true*,*false*), (*false*,*true*), and (*false*,*false*) always with *true* in column I1, *true* in O1 and *false*

in O2. Another operator is the question mark (?). It does not restrict the input or output on the channels it spans. In the last line of the table, the only information that is needed, is just the *false* on I1, because then, *false* is sent over O1 and the output on O2 is not restricted in this case.

Expressed more formally, the function will execute the following calculation (with input i_1 coming from $^{\rightarrow}I1$ and so on) and write the first element of the resulting pair to $^{\rightarrow}O1$ and the second part to $^{\rightarrow}O2$.

$$AAnd(p_1, p_2, i_1, i_2, i_3) = \begin{cases} (i_1, i_2 \wedge i_3), & \text{if } p_1 \wedge p_2 \wedge i_1 \\ (i_1, ?), & \text{if } p_1 \wedge p_2 \wedge \neg i_1 \\ (?, ?), & \text{else} \end{cases}$$

The following table specification is equivalent to the equation above. The primed channel names in the table specification are output channels. O1' means, that the values in this column are written to $^{\rightarrow}O1$.

AAnd				
PI1 = *true* PI2 = *false*				
I1	I2	I3	O1'	O2'
true	*true*	*true*	*true*	*true*
true	Else		*true*	*false*
false	?		*false*	?

A.3 System inputs and outputs

A.3.1 Sensors

Sensor data is the main input into the system. Table 2 shows the sensors with their names and location, as well as the type. A light sensor signals *true*, if and only if it detects a workpiece. A touch sensor sends *true*, if and only if something touches it and the inductive sensors send *true*, if and only if something gets close to it.

A.3.2 User inputs

In order to receive inputs from the user (over a PC), there have to be some channels a user can utilize to send data to the system. At first, the data that can be send has to be defined:

Data types

$OnOff = \{On, Off\}$ The users sends *On*, to turn the system on and *Off* to turn it off.

$AutoManual = \{Auto, Manual\}$ The user sends *Auto*, to switch to the automatic production mode and *Manual* to manually produce cylinder heads.

$SC = \{+X, -X, +Y, -Y, +Z, -Z, Grab, Drop\}$ The user alters the position of a crane by sending $+X, -X, \ldots$. The user sends *Grab* to make a crane grab and *Drop* to make a crane drop.

Channel	Description	Type
LS1In	Light sensor at beginning of supply band	$\mathbb{B}$
LS2In	Light sensor at end of supply band	$\mathbb{B}$
LSMi	Light sensor at milling station	$\mathbb{B}$
LSGr	Light sensor at grinding station	$\mathbb{B}$
LSMe	Light sensor at measuring station	$\mathbb{B}$
LSAs	Light sensor at assembly station	$\mathbb{B}$
LS1Out	Light sensor at beginning of delivery band	$\mathbb{B}$
ISC1	Inductive sensor at crane 1	$\mathbb{B}$
ISC2	Inductive sensor at crane 2	$\mathbb{B}$
TSC1	Touch sensor at crane 1	$\mathbb{B}$
TSC2	Touch sensor at crane 2	$\mathbb{B}$
TSGR2S	Touch sensor guide rail 2 (at supply)	$\mathbb{B}$
TSGR2D	Touch sensor guide rail 2 (at delivery)	$\mathbb{B}$

Table 2: Overview over sensor inputs, their names and types

Channels The channels for the user consist of two switches, UOnOff and UAutoManual. With those, the user can switch the system on and off as well as between automatic and manual production mode. Furthermore, the user can steer both cranes. Therefore, there are two channels USC1 and USC2, the user sends steering commands over.

Channel	Description	Type
UOnOff	Switch (On/Off)	*OnOff*
UAutoManual	Switch (Automatic/Manual)	*AutoManual*
USC1	Steering commands for crane 1	*SC*
USC2	Steering commands for crane 2	*SC*

A.3.3 Outputs

The system's output channels are not only data channels that are linked back to the user's terminal, but also channels for control signals for the production cell. In this section, the output channels of the leaf functions of the decomposition tree are explained.

Bands The supply and the delivery band have to be started and stopped. The system sends *true*, if a band has to start and *false* if the band has to stop. Therefore, the functions that control the supply and delivery band have channels attached ($^{\rightarrow}$*MovingIn* and $^{\rightarrow}$*MovingOut*) that are used to send the signals to the bands.

Alert The function $^{\circ}$*Alert* just sends *true*, if an error occurs.

Cranes For enabling a user to steer a crane, there need to be several actions that can be performed by the crane. *CraneUserAction* contains the actions the system can send to a crane:

$$CraneUserAction = \{ \begin{array}{l} Move+X, Move-X, \\ Move+Y, Move-Y, \\ Move+Z, Move-Z, \\ Drop, Grab \end{array} \}$$

Action	Description
$MoveD$	Signals the crane to move to direction D.
$Grab$	Signals the crane to grab. If there is a workpiece at the current position, the crane grabs this workpiece.
$Drop$	Signals the crane to drop. If the crane has a workpiece, the workpiece will be dropped at the current position.

The outputs to control cranes in the automatic mode are kept very abstract. The function just sends the job a crane has to do. The possible jobs are contained in the set *CraneJobAction*:

$$CraneJobAction = \{H1, 12, 23, 34, G4, 0\}$$

Action	Description
$H1$	Grab a workpiece at the end of the supply band and drop it at the milling station.
12	Grab a workpiece at the milling station and drop it at the grinding station.
23	Grab a workpiece at the grinding station and drop it at the measuring station and move back to milling or grinding. (To avoid collisions)
34	Grab a workpiece at the measuring station and drop it at the assembly station and move back to delivery band. (To avoid collisions)
$4G$	Grab a workpiece at the assembly station and drop it at the beginning of the delivery band.
0	Do nothing.

There are two output channels for each crane controlling function: $^{\rightarrow}ActionCn$ and $^{\rightarrow}JobCn$, where $n \in \{1, 2\}$. The function sends the action its crane has to do over those channels. The actions can be either from the set *CraneUserAction* or *CraneJobAction*. Therefore, the type of the channels is

$$CraneAction = CraneUserAction \cup CraneJobAction$$

A.4 Interaction

This section describes all interfaces that allow user interaction. First, the syntactic interfaces are illustrated. Afterwards the behavior is specified by a table.

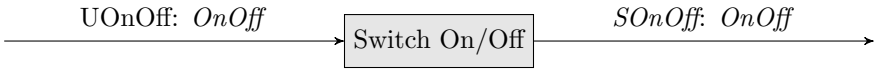

Figure 12: Syntactic interface of $^{\circ}$*Switch On/Off*

A.4.1 Switch On/Off

$^{\circ}$*Interaction* allows the user to switch the system on or off. The state of the function changes to *On*, if the user inputs *On*. If the input is *Off*, the state switches to *Off*. The syntactic interface is shown in Figure 12. The channel $^{\rightarrow}$*UOnOff* is used to receive data from the user. The corresponding mode is propagated over the mode channel $^{\rightarrow}$*SOnOff*. The following table states, that if *On* is received by $^{\circ}$*Switch On/Off* over the channel $^{\rightarrow}$*UOnOff*, then the output on the (mode) channel $^{\rightarrow}$*SOnOff* is also *On*. Analog for *Off*.

Switch On/Off	
UOnOff	SOnOff'
On	*On*
Off	*Off*

A.4.2 Switch Automatic/Manual

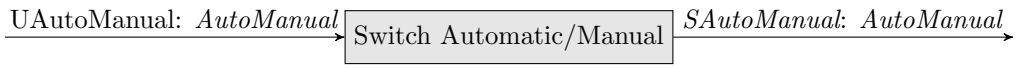

Figure 13: Syntactic interface of $^{\circ}$*Switch Automatic/Manual*

$^{\circ}$*Switch Automatic/Manual* offers the possibility to switch between automatic and manual production. The switch works like the interface for switching the system on or off. This is why the syntactic interface is very similar. Figure 13 shows the channel the user utilizes to signal the system whether it should run in automatic or manual mode: $^{\rightarrow}$*UAutoManual*. The system propagates the input over the mode channel $^{\rightarrow}$*SAutoManual*.

Switch Automatic/Manual	
UAutoManual	SAutoManual'
Auto	*Auto*
Manual	*Manual*

A.4.3 User Steering

The following functions describe the ability of the user to steer both cranes manually.

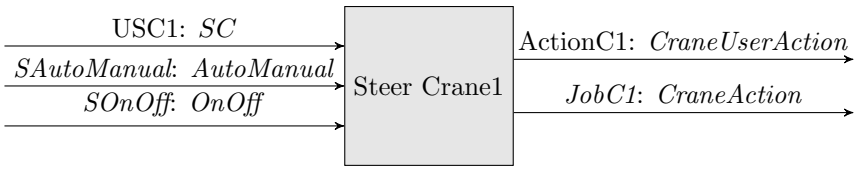

Figure 14: Syntactic interface of ○*Steer Crane1*

Steer Crane1 ○*Steer Crane1* is the user interface for steering crane 1. The user has several possibilities to control a crane. If the user enters $+X$, the crane changes its x-coordinate by one unit. The crane behaves similarly for the other user inputs. Figure 14 shows the syntactic interface for the function. It depends on the modes →*SOnOff* and →*SAutoManual*. Furthermore, the user inputs are received from channel →*USC1*. There are two output channels. The mode channel →*JobC1* is used to send the current job to ○*Alert* and ○*Crane2*. The channel →*ActionC1* is used to send actions to the production cell.

The line containing SOnOff = *On* in the tabular expression expands to a column with the head SOnOff and in every cell of this column stands *On*. This can be seen as a precondition for the function. If the precondition does not hold, no behavior is specified. That means that it is not defined what happens in that case. Here, we did not specify what ○*Steer Crane 1* does, if the system is turned off.

The question mark is another important notation in this table. It means that there is no restriction on the value of a cell with a question in it. In this case, that means: If the system mode is automatic, there is no restriction to the input and on the output. If there is another function (in this case ○*Crane1*), that restricts the output, the more restrictive output will be taken.

Steer Crane1			
SOnOff = *On*			
SAutoManual	USC1	ActionC1'	JobC1'
Manual	$+X$	*Move* $+ X$	*Move* $+ X$
Manual	$-X$	*Move* $- X$	*Move* $- X$
Manual	$+Y$	*Move* $+ Y$	*Move* $+ Y$
Manual	$-Y$	*Move* $- Y$	*Move* $- Y$
Manual	$+Z$	*Move* $+ Z$	*Move* $+ Z$
Manual	$-Z$	*Move* $- Z$	*Move* $- Z$
Manual	*Drop*	*Drop*	*Drop*
Manual	*Grab*	*Grab*	*Grab*
Auto	?	?	?

Steer Crane2 This function interface for steering crane 2. It works exactly like steering crane 1.

Figure 15: Syntactic interface of $^{\circ}$*Steer Crane2*

Steer Crane2			
SOnOff = *On*			
SAutoManual	USC2	ActionC2'	JobC2'
Manual	$+X$	$Move + X$	$Move + X$
Manual	$-X$	$Move - X$	$Move - X$
Manual	$+Y$	$Move + Y$	$Move + Y$
Manual	$-Y$	$Move - Y$	$Move - Y$
Manual	$+Z$	$Move + Z$	$Move + Z$
Manual	$-Z$	$Move - Z$	$Move - Z$
Manual	*Drop*	*Drop*	*Drop*
Manual	*Grab*	*Grab*	*Grab*
Auto	?	?	?

A.5 Transport

For the supply and delivery band, we can assume another system taking over. So, we are modeling these bands very simple. The channels $^{\rightarrow}$*MovingIn* of type $\mathbb{B}$ and $^{\rightarrow}$*MovingOut* of the same type just signal whether the bands start or stop moving.

A.5.1 In

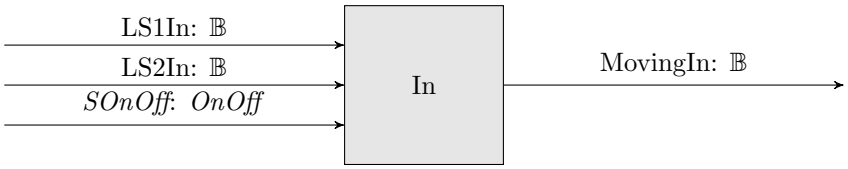

Figure 16: Syntactic interface of $^{\circ}$*In*

The control of the supply band. If $^{\rightarrow}$*LS1In* is *true*and $^{\rightarrow}$*LS2In* is *false*, the band starts to move. If $^{\rightarrow}$*LS2In* is *true*, the band stops. This data is sent over the channel $^{\rightarrow}$*MovingIn*. The inputs for this function, as shown in Figure 16, are the mode $^{\rightarrow}$*SOnOff* to determine whether the system is switched on and the light sensors at the supply band: $^{\rightarrow}$*LS1In* and $^{\rightarrow}$*LS2In*.

In		
SOnOff = *On*		
LS1In	LS2In	MovingIn'
true	*false*	*true*
?	*true*	*false*

A.5.2 Out

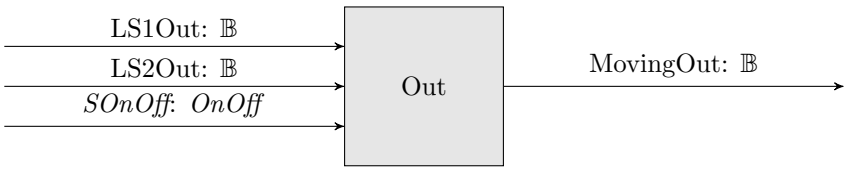

Figure 17: Syntactic interface of $^{\circ}Out$

The delivery band's control. It works just like the supply band. The difference is just that the input channels are fed by the light sensors at the delivery band, $^{\rightarrow}LS1Out$ and $^{\rightarrow}LS2Out$.

Out		
SOnOff = *On*		
LS1Out	LS2Out	MovingOut'
true	*false*	*true*
?	*true*	*false*

A.6 Control

$^{\circ}Control$ represents the control for automatic production. This model is focused on transporting workpieces. This is why the stations are not modeled. This would be an easy task: The stations' input is the corresponding light sensor. The station is turned on, if there is a workpiece detected and turned off if there is no workpiece or after the station finished a workpiece.

A.6.1 Crane1

This function steers crane 1. Because of some priorities of jobs, the availability of a job depends not only on the stations it moves to, but also on other stations that correspond to jobs with a higher priority. For example: $H1$ has less priority than 23. That means, if the sensors tell the system, that 23 is available, the system will not execute $H1$ but 23. In order to avoid collisions with the second crane, the system executes 23 only, if the second crane is not executing 34.

The syntactic interface is illustrated in Figure 18. The input channels are sensors, both of the switches and $^{\rightarrow}JobC2$. The last one is needed to avoid collisions.

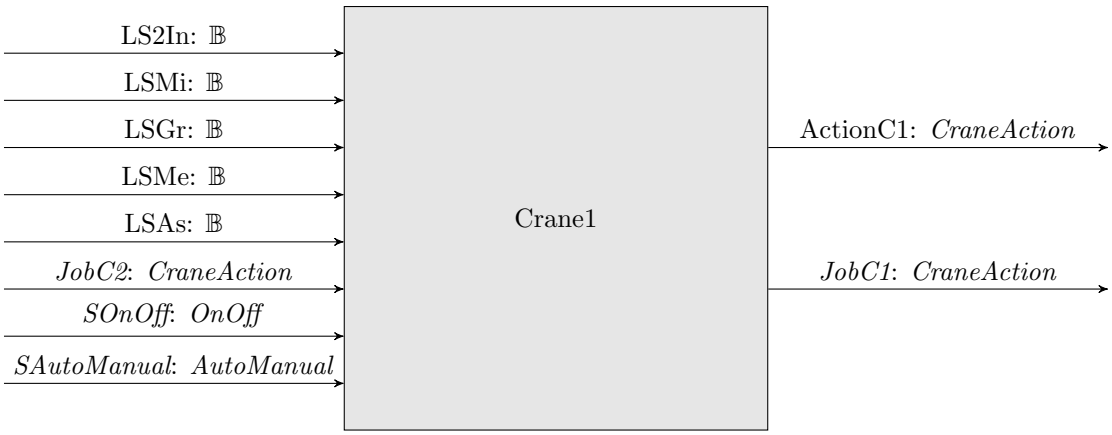

Figure 18: Syntactic interface of $^{\circ}$*Crane1*

This specification has one peculiarity: The Else-operator. Else expands to all combinations of the columns it spans that are not covered by any line yet. The difference to a line containing only question marks is, that with the Else-operator, all cases that are already covered, are excluded. If a question mark spans several columns, this means for all the columns it spans, that there is no restriction.

The notation $\neg$*Action* stands for every action, but not the action *Action*. In this case, this shortens the tabular specification.

Crane1							
SOnOff = *On*							
SAutoManual	LS2In	LSMi	LSGr	LSMe	JobC2	ActionC1'	JobC1'
Auto	*true*	*false*	*false*	?	?	$H1$	$H1$
Auto	*true*	*false*	?	*true*	?	$H1$	$H1$
Auto	?	*true*	*false*	?	?	12	12
Auto	?	?	*true*	*false*	$\neg$34	23	23
Auto	Else				?	0	0
Manual	?					?	?

A.6.2 Crane2

This function works like $^{\circ}$*Crane1*, except it corresponds to different jobs. Again, to avoid collisions, this crane does the jobs 34 and $4G$ only, if the other crane does not execute 23 at the moment. The syntactic interface (Figure 19) is similar, except the different sensor inputs. The differences are caused by the different stations, crane 1 handles.

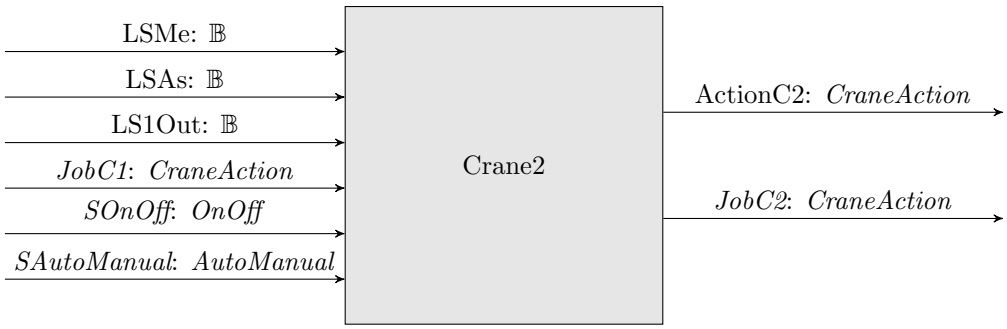

Figure 19: Syntactic interface of $\bigcirc$ *Crane2*

Crane2						
SOnOff = *On*						
SAutoManual	LSMe	LSAs	LS1Out	JobC1	ActionC2'	JobC2'
Auto	*true*	*false*	?	¬23	34	34
Auto	?	*true*	*false*	¬23	4G	4G
Auto	Else			?	0	0
Manual	?				?	?

Freedom from deadlocks and collisions has to be verified. The system works, because the cranes move back from the critical position at the measuring and assembly stations once they have done their work there. A formal proof is yet to be done.

A.7 Alert

The user has to be alerted in many cases. At this level of abstraction, not all of the required alerts can be modeled, because of insufficient information. Most of the required information will be coded into the logical or the technical perspective of the system.

A.7.1 Alert Collision

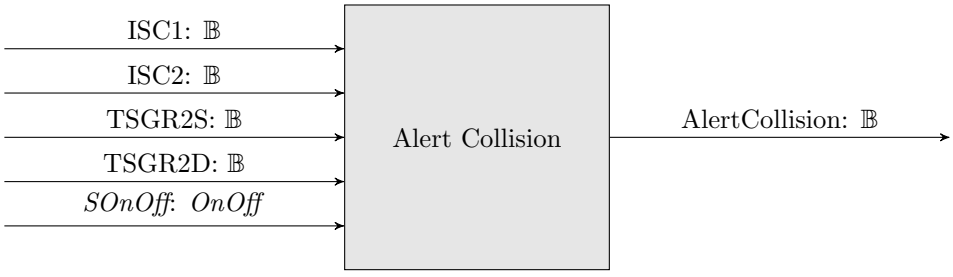

Figure 20: Syntactic interface of $\bigcirc$ *Alert Collision*

In order to alert the user in case of an impending collision, $\bigcirc$*Alert Collision* is introduced. Errors defined in [Den07] should be handled here. With just the sensor data, the system can decide if a collision of the cranes is imminent. The function $\bigcirc$*Alert Collision* decides, whether a crane collided with the ends of the guide rail or the cranes crashed.

The syntactic interface is illustrated in Figure 20. This function has to work in automatic and in manual mode, therefore it does not depend on the mode $^{\rightarrow}$*SOnOff*. To detect collisions, the input channels are the sensors $^{\rightarrow}$*ISC1*, $^{\rightarrow}$*ISC2*, $^{\rightarrow}$*TSGR2S*, and $^{\rightarrow}$*TSGR2D*. The only output channel of this function is $^{\rightarrow}$*AlertCollision*. This channel is used to notify the user of a collision.

Alert Collision				
SOnOff = *On*				
ISC1	ISC2	TSGR2S	TSGR2D	AlertCollision'
true	?	?	?	*true*
?	*true*	?	?	*true*
?	?	*true*	?	*true*
?	?	?	*true*	*true*
Else				*false*

A.7.2 Automatic Mode Alert

In Automatic mode, the production can be disturbed by some events. Mainly by arbitrarily placing workpieces into the manufacturing cell.

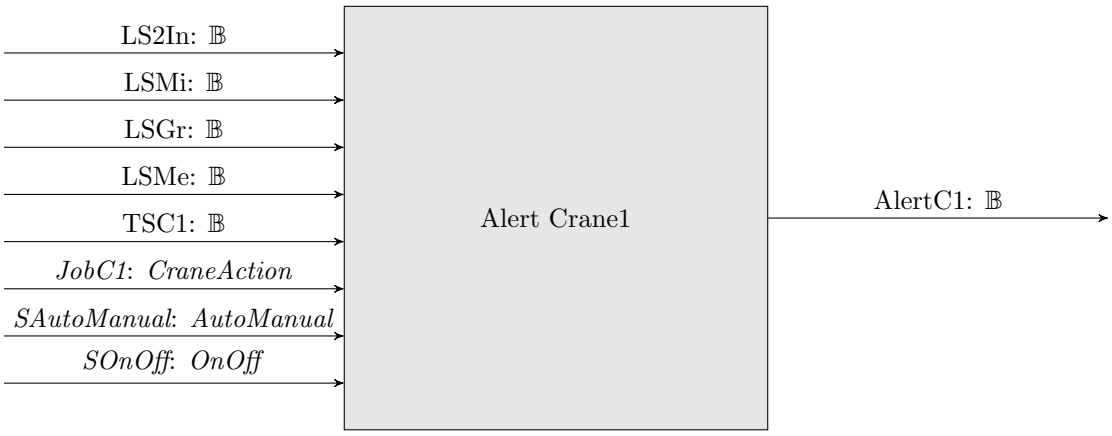

Figure 21: Syntactic interface of $\bigcirc$*Alert Crane1*

Alert Crane1 $\bigcirc$*Alert Crane1* gives an alert, if during executing a job, the configuration of workpieces becomes illegal. For example, if another workpiece appeared on the milling station during transporting a workpiece from the supply band to the milling station. The syntactic

interface is described in Figure 21. The input channels are mainly sensors, but also the current job, the crane has to do. The output channel $^{\rightarrow}$*AlertC1* is used to notify the user of an error.

Alert Crane1						
SOnOff = *On* SAutoManual = *Auto*						
LS2In	LSMi	LSGr	LSMe	TSC1	JobC1	AlertC1'
false	?	?	?	*false*	$H1$	*true*
?	*true*	?	?	*true*	$H1$	*true*
?	*false*	?	?	*false*	12	*true*
?	?	*true*	?	*true*	12	*true*
?	?	*false*	?	*false*	23	*true*
?	?	?	*true*	*true*	23	*true*
Else						*false*

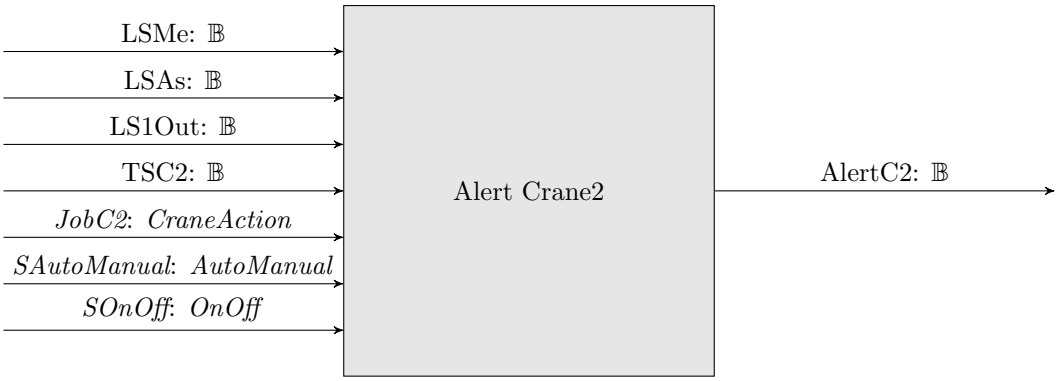

Figure 22: Syntactic interface of $^{\circ}$*Alert Crane2*

Alert Crane2 $^{\circ}$*Alert Crane2* works just like $^{\circ}$*AlertCrane1*. Also, the syntactic interface follows the same system.

Alert Crane2					
SOnOff = *On* SAutoManual = *Auto*					
LSMe	LSAs	LS1Out	TSC2	JobC2	AlertC2'
false	?	?	*false*	34	*true*
?	*true*	?	*true*	34	*true*
?	*false*	?	*false*	$4G$	*true*
?	?	*true*	*true*	$4G$	*true*
Else					*false*

B Behavior specifications for the logical architecture

The state machines introduced in this section are implementations of the components in Section 5. Because of the limited possibilities of illustrations, annotations of transitions are kept very simple. In [Vog11], all details are contained.

B.1 Switches

Figures 23 and 24 show how the corresponding components handle user inputs. If the user sends a signal that does not correspond to the current state, the state is changed. Otherwise, the system remains in the current state.

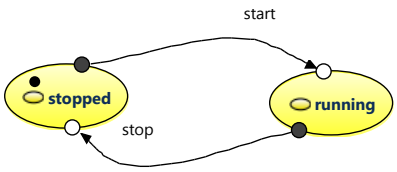

Figure 23: Automaton for switching the system on and off (Component □ *OperationController*)

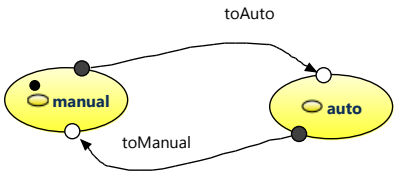

Figure 24: Automaton for switching the system into automatic and manual mode (Component □ *ModeController*)

B.2 Bands

Like the switches, also the state machines for the supply band (Figure 25) and the delivery band (Figure 26) are simple. State transitions are made according to the sensor data, the system receives. If the light sensor at the beginning of a band signals an object and the sensor at the end does not signal an object, the band is turned on (if the system is turned on, otherwise the bands are off).

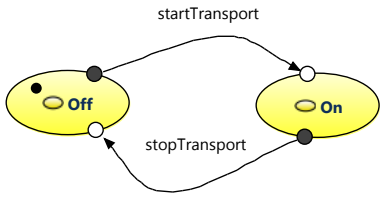

Figure 25: Automaton for switching the supply band on and off (Component □ *SupplyBand*)

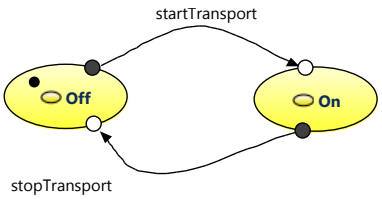

Figure 26: Automaton for switching the delivery band on and off (Component $\square$ *DeliveryBand*)

B.3 Cranes

Figure 27 shows the state machine for the component $\square$ *JobDecision*. According to sensor data and the state of each crane, it decides which job to do next for each crane. After reading the sensor inputs, the decision is made for the second crane, because its jobs have higher priority. Then, the next job for the first crane is determined. After this, the jobs are sent to the cranes.

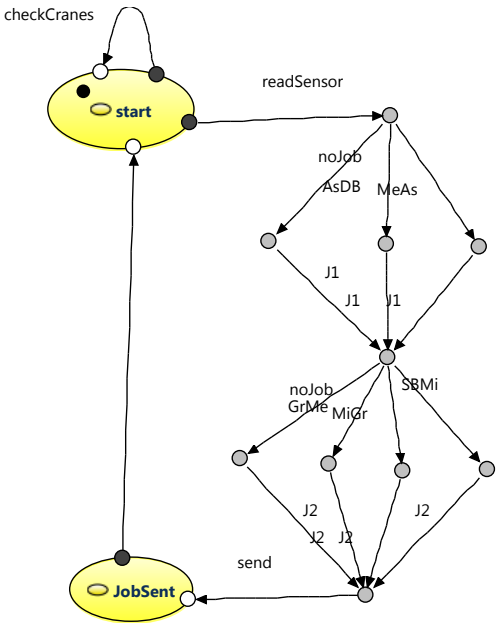

Figure 27: Automaton for $\square$ *JobDecision*

Each job is executed by the same sequence of actions: First, the crane moves to the position where a workpiece has to be picked up. Then it picks it up and moves to the destination of the workpiece. Then, the workpiece is dropped onto the station. Depending on the job that has to be done, the stations vary. Figures 28 and 29 illustrate the behavior.

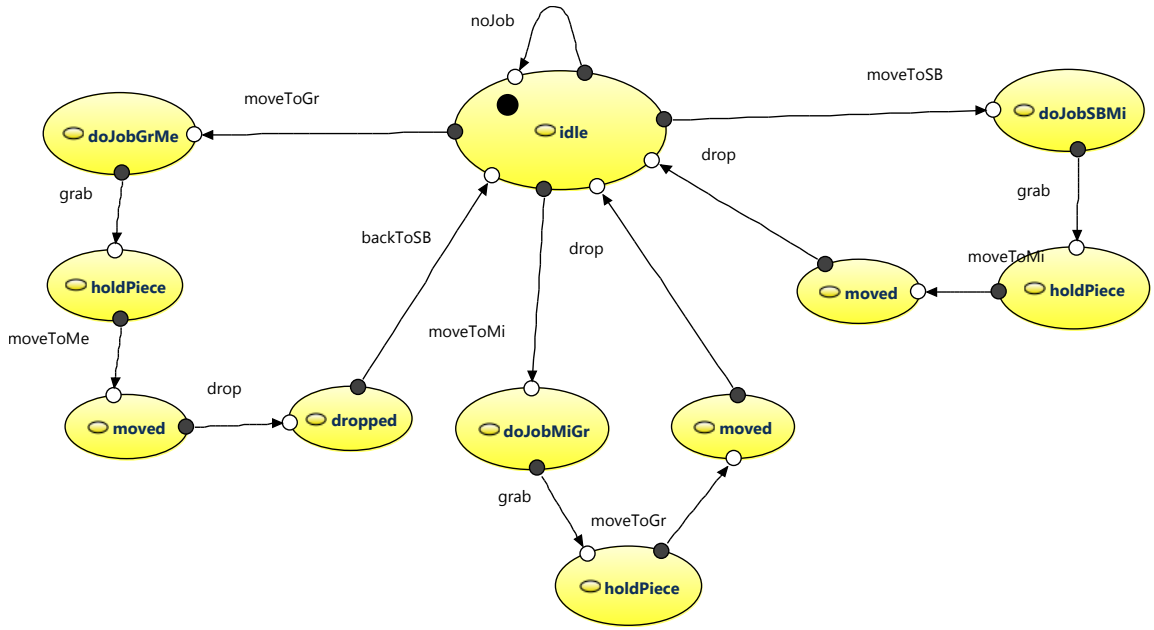

Figure 28: Automaton for $\square$ *Crane1*

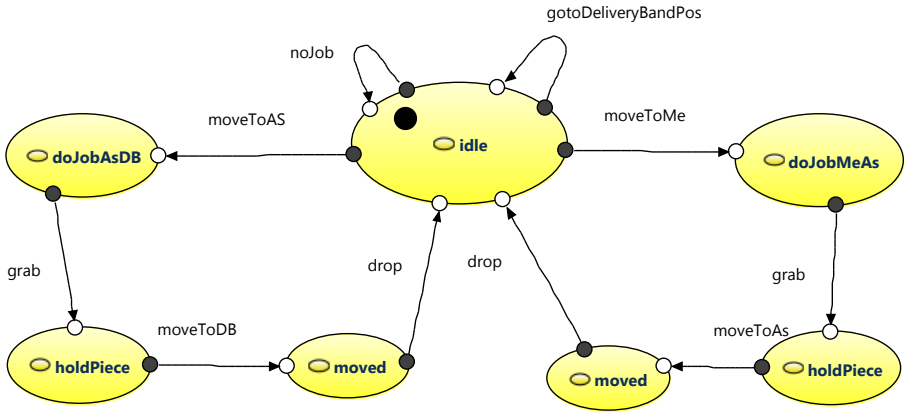

Figure 29: Automaton for $\square$ *Crane2*