Methodological Support for Formally Specifying Safety-Critical Software

Maritta Heisel
Technische Universität Berlin*

Carsten Sühl
GMD FIRST†

Abstract

We present the concept of an *agenda* and apply this concept to the formal specification of software for safety-critical applications. An agenda describes a list of activities to solving a task in software engineering, and validations of the results of the activities. Agendas used to support the application of formal specification techniques provide detailed guidance for specifiers, schematic expressions of the used specification language that only need to be instantiated, and application independent validation criteria. We present an agenda for a frequently used design of safety-critical systems and illustrate its usage by an example. Using agendas to systematically develop formal specifications for safety-critical software contributes to system safety because, first, the specifications are developed in a standardized way, making them better comprehensible for other persons. Secondly, using a formal language yields specifications with an unambiguous semantics as the starting point of further design and implementation. Thirdly, the recommended validation criteria draw the specifier's attention to common mistakes and thus enhance the quality of the resulting specification.

1 Introduction

Although every software-based system potentially benefits from the application of formal methods, their use is particularly advantageous in the development of safety-critical systems. The potential damage operators and developers of a safety-critical system have to envisage in case of an accident may be much greater than the additional costs of applying formal methods in system development. It is therefore worthwhile to develop formal methods tailor-made for the development of safety-critical systems.

A major drawback of formal techniques is that they are not easy to apply. Users of formal techniques need an appropriate education. They have to deal with lots of details, and often they are left alone with a mere formalism without any guidance on how to use it.

While nothing can be done about the first two points, it is definitely possible to provide guidance for the users of formal techniques. In this paper, we introduce the concept of an *agenda* that makes explicit the activities to be performed when developing an artifact in software engineering. We present a concrete agenda that supports specifiers in the development of specifications of software for safety-critical applications. The agenda not only makes all steps of the specification process explicit. It also provides schematic expressions of the specification language to be used and validation criteria to check the specification for consistency and completeness.

In the following, we first describe the class of systems we consider and the specification language we will use (Section 2). We then describe the concept of an agenda in more detail in Section 3. In Section 4, we present an agenda for specifying software suitable for a common class of safety-critical systems. We illustrate the usage of this agenda in Section 5. Finally, we discuss related work in Section 6 and summarize our achievements in Section 7.

*Franklinstr. 28/29, Sekr. FR 5-6, D-10587 Berlin, Germany, email: heisel@cs.tu-berlin.de, fax: +49-30-314-73488
†Rudower Chaussee 5, D-12489 Berlin, Germany, email: suehl@first.gmd.de, fax: +49-30-6392-1805

2 System Model and Specification Language

The purpose of the systems we want to consider is to control some technical process, where the control component is at least partially realized by software. Such a system consists of four parts: the technical process, the control component, sensors to communicate information about the current state of the technical process to the control component, and actuators that can be used by the control component to influence the behavior of the technical process.

A software-based control component affects certain process variables (*manipulated variables*) by sending commands to actuators. By evaluating the current state of certain process variables which are measured by sensors (*controlled variables*), the control component approximates the current state of the real process to verify the effect of the commands sent to the actuators and to determine further commands to be sent. It is very important that the image of the state of the technical process that is built up in the software control component is sufficiently accurate and up-to-date. In the following, we will call this state the *internal* state, because it is internal to the software control component; the state of the technical process we will call the *external* state.

Most safety-critical systems are *reactive*. Hence, two aspects are important for the specification of software for safety-critical systems. First, it must be possible to specify behavior, i.e. how the system reacts to incoming events. Second, the structure of the system's data state and the operations that change this state must be specified. These requirements lead us to use a combination of the process algebra real-time CSP [Dav93] and the model-based specification language Z [Spi92].

For each system operation *Op* specified in the Z part of a specification, the CSP part is able to refer to the event *OpExecution*. For each input or output of a system operation defined in Z, there is a communication channel within the CSP part onto which an input value is written or an output value is read from. The dynamic behavior of a software component may depend on the current internal system state. To take this requirement into account, a process of the CSP part is able to refer to the current internal system state via predicates which are specified in the Z part by schemas. For a more detailed description, see [HS96, Hei97].

3 Agendas and Reference Architectures

An agenda is a list of activities to be performed when carrying out some task. In this paper, we consider the application of formal specification techniques in the context of system safety. Here, agendas contain informal descriptions of the activities. They may also contain schematic expressions of the formal specification language that can be instantiated in carrying out the activity. The activities listed in an agenda may depend on each other. Usually, they will have to be repeated to achieve the goal, like in the spiral model of software engineering.

As one of the major reasons for applying formal techniques is to guarantee semantic properties of an artifact, the activities of an agenda may have validation conditions associated with them. These validation conditions state *necessary* semantic conditions that the artifact must fulfill in order to serve its purpose properly. Since the verification conditions that can be stated in an agenda are necessarily application independent, the developed artifact should be further validated with respect to application dependent needs.

Following an agenda gives no guarantee of success. Agendas cannot replace creativity, but they can tell the user what needs to be done and can help avoid omissions and inconsistencies. Their use lies in an improvement of the quality of the developed products and in the possibility for reusing the knowledge incorporated in an agenda.

In this paper, we show how agendas can be profitably employed to specify software to control safety-critical systems, as the ones described in Section 2. There are several ways to design safety-critical systems, according to the manner in which activities of the control component take place, and the manner in which system components trigger these activities. These different approaches to the design of safety-critical systems can be expressed as *reference architectures*. The architecture we consider here assumes that sensors are no passive measuring devices but can cause interrupts in the control component.

The agenda we present for this *active sensors architecture* describes the steps to be taken to specify a suitable software control component. It provides schematic expressions of Z or real-time CSP that only need to be instantiated, and states validation obligations that should be fulfilled. Our general approach to

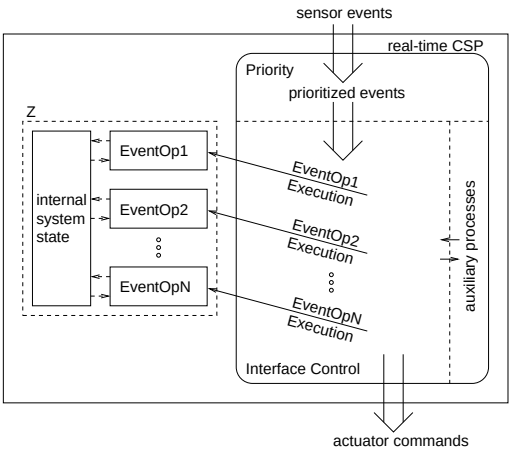

Figure 1: Software Control Component for Active Sensors Architecture

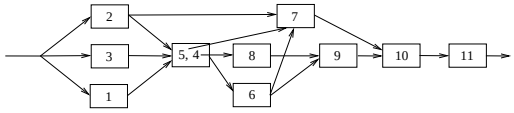

Figure 2: Dependencies of Steps

the specification of safety-critical software is to first decide on the architecture of the system for which a software control component must be specified, and then to follow the steps of the corresponding agenda.

The aim of this work is not to completely cover the area of specification of safety-critical systems, but to show that detailed guidance can be provided to specifiers of software systems if special contexts are considered. Consequently, other reference architectures for the design of safety-critical systems are useful, too. An example can be found in [Hei97]. Furthermore, concrete safety-critical systems need not be "pure" instances of predefined architectures. When necessary, reference architectures can be combined as appropriate.

4 An Agenda for the Active Sensors Architecture

An active sensor controls a certain variable of the technical process and independently reports certain changes of the controlled variable to the control component at arbitrary time instants. Such a report immediately triggers a handling operation within the control component. The active sensors architecture is applicable to systems with only active sensors. Moreover, we assume that it is appropriate to distinguish several *operational modes* of the system. Within distinct modes, which can model different environmental or internal conditions, the behavior of the system – and thus of the control component – may be totally different.

Figure 1 shows the structure of a software control component associated with the active sensors architecture. The CSP part of such a control component consists of three parallel processes. A *Priority* process receives the sensor events from the environment. If several events occur at the same time, this process defines which of these events is treated with priority. Depending on the prioritized events passed on from the *Priority* process, an *InterfaceControl* process invokes a Z operation to update the internal state of the software controller. Hence, the Z operations correspond to events that cause transitions between operational modes. The *InterfaceControl* process is also responsible for sending actuator commands to the environment. Finally, there may be auxiliary processes that interact only with the *InterfaceControl* process, not with the environment or with the *Priority* process. The active sensors architecture is suitable for systems that continuously have to react to user commands or other stimuli from the environment.

An overview of the agenda for this architecture is given in Table 1. The steps shown there need not be executed exactly in the given order. Figure 2 shows the dependencies between them. In the following, we describe the steps in more detail; their associated validation conditions are explained when necessary. For a more comprehensive presentation, see [Hei97].

Step 1 *Model the sensors and actuators as sets of CSP events or Z types.*
Sensors trigger operations of the control component. If a sensor carries a measured value, it is modeled as

No.	Step	Validation Condition
1	Model the sensors and actuators as sets of CSP events or Z types.	
2	Decide on auxiliary processes.	
3	Decide on the operational modes of the system and the initial modes.	
4	Set up a mode transition relation, specifying which events relate which modes.	4.1. All events identified in Step 1 and all modes defined in Step 3 must occur in the transition relation. 4.2. The omission of a mode-event pair from the relation must be justified. 4.3. All modes must be reachable from an initial mode.
5	Define the internal system states and the initial states.	5.1. The internal system state must be an appropriate approximation of the state of the technical process. 5.2. Each legal state must be safe. 5.3. There must exist legal initial states. 5.4. For each initial internal state, the controller must be in an initial mode.
6	Specify a Z operation for each mode transition contained in the mode transition relation.	6.1. These operations must be consistent with the mode transition relation.
7	Define the auxiliary processes identified in Step 2.	7.1. The alphabets of these processes must not contain external events or events related to the Z part of the specification.
8	Specify priorities on events (optional).	8.1. The priorities must not be cyclic.
9	Specify the interface control process.	9.1. All prioritized external events and all internal events must occur as initial events of the branches of the interface control process. 9.2. The preconditions of the invoked Z operations must be satisfied.
10	Define the overall control process.	10.1. The auxiliary processes must communicate with the interface control process.
11	Define further requirements or environmental assumptions if necessary.	

Table 1: Agenda for the Active Sensors Architecture

a Z type whose members are communicated via a communication channel. If a sensor just carries boolean information (i.e., something happens or not), it is modeled as a set of CSP events. Modeling the sensors yields a set of events *Sensor_Events*. Modeling the actuators proceeds analogously.

Step 2 *Decide on auxiliary processes.*
One can regard these auxiliary processes as subcomponents of the controller that do not need a state. Examples are timers that are controlled by the software component. The events that are used in the auxiliary processes to communicate with the rest of the control component form the set *Internal_Events*.

Step 3 *Decide on the operational modes of the system and the initial modes.*
According to different environmental conditions in which the control component has to react differently, a set of operational modes has to be defined yielding an enumeration type *MODE*. Moreover, modes must be identified in which the controller can be initialized.

Step 4 *Set up a mode transition relation, specifying which events relate which modes.*
This transition relation can be defined in Z, or it can be given as a state transition diagram. For each

operational mode m and each event e (which can be internal or external), it must be decided on the successor modes that are possible when event e occurs in mode m. It should also be specified what happens when the sensors report an event that cannot normally occur in the respective mode. Hence, the mode transition relation should be made as complete as possible, and a justification should be given, if no successor mode is defined for a pair (m, e).

Step 5 *Define the internal system state and the initial states.*
The legal states of the control component as well as its initial states must be specified by means of Z schemas. Furthermore, safety constraints on controller must be defined. These safety constraints have to be ensured by the definition of the legal states, see validation condition 5.2.

Step 6 *Specify a Z operation for each mode transition contained in the mode transition relation.*
The Z operations correspond to mode transitions caused by the occurrence of events. Different mode transitions might be associated with the same operation. The operations must be consistent with the state transition relation.

Step 7 *Define the auxiliary processes identified in Step 2.*
This step can be performed by defining process terms or by specifying predicates that restrict the behavior of the respective processes. The auxiliary processes should neither receive external sensor messages nor invoke Z operations or depend on the internal system state. They should exclusively interact with the *InterfaceControl* process, see Figure 1.

Step 8 *Specify priorities on events if necessary.*
To determine if priorities are necessary, we have to analyze the state transition diagram. If more than one event can occur at the same time when the system is in a certain operational mode, it must be decided how the system reacts when several events occur simultaneously. Usually, the event with the highest importance for safety will be treated; the other ones will be ignored.

Technically, this means to define derived events and a process *Priority* that relates the original events with the derived ones. If we have a high priority event *high* and a low priority event *low*, then the system will only react to the event *low* if *high* does not occur at the same time. Therefore, an event *excl_low* is derived that occurs at time t exactly when *low* but not *high* occurs at time t.

Step 9 *Specify the interface control process.*
The interface control process handles the prioritized events coming from the sensors. According to the internal or sensor events that occur, the interface control process triggers the execution of Z operations and sends events to actuators or auxiliary processes. The interface control process will usually contain an external choice of prioritized events. Each branch of this external choice should be robust, i.e., if the sensors send signals that contradict the internal state of the system, then the system must handle the faulty situation. Consequently, a possible form of interface control process that is executed after the system is initialized is

$$
\begin{array}{l}
InterfaceControl_{READY} \mathrel{\widehat{=}} \mu X \bullet \\
\quad event_1 \rightarrow \text{if } \langle consistency\ condition \rangle \\
\qquad \text{then } \langle execute\ event_1\text{-}Z\text{-}operation \rangle \ \langle send\ actuator\ commands \rangle ;\ Wait_{\epsilon} ;\ X \\
\qquad \text{else } \langle emergency\ shutdown \rangle \ \langle send\ actuator\ commands \rangle ;\ Stop \text{ fi} \\
\Box \ldots \Box \\
\quad event_n \rightarrow \ldots
\end{array}
$$

All branches are defined similarly. To ensure that the execution of each branch takes time, we need a *Wait* process in each of them. This form is only possible if there is a fail-safe state. Then the system can shut down when an inconsistency is detected. To express the consistency conditions, predicates on the current internal system state must be defined in Z.

According to validation condition 9.2, the preconditions of the invoked Z operations must be satisfied. This must be guaranteed by appropriate consistency conditions guarding the invocation of the Z operations in the interface control process.

Step 10 *Define the overall control process.*

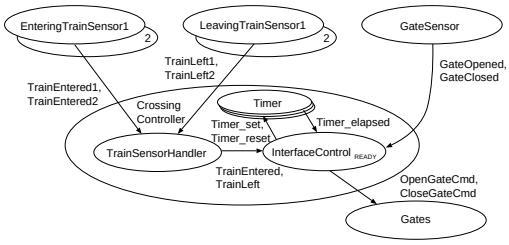

Figure 3: Architecture of Controller

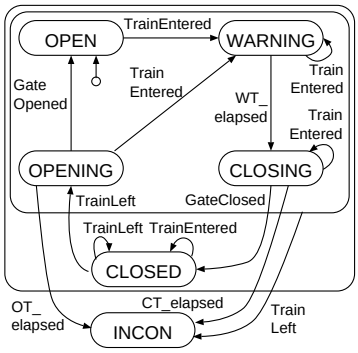

Figure 4: Operational Modes

The control process combines the processes defined in Steps 7, 8, and 9. Let $Aux_1, \ldots, Aux_k$ be the auxiliary processes defined in Step 7. Then a possible form of the overall control process is

$$ControlComponent \mathrel{\widehat{=}} (InterfaceControl \parallel Aux_1 \parallel \ldots \parallel Aux_k) \setminus Internal\_Events$$

$$InterfaceControl \mathrel{\widehat{=}} Init \rightarrow (InterfaceControl_{READY} \parallel Priority) \setminus (\alpha Priority \setminus (Sensor\_Events \cup Internal\_Events))$$

where *Init* establishes an initial internal system state. The internal events are hidden from the environment, and the prioritized events newly introduced in the alphabet of the *Priority* process are hidden from the other components of the controller (and hence from the environment).

Checking validation condition 10.1, it must be verified that the auxiliary processes communicate with the interface control process. Technically, this means that the alphabets of $(Aux_1 \parallel \ldots \parallel Aux_k)$ and *InterfaceControl* have a non-empty intersection.

Step 11 *Define further requirements or environmental assumptions if necessary.*
Usually, these will be assumptions on the environment and real-time requirements on the execution time of Z operations.

5 Example: Railroad Crossing

We illustrate the agenda by specifying a software component controlling the gates of a railroad crossing, known as the Generalized Railroad Problem, see e.g. [HM96]. We assume that there are two tracks at the crossing, one in each direction.

We begin with **Step 1** of the agenda, i.e. the definition of the sensors and actuators. For each direction, a sensor is situated in front of the crossing to report the arrival of trains. The arrival of trains are represented by the CSP events $TrainEntered1$ and $TrainEntered2$, respectively. Analogously, behind the crossing there are sensors reporting the departure of trains represented by the events $TrainLeft1$ and $TrainLeft2$.

The controller manipulates the gates of the crossing by giving the commands to close and to open the gates (events *CloseGateCmd* and *OpenGateCmd*). Sensors detect the transition of the gates into a completely closed or completely opened state.[1] These transitions are modeled by the events *GateOpened* and *GateClosed*. The interface of the controller to its environment is shown in Figure 3.

According to **Step 3** of the agenda, we define a data type *Modes* that contains the operational modes of the controller.

$$Modes ::= OPEN \mid OPENING \mid CLOSED \mid CLOSING \mid WARNING \mid INCON$$

The mode *OPEN* will be the only initial mode of the controller.

[1] We only regard the state of the gates as a whole, i.e. the gates are regarded to be open or closed if and only if all gates are open or closed.

To define the transitions between the operational modes, which are associated with sensor events and internal events, we have to decide on auxiliary processes (**Step 2**). The opening and closing of the gates must be finished within certain time intervals, whose lengths are expressed as constants *MaxOpeningDur* and *MaxClosingDur*, respectively. Car drivers and pedestrians are warned *WarningDur* time units before the gates are closed.

$$MaxOpeningDur, MaxClosingDur, WarningDur : \mathbb{N}_1$$

To monitor these intervals we use three timer components *WarningTimer*, *OpeningTimer*, and *ClosingTimer*, which will mark the end of these intervals by an event *Timer_elapsed*. Their alphabets, i.e. the set of events in which they can participate is

$$Internal\_Events \mathrel{\widehat{=}} \{WT\_set, WT\_reset, WT\_elapsed, OT\_set, OT\_reset, OT\_elapsed, CT\_set, CT\_reset, CT\_elapsed\}$$

The operational modes and the transitions between them are depicted in Figure 4. This figure graphically defines the mode transition relation to be set up in **Step 4** of the agenda. As can be easily verified, all external and internal events are taken into account in the state transition diagram, for each mode-event pair the consequence is considered, and each mode is reachable from the initial mode *OPEN*. Hence, the validation conditions associated with Step 4 are fulfilled.

Performing **Step 5** of the agenda, we decide that the state components of the controller are the current operational mode and the number of trains currently being in the crossing area. The safety constraint for the controller is that there must not be any train in the area while the gates are open. An availability requirement is that the gates may only be closed if there are trains in the crossing area. These conditions are reflected in the state schema.

State

$$mode : Modes$$
$$num\_trains : \mathbb{N}$$

$$mode \in \{OPEN, OPENING\} \Rightarrow num\_trains = 0$$
$$num\_trains = 0 \Rightarrow mode \in \{OPEN, OPENING, INCON\}$$

While validation condition 5.1 cannot be shown formally, validation condition 5.2 is satisfied by construction because the safety constraint is part of the state invariant.

When starting the controller, it is assumed that no train is in the area and that the gates are open.

$$Init \mathrel{\widehat{=}} [\, State' \mid mode' = OPEN \land num\_trains' = 0 \,]$$

The *Init* schema uniquely defines the initial state, whose compliance with the state invariant is straightforward to prove (validation condition 5.3). By the definition of the *Init* schema, validation condition 5.4 is trivially fulfilled.

To carry out **Step 6**, we model the possible commands to the gates by the data type *GateCmd*.

$$GateCmd ::= CMD\_OPEN \mid CMD\_CLOSE \mid CMD\_NONE$$

The schema *Actuators* defines how the actuator commands are derived from the operational mode of the controller.

Actuators

$$State'$$
$$gate! : GateCmd$$

$$gate! = CMD\_OPEN \Leftrightarrow mode' = OPENING$$
$$gate! = CMD\_CLOSE \Leftrightarrow mode' \in \{CLOSING, INCON\}$$

We can now define the operations on the abstract state. They reflect the mode transition diagram of Figure 4 and are straightforward to define.

TrainEntered

$\Delta State;\ Actuators$

$mode \neq INCON$

$num\_trains' = num\_trains + 1$
$mode \in \{OPEN, OPENING\} \Rightarrow mode' = WARNING$
$mode \in \{CLOSING, CLOSED, WARNING\} \Rightarrow mode' = mode$

TrainLeft

$\Delta State;\ Actuators$

$mode = CLOSED$

$num\_trains' = num\_trains - 1$
$mode' =$ **if** $num\_trains' = 0$
then *OPENING* **else** *CLOSED*

$$GateOpened \mathrel{\widehat{=}} [\, \Delta State;\ Actuators \mid mode = OPENING \land mode' = OPEN \land num\_trains' = num\_trains \,]$$

The remaining operations *GateClosed*, *WarningFin* and *Incon* (that enters the fail-safe state), are defined analogously.

The relationship between the modes in the pre and post state, as defined by the operation schemas, directly reflects the state transition diagram. Therefore, validation condition 6.1 can be checked successfully.

Continuing with **Step 7**, we specify the auxiliary timer subcomponents which monitor the maximal closing and opening duration of the gates as well as the warning period.

$$WarningTimer \mathrel{\widehat{=}} \mu X \bullet (WT\_set \rightarrow (Wait\, WarningDur;\ WT\_elapsed \rightarrow X) \triangle WT\_reset \rightarrow X)$$

The processes *ClosingTimer* and *OpeningTimer* are defined analogously.

Inspecting the mode transition diagram reveals that there is no need to give priority to some event in any mode. Hence, **Step 8** is omitted.

To accomplish **Step 9**, we first specify a process *TrainSensorHandler*, which serves two purposes. First, it abstracts from the concrete sensor reporting the event of a train arrival or a train departure. Second, it buffers incoming events, thus guaranteeing that no sensor report is ignored by the controller.

$$\begin{aligned} &TrainSensorHandler \mathrel{\widehat{=}} \\ &(\mu X \bullet TrainEntered1 \rightarrow TrainEntered \rightarrow X) \;|||\; (\mu X \bullet TrainEntered2 \rightarrow TrainEntered \rightarrow X) \\ &|||\; (\mu X \bullet TrainLeft1 \rightarrow TrainLeft \rightarrow X) \;|||\; (\mu X \bullet TrainLeft2 \rightarrow TrainLeft \rightarrow X) \end{aligned}$$

We can then specify the interface control process as follows.

$$InterfaceControl \mathrel{\widehat{=}} Init \rightarrow (InterfaceControl_{READY} \,|[\, EventSet \,]|\, TrainSensorHandler) \setminus EventSet$$

$$EventSet \mathrel{\widehat{=}} \{TrainEntered, TrainLeft\}$$

The process $InterfaceControl_{READY}$ reacts to the derived events contained in *EventSet* that are supplied by the process *TrainSensorHandler*. The derived events are hidden from the environment.

With the subprocess *ActuatorOut*, we specify how the actuator command is derived from the respective operation output.

$$\begin{aligned} &ActuatorOut \mathrel{\widehat{=}} gate?CMD\_OPEN \rightarrow OpenGateCmd \rightarrow Skip \\ &\quad \Box\ gate?CMD\_CLOSE \rightarrow CloseGateCmd \rightarrow Skip \;\Box\; gate?CMD\_NONE \rightarrow Skip \end{aligned}$$

The process $InterfaceControl_{READY}$, specifying the reactive behavior after initialization, contains an instance of the schematic process presented in Section 4. To define it, we need some predicates on the internal state, which are defined in Z.

$$IsClosed \mathrel{\widehat{=}} [\, State \mid mode = CLOSED \,] \qquad IsWarning \mathrel{\widehat{=}} [\, State \mid mode = WARNING \,]$$
$$IsOpening \mathrel{\widehat{=}} [\, State \mid mode = OPENING \,] \qquad AreaEmpty \mathrel{\widehat{=}} [\, State \mid num\_trains = 0 \,]$$
$$IsClosing \mathrel{\widehat{=}} [\, State \mid mode = CLOSING \,] \qquad FirstTrain \mathrel{\widehat{=}} [\, State \mid num\_trains = 1 \,]$$

$$
\begin{array}{l}
\mathit{InterfaceControl}_{\mathit{READY}} \mathrel{\widehat{=}} \mu\, X \bullet \\
\quad (\mathit{TrainEntered} \rightarrow \mathit{TrainEnteredExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut}; \\
\qquad (\text{if } \mathit{FirstTrain} \text{ then } \mathit{WT\_set} \rightarrow \mathit{Skip} \text{ else } \mathit{Skip} \text{ fi});\ X \\
\quad \Box\ \mathit{TrainLeft} \rightarrow \text{if } \mathit{IsClosed} \text{ then } \mathit{TrainLeftExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut}; \\
\qquad\quad (\text{if } \mathit{AreaEmpty} \text{ then } \mathit{OT\_set} \rightarrow \mathit{Skip} \text{ else } \mathit{Skip} \text{ fi});\ X \\
\qquad \text{else } \mathit{InconExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{Stop} \text{ fi} \\
\quad \Box\ \mathit{GateOpened} \rightarrow \text{if } \mathit{IsOpening} \\
\qquad \text{then } \mathit{GateOpenedExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{OT\_reset} \rightarrow X \\
\qquad \text{else } \mathit{InconExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{Stop} \text{ fi} \\
\quad \Box\ \mathit{GateClosed} \rightarrow \text{if } \mathit{IsClosing} \\
\qquad \text{then } \mathit{GateClosedExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{CT\_reset} \rightarrow X \\
\qquad \text{else } \mathit{InconExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{Stop} \text{ fi} \\
\quad \Box\ \mathit{WT\_elapsed} \rightarrow \text{if } \mathit{IsWarning} \\
\qquad \text{then } \mathit{WarningFinExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{CT\_set} \rightarrow X \\
\qquad \text{else } \mathit{InconExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{Stop} \text{ fi})) \\
\triangle\ (\mathit{OT\_elapsed} \rightarrow \mathit{Skip} \Box\ \mathit{CT\_elapsed} \rightarrow \mathit{Skip});\ \mathit{InconExecution} \rightarrow \mathit{Wait}\,\epsilon;\ \mathit{ActuatorOut};\ \mathit{Stop}
\end{array}
$$

The reactive behavior is recurrent which is indicated by the recursion operator μ. When an event is reported by a sensor, the controller checks whether this report is consistent with the current state. Such a consistency check is modeled by referring to the respective Z predicate. If the event is consistent with the internal state, the controller executes the system operation related to the sensor report. Afterwards, the actuator command determined by the operation are sent to the actuators. In case of an inconsistency, the controller changes to the mode *INCON* and attempts to fail safe by closing the gates.

Each branch of the external choice operator $\Box$ defines the reaction to the occurrence of a certain event. All derived sensor events and internal timer events are initial events of the branches (validation condition 9.1). Moreover, the predicates guarding the branches guarantee that the preconditions of the corresponding Z operations are fulfilled (validation condition 9.2).

This "normal" behavior in front of the interrupt operator $\triangle$ can be interrupted at any time, if a timer, monitoring the gates' opening or closing duration, elapses without being reset.

Having reached **Step 10** of the agenda, we specify the overall control process according to the schematic expression given in Section 4.

$$
\begin{array}{l}
\mathit{CrossingController} \mathrel{\widehat{=}} \\
\quad (\mathit{InterfaceControl} \,|[\, \mathit{Internal\_Events} \,]|\, (\mathit{WarningTimer} \,|||\, \mathit{OpeningTimer} \,|||\, \mathit{ClosingTimer})) \\
\quad \setminus\ \mathit{Internal\_Events}
\end{array}
$$

The auxiliary timer processes communicate with the interface control process via the events contained in the set *Internal_Events* defined in Step 2. Hence, validation condition 10.1 is fulfilled.

Finally, performing **Step 11** of the agenda, we require that, at each time instant, the actuators are able to accept the commands to open or close the gates. Furthermore, the sensors are supposed not to send contradictory events[2].

$$
\begin{array}{l}
\mathit{EnvironmentalAssumption} \mathrel{\widehat{=}} \forall\, t : [0, \infty) \bullet \\
\quad ((\mathit{OpenGateCmd}\ \mathsf{open}\ t \wedge \mathit{CloseGateCmd}\ \mathsf{open}\ t) \wedge \\
\quad \neg\ (\mathit{GateOpened}\ \mathsf{open}\ t \wedge \mathit{GateClosed}\ \mathsf{open}\ t))
\end{array}
$$

This concludes the development of a specification for the controller of the railway crossing. The example shows that, following an agenda, specifications for safety-critical software can be developed in a fairly routine way. When specifiers are relieved from the task to find new ways of structuring a specification for each new application, they can better concentrate on the peculiarities of the application itself.

[2] The predicate e open t means that the environment of a process offers event e at time t.

6 Related Work

The work presented in this paper further elaborates the results of an earlier paper [HS96]. There, we described related work presenting different formalisms for specifying safety-critical software and case studies using these formalisms. The focus of this paper, however, is on *methodological* support for applying formal techniques. Related to this aim is the work of Souquières and Lévy [SL93]. They support specification acquisition with *development operators* that reduce *tasks* to subtasks. However, they do not consider safety-related issues, and the development operators do not provide means to validate the developed specification. Chernack [Che96] uses a concept called *checklist* to support inspection processes. In contrast to agendas, checklists presuppose the existence of a software artifact and aim at detecting defects in this artifact.

7 Conclusions

With the concept of an agenda, we have introduced a means for organizing work that has to be carried out in a particular context. The purpose of agendas is to capture the knowledge used by the domain experts when carrying out their tasks. Agendas are specific to the task to be performed, not to the formalism to be used. Therefore, the use of agendas can be smoothly introduced into an organization. Developers essentially proceed as before, only that the steps to be taken in performing the task are made explicit. Thus, agendas contribute to making formal techniques applicable for non-experts.

Based on a reference architecture capturing a common design of safety-critical systems, we have presented an agenda that gives detailed guidance (i) for developing specifications of software components suitable for the architecture and (ii) for validating the component specifications. An example demonstrated the practicality of the agenda.

Using agendas, specification acquisition is performed in a standardized way. This not only supports specifiers but also other persons who must understand the specification process and its results, for example, because they must change or further develop the specification.

All in all, our approach to supporting the specification of software for safety-critical applications enhances the safety of the entire technical system, because the embedded software is specified in a systematic way, the specification has an unambiguous semantics, and the specification is validated more rigorously than this would be possible with informal specifications.

Acknowledgment. We thank Thomas Santen for his comments on this work.

References

[Che96] Yuri Chernack. A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering*, 22(12):866–874, December 1996.

[Dav93] Jim Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[Hei97] Maritta Heisel. *Improving Software Quality with Formal Methods: Methodology and Machine Support*. Habilitation Thesis, TU Berlin, 1997. submitted.

[HM96] Constance Heitmeyer and Dino Mandrioli, editors. *Formal Methods for Real-Time Computing*, chapter 1. Trends in Software. John Wiley & Sons, 1996.

[HS96] Maritta Heisel and Carsten S¨uhl. Formal specification of safety-critical software with Z and real-time CSP. In E. Schoitsch, editor, *Proceedings 15th International Conference on Computer Safety, Reliability and Security*, pages 31–45. Springer, 1996.

[SL93] Jeanine Souquières and Nicole Lévy. Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223, 1993.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.