

# Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components

Wolfgang Grieskamp<sup>1</sup>, Maritta Heisel<sup>2</sup>, and Heiko Dörr<sup>3</sup>

<sup>1</sup> Technische Universität Berlin, Institut für Kommunikations- und Softwaretechnik, Sekr. FR5-13, Franklinstr. 28/29, D-10587 Berlin, Germany, email: wg@cs.tu-berlin.de

<sup>2</sup> Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, D-39016 Magdeburg, Germany, email: heisel@cs.uni-magdeburg.de

<sup>3</sup> Daimler-Benz AG, Forschung Systemtechnik, Alt-Moabit 96a, D-10569 Berlin, Germany, email: Heiko.Doerr@DBAG.Bln.DaimlerBenz.com

**Abstract.** The application of formal techniques can contribute much to the quality of software, which is of utmost importance for safety-critical embedded systems. These techniques, however, are not easy to apply. In particular, methodological guidance is often unsatisfactory. We address this problem by the concept of an *agenda*. An agenda is a list of activities to be performed for solving a task in software engineering. Agendas used to support the application of formal specification techniques provide detailed guidance for specifiers, templates of the used specification language that only need to be instantiated, and application independent validation criteria. We apply the agenda approach to a particular class of embedded safety-critical systems, the formal specification of which has been investigated in the case-studies of the German ESPRESS project during the last two years.

## 1 Introduction

Every software-based system potentially benefits from the application of formal techniques. For the development of mission or even safety-critical embedded systems, however, their use is of particular advantage, 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.

A major drawback of formal techniques is that they are not easy to apply for the average software engineer. Besides the facts that users of formal techniques need an appropriate education and have to deal with lots of details, they are often left alone with a mere formalism without any guidance on how to use it. Hence, *methodological support* is a key issue to bring formal techniques into practice.

Methodological support for specification development must be abstract enough to cover a significant range of applications, but also detailed enough

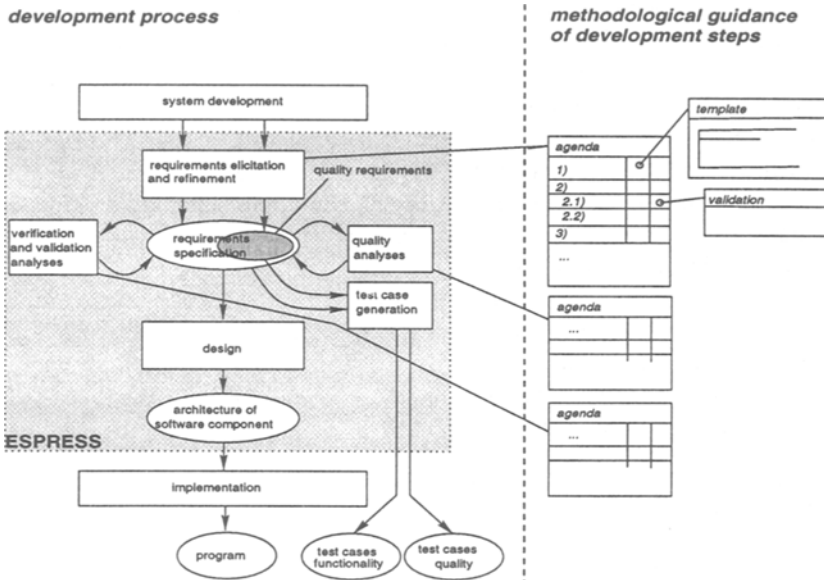


Fig. 1. Basic ESPRESS process model

to provide real guidance to developers. *Agendas*, as introduced in [9,10], provide a concept for representing methodological process knowledge for particular software architectures on a relatively fine-grained level of detail.

In this paper, we demonstrate the application of the concept of agendas to a particular class of embedded safety-critical systems. The architecture we study is that of a *cyclic software component* – a piece of software in a technical system which is triggered in regular time intervals by its environment to compute output values (usually commands to some actuators) from given input values (usually sensor values), and an internal state. The agenda we represent for this architecture is condensed from experiences with case studies performed in the ESPRESS project during the last two years<sup>1</sup>.

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.

Our agenda for cyclic software components elaborates on two particular aspects of embedded systems, motivated by the demands of the ESPRESS application context. First, special care is taken to accurately develop the embedding of the software in its surrounding technical system. Second, *quality requirements* such as high-level or safety-related conditions which have to be guaranteed by the software are treated systematically. The general ESPRESS methodology re-

<sup>1</sup> The ESPRESS project is a cooperation of industry and research institutes funded by the German ministry BMBF (“Förderschwerpunkt Softwaretechnologie”).

quires that – if possible – these requirements are specified as *properties*, which have to be logical *consequences* of an explicitly constructed model of the software. The redundancy introduced by this approach increases the potential for checking the consistency of the formal specification.

We use the ESPRESS notation  $\mu SZ$  [2] to express the specifications developed with our agenda for cyclic software components. This notation provides a semantically well-defined combination of the StateMate languages [7] (namely statecharts and activity charts), the formal specification language Z [16], and an extension of Z by temporal logics [4]. The StateMate languages and Z have been chosen for ESPRESS because of their relevance in industrial contexts and their fairly good tool support. For reasons of space, we cannot systematically explain  $\mu SZ$  and its constituting languages; we only give an informal explanation of the constructs used in this paper as they appear, and assume some familiarity of the reader with the Z and StateMate languages.

## 2 Agendas

An agenda gives guidance on how to perform a specific software development activity. Agendas can be used for structuring quite different activities in different contexts. We have set up and used agendas that support requirements engineering, specification acquisition, software design using architectural styles, and developing code from specifications [8]. Agendas are especially suitable to support the application of formal techniques in software engineering.

An agenda is 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 achieve the goal, similar to the general process proposed by the spiral model of software engineering. Agendas are presented as tables, see Fig. 3 on page 5. Agendas may be nested, and we call the “super-steps” *stages* (see, e.g., Fig. 2 on the following page).

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. 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. This first step requires a basic understanding of the

problem to be solved. 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 just be filled in (which nevertheless requires creativity) 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.

Agendas cannot replace creativity, but they can tell the software engineer what needs to be done and help avoid omissions and inconsistencies. Their advantage lies in an improvement of the quality of the developed products and in the possibility for reusing the knowledge incorporated in an agenda.

### 3 Agenda for Cyclic Software Components

Stage
1 Context embedding
2 Quality requirements
3 Model construction

**Fig. 2.** Stages of the agenda for cyclic software components

we use an *intelligent cruise control* system, which serves to automatically adjust the speed of a vehicle according to the driver's request. In addition to this conventional cruise control functionality, our version uses a sensor to detect a vehicle driving ahead, and adjusts the speed to maintain a certain safety distance. This example is extracted from one of the internal ESPRESS case studies, and modified for our illustration purposes.

The agenda for cyclic software components consists of three stages, which are shown in Fig. 2. Stage 1 must be performed first; Stage 2 and 3 can be performed independently of each other. Each of the stages is performed following a sub-agenda, as described below. As a running example,

#### Stage 1: Context embedding

Embedded software is characterized by the fact that the interfaces to the environment are not standardized to a degree as it is nowadays common for software running on e.g. workstations. Hence, a developer should take special care to model the *context embedding* of the software. In ESPRESS, the context definition also serves as a starting point for a simulation of the software, using the Statemate tool. The sub-agenda for context embedding is shown in Fig. 3 on the following page.

**Step 1.1: Specify technical interfaces.** The technical interfaces of an embedded software component are usually determined during system design, and

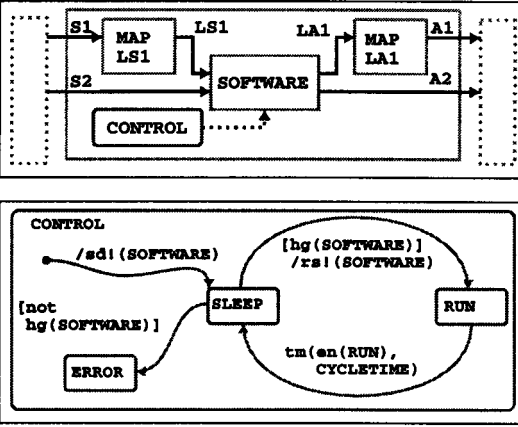
Step	Validation Conditions
<p><b>1.1 Specify technical interfaces:</b></p> <p><i>TechnicalDefs</i> _____</p> <p>  ...                      [...]                      ...</p> <hr/> <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p><i>TechnicalSensors</i> _____</p> <p><i>TechnicalDefs</i> _____</p> <p>  <i>PORT S1</i> _____</p> <p>  ...</p> <p>  ...</p> </div> <div style="width: 45%;"> <p><i>TechnicalActuators</i> _____</p> <p><i>TechnicalDefs</i> _____</p> <p>  <i>PORT A1</i> _____</p> <p>  ...</p> <p>  ...</p> </div> </div>	<ul style="list-style-type: none"> <li>◦ ranges of values of sensors and actuators are consistent with the technical specifications of the interfaces</li> <li>◦ errors of sensors and actuators are taken into account</li> <li>⊢ invariants are consistent</li> </ul>
<p><b>1.2 Design and specify logical interfaces and their mapping to technical ones:</b></p> <p><i>LogicalDefs</i> _____</p> <p>...</p> <hr/> <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p><i>LogicalSensors</i> _____</p> <p><i>LogicalDefs</i> _____</p> <p>  <i>PORT LS1</i> _____</p> <p>  ...</p> <p>  ...</p> </div> <div style="width: 45%;"> <p><i>MapLS1</i> _____</p> <p><i>LogicalSensors</i>; ...</p> <p>  <i>BEHAVIOR</i> _____</p> <p>  ...</p> <p>  ...</p> </div> </div> <hr/> <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p><i>LogicalActuators</i> _____</p> <p>...</p> <p>...</p> </div> <div style="width: 45%;"> <p><i>MapLA1</i> _____</p> <p>...</p> <p>...</p> </div> </div>	<ul style="list-style-type: none"> <li>◦ errors of technical sensors are taken into account</li> <li>⊢ invariants are consistent</li> <li>⊢ mappings are unique</li> <li>⊢ mappings are total</li> </ul>
<p><b>1.3 Derive software/context information flow and cycle control:</b></p> <p><i>Context</i> _____</p> <p><i>TechnicalSensors</i>; <i>TechnicalActuators</i></p> <p><i>LogicalSensors</i>; <i>LogicalActuators</i></p> <hr/> 	<p><i>no conditions</i></p>

Fig. 3. Steps of Stage 1: context embedding

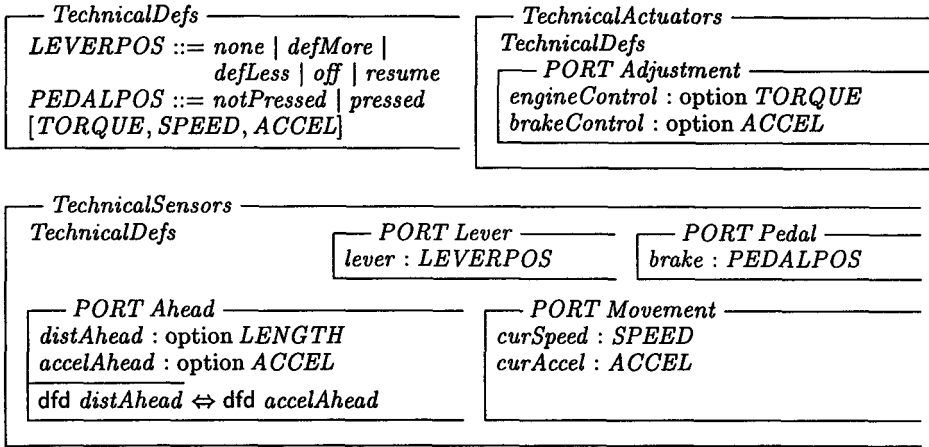


Fig. 4. Technical interfaces of the cruise control

cannot be modified by the software developer. Since their characteristics and capabilities may have significant influence on the further development, the first step of context embedding is to describe the technical interfaces in the modeling language.

Figure 3 on the page before gives templates for describing technical interfaces in our modeling language  $\mu SZ$ . The structuring entities of  $\mu SZ$  are *process classes* (the outer boxes in the figure, e.g., *TechnicalDefs*), which 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. Interpreted standalone, *PORT* schemata do not differ from plain Z schemata. However, they contribute to the semantics of an entire process class, defining the variables belonging to the shared data state of instances of the class.

For Step 1.1, the agenda in Fig. 3 suggests to collect the technical interfaces in process classes called *TechnicalSensors* and *TechnicalActuators*, respectively, which contain sets of *PORT* schemata. The types and constants used to define these ports are collected in a third process class, *TechnicalDefs*, which is included by the other classes. The inclusion of process classes can be interpreted as textual expansion.

The validation conditions associated with Step 1.1 first require the developer to carefully check whether the types defined to model the values of sensors and actuators really capture the technical properties of the technical sensors and actuators. The second validation condition suggests to define appropriate error values for the types. Finally, all invariants must be satisfiable, i.e., there must exist legal states of the system ports. Note that validation conditions marked with “o” are informal, whereas validation conditions marked with “⊢” are formal and hence can be checked with appropriate tool support.

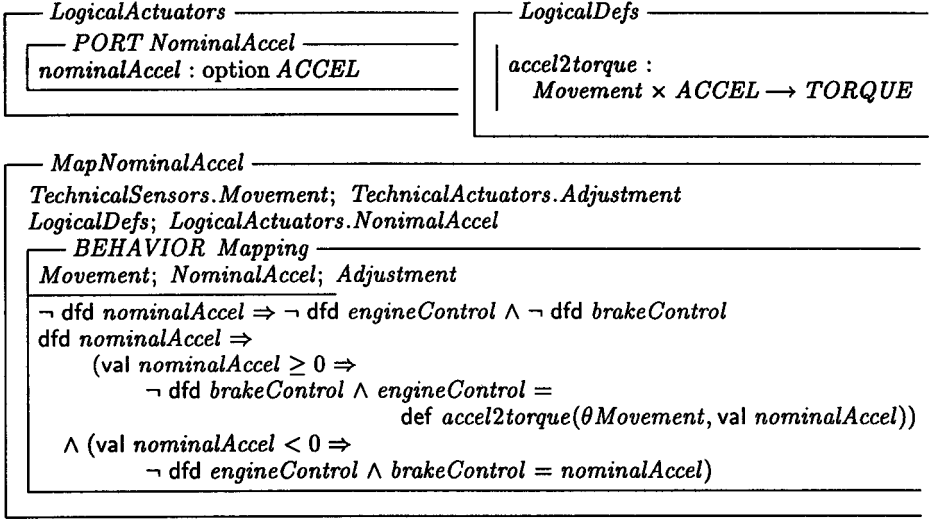


Fig. 5. Logical interfaces of the cruise control

*Cruise Control.* Fig. 4 on the preceding page shows how we apply Step 1.1 to the specification of the cruise control. The port *Lever* describes the driver's control lever, which can be used to turn off the cruise control, to increase or decrease the requested speed, to turn off the cruise control, and to resume its operation. The port *Pedal* models the brake pedal, the port *Ahead* the distance and relative acceleration with respect to a vehicle driving ahead, and the port *Movement* provides information about the current speed and acceleration of the vehicle. The port *Adjustment* describes the output of the cruise control, which consists of an engine torque and a (negative) acceleration for controlling the brake. Variables declared as  $x : \text{option } A$  carry values of  $A$  which may be available or not; we use  $\text{dfd } x$  to indicate whether the value of the optional variable  $x$  is available,  $\text{val } x$  to refer to that value (if it is defined), and  $\text{def } v$  to construct a defined value from  $v$ . If, e.g., the value of the sensor *distAhead* is not defined, then no vehicle driving ahead is detected, and if the actuator *engineTorque* is not defined, then the cruise control does not affect the engine.

For reasons of space, we cannot present the full specification of the technical interfaces of the cruise control system. Hence, we cannot check the validation conditions in detail. Let us just note that the only given invariant,  $\text{dfd } \text{distAhead} \Leftrightarrow \text{dfd } \text{accelAhead}$ , is indeed consistent, and that – for reasons of simplicity – we assume to have perfect sensors without errors in this example.

**Step 1.2: Design and specify logical interfaces.** Apart from being non-standardized, the technical interfaces of an embedded software component may be also on a relatively low technical level, which hinders a problem-oriented specification. It may therefore be useful to introduce *abstractions* of the technical interfaces, which is achieved by defining *logical interfaces*.

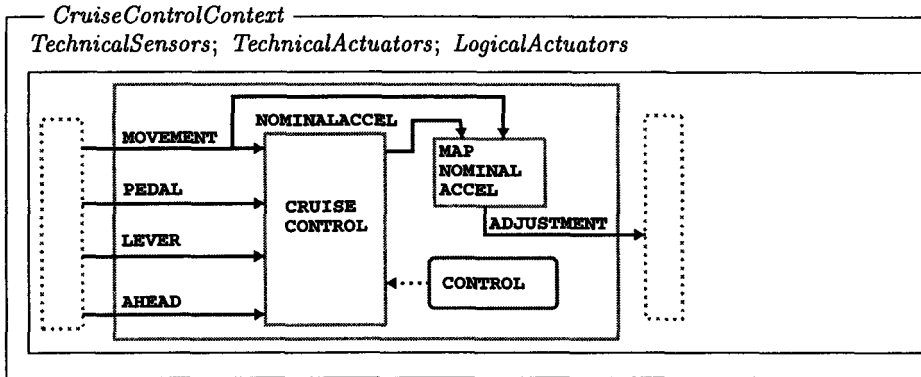


Fig. 6. Activity chart of the cruise control context

The values of sensors and actuators of a logical interface should be totally and uniquely defined by the technical values (see validation conditions associated with this step). In the simplest case, this mapping can be defined by a conversion function which maps a technical sensor to a logical sensor or a logical actuator to a technical actuator, respectively. In more complex situations, the mapping may require an internal state, for example if a logical sensor accumulates the values of a technical one. In any case, we define the mapping by a dedicated process class that describes the conversion by a property schema or by a state-chart (in the template of Fig. 3 on page 5, these classes are called *MapLS1* and *MapLA1*, respectively). These process classes are instantiated as sub-processes of the overall process modeling the system context, as will be seen in the next step.

The first validation condition associated with Step 1.2 suggests to apply fault tolerance techniques, e.g., consistency checks on sensor values and feedback control to check if actuator commands have been executed appropriately.

*Cruise Control.* We introduce a logical actuator *nominalAccel*, which abstracts from the two quite technical values *engineControl* and *brakeControl* given in the output port *Adjustment*, as sketched in Fig. 5 on the preceding page. The process class *MapNominalAccel* performs the mapping of the logical to the technical actuators. A schema introduced with the role *BEHAVIOR* describes an invariant which holds whenever a process is running.

Proving the last two validation conditions amounts to proving that the function *accel2torque* defined in the class *LogicalDefs* is indeed a total function.

**Step 1.3: Derive software/context information flow.** The description of the technical interfaces, the logical interfaces, and their mapping induces an activity chart, which is derived from the template given for Step 1.3 in Fig. 3 on page 5. The activity charts of Statemate used in  $\mu\text{SZ}$  combine the descriptions of information flow, of instantiation of sub-processes from process classes (rectangular boxes), and of behavior described by statecharts (rounded boxes). In Fig. 3, the overall description of the system's behavior aggregates a sub-process *Software*, as well as sub-processes for mapping technical to logical interfaces. The



Step	Validation Conditions
2.1 Collect relevant quality requirements	o requirements are realizable
2.2 Specify model properties: <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <i>Software</i> _____  <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px;"> <i>Obs1</i> _____  ... </div> <div style="border: 1px solid black; padding: 2px;"> <i>Obs2</i> _____  ... </div> </div>   <div style="border: 1px solid black; padding: 2px;"> <i>PROPERTY DYN</i> _____  ... <math>[Obs1] \wedge [Obs2]</math> ... </div> </div>	o observations form a problem-oriented classification of possible situations $\vdash$ properties are consistent

Fig. 7. Steps of Stage 2: quality requirements

information flows between these processes are labeled with ports, and semantically describe visibility of shared variables between processes. The aggregation of the statechart *Control* and the dotted lines are only for documentation purposes; they indicate that *Control* schedules the activity of the sub-process *Software*.

The scheduler defined by *Control* applies to any cyclic software component developed using this agenda, and is quite simple. It assumes that the software, once running, reads the sensors, computes the actuators, and then suspends itself. The scheduler thus periodically resumes the software in intervals of a certain cycle time. Notationally, Statemate's mechanism for suspending and resuming processes (processes are also called *activities* in Statemate) is used. The Statemate action *sd ! (SOFTWARE)* stands for suspending a process, *rs ! (SOFTWARE)* for resuming, and the condition *hg(SOFTWARE)* tests whether a process is "hanging", that is suspended. The event *tm(en(RUN), CYCLETIME)* appears *cycleTime* time units after the state *RUN* has been entered, where *en* stands for "entered", and *tm* stands for "timeout". Because the result of Step 1.3 can be derived schematically from the parts of the specification defined in Steps 1.1 and 1.2, there are no validation conditions associated with this step.

*Cruise Control*. Fig. 6 on the preceding page shows the result of Step 1.3. We only need to draw the activity chart (where the information flows are already induced by Steps 1.1 and 1.2); the statechart *Control* can be taken as is from the template in Fig. 3 on page 5. Only *Software* is renamed to *CruiseControl*.

## Stage 2: Quality requirements

The systems we study have to fulfill certain *quality requirements*. Typical examples are safety requirements, but also high-level requirements from earlier development phases may be transferred to the software development phase. A common characteristic of quality requirements is that they only address certain *selected aspects* to be realized by the software – these aspects are important enough to be emphasized explicitly in the specification. Technically, quality requirements are formulated as *model properties*, which have to be logical consequences of the model of the software as it is constructed in Stage 3. With model properties, redundancy is deliberately introduced in the specification. This contributes to the

potential for checking consistency by deduction, model checking, and systematic testing. Figure 7 on the preceding page describes the agenda for treating quality requirements.

**Step 2.1: Collect relevant quality requirements.** The quality requirements are usually defined during system design. In this step, the ones that are relevant for the software component under development are collected and documented.

*Cruise Control.* A few of the quality requirements are the following:

- *Activity.* The cruise control is allowed to adjust speed only if the driver has activated it through the control lever, and did not deactivate it since then.
- *Asymptotic String Stability.* If several vehicles using the cruise control drive in a queue, a sudden change of the speed of one of them must lead to changes of speed of the the following vehicles which fade away along the queue.

Because cruise control systems are already on the market, these requirements are well understood and known to be realizable with our technical interfaces. Hence, the validation condition associated with this step is fulfilled.

**Step 2.2: Specify model properties.** It is not realistic to demand that all quality requirements be specified formally as model properties. For example, the property of string stability cannot be expressed easily, because it would require to formalize aspects of the mathematics of control theory<sup>2</sup>. However, where it is possible, the quality requirements should be expressed as model properties, to be treated automatically in a review stage later on.

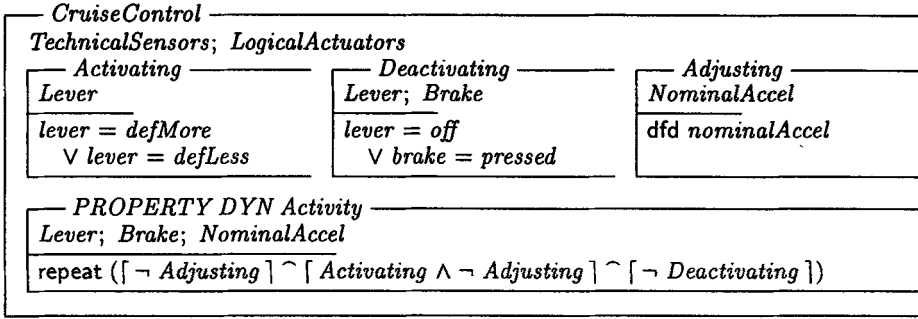
Our modeling language allows us to formalize properties as *temporal observations* of the sensors and actuators of the technical and logical interfaces. The logic used for this purpose is based on *discrete temporal interval logic*, including real-time constraints [4]. The modifier *DYN* in a schema declaration (possibly in conjunction with roles such as *PORT* or *PROPERTY*) signals that the schema uses temporal logic.

A useful guideline for specifying the model properties is to first introduce abstractions of common situations observable on the interfaces. In the template of Fig. 7 on the page before, these are introduced by the schemata *Obs1*, *Obs2*, and applied in the temporal formula of the dynamic property box.

The first (informal) validation condition suggests that the model properties be oriented on a classification of the relevant situations of the observable behavior of the system, whereas the second validation condition is an obvious consistency requirement.

*Cruise Control.* In Fig. 8 on the following page, we define schemata for observing the situations where the driver activates and deactivates the cruise control, and where the cruise control produces an output value to adjust speed. These schemata are used to formalize one of the quality requirements, namely the safety-condition “Activity”. Intuitively, the temporal formula given in the property-box *Activity* can be interpreted as a kind of regular expression: the

<sup>2</sup> In fact, notions of control theory could be expressed in Z, but ESPRESS does not aim at these goals.



**Fig. 8.** Model properties of the cruise control

admissible traces of the behavior of the cruise control repeatedly consists of an interval where adjustment of speed is not performed, followed by an interval where the driver activates the cruise control (adjustment of speed still does not take place), followed by an interval where the driver continuously does not deactivate the cruise control. Thereby, the temporal predicate  $[p]$  holds for those finite or infinite intervals where the predicate  $p$  holds in each state. Note that we do not say anything about whether the cruise control actually *ever* adjusts speed; we just say when it should *not* do so. This is typical for specifying model properties, where we are only interested in selected aspects of the software.

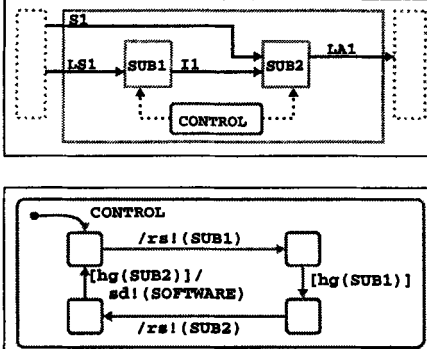
### Stage 3: Model construction

In this stage, we construct a model for the cyclic software component under consideration. There are several strategies for doing so, which depend on the problem to solve. Here we consider two variants: model construction by functional decomposition, and model construction by partitioning behavior into operational modes.

**Variant 3a: Model construction by functional decomposition** The problem to solve by the cyclic software component might be more adequately solved by decomposing it into subproblems, instead of giving a monolithic solution. The reasons for this may be that the problem is too large to be tackled in a monolithic way, that a decomposition follows naturally from the structure of the problem, or that existing components should be integrated into the design.

For a cyclic software component which computes output values from input values and internal state, a decomposition is naturally achieved in a *functional* style, based on information flow between the subcomponents (Fig. 9 on the next page). This is also the approach the Statemate tool supports best.

**Step 3a.1: Design functional decomposition using an activity chart.** Guidelines on how to perform a functional decomposition depend on the application. A useful approach is data-oriented, and considers intermediate values to be computed by the subcomponents. If we reuse existing components, these intermediate values are naturally their output interfaces. However, in general, decomposition is a problem that requires creativity. Hence our agenda suggests

Step	Validation Conditions
<p><b>3a.1 Design functional decomposition:</b></p> <p><i>Software</i></p> <p><i>InternalInterfaces</i></p> 	<p>⊢ information flow is free of cycles</p> <p>⊢ all sensors are used and all actuators are served</p>
<p><b>3a.2 Specify the internal interfaces:</b></p> <p><i>InternalInterfaces</i></p> <p><i>PORT I1</i></p> <p>...</p>	<p>⊢ invariants of ports are consistent</p>
<p><b>3a.3 For each subcomponent obtained in 3a.1, apply Stage 3 again.</b></p>	<p><i>no conditions</i></p>

**Fig. 9.** Steps of Stage 3, Variant *a*: model construction by functional decomposition

to first design the *principle* information flow between subcomponents by drawing an activity chart. The precise specification of the intermediate interfaces themselves is postponed until the next step<sup>3</sup>.

Once a information flow between the subcomponents has been defined, the data dependencies canonically induce a scheduling as described by the statechart *Control* of the template for Step 3a.1 in Fig. 9. Each subcomponent is treated similarly to a cyclic-software component: once it is resumed, it is expected to compute its output values and then to suspend itself. The scheduler activates the subcomponents one after the other in the order induced by the information flow.

The validation conditions associated with this step ensure that the component eventually produces an output if the subcomponents do so, and that all sensors and actuators are actually used by the system.

*Cruise Control.* We assume that we can reuse an existing component that implements a speed adjustment: it calculates a nominal acceleration from a given

<sup>3</sup> This differs from the order used in Stage 1, where we first specified the interfaces, and then the information flow; but there the exact definition of the interfaces had been given by the environment.

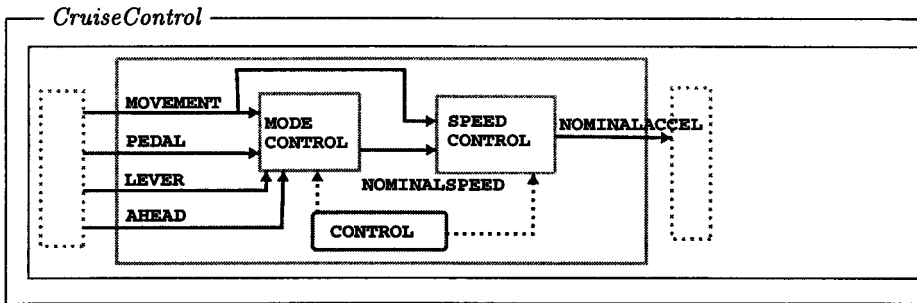


Fig. 10. Functional decomposition of cruise control

nominal speed and the vehicle movement. What is left to do is to introduce a subcomponent which controls activation and deactivation of the cruise control, and which decides to use the speed requested by the driver or a speed lower than the nominal speed to keep a certain safety distance. The decomposition leads to the activity chart given in Fig. 10, where *NominalSpeed* is a newly introduced internal interface, *ModeControl* is the subcomponent controlling the activation of the cruise control, and *SpeedControl* is the reused component.

**Step 3a.2: Specify the internal interfaces.** In this step, we specify the internal interfaces as they have been introduced in the last step. This step is similar to the introduction of interfaces in Stage 1; therefore, details are omitted here.

**Step 3a.3: Recursively apply Stage 3.** For subcomponents yielded by the decomposition and which are not reused, we apply Stage 3 again. For the cruise control, this applies to the subcomponent *ModeControl*, which we specify using a different sub-agenda, shown in Fig. 11 on the following page.

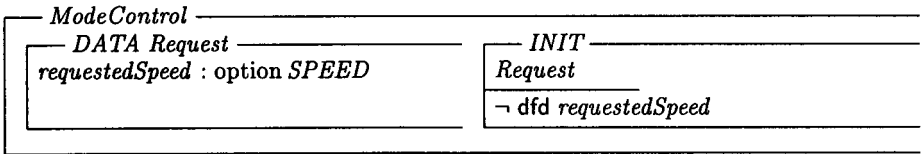
**Variant 3b: Model construction by mode-based design** The problem to solve by the software component might be adequately modeled by introducing *operational modes* for the software component (such as passive, active, emergency, etc.). A cyclic computation then triggers transitions between the operational modes. The agenda in Fig. 11 on the next page describes how to proceed for this modeling technique.

**Step 3b.1: Define modes by initial statechart.** In this step we introduce the different operational modes of the software component. Technically, this is done by defining an initial statechart (without transitions), where states or combinations of parallel states represent modes. We introduce this chart before the internal data (next step), because we might want to specify invariants on the data that depend on the current operational mode.

The initial statechart contains a so-called static reaction (*en*(M1) or *en*(M2)/*sd*!(SOFTWARE)) which suspends the software whenever a mode is entered, signaling to the environment that the computation of the current cycle has been finished. A static reaction in Statemate is syntactically similar to a transition label guard/action; semantically, its action is executed whenever the guard becomes true.

Step	Validation Conditions
<b>3b.1 Define modes by initial statechart:</b> <div style="border: 1px solid black; padding: 10px; margin: 10px;"> <i>Software</i> _____  <div style="border: 1px solid black; padding: 10px; margin: 10px;"> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 5px; text-align: center;">M1</div> <div style="border: 1px solid black; padding: 5px; text-align: center;">M2</div> </div> <p>en(M1) or en(M2)/sd!(SOFTWARE)</p> </div> </div>	<i>no conditions</i>
<b>3b.2 Define internal data state:</b> <div style="border: 1px solid black; padding: 10px; margin: 10px;"> <i>Software</i> _____  <div style="display: flex; justify-content: space-between; align-items: center;"> <div style="border: 1px solid black; padding: 5px;"> <i>DATA Internal</i> _____  ... </div> <div style="border: 1px solid black; padding: 5px;"> <i>INIT</i> _____  ... </div> </div> </div>	$\vdash$ invariants of data schemata are consistent $\vdash$ initial data state exists
<b>3b.3 Define transitions:</b> <div style="border: 1px solid black; padding: 10px; margin: 10px;"> <i>Software</i> _____  <div style="border: 1px solid black; padding: 10px; margin: 10px;"> <p>en(M1) or en(M2)/sd!(SOFTWARE)</p> <div style="display: flex; justify-content: space-between; align-items: center; margin-top: 10px;"> <div style="border: 1px solid black; padding: 5px;"> <i>GUARD G1</i> _____  ... </div> <div style="border: 1px solid black; padding: 5px;"> <i>OP Op1</i> _____  ... </div> </div> </div> </div>	<ul style="list-style-type: none"> <li>◦ for each mode, expected inputs are systematically treated</li> <li><math>\vdash</math> for each two transitions leaving a state, guards are exclusive</li> <li><math>\vdash</math> all states are reachable</li> <li><math>\vdash</math> from each mode, all possible (transitive) transitions reach another mode with a finite number of steps</li> </ul>

**Fig. 11.** Steps of Stage 3, Variant *b*: model construction by mode-based design



**Fig. 12.** Internal data of the *ModeControl* subcomponent of the cruise control

*Cruise Control.* The complete statechart with transitions as it is obtained in Step 3b.3 is given in Fig. 13 on page 16, and will be explained there.

**Step 3b.2: Define internal data state.** Internal data is introduced in  $\mu SZ$  by a schema with the *DATA* role, its initialization by a schema with the *INIT* role. The validation conditions associated with this step stem from the recommended Z methodology [17].

*Cruise Control.* In Fig. 12, the internal data of the subcomponent *ModeControl* is defined. It declares a variable *requestedSpeed*, whose value (if defined) describes the nominal speed which the last time has been requested by the driver. Initially, *requestedSpeed* is undefined.

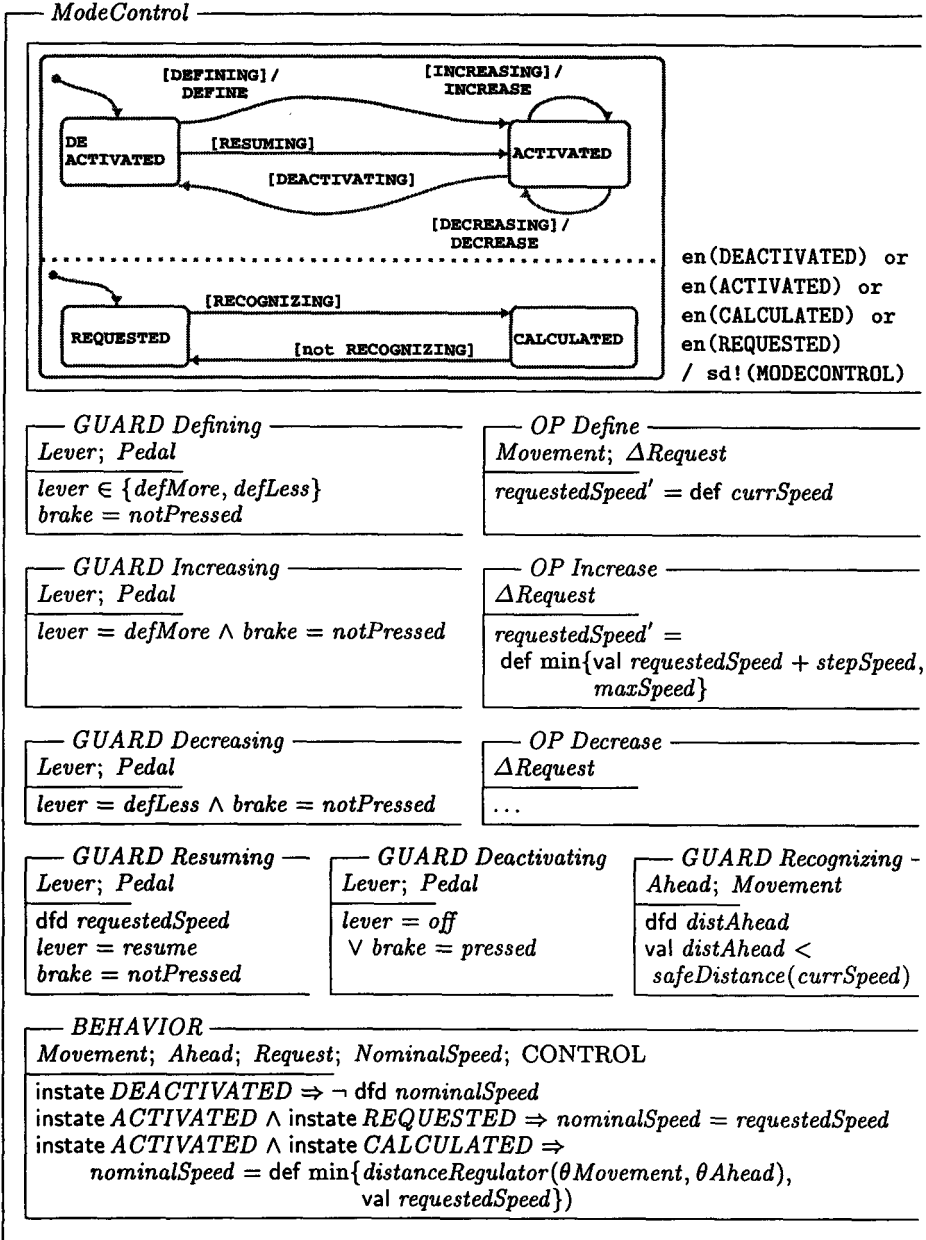
**Step 3b.3: Define transitions between states of statechart.** In this step, we refine the statechart developed in Step 3b.1 by transitions and possibly by intermediate states. Transitions are labeled systematically as  $[G]/Op$ , where  $G$  and  $Op$  are Z schemata introduced with the roles *GUARD* and *OP*, respectively.

Due to the static reactions introduced in Step 3b.1, the software is suspended whenever a transition reaches a state corresponding to an operational mode. Intermediate states do not necessarily lead to a suspension, as it is the case, e.g., for the internal state  $S$  in the template for Step 3b.3 in Fig. 11. The validation conditions associated with this step require the developer to check if all inputs are treated appropriately, and to show that the system behaves deterministically. Moreover, useless states that cannot be reached are not allowed. An important condition to check is whether mode transitions terminate, that is starting from any mode, for all possible inputs another mode is reached in a finite number of steps. The template state-chart given for Step 3b.3 in Fig. 11 shows that this condition is not trivial if intermediate states are used (it is possible that the process hangs in state  $S$ ).

*Cruise Control.* Applying Step 3b.3 to the subcomponent *ModeControl* leads to the statechart, guards and operations given in Fig. 13 on the following page. The statechart does not contain intermediate states. The cartesian product of the state sets  $\{ACTIVATED, DEACTIVATED\}$  and  $\{REQUESTED, CALCULATED\}$  makes up the set of operational modes. In addition to the declared objects, we use the following Z constants: *stepSpeed* : *SPEED* is the offset how to increase or decrease the requested speed, and *maxSpeed* : *SPEED* is the maximum requested speed the cruise control is allowed to manage. The function *safeDistance* : *SPEED*  $\rightarrow$  *LENGTH* yields the safe distance to a vehicle ahead in dependency of a driving speed. The function *distanceRegulator* : *Movement*  $\times$  *Ahead*  $\rightarrow$  *SPEED* represents an algorithm calculating a nominal speed from the movement of the vehicle and information about a vehicle ahead.

## 4 Conclusions

We have demonstrated that the agenda approach supports the systematic development of requirement specifications for high quality embedded systems on a non-trivial level of detail which gives substantial guidance to developers. As already noted, agendas are not intended to replace creativity and do not aim at completely automating development processes. Hence, in the first steps of an agenda, high-level decisions have to be taken. The validation conditions associated with the early steps of an agenda are mostly informal, encouraging developers to carefully re-consider their decisions, see e.g. Step 1.1 of the agenda of Fig. 3. Later steps in an agenda, on the other hand, often have validation conditions associated with them that can be formally expressed and proven. The reason is that in the later steps consistency conditions between the various parts of the specification that are already developed can be stated. Step 1.2 of the agenda shown in Fig. 3 is an example. Finally, some steps of an agenda



**Fig. 13.** Modes and transitions of the *ModeControl* subcomponent of the cruise control



(usually the last steps) can be performed in an entirely schematic way, because they merely consist in an appropriate combination of parts of the specification developed in earlier steps, see e.g. Step 1.3 of the agenda of Fig. 3.

Agendas are language-independent to a large extent. Changing the language in which the developed specification is expressed consists mostly in replacing the templates of the various steps, and effects the steps themselves very little, see the agenda presented in [10].

The validation conditions are a very important aspect of agendas. Clearly, the errors revealed by failing to demonstrate validation conditions of an agenda can only be of an application-independent nature. Checking the validation conditions cannot guarantee, e.g., that a system is adequately modeled by a developed specification. Nevertheless, many common errors can be discovered. As reported by Heitmeyer et al. [11], in the certification of the Darlington plant (which cost \$ 40M), “the reviewers spent too much of their time and energy checking for simple, application-independent properties.” To improve this situation, Heitmeyer et al. have implemented a tool that performs consistency checks. Since this tool is not tailored for any application domain, it can only check very general consistency conditions. In comparison, the validation conditions provided by agendas are much more to the point (see e.g. the validation conditions of Step 3.b.3 of the agenda shown in Fig. 11), such that more specific tool support for checking validation conditions generated by agendas is conceivable. But even if no specific support tools for agendas are available, agendas allow developers to use existing tools, e.g., Statemate to check the specification by simulation, or type checkers and theorem provers for Z to check some of the formal validation conditions.

Besides providing guidance for developers and ensuring some application independent quality aspects of the developed product, agendas offer the following advantages:

- *Agendas make software processes explicit, comprehensible, and assessable.* Giving concrete steps to perform an activity and defining the dependencies between the steps make processes explicit. The process becomes comprehensible for third parties because the purpose of the various steps is described informally in the agenda. Thus, agendas may be subject to evaluation.
- *Agendas standardize processes and products of software development.* Agendas structure development processes. The development of an artifact following an agenda always proceeds in a way consistent with the steps of the agenda and their dependencies. Thus, processes supported by agendas are standardized. The same holds for the products: since applying an agenda results in instantiating the templates given in the agenda, all products developed with an agenda have a similar structure.
- *Agendas support maintenance and evolution of the developed artifacts.* Understanding a document developed by another person is less difficult when the document was developed following an agenda than without such information. Each part of the document can be traced back to a step in the agenda, which reveals its purpose. To change the document, the agenda can be “replayed”. The agenda helps focus attention on the parts that actually are subject to change.

- *Agendas are a promising starting point for sophisticated machine support.* They can form the basis of a process-centered software engineering environment (PSEE) [6]. Such a tool would lead its users through the process described by the agenda.

For these reasons, agendas play a central role in the ESPRESS methodology.

**Related Work.** Recently, efforts have been made to support re-use of special kinds of software development knowledge: *Design patterns* [5] have had much success in object-oriented software construction. They represent frequently used ways to combine classes or associate objects to achieve a certain purpose. Furthermore, in the field of software architecture [14], *architectural styles* have been defined that capture frequently used design principles for software systems. In contrast to these, the general concept of an agenda is not specialized to a programming paradigm such as object-orientedness or an activity such as software design, as is the case for design patterns and architectural styles. Apart from the fact that these concepts are more specialized in their application than agendas, the main difference is that design patterns and architectural styles do not describe *processes* but *products*.

Agendas have much in common with approaches to software process modeling [12]. The difference is that software process modeling techniques cover a wider range of activities, e.g., management activities, whereas with agendas we always develop a document, and we do not take roles of developers etc. into account. Agendas concentrate more on technical activities in software engineering. On the other hand, software process modeling does not place so much emphasis on validation issues as agendas do.

Related to our aim to provide methodological support for applying formal techniques is the work of Souquières and Lévy [15]. They support specification acquisition with *development operators* that reduce *tasks* to subtasks. However, development operators do not provide means for validation conditions.

Astesiano and Reggio [1] also emphasize the importance of method when using formal techniques. In the “method pattern” they set up for formal specification, agendas correspond to *guidelines*.

**Future Work.** In ESPRESS, we are currently working on agendas supporting further activities of the general development process as shown in Fig. 1 on page 2. We already have a first version of an agenda for the activity of safety analyses, which is based on common techniques such as FTA (failure-tree analysis) and SHARD (software hazard analysis and resolution design). We are working on an agenda for the verification and validation analyses, which captures the process how to check model properties. Verification is based on deduction techniques being developed in ESPRESS [13, 4], and on an adaption of existing model checking techniques. How the testing process is described by an agenda is a topic of ongoing research. Another important task is to support software design with agendas, where the starting point is a requirements specification as developed in this paper.

Cyclic software components, though important in practice, are indeed a rather simple software architecture. We are currently working on an extension

of our agenda for this architecture to certain kinds of event-triggered software components, which are also studied in the case studies of ESPRESS [3]. We expect to reuse significant parts of the given agenda, in particular from Stage 1, context embedding, and Stage 2, quality requirements.

**Acknowledgments.** Many results and ideas presented in this paper stem from the broader context of the ESPRESS project and the work of its many collaborators in Berlin. We would especially like to thank Mirco Conrad and Eckard Lehmann for their work on the cruise control case study, and Thomas Santen for his comments on a draft of this paper.

## References

1. E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT'97*, LNCS 1214, pages 93–114. Springer-Verlag, 1997.
2. 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.
3. R. Büssow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with statecharts and Z: a case study. this volume, 1997.
4. Robert Büssow and Wolfgang Grieskamp. Combining 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.
5. E. Gamma, R. Helm, R. Johnson, and Vlissides. J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.
6. P. Garg and M. Jazayeri. Process-centered software engineering environments: A grand tour. In A. Fuggetta and A. Wolf, editors, *Software Process*, number 4 in *Trends in Software*, chapter 2, pages 25–52. Wiley, 1996.
7. 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 TSE*, 16 No. 4, April 1990.
8. M. Heisel. *Methodology and Machine Support for the Application of Formal Techniques in Software Engineering*. Habilitation Thesis, TU Berlin, 1997.
9. M. Heisel and C. Sühl. Methodological support for formally specifying safety-critical software. In P. Daniel, editor, *Proceedings 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 295–308. Springer-Verlag London, 1997.
10. Maritta Heisel. Agendas – a concept to guide software development activities. In *Proc. Systems Implementation 2000*, 1998. to appear.
11. C. Heitmeyer, R. Jeffords, and B. Lebow. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
12. K. Huff. Software process modelling. In A. Fuggetta and A. Wolf, editors, *Software Process*, number 4 in *Trends in Software*, chapter 2, pages 1–24. Wiley, 1996.
13. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher-Order Logics*, LNCS 1125. Springer-Verlag, 1996.
14. M. Shaw and D. Garlan. *Software Architecture*. IEEE Computer Society Press, Los Alamitos, 1996.
15. Jeanine Souquières and Nicole Lévy. Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223, 1993.
16. J.M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 1992.
17. J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.