# An Agenda for Specifying Software Components with Complex Data Models

Kirsten Winter[1], Thomas Santen[1], and Maritta Heisel[2]

[1] GMD FIRST,Rudower Chaussee 5, D-12489 Berlin, Germany,
kirsten.winter@first.gmd.de, santen@first.gmd.de
[2] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, D-39016 Magdeburg, Germany, heisel@cs.uni-magdeburg.de

**Abstract.** We present a method to specify software for a special kind of safety-critical embedded systems, where sensors deliver low-level values that must be abstracted and pre-processed to express functional and safety requirements adequately. These systems are characterized by a *reference architecture*. The method is expressed as an *agenda*, which is a list of activities to be performed for setting up the software specification, complemented by validation conditions that help detect and correct errors. The specification language we use is a combination of the formal notation Z and the diagrammatic notation statecharts. Our approach not only provides detailed guidance to specifiers, but it is also part of a more general engineering concept for engineering safety-critical embedded systems that was developed in the ESPRESS project, a joint project of academia and industry.

## 1 ESPRESS: Engineering of Safety-Critical Embedded Systems

The work we present in this paper has been carried out in the context of the ESPRESS project during the last two years[1]. In ESPRESS, we investigate development methods for software to be used as part of safety-critical embedded systems. We favor the application of formal methods for this purpose. Even though every software-based system potentially benefits from the application of formal techniques, their use is of particular advantage for the development of safety-critical embedded systems, because the potential damage operators and developers have to envisage in case of malfunction may be much worse than the additional costs of applying formal techniques in system development.

Figure 1 shows the basic ESPRESS process model. The agenda presented in this paper guides the development of a requirements specification. Such a requirements specification is further validated and serves as a basis for safety analyses, test case generation, and software design.

We use the ESPRESS notation $\mu\mathcal{SZ}$ [1] to express the specifications developed with our agenda. This notation provides a semantically well-defined combination of the Statemate languages [6] (namely statecharts and activity charts), the formal specification language Z [15], and an extension of Z by temporal logics [2]. The Statemate languages and Z have been chosen for ESPRESS because of their relevance in industrial

[1] The ESPRESS project is a cooperation of industry and research institutes funded by the German ministry BMBF ("Förderschwerpunkt Softwaretechnologie", grant 01 IS 509 C6).

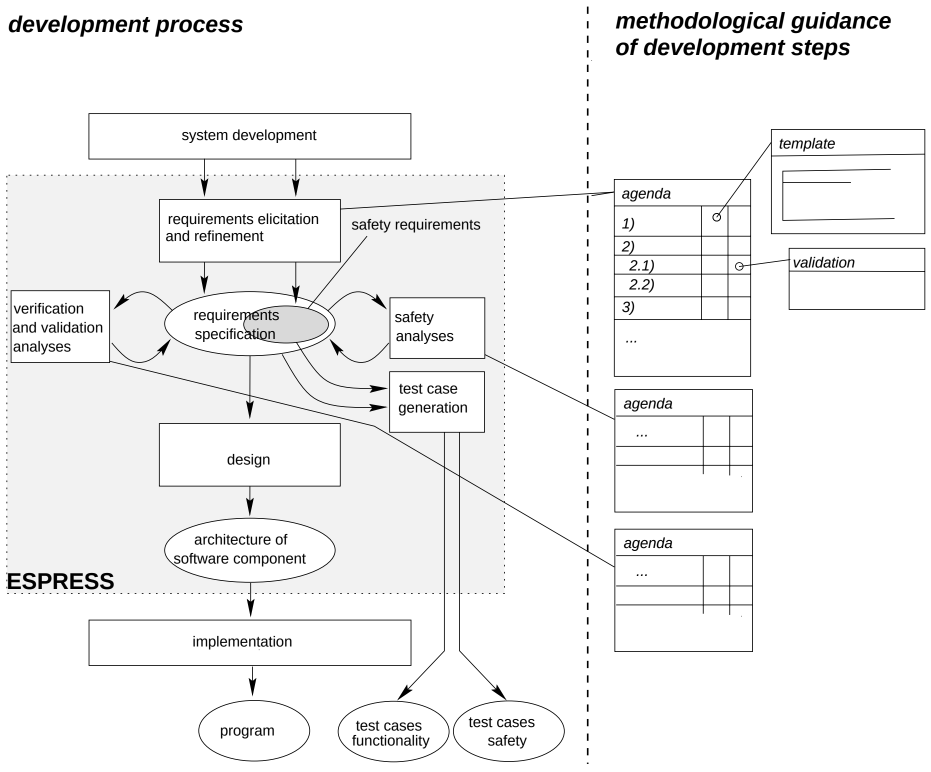


**Fig. 1.** Basic ESPRESS process model

contexts and their fairly good tool support. For reasons of space, we cannot systematically explain $\mu\mathcal{SZ}$ and its constituting languages; we only give an informal explanation of the constructs used in this paper as they appear.

## 2 Agendas

An agenda [7] gives guidance on how to perform a specific software development activity. It consists of a list of steps to be performed when carrying out some task in the context of software engineering. The result of the task will be a document expressed in a certain language. Agendas contain informal descriptions of the steps. With each step, *templates* of the language in which the result of the task is expressed are associated. The templates are instantiated when the step is performed. The steps listed in an agenda may depend on each other. Usually, they will have to be repeated to solve the given task. Agendas are presented as tables, see Figure 5. Agendas may be nested, and we call the "super-steps" *stages* (see Table 1).

Agendas are not only a means to guide software development activities. They also support quality assurance because the steps of an agenda may have validation conditions associated with them. These validation conditions state necessary conditions that the artifact must fulfill in order to serve its purpose properly. When formal techniques are applied, some of the validation conditions can be expressed and proven in a formal way. Such validation conditions are marked "$\vdash$". Since the validation 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.

Working with agendas proceeds as follows: first, the software engineer selects an appropriate agenda for the task at hand. Usually, several agendas will be available for the same development activity, which capture different approaches to perform the activity. Once the appropriate agenda is selected, the further procedure is fixed to a large extent. Each step of the agenda must be performed, in an order that respects the dependencies of steps. The informal description of the step informs the software engineer about the purpose of the step. The templates associated with the step provide the software engineer with patterns that can be filled in or modified according to the needs of the application at hand. The result of each step is a concrete expression of the language that is used to express the artifact. If validation conditions are associated with a step, they should be checked immediately to avoid unnecessary dead ends in the development. When all steps of the agenda have been performed, a product has been developed that can be guaranteed to fulfill certain application-independent quality criteria. This product should then be subject to further validation, taking the specific application into account.

## 3 Reference Architecture: Software Components with Complex Data Models

A reference architecture describes a class of software components that share common principles. We sketch the reference architecture for embedded software components with complex data models, henceforth called CDM reference architecture. An instance of this class of software components, the safety-controller of a traffic light system, serves us to illustrate the agenda for that reference architecture. We introduce the traffic light system in Section 4.

Software components of embedded systems often have a relatively simple data model. Although a mathematical model of the requirements of such a system may be complex, e.g., a system of differential equations, and the resulting software may involve non-trivial algorithms, it is often possible to express the functional requirements as a direct relation between the values of controlled variables, which are measured by sensors, and values of manipulated variables. Examples are small automotive controllers, such as cruise control systems, or controllers of household appliance. In earlier work, we identified two reference architectures, called *cyclic software component* and *active sensors*, for systems with a simple data model. Agendas for these systems are described in [5, 9].

In the present paper, we consider software components that are characterized by the fact that the sensors deliver low-level values (e.g., sequences of "on" and "off" values), and for which no theory exists that relates them to the high-level notions as they are used by domain experts in their discourses about the problem domain. In such a situation, we cannot easily describe the requirements for a software component by a direct relation between sensor values and actuator commands. Instead, the sensor values must first be abstracted and interpreted appropriately to deduce the state of the technical system in which the software is embedded and which is modeled on a higher level of abstraction. Sometimes it is necessary to accumulate several consecutive sensor inputs. We use the term *controlled entities* to name the higher-level notions that represent abstractions of sensor value variations over time.

The relation between sensor values and actuator commands is then divided into two relations, one relation between sensor values and controlled entities, and a second relation between controlled entities and actuator commands. The CDM reference architecture shown in Figure 2 reflects these considerations. The two outermost components map technical data (e.g., a lamp being $on$ or $off$) to "logical" data suitable for specifying the software component (e.g., a lamp being $on$, $off$ or $flashing$). The specification of the software component that determines logical actuator commands based on logical sensor values consists of two parts: the *internal domain model* and the *regulator*.

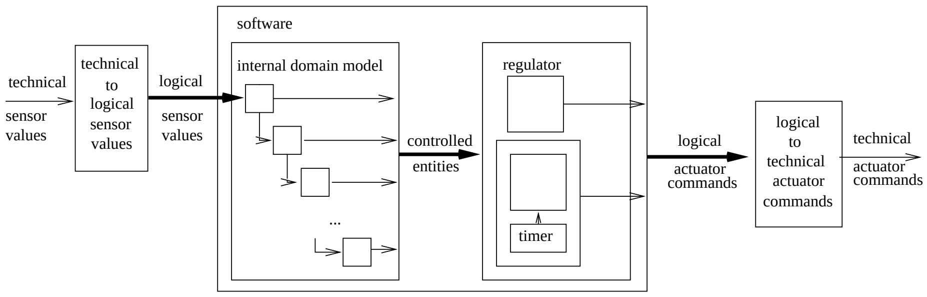


**Fig. 2:** The CDM reference architecture

The internal domain model derives the values of controlled entities. These entities describe the state of the controlled system in terms that are adequate to specify the control task on a level of abstraction as it is used by domain experts. The regulator specifies a relation between controlled entities and actuator commands in a "direct" way (similar to the relation between sensor values and actuator commands in simpler systems).

## 4 Case Study: Traffic Light Safety-Control

Most traffic light systems today are controlled by software. The software controller usually consists of two largely independent modules (see Figure 3): the phase control program and the safety controller. The phase control program receives signals about traffic flow from various sensors in the streets and sends commands to the switchboard to turn on and off the signal heads.

The second module, the safety controller, is responsible to guarantee that the *real* signals as shown by the signal heads allow only safe traffic flow at any time. To this end, it monitors not only the commands produced by the phase control, but it also receives sensor data from the signal heads about their current state relative to the last received switch command.

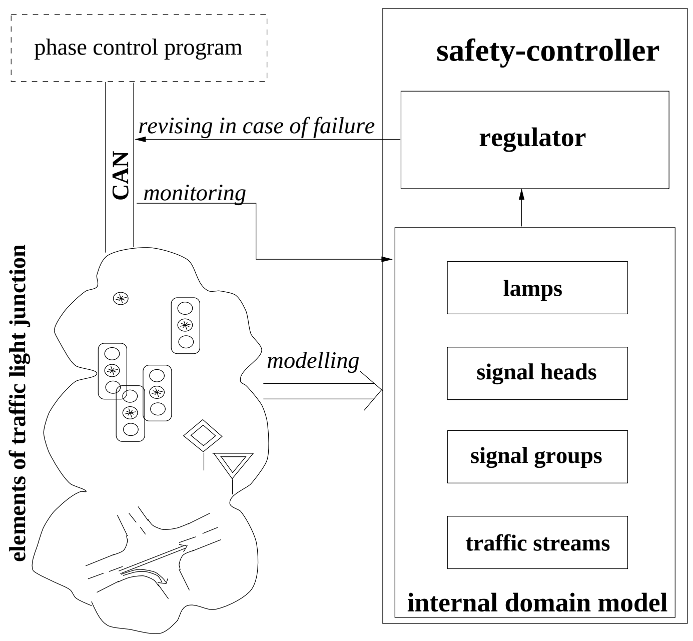


**Fig. 3:** Entities of a traffic light system

Figure 4 illustrates the interaction between phase control, safety control, and the switchboard. The upper half of the timing diagram shows the normal operation of the traffic light control. Every 500 ms, the phase control issues a burst of commands to the switchboard. The switchboard acknowledges these commands with a delay of less than 100 ms, which is the time the sensors in each lamp need to determine its working state. Acknowledgments come in pairs describing the state of the lamp relative to the latest switching command: one bit for each lamp tells whether it is unintentionally on ($uon$), a second bit whether it is unintentionally off ($uoff$). If neither an $uon$ nor an $uoff$ is acknowledged, then this indicates that the signal head shows the intended signal. If both bits are set, then faulty sensor data are received. The switchboard may also issue $uon$ and $uoff$ messages spontaneously if a failure occurs in the otherwise silent time between two command bursts.

The safety controller monitors all commands and acknowledgments. If it detects an unsafe state, it issues appropriate commands to the switchboard to re-establish a safe situation (see bottom of Figure 4). Our task is to specify such a controller with the following general requirements:

1. The controller is generic: it is parameterized with data describing the configuration of a traffic light junction.
2. The controller must detect all unsafe signal conditions. In particular, it may not assume that the phase control commands lead to safe signal conditions.
3. The controller must take appropriate action to establish a safe situation, such that an unsafe situation does not last for more than 300 ms.

The major objective of the specification is to *precisely* capture the meaning of a “safe condition” of a traffic junction *in general*, i.e., not for a particular junction only, but for an (almost) arbitrary configuration of lanes, traffic lights, etc. A detailed analysis of the problem reveals that it is appropriate to base a judgment on the junction’s safety condition on the signals issued for the different *traffic streams*. A traffic stream is the logical entity of all vehicles (or pedestrians) entering and leav-

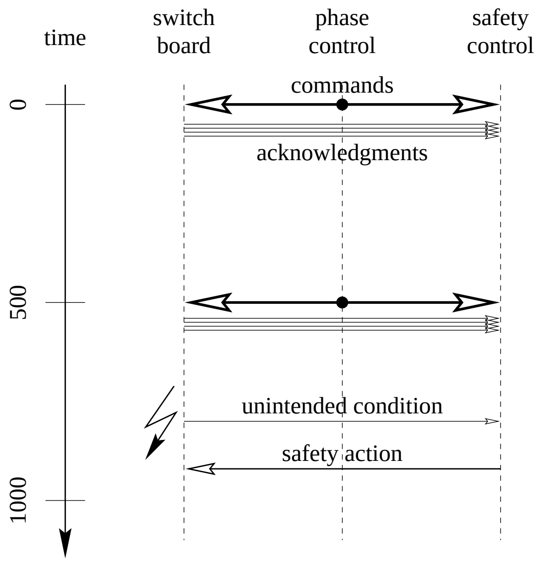


**Fig. 4:** Interaction between the modules

ing the junction at the same points, e.g., the stream of vehicles entering at one point and turning left. Several traffic streams may share the same lane. Based on this abstraction, judging the safety condition of a junction amounts to considering the concurrently open traffic streams and the timing requirements between opening and closing traffic streams.

The safety controller can judge the safety condition only based on its observation of the switching commands of the phase control and acknowledgments of the switchboard. To specify the action of the safety controller depending on the states of traffic streams, we first must describe how the flow of information about states of single lamps can be assembled to an *internal model* of the traffic junction in terms of useful abstractions such as signal heads (consisting of several lights), signal groups (which work synchronously), and traffic streams. This is actually the most complex part of the task.

The next section introduces an agenda to specify software components with such a complex data model and illustrates the application of the agenda to the case study.

## 5 Agenda for Software Components with Complex Data Models

| Stage |
|---|
| **1** Context embedding |
| **2** Controlled entities |
| **3** Software model construction |

**Table 1:** Stages of the CDM agenda

The CDM agenda consists of three stages, shown in Table 1. In the first stage, the embedding of the software in its environment must be defined. This stage consists of defining the technical and the logical software interfaces, and the mappings between them. It is performed as described in [5], and will not be discussed further in this paper (see [8] for a complete description). Stage 2 is characteristic for the CDM reference architecture that needs complex data models. It is described in more detail in Section 5.1. In the third stage, the internal domain model and the regulator must be specified (see Figure 2). Again, only the internal domain model is characteristic for the CDM reference architecture and hence discussed in more detail in Section 5.2. As for the specification of the regulator, we only note that it may contain two different kinds of components. “Passive” components are triggered by a change of some controlled entity, whereas “active” components are activated internally by the regulator when an internal timer times out.

## 5.1 Sub-Agenda for the Definition of Controlled Entities

The internal domain model has to construct the controlled entities based on the logical sensor values. The regulator must produce the logical actuator commands from the controlled entities, see Figure 2. Figure 5 shows the agenda for defining controlled entities. First, the appropriate entities must be identified and given a Z type ($CEtype_1, \ldots CEtype_n$). Step 2.1 also contains a template for an activity chart defining the overall data flow that occurs in the software component.

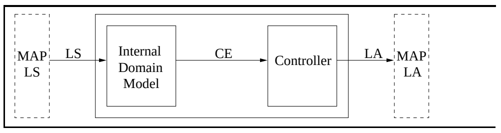
| Step | Validation Conditions |
|---|---|
| **2.1** Identify controlled entities that are needed to express safety requirements and to specify the regulator, and define their types.<br>— $DomainModelDefs$ —<br>$CEtype_1 : \ldots$   $\ldots$   $CEtype_n : \ldots$<br>$\ldots$   $\ldots$<br>— $Software$ —<br> | *no validation conditions* |
| **2.2** Identify groups of controlled entities that change simultaneously. For each group, define a $deliver$ event that will be produced by the internal domain model whenever the values of the entities in the group change. | ⊢ The signatures of $CE_1, \ldots CE_n$ are pairwise disjoint. |
| — $ControlledEntities$ —<br>ENRICH $DomainModelDefs$<br>— PORT $CE$ —<br>$CE_1, \ldots CE_n$<br>— $CE_i$ —<br>$ce^i_1 : CEtype_k$<br>$\ldots$<br>$ce^i_n : CEtype_l$<br>$deliver\_CE_i$ : event<br>— Property DYNAMIC $CE\_deliver_i$ —<br>$CE_1, \ldots CE_n$<br>$whenever\ \{ce^i_1, \ldots, ce^i_n\}\ change$<br>df $deliver\_CE'_i$ | |

**Fig. 5.** Sub-agenda for definition of controlled entities

Step 2.2 defines the dynamics that must occur in connection with controlled entities: when the values of the controlled entities change, the regulator must be notified, because it may be necessary that new actuator commands must be determined. This is achieved by generating *events* that will cause the regulator to take appropriate actions. It is not necessary to define an event for each controlled entity. Instead, events are defined for *groups* of controlled entities that may change their values simultaneously, i.e., if one entity of the group changes its value, then the others *may* also change their values.

```
┌─ ControlledEntities ───────────────────────────────────────────────
│ ENRICH DomainModelDefs                  ┌─ CE_Lamp ─────────────────
│                                         │ transPoint_LA : Lamp ⇸ TIME
│                                         │ errSignal : Lamp ⇸ STATE_ErrSig
│                                         │ deliver_CE_Lamp : event
│                                         └───────────────────────────
│ ┌─ CE_SignalGroup ─────────────────     ┌─ CE_TrafficStream ─────────
│ │ state_SG : SignalGr ⟶ STATE_SG        │ state_TS : TrafficStr ⟶ STATE_TS
│ │ transPoint_SG : SignalGr ⟶ TIME       │ transPoint_TS : TrafficStr ⟶ TIME
│ │ deliver_CE_SignalGroup : event        │ deliver_CE_TrafficStream : event
│ └──────────────────────────────────     └───────────────────────────
└────────────────────────────────────────────────────────────────────
```

**Fig. 6.** Controlled entities for the safety

The schema *ControlledEntities* defines a *process class*. Process classes are the structuring entities of $\mu SZ$. They are containers for sets of plain Z declarations, of schema definitions, and of Statemate statecharts and activity charts. The schema definitions inside a class may have assigned certain *roles*. For example, the role of schema definitions introduced with the keyword *PORT* is to describe data variables that can be shared by a process with its environment.

In the process class *ControlledEntities*, the grouping of controlled entities is defined using the schemas $CE_i$. These groups of entities are collected in the port schema $CE$. The schema with the keyword *Property DYNAMIC* expresses that an event $deliver\_CE_i$ must be generated (df $deliver\_CE_i'$) whenever one of the corresponding controlled entities has changed its value.

*Safety Controller.* Step 2.1 of the CDM agenda requires us to identify entities that allow us to describe the requirements on the safety controller at an adequate level of abstraction. This task encompasses a detailed requirements analysis, and for the safety controller, it needed considerable effort to find a suitable set of entities that allows us to clearly express the control task. We found three important groups of failures that the safety controller may need to react to (c.f. Figure 3):

1. For single lamps, the controller must evaluate *uoff* and *uon* messages to record failures, and detect sensor failures such as omissions of acknowledgments and inconsistent sensor data.
2. For signal groups, the controller must decide whether a sequence of signals is admissible (e.g., in Germany, a transition from “red” to “green” must always go via a combined “red-and-yellow” signal) and whether timing constraints for single signals are satisfied.
3. For traffic streams, the controller must evaluate the safety of simultaneously open streams, and it must monitor intermediate green times, which are required, e.g., between closing a stream and opening a crossing stream that turns left.

In Step 2.1, we define the types describing the information about lamps, signal groups, and traffic streams, that the regulator needs to evaluate the three groups of failures. In Step 2.2, we use these types to specify three groups of controlled entities, as shown in Figure 6. The data of a particular traffic light junction are parameters to our specification. Therefore, the components of the groups *CE_Lamp*, *CE_SignalGroup*, and *CE_TrafficStream* are *functions* mapping identifiers of lamps, etc., to states or transition points. The transition point is the most recent point in time when the state of an element changed. The type *STATE_ErrSig* has four elements: *ok*, *uoff*, *uon*,

| Step | Validation Conditions |
|---|---|
| **3.1.1** Identify abstraction layers of controlled entities, and associate exactly one such layer to each controlled entity.<br>$AbstractionLayer_i$<br>*ENRICH* $ControlledEntities$<br>*PORT* $CELayer_i$<br>$ce_j^k$ : . . .<br>. . .<br>$deliver\_CE_j$ : event | ⊢ If a port $CELayer_i$ contains a controlled entity $ce_j^k$, and it is the port of the maximal abstraction layer index $i$ that contains controlled entities of the group $CE_j$, then it also contains the event $deliver\_CE_j$.<br>⊢ Each controlled entity is contained in exactly one port $CELayer_i$ |
| **3.1.2** Identify the interface between each pair of consecutive layers, and the internal data for each layer. The input port of $layer_1$ are the logical sensor values.<br>$Interface\_0\_1$<br>*ENRICH* $LogicalSensors$<br>$IF\_0\_1 \widehat{=} LS$<br>$Interface\_i\_i+1$<br>*PORT* $IF\_i\_i+1$<br>. . .<br>$update\_model_i$ : event<br>$AbstractionLayer_i$<br>*ENRICH* $Interface\_i-1\_i$ *INPUT* $IF\_i-1\_i$<br>*ENRICH* $Interface\_i\_i+1$<br>*DATA* $State_i$<br>. . .<br>*INIT* . . . | *no validation conditions* |

**Fig. 7.** Steps for Stage 3: definition of abstraction layers

and $fail$, which describe the possible constellations of acknowledge messages ($fail$ indicating that both, a $uoff$ and a $uon$ message have been received). The functions in $CE\_lamp$ are partial, because acknowledgment messages arrive sequentially, in groups of twelve, from the switchboard. The domains of these functions are the identifiers of lamps for which an acknowledgment has been received after the last command burst from the phase control. Transition points need to be evaluated by the regulator to find missing acknowledgments.

The functions in the other two groups are total, because they describe the complete state of the junction, which is accumulated from incoming messages about state changes of lamps. The types $STATE\_SG$ and $STATE\_TS$ are parameters to the specification, because the concrete information to evaluate safety of a junction depends on the legal context: what is tolerable in one community may not be legal in another.

### 5.2 Specification of the Internal Domain Model

Stage 3 of the CDM agenda (cf. Table 1) consists of two steps: specifying the internal domain model, and specifying the regulator. We consider only specifying the internal domain model, for which the sub-agenda is given in Figures 7 and 8.

Having identified the necessary controlled entities, and decided on their types and dynamics in Stage 2 of the agenda, we must now specify how the controlled entities can

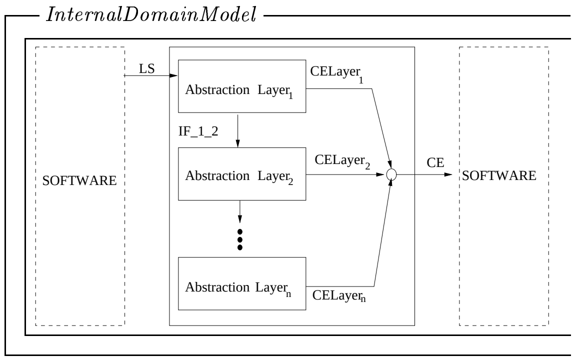
| Step | Validation Conditions |
|---|---|
| **3.1.3** Specify the behavior of each abstraction layer. | ⊢ If $AbstractionLayer_i$ changes data in $IF\_i\_i+1$, then df $update\_model_i$. ⊢ $AbstractionLayer_i$ reacts to $update\_model_{i-1}$ events, i.e. the corresponding statechart performs a state transition. ⊢ If $AbstractionLayer_i$ changes entities in $CELayer_i$, then it also generates the corresponding $deliver$ events. . . . |
| **3.1.4** Assemble abstraction layers into the internal domain model.   InternalDomainModel LS Abstraction Layer1 CELayer1 IF_1_2 Abstraction Layer2 CELayer2 CE SOFTWARE SOFTWARE Abstraction Layern CELayern | *no validation conditions* |

**Fig. 8.** Steps for Stage 3: structure of the internal domain model

be obtained from the logical sensor values. This is the purpose of the internal domain model.

We begin by identifying appropriate *abstraction layers* in Step 3.1.1. Controlled entities that can be derived directly from the logical sensor values belong to the first layer $AbstractionLayer_1$, whereas controlled entities that are defined in terms of other controlled entities belong to a higher abstraction layer.

In Step 3.1.2, we must decide what kind of information must be propagated from one level to the next one. This information is collected in the port schemas $IF\_i\_i+1$. An event $update\_model_i$ notifies the next abstraction layer when relevant information changes. Furthermore, each abstraction layer may have a memory, e.g., for accumulating values. This results in a local state $State_i$.

So far, we have modeled the data aspects of the abstraction layers. It remains to specify their behavior, which is the purpose of Step 3.1.3. We distinguish two kinds of behavior: an abstraction layer may either immediately react to an $update\_model_i$ event, or it may buffer incoming values and only take action when some internal condition is fulfilled. These alternatives are discussed in Section 5.3. Step 3.1.4, finally, is automatic and consists in assembling the abstraction layers in a cascade-like manner, see also Figure 2.

*Safety Controller.* While the order of Steps 3.1.1 through 3.1.4 describes their logical dependencies and is appropriate when developing a specification, it is easier to explain

$InternalDomainModel$

$ENRICH\ LogicalSensors,\ ENRICH\ ControlledEntities$

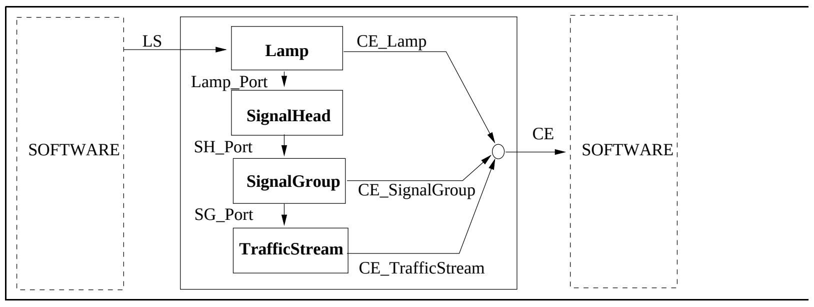


**Fig. 9.** Internal domain model for the safety controller

the resulting product in the reversed order: The internal domain model of the traffic light safety controller, shown in Figure 9, is a cascade of abstraction layers, which resembles the informal "entity-relationship" analysis sketched in Figure 3.

The entities of the lowest abstraction layer are **Lamps**. They are grouped together to **SignalHeads**. Several signal heads form a **SignalGroup**, each of which determines the state of one or more **TrafficStream**s. The state of the elements of one abstraction layer determines the state of the following layer. The relevant data are transmitted along the interfaces, namely $Lamp\_Port$, $SH\_Port$ and $SG\_Port$. Three of the four internal data layers deliver the controlled entities that were introduced in Section 5.1. The union of all controlled entities yields the interface $CE$, which is linked to the regulator.

## 5.3 Alternatives for the Definition of Abstraction Layers

We discuss two approaches to Step 3.1.3. The templates for both fulfill most of the verification conditions of that step by construction.

Table 2 shows a template for an abstraction layer that *immediately* reacts to the event $update\_model_{i-1}$. The statechart describing its behavior has only one state, called $working$. Whenever the event $update\_model_{i-1}$ occurs, the operation $Update\_Layer_i$ is invoked. As indicated by the prefix $\Delta$, this operation may change the internal data of the abstraction layer and the values of its outgoing interfaces. If the interface information

$AbstractionLayer_i$

$OP\ Update\_Layer_i$

$IF\_i-1\_i;\ \Delta State_i$

$\Delta IF\_i\_i+1;\ \Delta CELayer_i$

df $update\_model_{i-1}$

df $update\_model_i' \Leftrightarrow$

$\quad \theta IF\_i\_i+1 \neq \theta IF\_i\_i+1'$

df $deliver\_CE_i' \Leftrightarrow \ldots$

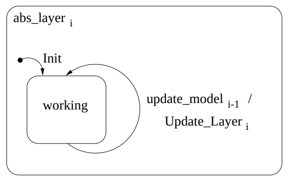


**Table 2:** Steps for Stage 3: template for immediate reaction

changes ($\theta IF\_i\_i+1 \neq \theta IF\_i\_i+1'$), then the next abstraction layer must be notified by generating an $update\_model_i$ event (df $update\_model_i'$). Similarly, $deliver\_CE_i$ events notify the regulator about changed controlled entities.

Table 3 shows the template for *buffered abstraction layers*. They behave differently than immediately reacting layers, although the corresponding state-charts look the same for both. The operation $Update\_Layer_i$ is a composition of the operations $Fill\_Buffer_i$ and $Process\_Buffer_i$. The operation $Fill\_Buffer_i$ works only on the internal state of the abstraction layer and its input interface $IF\_i-1\_i$. The operation $Process\_Buffer_i$ does nothing if a predicate $trigger_i$ defined on the internal state is false, as indicated by the prefix $\Xi$. If the trigger predicate is true, the operation $Generate_i$ computes new values for the controlled entities.

$AbstractionLayer_i$

$trigger_i$

$State_i$

---

$\ldots$

---

$OP\ Fill\_Buffer_i$

$\Delta State_i$

$IF\_i-1\_i$

---

df $update\_model_{i-1}$

$\ldots$

---

$OP\ Generate_i$

$\Delta State_i$

$\Delta IF\_i\_i+1$

$\Delta CELayer_i$

---

$\ldots$

---

$OP\ Process\_Buffer_i$

$\Delta State_i$

$\Delta IF\_i\_i+1$

$\Delta CELayer_i$

---

$trigger_i \Rightarrow Generate_i$

$\neg\ trigger_i \Rightarrow \Xi State_i \wedge \Xi IF\_i\_i+1$

df $update\_model_i' \Leftrightarrow \theta IF\_i\_i+1 \neq \theta IF\_i\_i+1'$

df $deliver\_CE_i' \Leftrightarrow \ldots$

---

$Update\_Layer_i \mathrel{\widehat=} Fill\_Buffer_i \mathbin{;} Process\_Buffer_i$

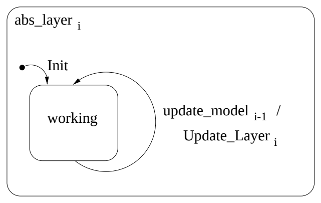


**Table 3:** Steps for Stage 3: template for buffered behavior

Similar to the immediately reacting abstraction layer, the operation $Process\_Buffer_i$ generates an event $update\_model_i$ to indicate that data of the output interface have changed.

For this variant of an abstraction layer, we have the validation conditions that the operation $Fill\_Buffer_i$ eventually leads to a state satisfying $trigger_i$, and that the operation $Generate_i$ falsifies $trigger_i$.

*Safety Controller.* The abstraction layer *SignalGroup* contributes to computing the current state and the transition points of the signal groups in the system. Incoming data are states and transition points of the signal heads. They are available at port $SH\_Port$. Outgoing data are the current states and transition points of the signal groups as defined in $SG\_Port$. The port $CE\_SignalGroup$ contains the controlled entities enabling the regulator to monitor the sequence of phases as well as the duration of each phase (cf.

*SignalGroup*

*ENRICH ControlledEntities*

*PORT CE_SignalGroup*
$state\_SG : SignalGr \longrightarrow STATE\_SG$
$transPoint\_SG : SignalGr \longrightarrow TIME$
$deliver\_CE\_SignalGroup :$ event

*PORT SH_Port*
$state\_SH : SignalHe \longrightarrow STATE\_SH$
$transPoint\_SH : SignalHe \longrightarrow TIME$
$update\_model\_SH :$ event

*PORT SG_Port*
$state\_SG : SignalGr \longrightarrow STATE\_SG$
$transPoint\_SG : SignalGr \longrightarrow TIME$
$update\_model\_SG :$ event

*INPUT* $state\_SH, transPoint\_SH, update\_model\_SH$

*DATA State_SG*
$bufStat\_SH : SignalHe \nrightarrow STATE\_SH$
$bufTime\_SH : SignalHe \nrightarrow TIME$
$first\_update : SignalGr \nrightarrow TIME$

*INIT Init_SG*
...

*OP Fill_Buffer_SG*
$\Delta State\_SG;\ SH\_Port$

df $update\_model\_SH$
$bufStat\_SH' = bufStat\_SH \oplus state\_SH$
$bufTime\_SH' = bufTime\_SH \oplus transPoint\_SH$

*trigger*
$State\_SG$

$\exists\, sg : SignalGr \bullet$
...

*OP Generate_SG*
$\Delta State\_SG;\ \Delta SG\_Port$
$\Delta CE\_SignalGroup$

$\forall\, sg : SignalGr \bullet$
...

*OP Process_Buffer_SG*
$\Delta State\_SG;\ \Delta SG\_Port;\ \Delta CE\_SignalGroup$

$trigger \Rightarrow Generate\_SG$
$\neg\ trigger \Rightarrow \Xi State\_SG \wedge \Xi SG\_Port$
df $update\_model\_SG' \Leftrightarrow \theta SG\_Port \neq \theta SG\_Port'$
df $deliver\_CE\_SignalGroup' \Leftrightarrow \theta CE\_SignalGroup \neq \theta CE\_SignalGroup'$

$Update\_Layer\_SG \mathrel{\widehat=} Fill\_Buffer\_SG \fatsemi Process\_Buffer\_SG$

**Fig. 10.** The abstraction layer of signal groups

Figure 9). In this case the controlled entities are the same as the outgoing data, i.e. the ports *CE_SignalGroup* and *SG_Port* are the same except for the generated events.

The incoming values for signal heads have to be buffered because they are delivered sequentially instead of simultaneously. The specification shown in Figure 10 is an instance of Table 3. Whenever the event *update_model_SH* is read from the incoming port *SH_Port*, the operation *Update_Layer_SG* is performed. This operation is a composition of the operations *Fill_Buffer_SG* and *Process_Buffer_SG*. The shape of the statechart, which is omitted here, is exactly the one of the chart in Table 3.

The operation *Fill_Buffer_SG* just adds the data read from the incoming port *SH_Port* to the buffer functions called *bufStat_SH* and *bufTime_SH* by functional overwriting. The operation *Process_Buffer_SG* assembles the buffered values when *trigger* is satisfied. This is the case if there exists a signal group whose update information is complete, or whose transition point was set more than a certain time span ago. If *trigger* is not satisfied, then the data remain as they are. In case of change, two events

*TrafficStream*

$\mathit{ENRICH}\ \mathit{ControlledEntities}$

*PORT SG_Port*

$\mathit{state\_SG} : \mathit{SignalGr} \longrightarrow \mathit{STATE\_SG}$
$\mathit{transPoint\_SG} : \mathit{SignalGr} \longrightarrow \mathit{TIME}$
$\mathit{update\_model\_SG} : \text{event}$

*PORT CE_TrafficStream*

$\mathit{state\_TS} : \mathit{TrafficStr} \longrightarrow \mathit{STATE\_TS}$
$\mathit{transPoint\_TS} : \mathit{TrafficStr} \longrightarrow \mathit{TIME}$
$\mathit{deliver\_CE\_TrafficStream} : \text{event}$

$\mathit{INPUT}\ \mathit{state\_SG}, \mathit{transPoint\_SG}, \mathit{update\_model\_SG}$

*OP Update_Layer_TF*

$\mathit{SG\_Port};\ \Delta \mathit{CE\_TrafficStream}$

$\mathsf{df}\ \mathit{update\_model\_SG}$
$\mathsf{df}\ \mathit{deliver\_CE\_TrafficStream}' \Leftrightarrow \theta \mathit{CE\_TrafficStream} \neq \theta \mathit{CE\_TrafficStream}'$
$\mathit{state\_TF}' = \mathit{interpret\_signal\_TF} \circ \mathit{state\_SG} \circ \mathit{signals\_TF}$
$\mathit{transPoint\_TF}' = \mathit{transPoint\_SG} \circ \mathit{signals\_TF}$

**Fig. 11.** The abstraction layer of traffic streams

$\mathit{update\_model\_SG}$ and $\mathit{deliver\_CE\_SignalGroup}$ report about the new values along the outgoing ports.

An example for unbuffered immediate reaction is given with the highest abstraction layer of the cascade, namely the process class $\mathit{TrafficStream}$ in Figure 11. The update operation determines the current values of the traffic streams according to the actual data of the signal groups which control them. Incoming data are defined in the port $\mathit{SG\_Port}$. The states and the transition points of the traffic streams are the controlled entities which are delivered to the regulator via the port $\mathit{CE\_TrafficStream}$. The behavior is cyclic as before but there is no need to store values as internal data, because the values of all signal groups are determined simultaneously.

The updating operation is activated each time the event $\mathit{update\_model\_SG}$ is read from the input port. It computes the data for the actual state and transition point for each traffic stream in accordance with the signal groups that give the signals to them. To determine the actual state value means to interpret the signal from the signal group in terms of opening or closing. Both functions $\mathit{interpret\_signal\_TF}$ and $\mathit{signals\_TF}$ are external functions described by the planning documents. The event $\mathit{deliver\_CE\_TF}$ is generated if there is any change of data.

## 6 Related Work

The use of formal methods to specify software for safety-critical embedded systems is not uncommon, see e.g. [10–12]. However, few approaches provide an explicit methodology to develop formal specifications. Related to this aim is the work of Souquières and Lévy [14]. 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.

More agendas that support the specification of software for safety-critical embedded systems can be found in [5, 9]. There, different reference architectures and formalisms are supported. More details of the traffic light system can be found in [8, 13].

## 7 Conclusions

The requirements for safety-critical embedded systems can be non-trivial. In these cases, we cannot assume that highly abstract requirements are easily captured as a direct relation between sensor values and actuator commands.

Without formal specification techniques, the relation between high-level requirements and low-level sensor data often is established only in the design and implementation phases and in an ad hoc manner. This results in a gap between high-level requirements documents and low-level design and implementation documents. Consequently, errors in mapping high-level requirements to low-level data are either detected very late in the development process, or not at all. It is very hard to certify the safety of systems developed in that way.

Our approach avoids these problems by proposing to

1. Set up a formal requirements specification before beginning with the design and implementation of the software component.
   In this way, functional and safety requirements for the software component are stated explicitly and unambiguously.
2. Define the abstract notions, which are used to express requirements, in terms of low-level sensor data formally and early in the software development process.
   In this way, we establish a direct connection between the requirements analysis and the software modeling phases. As a result, the requirements are adequately reflected in the software model.

For the traffic light safety controller, finding appropriate abstract notions to characterize safe states of a traffic junction (such as signal groups and traffic streams) was a crucial point in developing an adequate specification, because these abstractions are not documented in the domain specific literature [3, 4].

In addition, we have identified a *systematic way of procedure* to achieve the abstraction of low-level values to high-level concepts. This systematic way of procedure and the formal nature of our specification language force software developers to analyze the system much more thoroughly than this is the case for traditional software engineering approaches. For example, classifying controlled entities into groups that may change simultaneously (see Step 2.2 of Figure 5) forces the specifier to carefully reconsider all the controlled entities introduced in Step 2.1.

Following the CDM agenda leads to a *clean architecture* of the software component. The clear cascade-like organization of abstraction layers leads to well structured and comprehensible specifications even for complex applications.

The validation conditions associated with the steps of the agendas ensure that the specification fulfills certain *quality criteria*. All of the validation conditions presented in this paper can be expressed formally and be demonstrated with machine support. Without formal techniques, such a rigorous validation of the specification would not be possible.

Apart from making *design knowledge* explicit and re-usable, an agenda provides a *documentation* of the specifications developed with it. Each part of the specification can be mapped to a step of the agenda that explains its purpose. In this way, the *evolution* of specifications is facilitated considerably.

The traffic light case study has provided a proof of concept for the approach presented in this paper. It is not an academic example but a real-life industrial application. Safety controllers for traffic light systems are highly non-trivial. The complete formal specification [13] is 50 pages long, and an informal analysis document takes another 21 pages. A first version of the formal specification had been developed without using the agendas presented in this paper. Revising this first version to make it conform to the agenda resulted in eliminating some ad hoc solutions and has lead to a better structured and more comprehensible specification that can be adjusted to new requirements in a systematic way.

## References

1. R. Büssow, H. Dörr, R. Geisler, W. Grieskamp, and M. Klar. $\mu$SZ – ein Ansatz zur systematischen Verbindung von Z und Statecharts. Technical Report TR 96-32, Technische Universität Berlin, 1996.
2. R. Büssow and W. Grieskamp. Combinig Z and temporal interval logics for the formalization of properties and behaviors of embedded systems. In R. K. Shyamasundar and K. Ueda, editors, *Asian '97*, LNCS 1345, pages 46–56. Springer-Verlag, 1997.
3. Deutsche Elektrotechnische Kommission im DIN und VDE (DKE). DIN Norm VDE 0832 – Straßenverkehrs-Signalanlagen (SVA), 1990.
4. Forschungsgesellschaft für Straßen- und Verkehrswesen. Richtlinien für Lichtsignalanlagen – RiLSA, 1992.
5. W. Grieskamp, M. Heisel, and H. Dörr. Specifying safety-critical embedded systems with Statecharts and Z: An agenda for cyclic software components. In E. Astesiano, editor, *Proc. ETAPS-FASE'98*, LNCS 1382, pages 88–106. Springer-Verlag, 1998.
6. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.
7. M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32, London, 1998. Chapman & Hall.
8. M. Heisel, T. Santen, and K. Winter. An agenda for software components with complex data models. Technical report, GMD FIRST, 1998. to appear.
9. M. Heisel and C. Sühl. Methodological support for formally specifying safety-critical software. In P. Daniel, editor, *Proc. 16th SAFECOMP*, pages 295–308. Springer-Verlag London, 1997.
10. J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, 1995.
11. J. McDermid and R. Pierce. Accessible formal method support for PLC software development. In G. Rabe, editor, *Proc. 14th SAFECOMP, Belgirate, Italy*, pages 113–127, London, 1995. Springer-Verlag.
12. A. Ravn, H. Rischel, and K. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, 1993.
13. T. Santen and K. Winter. Sicherung einer Lichtsignalanlage in $\mu$SZ. Technical report, GMD FIRST, 1998. to appear.
14. J. Souquières and N. Lévy. Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223, 1993.
15. J. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 1992.