Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study

Robert Büssow[1], Robert Geisler[1], and Marcus Klar[2]

[1] Technische Universität Berlin, Institut für Kommunikations- und Softwaretechnik Sekr. 6–1, Franklinstr. 28/29, D–10587 Berlin. {buessow, geislerr}@cs.tu-berlin.de

[2] Fraunhofer-Institut für Software- und Systemtechnik ISST Kurstr. 33, D–10117 Berlin. Marcus.Klar@isst.fhg.de

Abstract. In this paper we introduce a formal approach for the specification of safety-critical embedded systems. The specification formalisms Z and statecharts are integrated under a suitable structural model. The combined approach uses the advantages of the formalisms while avoiding their disadvantages. The different formalisms yield different, compatible views on the system: the functional view describing data and data-transformation, the reactive view, describing the system's reaction upon external stimuli, and the structural view, describing the components of the system and their interaction. The combination is discussed presenting parts of a case study: a traffic light control system. The case study is oriented at original planning documents. Besides its safety- and real-time-aspects, the case study is particularly interesting because structuring and reuse is of considerable importance in this example.

1 Introduction

Embedded systems are permanently increasing in size, complexity and responsibility. Failures of the control software can have disastrous consequences and, due to this, *safety* [18] of the software is becoming more and more important. In addition, embedded systems raise problems of concurrency, have to obey real-time requirements, and usually reside in a heterogeneous environment. This stresses the need of an adequate software specification technique and a suitable development method for large-scale (safety-critical) embedded systems. The used *specification formalisms* need to be comprehensive, expressive, and precise. We are using statecharts, an extension of finite automata, to describe reactive behaviors; because of its clear depiction of a system's reaction and states. For the data and data-transformations we are using the Z specification language because of its mathematical-like notation and expressiveness.

Instead of describing the whole system in one specification formalism, we use different formalisms to specify different aspects of the system, exploiting the advantages and avoiding the disadvantages of the particular formalisms; e.g. dynamic aspects like control flow can not be expressed very descriptive in Z, while the statecharts formalism provides only limited support for the description of data.

The *structural view* describes the components of the system and their relations with each other and their environment. Typically some variant of data flow diagrams [8] is used for this purpose. In the *dynamic view* the reaction of the system and its components to internal and external stimuli is specified. Such behavior can be described intuitively using state automata. The *functional view* describes data, data invariants and data transformations of the system and its components. The data transformations are controlled by the specification of the dynamic view. For the specification of the functional view, data type specification languages are an adequate tool.

The main emphasis of this work is the presentation of the case study—the specification of a traffic light system with statecharts and Z. The case study, the combination (called μSZ) as well as verification and validation techniques are investigated in detail in the ESPRESS project[1]. Methodological aspects of the ESPRESS project are presented in [11].

2 Specification Technique

We represent an embedded system as a collection of synchronous, communicating processes. Each process has a data space, an interface for the communication with other processes or the environment, and a statechart determining how it reacts upon external stimuli.

The description of a process is given by the specification of so called process classes. A process class describes a set of processes with common behavior. This description (a process-class) includes the structure of the process, i.e. its subprocesses, their communication relations, and the processes the specified process is linked to via associations. The description consists in its interface, its local variables, its dynamic behavior, its configuration, predicates and operations over its variables, as well as behavioral constraints. This combination is discussed in detail in [2], the different views are depicted in Figure 1.

The structure of a process is depicted in a so called *configuration* diagram. Processes communicate via shared variables and valued events. A process can have several interfaces called *ports* to read and write variables shared with other processes. In the configuration diagram, communication relations between processes are established by linking ports of different processes together. The structure and the interfaces of a process together are forming the *architectural* view of a process.

A process is storing, transforming, and exchanging data. This data is specified in the *functional* description of the process, using the Z language

[1] ESPRESS is a joint project of German industry and research institutes, funded by the German Bundesministerium für Bildung, Wissenschaft und Technologie (BMBF).

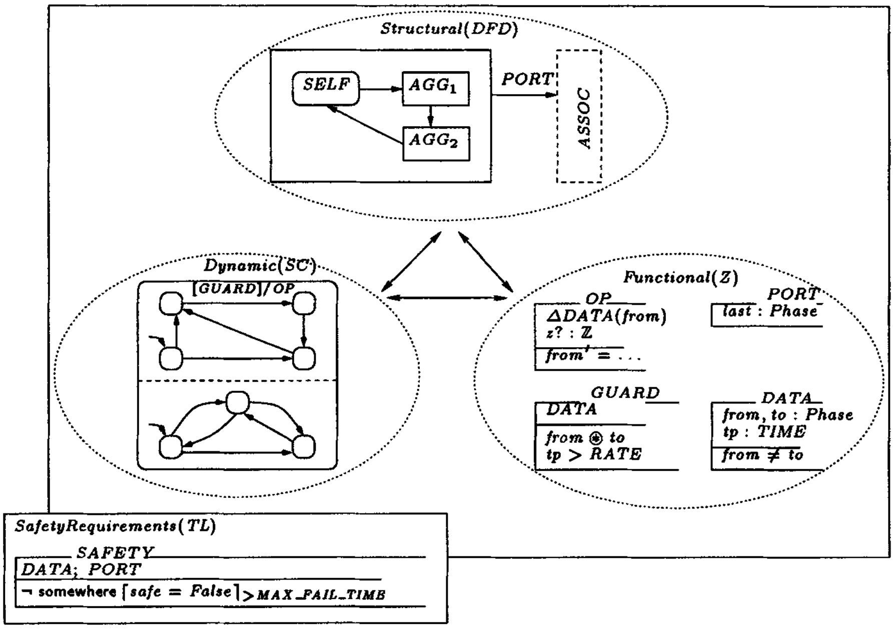


Fig. 1. The different Views on a Process

[21]. We distinguish local and shared data. The local data is described in so called *DATA*-schemas, whereas shared data is stored in *PORT*-schemas. Data transformation can be carried out via *operations*. In an *Op*-schema, the input-output relation of an operation is specified in Z.

The behavior of a process, i.e. the actions it performs during its lifetime, are subject to the statecharts [12] in the *dynamic view*. We are using the STATEMATE [13] tool to support the graphical languages used. Thus, for statecharts, the STATEMATE syntax and semantics [15,14] are adopted. The configurations are expressed with a subset of STATEMATE's *activity-charts*. Nevertheless, we specify the guards and actions of the statechart transitions in Z. The guards are predicates over a process's data space and the actions are operations over the data space.

Especially in the early development phases, it might not be possible or adequate to describe the process behavior in an operational manner, i.e. by using statecharts. Therefore, we express behavioral requirements (particularly safety requirements) in temporal logic. Here, we are using the logic introduced in [5]. These temporal requirements can later be "implemented" by a statechart. Thus, a process can as well include an abstract specification of behavioral constraints and requirements as a concrete implementation. It becomes a natural proof obligation in the development process to show that the implementation fulfills the constraints and requirements.

The semantics of a combined specification is a set of processes that is ordered in a tree-like structure according to the specification's configuration diagrams. The processes are connected via ports, allowing them to access and exchange the values of their external variables. The *static semantics* of a process is determined (according to the Z semantics) by the bindings of the process variables, by the statechart status (i.e. the set of statecharts states the process currently resides in), and recursively by the static semantics of the process's aggregated processes. The *dynamic semantics* of a process is given by the STATEMATE semantics described in [14]. The parallel statecharts of the processes are executed synchronously. Changes that occur during one step are visible only in the following step. We are applying the *asynchronous* time model of STATEMATE. We want to point out that the semantics of the basis formalisms Z and statecharts are preserved in the combined specification. This enables us to reuse existing tools for Z and Statecharts. Note that we do not aim at translating one formalism into the other, but rather use them in a supplemental way.

3 The Case Study: A Traffic Light System

In this case study we describe a fault tolerant traffic light system (TLS). The task of the TLS consists of (1) steering the signal heads[2] according to a given program (functionality), and (2) guaranteeing a safe signal situation on the junction, even in case of signal head failure (safety). To reduce complexity, this specification separates the functional and the safety aspects into different components. The means of structuring of the used specification formalism support this approach and allow a modular specification. A TLS has to comply to several norms and laws; for this case study the *German Road Traffic Law* and the German norms [9,10] are relevant. The requirements for a particular junction are given in the *planning documents*, made by an engineer's office. This specification is based on authentic planning documents of an existing junction.

The planning documents include the essential local informations for the traffic junction such as the road infrastructure, road markings, sidewalks, bike-ways, signing, signal heads, and detectors (see Figure 2). The street from north-east to south-west is the main road. The signal heads are numbered, the detectors (induction loops and pedestrian push buttons) are carrying numbers with a leading "D". The detectors are measuring the requirements of the road users. These requirements are influencing the control of the traffic light system, allowing to react in a flexible way on changing traffic volumes.

We model the signal heads of the TLS in Z as an enumeration type. The signal heads have different lamps, modeled as *Lamp*.

[2] A set of signal heads is a green, yellow, and red lamp or, for pedestrians, green and red lamps.

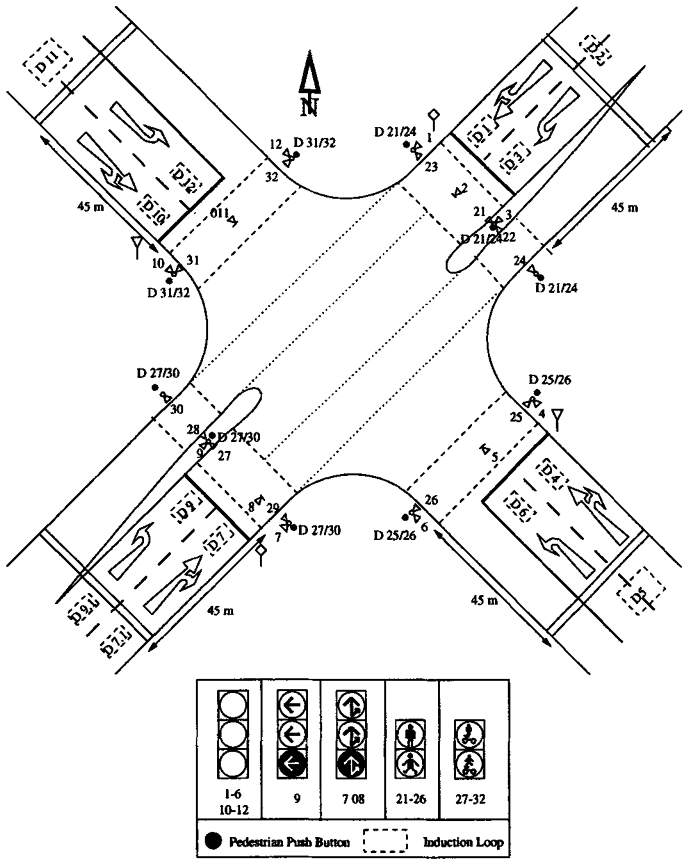


Fig. 2. The Signal Layout Plan

SignalHeads ::= $Sh_1 \mid Sh_2 \mid Sh_3 \mid \ldots \mid Sh_{31} \mid Sh_{32}$
Lamp ::= *red* | *yellow* | *green*

The signal heads that control the same traffic flows, are grouped together to *signal groups*. A signal group consists of a set of signal heads that, in absence of failure, indicate the same signal. In our traffic junction we have eleven signal groups (v denotes vehicle, p pedestrian, m the main road, and s the side road).

$$G^{v/m}_{1/2/3}, G^{v/s}_{4/5/6}, G^{v/m}_{7/8}, G^{v/m}_{9}, G^{v/s}_{10/11/12} : \mathbb{P}\, SignalHeads$$
$$G^{p/m}_{21/22}, G^{p/m}_{23/24}, G^{p/s}_{25/26}, G^{p/m}_{27/28}, G^{p/m}_{29/30}, G^{p/s}_{31/32} : \mathbb{P}\, SignalHeads$$
$$G^{v/m}_{1/2/3} = \{Sh_1, Sh_2, Sh_3\} \wedge \ldots \wedge G^{p/s}_{31/32} = \{Sh_{31}, Sh_{32}\}$$

$$SignalGroup == \{G^{v/m}_{1/2/3}, G^{v/s}_{4/5/6}, G^{v/m}_{7/8}, G^{v/m}_{9}, G^{v/s}_{10/11/12}, G^{p/m}_{21/22}, G^{p/m}_{23/24}, G^{p/s}_{25/26}, G^{p/m}_{27/28}, G^{p/m}_{29/30}, G^{p/s}_{31/32}\}$$

Signal groups that must not be opened (i.e. show green) at the same time, are said to be in *conflict*, e.g. $G^{v/m}_{1/2/3}$ and $G^{v/s}_{4/5/6}$ are in conflict. This is expressed by the relation *conflict*. The conflict relation is irreflexive and symmetric. Although not in conflict, bending traffic flows may still interfere with non-bending. The road traffic law determines the *priority* relation between two traffic flows with a common conflict area that might pass the junction at the same time. For example, pedestrians have priority over turning traffic, e.g. we have $G^{p/s}_{31/32}$ has priority over $G^{v/m}_{1/2/3}$. The priority relation is important for the safety conditions as shown in section 3.3. It has no relevance for the control program.

$$\begin{array}{|l}
conflict, priority : SignalGroup \leftrightarrow SignalGroup \\
\hline
\forall s_1, s_2 : SignalGroup \bullet \\
\quad (s_1, s_1) \notin conflict \land (s_1, s_2) \in conflict \Leftrightarrow (s_2, s_1) \in conflict \\
(G^{v/m}_{1/2/3}, G^{v/s}_{4/5/6}) \in conflict \land \ldots \\
(G^{p/s}_{31/32}, G^{v/m}_{1/2/3}) \in priority \land \ldots
\end{array}$$

Besides not being opened at the same time, conflicting traffic flows have to obey to their *intergreen time*. The intergreen time between two conflicting traffic streams specifies how long the vacating traffic stream has to be blocked until the starting stream can be opened. The *intergreen time table* (Fig. 3), shows the intergreen times as given in the planning documents for all pairs of conflicting signal groups. It shows the necessary intergreen times between starting (abscissa) and vacating (ordinate) traffic streams.

The intergreen time table is modeled in Z as a function that maps a pair of conflicting signal groups to its intergreen time. $IGT(grp_1, grp_2)$ denotes the time grp_1 has to be closed before grp_2 may be opened.

$$\begin{array}{|l} IGT : conflict \rightarrow TIME \end{array}$$

3.1 Structure of the Traffic Light Control System

We describe our system as a hierarchical set of interacting processes. The top level process is called *TRAFFIC_LIGHT_CONTROL*. Its configuration shows how it is divided into four aggregated processes as well as the interfaces between the processes and their environment. Here, the environment consists of the external traffic facilities, such as the signal heads and detectors of the system. The processes are connected via arrows that are labeled with port names. The processes communicate via the variables declared in their ports. These ports are specified in the process classes of the processes. In the

	1/2/3	4/5/6	7/8	9	10/11/12	21/22	23/24	25/26	27/28	29/30	31/32
1/2/3	×	5	-	4	7	4	4	7	8	8	6
4/5/6	7	×	4	5	-	5	5	4	8	8	7
7/8	-	6	×	-	5	8	8	7	5	5	-
9	7	5	-	×	6	-	-	-	4	4	9
10/11/12	6	-	5	4	×	8	8	8	6	6	4
21/22	6	3	1	-	2	×	-	-	-	-	-
23/24	6	3	1	-	2	-	×	-	-	-	-
25/26	8	10	9	-	8	-	-	×	-	-	-
27/28	1	1	6	6	2	-	-	-	×	-	-
29/30	1	1	6	6	2	-	-	-	-	×	-
31/32	9	8	-	8	10	-	-	-	-	-	×

Fig. 3. The Intergreen Time Table

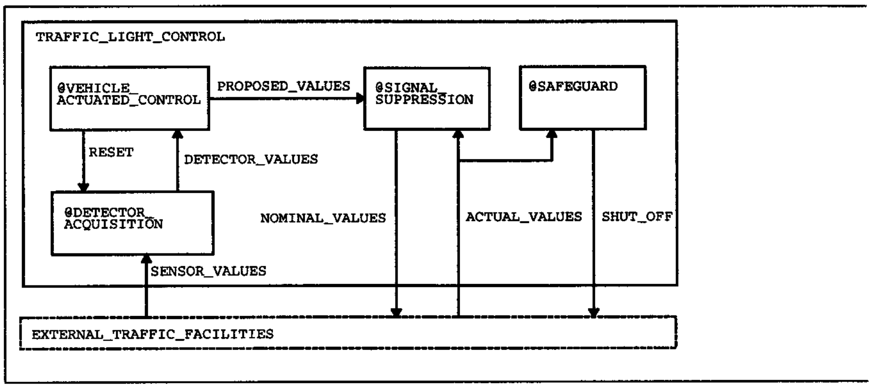
following we give a short description of the four aggregates of the top-level process:

The *VEHICLE_ACTUATED_CONTROL* realizes the traffic control algorithm, i.e., it computes when the lamps of the signal heads are to be switched on or off and sends the relevant signals. The *SAFEGUARD* guarantees the safety of the traffic junction. If an unsafe signal indication occurs due to hardware or software failure, the safeguard has to take counter action. If, for example, a red light breaks, the safeguard and the suppression have to decide whether the resulting signal indication is still safe. If it is not, it has to take measures to reestablish a safe indication. If an unsafe signal indication is detected, the suppression tries to reestablish safety in, e.g. switching defective signal heads whereas the safe guard shuts off the entire TLS if an unsafe situation is imminent to last longer than 0.3 seconds. The safe guard is discussed in detail in section 3.3.

Signals that are sent by the control might interfere with measures taken by the *SAFEGUARD* while handling failures. Such signals are filtered out by the *SIGNAL_SUPPRESSION*. Moreover, the *SIGNAL_SUPPRESSION* suppresses all signals that would lead to an unsafe situation of the junction. By that, safety can be ensured independently of the *VEHICLE_ACTUATED_CONTROL*. The signals of the system's detectors (induction loops and pedestrian push buttons) are recorded and prepared for the control by the *DETECTOR_ACQUISITION*, because these detector values serve as parameters for the vehicle actuated control.

There are four interfaces between the traffic control system and the external traffic facilities: The signals, indicated to the road users, are transmitted via the port *ACTUAL_VALUES*. They are assumed to be measurable in a fail-safe way and can be always requested. The nominal values for signal heads are sent via the port *NOMINAL_VALUES*. The port *SENSOR_VALUES* serves to read the signals occurring at the detectors. The entire traffic light system can be shut off via the port *SHUT_OFF*. This transfers the system in a fail-safe state, if an error state can not be recovered by the signal suppression.

TRAFFIC_LIGHT_CONTROL



3.2 The Control Program

In the process class *VEHICLE_ACTUATED_CONTROL*, the control algorithm is specified. It switches between different phases. A phase can be characterized by the set of signal groups that are opened together during this phase. In many traffic light systems the order and duration of the phases is predefined (*fixed-time control*). Nevertheless, the control program, presented here, is driven by the actual traffic, measured by the induction loops and the pedestrian push buttons. It allows phase transitions between all phases. Note that the program has two rather unusual phases: in *phase*0 all signal groups are closed and in *phase*1 only the pedestrians have green. Obviously, groups that are in the same phase must not be in conflict.

$$phase0, phase1, phase2, phase3, phase4 : \mathbb{P}\, SignalGroup$$

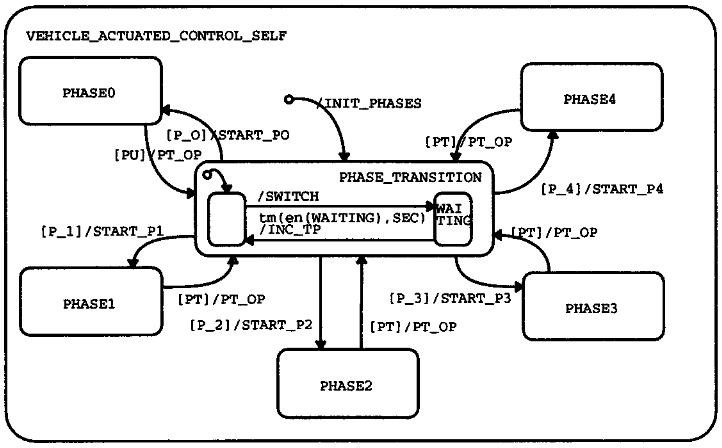
$$\begin{aligned}
&phase0 = \varnothing\\
&phase1 = \{\, G^{p/m}_{21/22}, G^{p/m}_{23/24}, G^{p/s}_{25/26}, G^{p/m}_{27/28}, G^{p/m}_{29/30}, G^{p/s}_{31/32} \}\\
&phase2 = \{\, G^{v/m}_{7/8}, G^{v/m}_{9} \}\\
&phase3 = \{\, G^{v/m}_{1/2/3}, G^{v/m}_{7/8} \}\\
&phase4 = \{\, G^{v/s}_{4/5/6}, G^{v/s}_{10/11/12} \}
\end{aligned}$$

$$Phase == \{\, phase0, phase1, phase2, phase3, phase4 \}$$

| $\forall p : Phase \bullet \forall grp_1, grp_2 : p \bullet (grp_1, grp_2) \notin conflict$

In the statechart[3] of the class *VEHICLE_ACTUATED_CONTROL*, the control is either in one of the five phases or a phase transition is performed.

— *VEHICLE_ACTUATED_CONTROL* ———



During a phase transition, the signal groups are switched according to the phase transition tables in the planning documents (Figure 4). The phase transition table denotes for each signal group the signaling over the time (measured in seconds). The traffic flows that are blocked in the new phase are closed and the flows that are to be opened are given green. The phase transition tables assure that the intergreen times are not violated during a phase transition.

In Figure 4, the phase transition table from phase 1 to phase 2 is shown. Initially, all pedestrian signal heads indicate green. One second after the beginning of the phase transition, signal group 27/28 is switched to red, three seconds later signal group 25/26 follows and so on.

The internal state variables of a process class are declared in a *DATA* schema. The variables *from* and *to* are used to record the actually desired phase transition. The variable *tp* of type *TIME* keeps track of the progress of the actual phase transition, i.e. the column of the phase transition table. The relation *Transition* describes the possible phase transitions, where $(phase0, phase1)$ stands for the transition from phase 0 to phase 1.

[3] In the statechart, the transition labels are of the form $[cond]/action$, where *cond* is a condition specified in a Z schema or in STATEMATE syntax and *action* is a Z operation. The transition fires if the condition is true; the action is executed then.

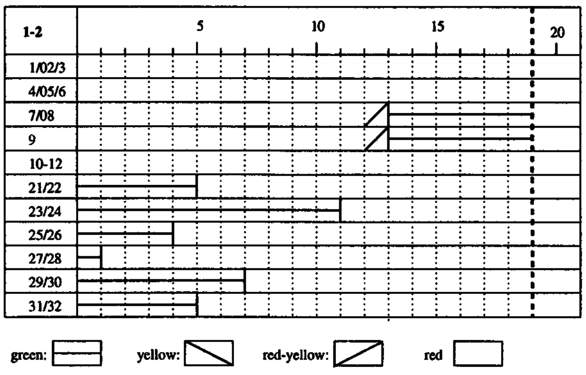


Fig. 4. The Phase Transition from Phase 1 to Phase 2

The function *TransitionTable* models the phase transition tables. It assigns to every phase transition and every signal group the moment of signal change and the new signal indication. The duration of each phase transition is expressed by the function *TransitionDuration*. It is desired that all changes in the signal indication occur within the specified duration of the phase transition.

VEHICLE_ACTUATED_CONTROL

$DATA\ PhaseTransition \mathrel{\widehat{=}} [from, to : Phase;\ tp : TIME]$

$Transition : Phase \leftrightarrow Phase$

$\forall p : Phase \bullet (p, p) \notin Transition$

$TransitionTable : (Transition \times SignalGroup) \rightarrow (TIME \nrightarrow \mathbb{P}\, Lamp)$

$TransitionDuration : Transition \rightarrow TIME$

$\forall u : Transition;\ s : SignalGroup \bullet$

$\quad \forall t : \mathrm{dom}(TransitionTable(u, s)) \bullet t < TransitionDuration(u)$

$TransitionDuration((phase1, phase2)) = 19$

$\forall Gr : \{G^{v/m}_{1/2/3}, G^{v/s}_{4/5/6}, G^{v/s}_{10/11/12}\} \bullet$

$\quad TransitionTable((phase1, phase2), Gr) = \varnothing$

$TransitionTable((phase1, phase2), G^{v/m}_{7/8}) =$

$\quad \{12 \mapsto \{red, yellow\}, 13 \mapsto \{green\}\}$

$\ldots$

$TransitionTable((phase0, phase1), G^{p/s}_{31/32}) = \{5 \mapsto \{red\}\}$

3.3 The Safeguard

The safeguard guarantees the safety of the system. It works independently from the other parts and shuts the TLS off if an unsafe situation has been encountered. According to the norm DIN VDE 0832 [9], the TLS must not stay longer than 0.3 seconds in an unsafe situation. For the safeguard, firstly the requirements are specified using abstract temporal formulae. Then, these formulae are translated into more concrete, non-temporal predicates and a statechart. Here, we present three of the TLS's safety requirements. The signal heads sensor their own state, i.e. which bulbs are shining or not and make these values available to the safeguard. The safeguard reads these values through its port *ACTUAL_VALUES*. For the safety consideration, we assume that the actual values are supplied immediately (with no time delay). For the concrete implementation, this requirement can be weakened. If the TLS is to be shut off, the safeguard sets the variable *OFF* in the port *SHUT_OFF* to *True*.

We model the *actual* signal stage of the junction as a function, assigning to each set of signal heads the set of on-lamps. Based on the signaling situation, the signal groups are partitioned into *open* (i.e. showing green, yellow, etc.), *closed* (i.e. showing red), and *free* (i.e. are off). Note that normally signal heads of the same group should show the same signals, which might not be true in case of an defective signal heads. It might also happen that the green and red lights of one set of signal heads are on simultaneously. In these situations, the partition of the signal groups is quite intricate. Moreover the actual assessment sometimes changes with local regulations. We therefore omit a precise definition here.

SAFEGUARD

PORT ACTUAL_VALUES

$actual : SignalHeads \rightarrowtail \mathbb{P}\, Lamp$

PORT SHUT_OFF

$OFF : \mathbb{B}$

GroupStates

ACTUAL_VALUES

$opened, closed, free : \mathbb{P}\, SignalGroup$

disjoint $\langle\, opened, closed, free \,\rangle$

...

DATA SAFE

$safe : \mathbb{B}$

$MAX\_FAIL\_TIME : TIME$

$SAMPLING\_RATE : TIME$

Basing on the actual signal stage, the TLS has to judge whether the junction is in a safe situation or not. The safe guard introduces a predicate $safe : \mathbb{B}$, denoting whether the junction is in a safe situation in this particular state or not. In fact, the safety conditions presented here discriminate only states that are definitely not safe. We firstly define for each safety requirement the auxiliary schemas *Conflict_Abs*, *Priority_Abs*, and *ObeysIGT_Abs*. They are then used to formulate the safety requirement for the class *SAFEGUARD*.

We are using discrete interval logic to describe the safety requirements. A complete description of its syntax and semantics is beyond the scope of this paper. Here we only give a short introduction. The logic can be seen as a discrete variant of the Duration Calculus [7] adopted for Z. It is presented in [5]. $\lceil x = 0 \rceil$ denotes an interval where x equals zero all the time. By $\lceil x > 0 \rceil \frown \lceil x < 0 \rceil$ an interval is denoted, where x is greater zero in the beginning and is less than zero immediately afterwards. somewhere $\lceil x = 0 \rceil$ denotes an interval where for some sub-interval $x = 0$. The formula $\lceil x = 0 \rceil_{<t}$ describes an interval shorter than t, where x equals zero.

SAFEGUARD

DYNAMIC Conflict_Abs

GroupStates; *SAFE*

$\forall\, grp_1, grp_2 : Signalgroup \mid (grp_1, grp_2) \in conflict \bullet$
$\quad \neg\, \mathsf{somewhere}\, \lceil grp_1 \in opened \wedge grp_2 \in opened \wedge safe = True \rceil$

DYNAMIC Priority_Abs

GroupStates; *SAFE*

$\forall\, grp_1, grp_2 : SignalGroup \mid (grp_1, grp_2) \in priority \bullet$
$\quad \neg\, \mathsf{somewhere}\, (\lceil grp_1 \notin opened \wedge grp_2 \in opened \rceil \frown$
$\qquad \lceil grp_1 \in opened \wedge grp_2 \in opened \wedge safe = True \rceil)$

DYNAMIC ObeysIGT_Abs

ACTUAL_VALUES; *GroupStates*; *SAFE*

$\forall\, grp_1, grp_2 : SignalGroup \mid (grp_1, grp_2) \in conflict \bullet$
$\quad \neg\, \mathsf{somewhere}\, (\lceil grp_1 \in opened \rceil \frown$
$\qquad \lceil true \rceil_{<IGT(grp_1, grp_2)} \frown \lceil grp_2 \in opened \wedge safe = True \rceil)$

The first safety requirement *Conflict_Abs* states that the junction is unsafe if two conflicting signal groups are opened at the same time, i.e., there is no sub-interval in which a conflicting pair of signal groups ($(grp_1, grp_2) \in conflict$) is opened and $safe = True$.

The second safety requirement *Priority_Abs* states that a prioritized signal group must not be opened while a signal group of lower priority is open. A group has priority over another one, if the two groups can be open simultaneously (i.e. are not in conflict), but their traffic flows interfere. This is, e.g., the case for bending vehicles and pedestrians. In this case the pedestrians have priority over the vehicles and the pedestrian flow must not be opened while the vehicles are already driving.

The third safety requirement *ObeysIGT_Abs* denotes that an opening signal group must obey to the intergreen time table, i.e., all conflicting signal groups have to be closed for at least their intergreen time before the group is opened.

If the TLS enters an unsafe situation the program has to react and reestablish safety within *MAX_FAIL_TIME*. We can now formulate the central safety requirement *Safety_Abs* for the TLS. Note that the actual value of *MAX_FAIL_TIME* depends on how fast the hardware can shut off the system, thus depends on the hardware environment the control software is embedded in. The *SAFEGUARD* has to obey to the three safety conditions and there must not be any sub-interval with $safe = False$ for more than *MAX_FAIL_TIME* ($\lceil safe = False \rceil_{>MAX\_FAIL\_TIME}$). The behavior requirement for the *SAFEGUARD* says that the TLS has to obey to *Safety_Abs* or has to be shut off. Note that *Conflict_Abs*, *Priority_Abs*, *ObeysIGT_Abs*, and *Safety_Abs* are auxiliary schemas whereas a box labeled *PROPERTY* denotes a direct requirement for the behavior of the class.

SAFEGUARD

DYNAMIC Safety_Abs

GroupStates; *SAFE*

$Conflict\_Abs \wedge ObeysIGT\_Abs \wedge Priority\_Abs$

$\neg$ somewhere $\lceil safe = False \rceil_{>MAX\_FAIL\_TIME}$

PROPERTY DYNAMIC

GroupStates; *SAFE*

$Safety\_Abs \vee (Safety\_Abs \frown \lceil OFF = True \rceil)$

After having specified the requirements of the safeguard we can implement them. The safeguard needs to store the time when a signal group was closed and which signal groups have just been opened resp. closed. The function *blocking_time* assigns to each closed signal group the time it was closed. The set *last_step_opened* holds all newly opened signal groups. Analogously, *last_step_closed* holds all newly closed groups. The operation *UPDATE_BLOCKING_TIME*[4] updates these variables in *SAMPLING_RATE* intervals.

[4] Note that in this operation dom *blocking_time* is set of signal groups closed in the previous step, whereas *closed* holds the current value. Therefore, $closed \setminus (\mathrm{dom}\ blocking\_time)$ denotes the signal groups that were recently opened and are closed now. Correspondingly, $opened \cap (\mathrm{dom}\ blocking\_time)$ denotes the signal groups that were closed in the last step and are opened now. $opened \lhd blocking\_time$ restricts the domain of *blocking_time* to *opened*. Thus, the operation restricts *blocking_time* to the closed groups and adds the newly closed groups related to the current time *Time*.

SAFEGUARD

DATA BlockingTime

$blocking\_time : SignalGroup \pfun TIME$
$last\_step\_closed, last\_step\_opened : \mathbb{P}\, SignalGroup$

OP UPDATE_BLOCKING_TIME

$\Delta BlockingTime$
$GroupStates$

$$last\_step\_closed' = closed \setminus (\mathrm{dom}\ blocking\_time)$$
$$last\_step\_opened' = opened \cap (\mathrm{dom}\ blocking\_time)$$
$$blocking\_time' = (closed \lhd blocking\_time) \cup (last\_step\_closed' \times \{ Time \})$$

Conflict_Conc

$BlockingTime$

$$\forall grp_1, grp_2 : SignalGroup \mid (grp_1, grp_2) \in conflict \bullet$$
$$\quad (grp_1 \in (\mathrm{dom}\ blocking\_time) \lor grp_2 \in (\mathrm{dom}\ blocking\_time))$$

Priority_Conc

$BlockingTime$

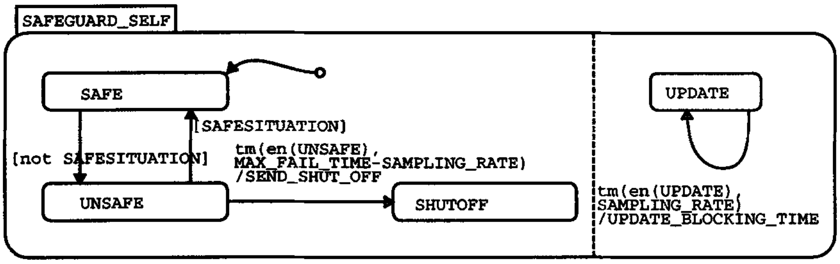
$$\forall grp_1, grp_2 : SignalGroup \mid (grp_1, grp_2) \in priority \bullet$$
$$\quad grp_1 \in last\_step\_opened \Rightarrow$$
$$\qquad (grp_2 \in \mathrm{dom}\ blocking\_time \lor grp_2 \in last\_step\_opened)$$

ObeysIGT_Conc

$BlockingTime$

$$\forall grp_1, grp_2 : SignalGroup \mid (grp_1, grp_2) \in conflict \bullet$$
$$\quad grp_1 \notin (\mathrm{dom}\ blocking\_time)$$
$$\qquad \Rightarrow (blocking\_time(grp_2) + IGT(grp_2, grp_1) < Time)$$

$$SAFESITUATION \mathrel{\widehat{=}} Conflict\_Conc \land Priority\_Conc \land ObeysIGT\_Conc$$


SAFEGUARD_SELF
SAFE
[SAFESITUATION]
[not SAFESITUATION]
tm(en(UNSAFE), MAX_FAIL_TIME-SAMPLING_RATE) /SEND_SHUT_OFF
UNSAFE
SHUTOFF
UPDATE
tm(en(UPDATE), SAMPLING_RATE) /UPDATE_BLOCKING_TIME

With these data values we can translate the temporal requirements into operational predicates that can be used in the statechart. We translate *Conflict_Abs* into *Conflict_Conc*, *Priority_Abs* into *Priority_Conc*, and *Obeys-*

IGT_Abs into *ObeysIGT_Conc*. *Priority_Conc* says that for a group that has just been opened ($grp_1 \in last\_step\_opened$) all other groups it has priority over ($(grp_1, grp_2) \in priority$) must be either closed or also just been opened.

The statechart implements the reactive behavior of the safeguard. It resides in the state SAFE if *SAFESITUATION* is true and switches to UNSAFE otherwise. If it stays in UNSAFE for $MAX\_DELAY\_TIME - SAMPLING\_RATE$ it shuts the system off.[5] Meanwhile, it periodical reads the actual values sent by the external facilities and updates *blocking_time* and *last_step_opened* accordingly. The statechart has to implement the behavior constraint of the process, i.e. the set of observable runs it defines has to be a subset of the set defined by the behavior constraint. Here, we have to prove that $in(SAFE) \Rightarrow (safe = True)$, where $in(SAFE)$ denotes that the statechart state *SAFE* is active.

4 Conclusion

In this paper we have presented a combination of formal specification techniques with a well-known structuring technique that has been successfully applied in software engineering. We are using Z and statecharts as basis formalisms for expressing the functional and reactive behavior of embedded systems. Both formalisms have found broad acceptance in the research area as well as in industry. The presented combination exploits the advantages of each formalism. Note that in this approach the semantics of the basic formalisms are preserved, which is very important for tool reuse. In contrast, in [22,6] the STATEMATE semantics were not preserved.

The usage of Z as language for the description of the functional aspects has turned out to be superior to e.g. the STATEMATE data description language [15], especially for the formulation of data invariants and safety requirements, and for the rather complex data-items as the phase transition table. The dynamic behavior of a process could be modeled very naturally with statecharts. Its graphical nature (in contrast to other process description languages and combinations of specification languages like [17,19,1,20,16]) provides a good overview over the different states and possible state transitions of a process. The architectural view was described by data flow diagrams. Here, a more powerful notation would have been helpful, supporting genericity in order to reuse components as well as collections of processes.

Supplementing Z with temporal logic and an adequate satisfaction relation for statecharts and temporal logic is still subject to further research. Nevertheless, we believe that the case study shows impressively the advantage of this approach. Within the ESPRESS project, work on the development of tools to support the presented technique and method is investigated. This includes type-checking, execution of Z specifications, and verification and

[5] `tm(E, T)` is true T seconds after event E occured. `en(UNSAFE)` denotes the event that state UNSAFE was entered.

validation of combined specifications. With that, it should be possible to simulate and analyze the model extensively.

With the formalization of safety requirements we were able to discover ambiguities and unprecise formulations in the natural language specification. We believe that the precise formulation of safety requirements in an early development phase will turn out to be very helpful to increase the developer's understanding of the problem, avoiding misconceptions and design errors caused by ambiguous or incomplete requirements.

The applicability of our approach has been demonstrated in a case-study where a traffic light system has been specified [3,4]. Even if only parts of the case study could be presented here (the entire specification contains over 70 pages), we found the division of a system into different views and the formulation of explicit safety requirements convincing for the specification of safety-critical embedded systems.

References

1. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
2. R. Büssow, H. Dörr, R. Geisler, W. Grieskamp, and M. Klar. *μSZ* – Ein Ansatz zur systematischen Verbindung von Z und Statecharts. Technical Report 96-32, Technische Universität Berlin, Feb. 1996.
3. R. Büssow, R. Geisler, M. Klar, and S. Mann. Spezifikation einer Lichtsignalanlagen-Steuerung mit *μSZ*. Technical Report 97-13, Technische Universität Berlin, 1997.
4. R. Büssow, R. Geisler, and M.Klar. Spezifikation eingebetteter Steuerungssysteme mit Z und Statecharts. In *Tagungsband zur 5. Fachtagung Entwurf komplexer Automatisierungssysteme.* TU Braunschweig, 1997.
5. 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, *Advances in Computing Science – Asian '97*, volume 1345 of *LNCS*, pages 46–56. Springer-Verlag, 1997.
6. R. Büssow and M. Weber. A steam-boiler control specification using statecharts and Z. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*. Springer, 1996.
7. Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
8. T. DeMarco. *Structured analysis and system specification.* Yourdon Press, Engelwood Cliffs, NY, USA, 1978.
9. Deutsche Elektrotechnische Kommission im DIN und VDE (DKE). DIN Norm VDE 0832 - Strassenverkehrs-Signalanlagen (SVA), 1990.
10. Forschungsgesellschaft für Strassen- und Verkehrswesen. Richtlinien für Lichtsignalanlagen - RiLSA, 1992.
11. W. Grieskamp, M. Heisel, and H. Dörr. Specifying safety-critical embedded systems with statecharts and Z: An agenda for cyclic software components. accepted for publication at ETAPS'98, 1998.

12. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
13. 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 No. 4, Apr. 1990.
14. D. Harel and A. Naamad. The statemate semantics of statecharts. Technical report, The Weizmann Institute of Science, Oct. 1995.
15. D. Harel and M. Politi. Modeling reactive systems with statecharts: The statemate approach. i-Logix Inc, Three Riverside Drive, Andover, MA 01810, USA, June 1996. Part No. D-1100-43, 6/96.
16. M. Heisel and C. Sühl. Combining Z and Real-Time CSP for the development of safety-critical systems. In *Proceedings 15th International Conference on Computer Safety, Reliability and Security*. Springer, 1996.
17. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, Eaglewood Cliffs, N.J., 1985.
18. N. Leveson. *Safeware – System Safety and Computers*. Addison Wesley, 1995.
19. LOTOS - A formal description technique based on temporal ordering of observational behaviour. Information Processing Systems - Open Systems Interconnection **ISO DIS 8807**, jul. 1987. (ISO/TC 97/SC 21 N).
20. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In *Proceedings of FME'97: Industrial Benefits of Formal Methods*, Graz, Austria, September 1997. Springer-Verlag.
21. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
22. M. Weber. Combining statecharts and Z for the desgin of safety-critical control systems. In *Industrial Benefits and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 307–326. Springer, 1996.