Combining Z and Real-Time CSP for the Development of Safety-Critical Systems

Maritta Heisel, Carsten Sühl
Technische Universität Berlin
FB Informatik – FG Softwaretechnik
Franklinstr. 28-29, Sekr. FR 5-6, D-10587 Berlin
email: {heisel, suehl}@cs.tu-berlin.de fax: 49-30-314-73488

Abstract

We present a method for the specification and development of safety-critical systems. It is based on a combination of the formal languages Z and real-time CSP. Different reference architectures are introduced that represent frequently used designs of safety-critical systems. For these reference architectures, schematic specifications are given that can serve as guidelines for specifiers. Once the specification of the system is developed, it can be validated using a checklist and by demonstrating properties of it. Further steps consist in the refinement of the specification and its implementation that can partially be supported by machine.

Keywords. Safety, formal specification, Z, CSP, program synthesis.

1 Introduction

The application of formal methods in software engineering is undeniably more cost-intensive than working exclusively with informal methods. One important reason to spend extra effort is that the developed product can be expected to be of a better quality than can be achieved without using formal methods.

Although every software-based system potentially benefits from the application of formal methods, their use is particularly advantageous in the development of safety-critical systems. These are systems whose malfunctioning can lead to accidents with loss of property or danger for human lives. The potential damage operators and developers of a safety-critical system have to envisage in case of an accident may be much greater than the additional costs of applying formal methods in system development. It is therefore worthwhile to develop formal methods tailor-made for the development of safety-critical systems. The present paper presents such a method.

Before we describe our ideas in more detail, it should be noted that we do not advocate to *replace* but to *complement* traditional methods for software development by formal ones. As will become apparent in the following, the kind of safety that can be guaranteed by formal methods is relative. Usually, compilers and operating systems of the target machines are not part of the system model (and should not be because the models would become far too complicated). Moreover, relevant properties of an implementation (like speed of execution) can hardly be established by formal methods. Hence, traditional methods like testing are still indispensable.

Most safety-critical systems are *reactive*. This means they do not just perform data transformations, like payroll systems. Instead, they are not intended to terminate, and their behavior depends on stimuli coming from the environment and their internal state which usually is an approximation of the state of the environment. Frequently, they have to fulfill real-time requirements.

From these characteristics, it follows that two aspects are important for the specification of safety-critical systems. First, it must be possible to specify behavior, i.e. what happens in the system in which order, how the system reacts to incoming events, and what signals it sends to the environment under which conditions. The specified behavior must additionally take place sufficiently fast. This is a crucial requirement for the system and thus should be expressed in the specification. Second, complementing the behavior specification, the structure of the system's data state and the operations that change this state must be specified.

Both of these parts are of equal importance, and a specification that ignores one of them would not be satisfactory. On the one hand, process algebras offer appropriate constructs to specify behavior. With some extensions, also real-time requirements can be expressed. On the other hand, model-based specification languages are suitable to specify the data-oriented part of the system. Since they allow the legal states of the system to be explicitely specified, they are to be preferred over algebraic languages in this context. A combination of a process algebra and a model-based specification language yields a suitable language for the specification of software for safety-critical systems.

In our approach, we choose to combine the specification notation Z [Spi92] with the process algebra real-time CSP [Dav93] which adds real-time constructs to CSP [Hoa85]. Both languages are fairly well known and frequently used. Other such combinations, however, (e.g. VDM and CCS) would also be conceivable. Although with the language LOTOS, a combined language already exists, we do not use it because the specification language contained in LOTOS is a simple algebraic language that is less appropriate than Z for the data-oriented part of the specification.

A combination of two different specification languages must be given a common semantics; otherwise, combined specifications cannot be regarded as completely formal. Once this is achieved, we obtain a specification language tailored for the modeling of safety-critical systems.

A mere language, however, does not suffice to improve product quality in practice. A methodology for its application that provides specifiers with guidance how to construct specifications is also indispensable. We provide such a methodology in identifying frequently used designs of safety-critical systems. These designs are expressed as reference architectures, and for each architecture we give schematic specifications that only have to be instantiated for concrete applications.

The validation of a specification is as important as a controlled process for its development. Therefore, our approach also contains guidelines for this purpose. First, general validation criteria can be stated that are independent of concrete applications but only refer to the chosen architecture. Second, we identify two different kinds of properties that are important for safety-critical systems, namely safety-related and liveness properties. Our method encourages specifiers to identify and demonstrate such properties.

The main focus of our work is on the development of the specification and its validation. This is justified because a formal specification is a necessary prerequisite for the usage of formal methods in the development process. Later phases like design and implementation can only be supported by formal methods in presence of a formal specification. But formal

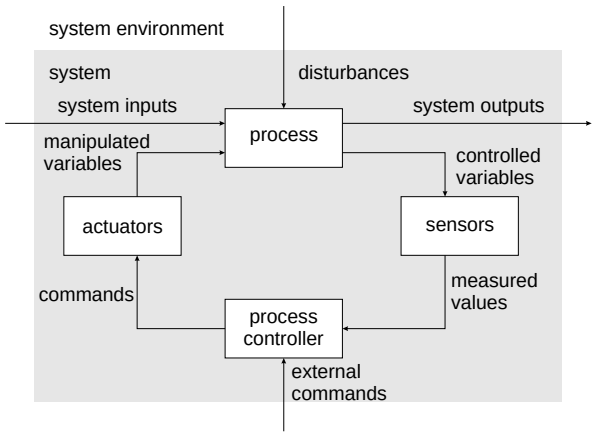

Figure 1: System Model

specifications are not only necessary. Often, it is even sufficient to develop a formal specification and perform the subsequent development steps with traditional methods to obtain a considerable gain in product quality.

Nevertheless, our approach also offers formal support for the later phases of the software development process. A notion of refinement for specifications in the combined language is defined, and it is shown how an existing program synthesis system can be used to synthesize code for the Z part of a combined specification.

In the following, the approach described above is elaborated. Sections 2 and 3 describe the specifics of safety-critical systems and their development. Section 4 presents the specification method that is based on a software model described in Section 4.1. This software model motivates how to combine the languages Z and real-time CSP whose common semantic model is outlined in Section 4.2. Guidelines to develop specifications in the combined language are provided by identifying frequently used architectures of software systems for safety-critical applications and by giving schematic specifications for each of these architectures (Section 4.3). An example illustrating the method is given in Section 5. How specifications of the combined language can be refined and implemented is briefly described in Section 6. A discussion of the approach and of related work concludes the paper.

2 Underlying System Model

The purpose of the systems we want to consider is to control some technical process, where the control component is at least partially realized by software, see Figure 1 and [Lev95]. Such a system consists of four parts: the technical process, the control component, sensors to communicate information about the current state of the technical process to the control component, and actuators that can be used by the control component to influence the behavior of the technical process.

A software-based control component affects certain process variables (*manipulated variables*) by sending commands to actuators. By evaluating the current state of certain process variables which are measured by sensors (*controlled variables*), the control component approximates the current state of the real process in order to verify the effect of the commands sent to the actuators (feedback control) and to determine the further commands to be sent.

The behavior of the technical process does not only depend on internal conditions within the process, e.g. the state of the manipulated variables, but it is also influenced by external disturbances. The basic objective of process control is to achieve the process control function in spite of disturbances from the environment.

Safety can be defined as the property of a system to be free from accidents or losses (cf. [Lev95]). It follows that a software component which is considered in isolation cannot be unsafe because it is not directly able to cause a loss event. Safety is a property of a whole system in the context of its environment rather than a property of a separate system component. A method concerned with *software* development for safety-critical systems must aim at *system* safety and can only be evaluated in this respect.

From these considerations, we can infer the subsystems of a technical process that have to be modeled to achieve system safety:

- *all* parts of the process-control component, i.e. software components, mechanical and electrical components, and interfaces to human operators,
- sensors, determining the projection of the real process state to the internal state of the control component, and
- actuators, which realize the execution of commands given by the control component within the real process.

Another desirable property of software systems is *correctness*. What is the relationship between safety and correctness? The latter is defined as the property of a software component to fulfill the relation between inputs and outputs prescribed in the component specification.

One might consider safety a weaker requirement than correctness. Leveson [Lev86] states "We assume that, by definition, the correct states are safe." However, safety concerns have an influence on what is considered a correct state. For example, incorrect measurements of process variables or the failed realization of given commands by the actuators are normally not relevant in the context of correctness. To achieve system safety, on the other hand, the thorough examination of the above situations is a necessary condition.

This leads to differences in the modeling of a system. If correctness in the usual sense of the word is striven for, the environment in which the system operates, hardware failures, and the credibility of inputs are of no interest. In contrast, to achieve system safety, the environment must explicitly be modeled, too. It is necessary to try to detect hardware failures, and not only the specified *relation* between input and output values must be guaranteed. It must also be checked if the input values can represent a possible situation in the real world, e.g. by consistency checks on different sensors and by redundant arrangements of sensors.

The approaches to achieve correctness on the one hand and safety on the other hand do not differ in a *technical*, but in a *methodological* way: in the end, safety requirements are expressed as functional requirements, and safety is guaranteed by developing software that is correct with respect to the safety requirements.

3 Phases of Software Development for Safety-Critical Systems

Although non-software-based components have to be taken into account in the modeling process, the subject of the development process are exclusively the software-based components of the control component. The development of these software components is performed in a

number of stages that will be applied with repetitions and in an interleaved manner. These stages are: hazard analysis, formal specification, validation of the formal specification, design and program development.

Hazard Analysis. The objective of this stage is to identify hazards, analyze their causes and consequences, design safeguards for their control or elimination, and assess risks.

After identifying the system-level hazards by using a hazard identification approach, the system hazards should be traced back to the interface of the intended software components by applying the technique of fault tree analysis. The subsequent qualitative analysis of the developed fault tree yields an informal description of the safety constraints for the software to be developed.

This activity can hardly be supported by formal methods; hence, we do not cover it in our approach.

Formal Specification. The essential task of this stage is to formally define the safety constraints related to software components and to show that the defined safety constraints are logical consequences of the formally specified properties of the software components.

It is often advocated to write constructive, executable specifications because such specifications can be animated which is an important means for validation. We acknowledge the usefulness of animation. However, executable specifications must usually be more detailed than non-executable ones. Moreover, a constructive specification can introduce implementation biases that exclude efficient algorithms. This leads us not to strive for executability. Instead, we are convinced that the most important properties of a specification – besides being a faithful representation of the requirements – are simplicity and comprehensibility.

Furthermore, a compromise has to be made between the safety and the availability of the system (an absolutely safe airplane is one that never leaves the ground). This means that the specification should also guarantee that the system fulfills its function as long as possible.

Hence, two kinds of properties of the formal specification are of interest: Safety-related properties that contribute to system safety, and liveness properties that guarantee that the system fulfills its function. This phase is described in detail in Section 4.

Validation of the Formal Specification. According to historical experience, a major part of the accidents which were caused by incorrect behavior of software components can be traced back to incomplete or inconsistent requirements specifications. It is therefore very important to validate the specification.

Our approach supports validation by use of checklists and demonstration of properties. Checklists are lists of criteria to check a formal specification developed in compliance with the proposed approach for completeness and consistency. These criteria are heuristic rules for identifying general sources of faults rather than formal completeness criteria. A checklist (consisting of 24 criteria) for our method is presented in [Süh96].

The use of a checklist is complemented by demonstrating safety-related and liveness properties of the specification. Examples are assertions that the system cannot be longer than a certain time in a certain state, that the violation of a safety constraint is noticed within some time bound, that certain conditions exclude each other, or that certain conditions always occur together. Concrete examples of such properties are given in Section 5.

Although our approach does not necessarily lead to executable specifications, it provides means to make a specification executable by refinement, as described in Section 6.1. In this way, also an animation is possible.

Refinement. When the transition from an abstract specification to an executable program is to be supported by formal methods, it is advisable to perform this transition in several steps. First, the specification is made more concrete by *refinement*. Refinement basically consists of eliminating non-determinism, adding detail and transforming abstract data structures into data types available in the target programming language. A definition of refinement for our combined language is given in Section 6.1.

Program Development. When the specification is detailed enough, it forms the basis for program development. In case formal methods are to be used in this step, it is either possible to write a program and afterwards verify that it is correct with respect to the specification. Or the program is synthesized in such a way that it can be guaranteed to be correct. For the Z part of our specifications, programs can be synthesized semi-automatically as described in Section 6.2.

4 Formal Specification

In general, the control component of a technical process refers to a *reactive* system, which is characterized to be mainly event-triggered. It continuously reacts to events occurring within the environment by invoking internal operations and subsequently emitting resulting events into the environment. In accordance with Harel [Har87], we split the specification of a software component into two parts.

1. In the *structural* and *dynamic* part the reactive behavior of the software component is specified, i.e. its reaction to the occurrence of events within the real process (detected by sensors) which is realized by invoking internal operations and giving commands to the actuators. In this part, real-time requirements and the ordering of events are crucial.
2. In the *functional* part the properties and the structure of the possible system states, i.e. data structures as well as system operations applied to these states are specified. System operations are defined by relations between inputs, outputs, and the system states before and after the execution of the respective operation.

The specification languages Z and real-time CSP provide constructs to adequately express both aspects.

4.1 Software Model

To achieve a suitable combination of both parts of the formal specification of a software component formulated in Z and real-time CSP, we propose the software model shown in Figure 2.

1. The innermost component which is expressed in Z specifies the functional aspects, i.e. the structure and the properties of the valid system states as well as the requirements for system operations.
2. Around this innermost component, a CSP process specifies the reactive behavior, i.e. the absorption of values provided by the sensors, the invocation and termination of internal operations, and the transmission of the operation results to the actuators.

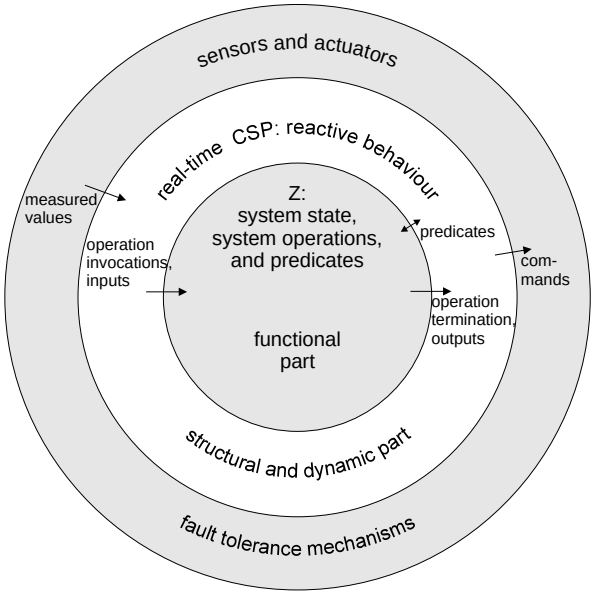

Figure 2: Software Model

3. The outermost component models the required behavior of the sensors and actuators. It offers the possibility to specify fault tolerance mechanisms, e.g. the redundant arrangement of sensors and actuators.

Both the Z specification and the sensors and actuators form the environment of the CSP process.

Two different styles of specifying the reactive behavior in the CSP part can be distinguished. First, a term of the syntax of real-time CSP can be given to model the dynamic behavior in a constructive manner which is amenable to further refinement. Second, predicates can be used to constrain the set of possible behaviors. This is a more abstract way of specification. Both approaches are semantically equivalent and can thus be combined arbitrarily.

Informally, the relation between the elements of the Z part and the CSP part of a formal specification can be explained as follows, see also Figure 3. For each system operation Op specified in the Z part which is intended to be externally available, the CSP part is able to refer to the events $OpInvocation$ and $OpTermination$, whose occurrences represent the invocation of the system operation Op by the software component and its termination, respectively. The two events mark the execution interval of an operation. This makes it possible to specify requirements for the maximal duration in terms of assumptions about the environment which constrain the availability of these events. An example can be found in Section 4.3.1. Alternatively, if the duration of the execution is assessed to be negligible, only one event $OpExecution$ is used to represent the execution of the operation Op.

For each input $in? : Type$ of a system operation Op, there is a communication channel in within the CSP part onto which an input value possibly derived from sensor measurements is written before operation invocation. The alphabet of this channel is identical to the type of the operation input.

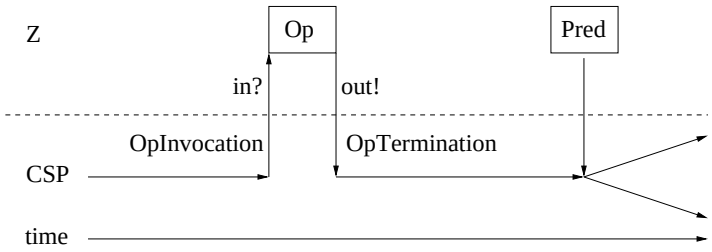

Figure 3: Connection between Z and CSP

Analogously, for each output $out! : Type$ of a system operation, there is a communication channel *out* in the CSP part from which the output value of the operation is read after termination and possibly used to derive commands to the respective actuator.

The dynamic behavior of a software component may depend on the current system state. To achieve this, a process of the CSP part is able to refer to the current system state via predicates which are specified in the Z part by schemas. This link between both parts contributes to a clear separation of the system states from the dynamic aspects.

The connection between the CSP part and the specification of the intended behavior of the sensors and actuators is as follows. The CSP part is linked with every sensor via a communication channel from which the measured values of the respective sensor are read. Analogously, the CSP part is connected with every actuator via a communication channel onto which the commands to the respective actuator are written.

Furthermore, the specification of communication channels in terms of CSP processes makes it feasible to model aspects of a distributed communication, for example the delay of transmission or the redundant arrangement of unreliable communication channels.

4.2 Common Semantic Model

In this section, we outline the formal definition of the semantics associated with a combined specification as explained informally in the previous sections.

The basis of this definition is the semantic function of the timed failures model of real-time CSP [Dav93]. This model associates with each process term a set of *timed failures* which represents possible observations of the process. A timed failure consists of a *timed trace* and a *timed refusal*. A timed trace is a sequence of *timed events*, where each timed event is a pair of an event and the time instant at when it was observed. A timed refusal is a set of timed events. In case the corresponding timed trace has been observed, an event can be refused by the system at a time instant if the corresponding pair is a member of the timed refusal.

$$timed\,failures : Process \rightarrow \mathbb{P}(\text{seq}\ TimedEvents \times \mathbb{P}\ TimedEvents)$$

Analogously, the set of all possible observations of a system specified by a combination of Z and real-time CSP has to be determined. In this context, a third component is of importance, namely the evolution of the system state within the observation interval. Hence an observation for a combined specification is a tuple consisting of a timed trace, a timed refusal, and a so-called *timed state*. A timed state is defined as a function which maps every time instant of the observation interval to the respective system state observed.

The Z part of a specification is characterized by a state schema *State*, an initial state schema *InitState*, a set of external operation schemas $Op1, \ldots, OpN$, and a set of predicates

on the system state $Pred1, \ldots, PredM$. The CSP part of a specification is characterized by a term of real-time CSP and a predicate of the timed failures model. The set *RESTR_RTCSP_PROCESS* contains all process terms of real-time CSP that do not allow subprocesses to perform an event concerning the execution of a system operation (and consequently causing a state change) in parallel with other subprocesses which either perform an operation event or evaluate a predicate on the system state. Furthermore, the set *TF_PREDICATE* contains all predicates of the timed failures model. Thus the signature of our semantic function is as follows.

$$\begin{aligned} &\textit{timed failures states} : SCHEMA \times SCHEMA \times \mathbb{P}\,SCHEMA \times \\ &\quad \mathbb{P}\,SCHEMA \times RESTR\_RTCSP\_PROCESS \times TF\_PREDICATE \nrightarrow \\ &\quad \mathbb{P}((\text{seq}\ TimedEvents \times \mathbb{P}\ TimedEvents) \times (TIME \nrightarrow STATES)) \end{aligned}$$

A possible observation $((s, X), tstate)$ of the behavior of the specified system can be interpreted in the following sense: the timed failure (s, X) consisting of the timed trace s and the timed refusal X is defined by the semantic function *timed failures* as a possible observation of the CSP process, and the timed state *tstate* maps each instant of the observation interval to a system state. This system state must be one of the states that can be reached at the respective time instant, starting from an initial state and proceeding in accordance with the operation events as well as their assigned input and output values which are recorded in the timed trace s up to the considered time instant. The formal definition of the function *timed failures states* can be found in [Süh96].

4.3 Reference Architectures

In practice, there are (at least) two different ways to design safety-critical systems, according to the manner in which activities of the control component take place and which system components trigger these activities. We express these as reference architectures that serve as frameworks supporting the specifier with general structures to be instantiated in the context of the specific application.

Concrete applications need not be "pure" instances of these architectures. When necessary, they can be combined as appropriate.

4.3.1 Centralized Coordination of Passive Sensors

For this architecture it is assumed that all sensors are passive, i.e. they cannot cause activities of the control component, and their measurements are permanently available. There is only one control operation which is executed at time instants uniquely defined by the current system state (e.g. equidistant points of time). Further assumptions are that all actuators are able to perform the possible commands at arbitrary time instants and that the sample rate is high enough to provide all relevant information about changes of the system state.

Functional view. The functional aspects of the control component comprise the structure and the invariant properties of the system state defined by the state schema *SystemState* as well as the functional aspects of the control operation. The control operation is specified by an operation schema *ControlOperation* within the Z part. It is assumed that the controller is always in one of the operational modes $Mode1, \ldots, ModeK$ that are defined with respect to the needs of the technical process. Within distinct modes, which can model different environmental or internal conditions, the behavior of the control component might be

totally different. The behavior within an operational mode $ModeI$ is specified by the internal operation $OpModeI$. This yields the following general schema[1]:

$$Sensors \mathrel{\widehat=} [input1? : IType1;\ \ldots;\ inputN? : ITypeN \mid \ldots]$$

All sensor measurements of the controlled variables of the technical process are introduced as inputs in a separate schema *Sensors*. If necessary, its predicate part should contain the specification of consistency checks concerning the sensor measurements as well as for the definition of redundancy mechanisms, e.g. the arrangement of several identical sensors and the derivation of a unique value from a set of measured values of the same controlled variable.

$$Actuators \mathrel{\widehat=} [output1! : OType1;\ \ldots;\ outputM! : OTypeM \mid \ldots]$$

All commands which are to be sent to the actuators are introduced in the separate schema *Actuators*. Here the derivation of these commands from the current system state is specified.

ControlOperation

$\Delta SystemState$
$Sensors;\ Actuators$

$mode = Mode1 \Rightarrow OpMode1$
$\wedge \ldots \wedge$
$mode = ModeK \Rightarrow OpModeK$

By importing the schemas *Sensors* and *Actuators* the operation has all relevant inputs from the sensors at its disposition. These inputs and the current operational mode determine the successor mode which is specified by the internal operations $OpModeI$. The outputs serve to give commands to the actuators.

At this place we want to outline the application of validation criteria to a formal specification complying with the described architecture. There are two important criteria referring to the conditions of the transitions between different operational modes. The first one requires that in every operational mode there is *at least* one successor mode for an arbitrary combination of sensor inputs. This guarantees that each possible situation is taken care of by the system.

The second validation criterion requires that in every operational mode there is *at most* one successor mode for each combination of sensor inputs. This validation criterion contributes to the determinism of the specification. Although determinism is not a necessary condition for safety, in the most cases it will enhance comprehensibility.

Validation criteria are a means to check a formal specification for consistency and completeness. The other means to validate the specification is to carry out mathematical proofs. The main emphasis of the validation criteria is on the detection of incompleteness, whereas the mathematical proofs concentrate on detecting inconsistencies. So both means are complementary tools.

Dynamic view. The following CSP process *ControlComponent* serves as a template to specify the reactive behavior of the control component as well as its structural connection to other system components. Its behavior is cyclic which is modeled by a recursive process

[1] Readers not familiar with Z are referred to [Spi92]. A summary of real-time CSP is given in Appendix A.

definition. Before invoking the control operation, all associated input values are read from the respective sensor channels ($sensor1, \ldots, sensorN$) in an arbitrary order which is modeled by the use of the parallel operator $\|$. When the control operation has terminated, all output values are written to the respective actuator channels ($actuator1, \ldots, actuatorM$). The parallel process *Wait INTERVAL* delays the process execution so that the next invocation of the control operation will happen exactly *INTERVAL* time units after the current invocation.

$$
\begin{array}{l}
ControlComponent \mathrel{\widehat=} \mu\, X \bullet \\
\quad ((sensor1?valueI1 \rightarrow input1!valueI1 \rightarrow Skip \parallel \ldots \parallel \\
\quad sensorN?valueIN \rightarrow inputN!valueIN \rightarrow Skip); \\
\quad ControlOperationInvocation \rightarrow ControlOperationTermination \rightarrow \\
\quad (output1?valueO1 \rightarrow actuator1!valueO1 \rightarrow Skip \parallel \ldots \parallel \\
\quad outputM?valueOM \rightarrow actuatorM!valueOM \rightarrow Skip) \\
\quad \parallel \\
\quad Wait\ INTERVAL);\ X
\end{array}
$$

In addition to the process term, the predicate *EnvironmentalAssumption* specifies an assumption about the duration of the operation execution. The maximal time distance between the invocation and the termination are *INTERVAL* time units[2]. Moreover, the invocation of the control operation must be possible at any time.

$$
\begin{array}{l}
EnvironmentalAssumption \mathrel{\widehat=} (\forall\, t : [0, \infty) \bullet \\
\quad ControlOperationInvocation\ \mathsf{open}\ t \wedge \\
\quad ControlOperationInvocation\ \mathsf{at}\ t \Rightarrow (\exists\, t' : (t, t + INTERVAL] \bullet \\
\qquad ControlOperationTermination\ \mathsf{open}\ t'))
\end{array}
$$

The above shows (i) that it is possible to give fairly detailed guidelines concerning the shape of formal specifications in a given context, (ii) how the proposed validation criteria contribute to validate a specification, and (iii) that both styles of specifying behavior (as process terms and as predicates) are useful and should be combined as appropriate.

4.3.2 Decentralized Coordination of Active Sensors

For this architecture it is assumed that all sensors are active, i.e. they control a certain variable of the technical process and independently report certain changes of the controlled variable to the control component at arbitrary time instants. Such a report immediately triggers the execution of a handling operation within the control component.

In contrast to the centralized architecture for which one "central" control operation suffices, a control component in the decentralized architecture has to make several "decentral" handling operations available at its external interface.

Functional View. We propose to define in the Z part of the specification for every active sensor exactly one *sensor handling operation*, which defines the reaction of the control component to the received sensor measurement.

Furthermore, it is taken into consideration that actuators may not be able to accept commands by the control component at arbitrary points in time. In general, actuators are technical devices which are exposed to a physical inertia, i.e. they cannot change their state

[2] e open t means that the environment of the process is ready to participate in event e at time t. e at t means that event e happens at time t.

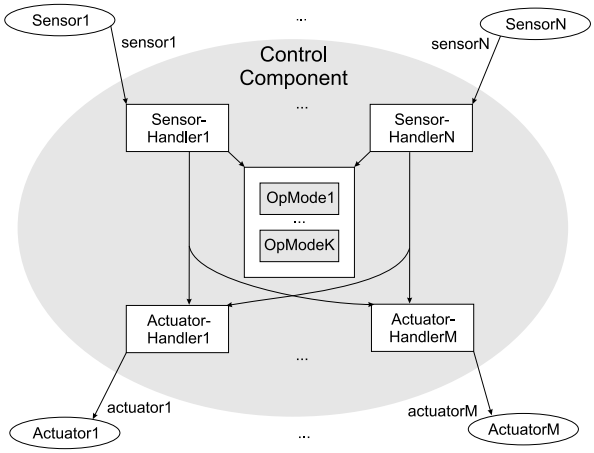

Figure 4: Decentralized Architecture of Active Sensors

arbitrarily fast. On the other hand they may require to get current commands from the control component in regular time distances. Consequently, the transmission of commands to the actuators may be independent of the execution of sensor handling operations, so that the specification of an *actuator handling operation* for every actuator is necessary, specifying the derivation of the command from the current state of the control component.

As in the centralized architecture, the functional behavior of the control component may depend on a set of operational modes. For each operational mode there is an internal operation specifying the general behavior of the control component in this mode independently of special inputs from sensors.

The relationship between the externally available sensor handling operations and the internal operations of the operational modes can be outlined as follows. A sensor handling operation specifies the entire reaction of the control component to incoming sensor values. If necessary, it should specify consistency conditions between sensor measurements and the current state of the control component as well as redundancy mechanisms, e.g. the derivation of a unique value from a set of measured values from redundantly arranged sensors. Depending on the current state of the control component, the result of consistency checks, and the derived sensor value, a sensor handling operation may make use of the definition of the different internal operations of the operational modes.

The actuator handling operations are either directly connected to the sensor handling operations, i.e. every execution leads to a transmission of commands to the actuators, or they are independently executed by the control component. This architecture is illustrated in Figure 4.

Dynamic View. The following process term *Coordination* schematically specifies the reactive behavior of the control component with a decentralized architecture. The version presented here assumes that all sensor handling operations are not time critical, i.e. each execution is represented by the occurrence of only one event. The actuator handling operations are included in the sensor handling operations, i.e. every incoming sensor value leads to the output of commands to the actuators.

$$
\begin{array}{l}
Coordination \mathrel{\widehat{=}} \mu\, X \bullet \\
\quad ((sensor1?valueI1 \rightarrow input1!valueI1) \rightarrow \\
\quad SensorHandler1Execution \rightarrow \\
\qquad ((output_{(1,1)}?valueO_{(1,1)} \rightarrow actuator_{(1,1)}!valueO_{(1,1)} \rightarrow Skip) \\
\quad \| \cdots \| \\
\qquad (output_{(1,M_1)}?valueO_{(1,M_1)} \rightarrow actuator_{(1,M_1)}!valueO_{(1,M_1)} \rightarrow Skip))) \\
\quad ;\ X \\
\Box \\
\quad \cdots \\
\Box \\
\quad ((sensorN?valueIN \rightarrow inputN!valueIN) \rightarrow \\
\quad SensorHandlerNExecution \rightarrow \\
\qquad ((output_{(N,1)}?valueO_{(N,1)} \rightarrow actuator_{(N,1)}!valueO_{(N,1)} \rightarrow Skip) \\
\quad \| \cdots \| \\
\qquad (output_{(N,M_N)}?valueO_{(N,M_N)} \rightarrow actuator_{(N,M_N)}!valueO_{(N,M_N)} \rightarrow Skip))) \\
\quad ;\ X
\end{array}
$$

In the context of the decentralized architecture there is a problem concerning the ability of the sensors to independently trigger actions of the control component. The limited time and space resources of the control component may impose several restrictions with respect to the number of measured values a sensor can send in an interval. On the other hand, it may be necessary to require that the sensors send fresh measurements in regular time distances as the control component has to make its decisions on the basis of the current state of the technical process. These assumptions about maximal and minimal numbers of sensor values in time intervals of a certain length as well as the reaction of the control component to a violation of such assumptions is specified in the dynamic view. They are expressed as predicates of the timed failures model expressing these assumptions about the behavior of the environment. For reasons of space their integration in this architecture is not described in this paper. An example of a specification for this architecture can be found in [HS96].

5 Example

The following case study is a variant of a specification problem used in [MP95]. The software controller of an inert gas release system to be operated from the control room of a plant is to be specified. The task of this system is to detect fire in one of the different machine rooms of the plant and to extinguish a detected fire with the help of inert gas. The architecture of the control component is central, i.e. the sensors are passive always allowing the controller to request the current value of the controlled process variable. The only control operation is executed at equidistant time instants.

5.1 Functional View

We first specify the internal operations of the various operational modes of the controller, together with the necessary data types and constants. Finally, the central control operation is defined in terms of these internal operations.

Data Types. The operational modes of the inert gas system are included in the following data type *MODE* and the possible transitions between them are depicted in Figure 5.

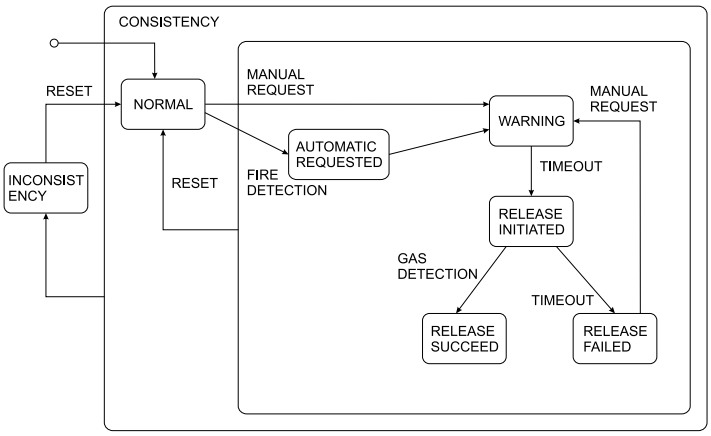

Figure 5: Operational Modes

$$MODE ::= NORMAL \mid AUTOMATIC\_REQUESTED \mid WARNING \mid RELEASE\_INITIATED \mid RELEASE\_FAILED \mid RELEASE\_SUCCEED \mid INCONSISTENCY$$

The remaining data types encompass the sets of values representing possible measurements made by the sensors or possible commands given to the actuators. The sensors and actuators of the system are described in the next paragraph.

$$BANK\_SELECTOR\_STATUS ::= BANK\_A \mid BANK\_B \mid INHIBIT$$

$$BUTTON\_STATUS ::= PRESSED \mid NOT\_PRESSED$$

$$DETECTION\_STATUS ::= DETECTION \mid NO\_DETECTION$$

$$OPEN\_CLOSED ::= OPEN \mid CLOSED$$

$$LIGHT\_STATUS ::= ON \mid OFF \mid FLASHING$$

$$BEEP\_STATUS ::= BEEPING \mid NOT\_BEEPING$$

$$YES\_NO ::= YES \mid NO$$

Sensors and Actuators. To detect the event of a fire in a certain machine room, the software controller of the inert gas system makes use of two redundantly arranged sensors (*fire_detector*1?, *fire_detector*2?) which are able to detect the presence of smoke. The controller only assumes the existence of fire if both sensors report smoke simultaneously. This redundancy mechanism is represented by the derived component *fire_detector* of the schema *SENSORS* given below.

The gas sensor (*gas_detector*) serves to observe if there really is an escape of inert gas into the machine room after an initiation of gas release. Thus, it realizes feedback control. There are two banks of extinguishant, bank A and bank B. By means of a bank selector switch (*bank_selector*?) within the control room, the operator is able to select one of them or to deselect both by choosing the *INHIBIT* position of the switch.

Inside the control room there is an inhibit switch (*inhibit_button*?) which — if being in the inhibit position — prevents that a fire alarm is automatically triggered somewhere on

the whole plant. The operator can by-pass a global inhibit by pressing the request button (*request_button*?). Moreover, the operator can abort the release of gas and reset the inert gas system at any time by pressing the reset button situated in the control room (*reset_button*?).

SENSORS

$$
\begin{array}{l}
InertGasSystem \\
\\
bank\_selector? : BANK\_SELECTOR\_STATUS \\
request\_button? : BUTTON\_STATUS \\
reset\_button? : BUTTON\_STATUS \\
inhibit\_button? : BUTTON\_STATUS \\
fire\_detector1?, fire\_detector2? : DETECTION\_STATUS \\
gas\_detector? : DETECTION\_STATUS \\
\\
fire\_detector : DETECTION\_STATUS \\
consistency : YES\_NO \\
\hline
fire\_detector = DETECTION \Leftrightarrow \\
\quad fire\_detector1? = fire\_detector2? = DETECTION \\
\\
consistency = NO \Leftrightarrow \\
\quad mode \neq RELEASE\_INITIATED \land gas\_detector? = DETECTION
\end{array}
$$

This schema imports the global system state *InertGasSystem* to be defined subsequently.

There is a consistency condition between the current state of the controller, i.e. the current operational mode, and the incoming sensor values. An inconsistency exists if and only if the controller is not in the mode *RELEASE_INITIATED* (and consequently not releasing inert gas) but the gas sensor is reporting the detection of escaping gas. This is represented by the component *consistency*. This condition makes it possible to detect leaks in the gas banks.

The controller guides the escape of inert gas from the two banks by means of two actuators (*release_bank_A*!, *release_bank_B*!). To inform the persons in the machine room that a release of gas will take place soon, that a release of gas currently happens, that a release has taken place recently, or that a gas leak was detected, a warning light (*warning_light*!) can change between the states *ON*, *OFF*, and *FLASHING*. The warning beeper (*warning_beeper*!) serves a similar purpose acoustically. The operator of the inert gas system is informed about the state of the system (*mode*!). All the outputs of the system are collected in the *ACTUATORS* schema.

ACTUATORS

$$
\begin{array}{l}
InertGasSystem' \\
\\
release\_bank\_A!, release\_bank\_B! : OPEN\_CLOSED \\
warning\_light! : LIGHT\_STATUS \\
warning\_beeper! : BEEP\_STATUS \\
mode! : MODE \\
\hline
release\_bank\_A! = release\_bank\_A' \\
release\_bank\_B! = release\_bank\_B' \\
warning\_light! = warning\_light' \\
warning\_beeper! = warning\_beeper' \\
mode! = mode'
\end{array}
$$

Constants. The constant *WARNING_DURATION* represents the duration of an interval after an automatic or manual request during which the persons in the machine room are warned before the release of inert gas actually takes place. The duration of the period during which the system tries to detect escaping gas after the initiation of gas release before assuming a failure is represented by the constant *CHECK_DURATION*. Finally, the length of the time interval between two consecutive executions of the control operation is characterized by the constant *EXECUTION_INTERVAL*.

$$
\begin{array}{|l}
WARNING\_DURATION : \mathbb{N}_1 \\
CHECK\_DURATION : \mathbb{N}_1 \\
EXECUTION\_INTERVAL : \mathbb{N}_1 \\
\hline
WARNING\_DURATION \ \mathsf{mod}\ EXECUTION\_INTERVAL = 0 \\
CHECK\_DURATION \ \mathsf{mod}\ EXECUTION\_INTERVAL = 0
\end{array}
$$

It is required that the warning and check durations are multiples of the time distance between two consecutive executions of the control operation.

Abstract State. The following schema *InertGasSystem* defines the set of abstract states of the software controller. The main component is the state variable *mode* representing the current operational mode. Furthermore, the controller must have at its disposal two timer components which are initialized with the duration of the warning period or the checking period and subsequently decrease their values until reaching zero. These timer components are represented by the state variables *warning_timer* and *check_timer*, respectively. The other state components define the current states of the actuators as assumed by the controller.

InertGasSystem

$$
\begin{array}{|l}
mode : MODE \\
warning\_timer : 0 \,..\, WARNING\_DURATION \\
release\_check\_timer : 0 \,..\, CHECK\_DURATION \\
release\_bank\_A, release\_bank\_B : OPEN\_CLOSED \\
warning\_light : LIGHT\_STATUS \\
warning\_beeper : BEEP\_STATUS \\
\hline
mode \neq RELEASE\_INITIATED \Rightarrow \\
\quad release\_bank\_A = release\_bank\_B = CLOSED \\
mode = WARNING \Leftrightarrow warning\_timer > 0 \\
mode = RELEASE\_INITIATED \Leftrightarrow release\_check\_timer > 0 \Leftrightarrow warning\_light = ON \\
mode \notin \{WARNING, RELEASE\_INITIATED, INCONSISTENCY\} \\
\quad \Leftrightarrow warning\_light = OFF \\
mode = NORMAL \Leftrightarrow warning\_beeper = NOT\_BEEPING
\end{array}
$$

Only in the mode *RELEASE_INITIATED* a release of gas from bank A or bank B is possible if the selector switch is in the corresponding position. The warning timer is only set in the mode *WARNING* (i.e. has a strictly positive value), and the release check timer is only set in the mode *RELEASE_INITIATED*. The warning light is *ON* when inert gas is released, and it is flashing in the warning period before the gas release and in the *INCONSISTENCY* mode. The warning beeper is always beeping outside the *NORMAL* mode.

When starting the software controller of the inert gas release system the current operational mode is *NORMAL*, since at this moment there may be no fire detection by the sensors and no manual request by the operator.

InertGasSystemINIT

$InertGasSystem'$

$mode' = NORMAL$

Operations. Under normal environmental conditions, i.e. there has been no fire detection by the sensors and no manual request by the operator, the controller is in the mode *NORMAL*. No inert gas is released and no visual or auditory signal is given.

OpNormal

$\Delta InertGasSystem$

$SENSORS;\ ACTUATORS$

$mode = NORMAL$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad (request\_button? = PRESSED \Rightarrow mode' = WARNING) \wedge$
$\quad\quad (request\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad\quad (fire\_detector = DETECTION \Rightarrow$
$\quad\quad\quad\quad mode' = AUTOMATIC\_REQUESTED) \wedge$
$\quad\quad\quad (fire\_detector = NO\_DETECTION \Rightarrow mode' = NORMAL))))$

If both redundantly arranged sensors report the detection of smoke, the controller changes to the mode *AUTOMATIC_REQUESTED*. If the request button is pressed in the mode *NORMAL* there is a transition into the mode *WARNING*. If the reset button is pressed, a transition from any mode to the mode *NORMAL* is the consequence.

In the case of an automatic request, the *WARNING* can only be entered if the global inhibit switch is not set.

OpAutomaticReq

$\Delta InertGasSystem$

$SENSORS;\ ACTUATORS$

$mode = AUTOMATIC\_REQUESTED$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad (inhibit\_button? = PRESSED \Rightarrow mode' = AUTOMATIC\_REQUESTED) \wedge$
$\quad\quad (inhibit\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad\quad mode' = WARNING \wedge warning\_timer' = WARNING\_DURATION)))$

In the mode *WARNING* which lasts exactly *WARNING_DURATION* time units, the warning light is flashing to inform the persons in the machine room about the following release of inert gas to give them the possibility to leave the danger area. When this warning period is elapsed there is a transition into the mode *RELEASE_INITIATED*. At each execution of the control operation in the mode *WARNING*, the warning timer either has to be reduced

by *EXECUTION_INTERVAL* or, if the controller leaves the *WARNING* mode, has to be set to zero to fulfill the state invariant.

OpWarning

$\Delta InertGasSystem$
$SENSORS;\ ACTUATORS$

$mode = WARNING$

$(warning\_timer' = warning\_timer - EXECUTION\_INTERVAL$
$\vee\ warning\_timer' = 0)$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad (warning\_timer - EXECUTION\_INTERVAL > 0 \Rightarrow mode' = WARNING) \wedge$
$\quad\quad (warning\_timer - EXECUTION\_INTERVAL = 0 \Rightarrow$
$\quad\quad\quad mode' = RELEASE\_INITIATED \wedge$
$\quad\quad\quad (release\_bank\_A' = OPEN \Leftrightarrow bank\_selector? = BANK\_A) \wedge$
$\quad\quad\quad (release\_bank\_B' = OPEN \Leftrightarrow bank\_selector? = BANK\_B) \wedge$
$\quad\quad\quad release\_check\_timer' = CHECK\_DURATION)))$

In the *RELEASE_INITIATED* mode, inert gas is released either from bank A or bank B, or no gas is released if the bank selector switch is in the *INHIBIT* position. The alarm light is *ON* to indicate the potential danger. During a period of *CHECK_DURATION* time units it is tested if inert gas is indeed escaping into the machine room. The detection of escaping gas by the respective sensor will cause a transition into the mode *RELEASE_SUCCEED*. If there is no gas detection within this period, a change into the mode *RELEASE_FAILED* results. At each execution of the control operation in the mode *RELEASE_INITIATED* the check timer either has to be reduced by *EXECUTION_INTERVAL* or, if the controller leaves the *RELEASE_INITIATED* mode, has to be set to zero to fulfil the state invariant.

OpReleaseInitiated

$\Delta InertGasSystem$
$SENSORS;\ ACTUATORS$

$mode = RELEASE\_INITIATED$

$(release\_check\_timer' = release\_check\_timer - EXECUTION\_INTERVAL$
$\vee\ release\_check\_timer' = 0)$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad (gas\_detector? = DETECTION \Rightarrow mode' = RELEASE\_SUCCEED) \wedge$
$\quad\quad (gas\_detector? = NO\_DETECTION \Rightarrow$
$\quad\quad\quad (release\_check\_timer - EXECUTION\_INTERVAL > 0 \Rightarrow$
$\quad\quad\quad\quad mode' = RELEASE\_INITIATED \wedge$
$\quad\quad\quad\quad release\_bank\_A' = release\_bank\_A \wedge$
$\quad\quad\quad\quad release\_bank\_B' = release\_bank\_B) \wedge$
$\quad\quad\quad (release\_check\_timer - EXECUTION\_INTERVAL = 0 \Rightarrow$
$\quad\quad\quad\quad mode' = RELEASE\_FAILED))))$

Being in the mode *RELEASE_FAILED* indicates that either the chosen bank is empty or defect or that the bank selector switch is in the *INHIBIT* position. Therefore the operator must have the possibility to change the selector position and to repeat the process of gas release. This is done by pressing the request button causing a transition into the mode *WARNING*.

OpReleaseFailed

$\Delta InertGasSystem$
$SENSORS;\ ACTUATORS$

$mode = RELEASE\_FAILED$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow$
$\quad\quad (request\_button? = PRESSED \Rightarrow mode' = WARNING) \wedge$
$\quad\quad (request\_button? = NOT\_PRESSED \Rightarrow mode' = RELEASE\_FAILED)))$

OpReleaseSucceed

$\Delta InertGasSystem$
$SENSORS;\ ACTUATORS$

$mode = RELEASE\_SUCCEED$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow mode' = RELEASE\_SUCCEED))$

If an inconsistency is noticed by the controller in an arbitrary mode it immediately changes to the mode *INCONSISTENCY*. No inert gas is released in this mode and the flashing warning light and warning beep warn the persons in the machine room.

OpInconsistency

$\Delta InertGasSystem$
$SENSORS;\ ACTUATORS$

$mode = INCONSISTENCY$

$(consistency = NO \Rightarrow mode' = INCONSISTENCY)$
$(consistency = YES \Rightarrow$
$\quad (reset\_button? = PRESSED \Rightarrow mode' = NORMAL) \wedge$
$\quad (reset\_button? = NOT\_PRESSED \Rightarrow mode' = INCONSISTENCY))$

The central control operation is executed by the software controller at equidistant points in time. According to the current operational mode, the corresponding internal operation of the mode is executed. The following schema *ControlOperation* is very straightforward only connecting the different modes to the corresponding internal operations.

ControlOperation

$\Delta InertGasSystem$
$SENSORS;\ ACTUATORS$

$mode = NORMAL \Rightarrow OpNormal$
$mode = AUTOMATIC\_REQUESTED \Rightarrow OpAutomaticReq$
$mode = WARNING \Rightarrow OpWarning$
$mode = RELEASE\_INITIATED \Rightarrow OpReleaseInitiated$
$mode = RELEASE\_FAILED \Rightarrow OpReleaseFailed$
$mode = RELEASE\_SUCCEED \Rightarrow OpReleaseSucceed$
$mode = INCONSISTENCY \Rightarrow OpInconsistency$

5.2 Dynamic View

The dynamic behavior of the controller of the inert gas release system is defined by the real-time CSP process *ControlSystem*. First, the controller is initialized by occurrence of the event *InertGasSystemINITExecution*. The behavior after the initialization is specified by the process $ControlSystem_{READY}$.

$$ControlSystem \mathrel{\widehat{=}} InertGasSystemINITExecution \rightarrow ControlSystem_{READY}$$

The architecture of the software controller is in accordance with the centralized control of passive sensors. Thus, the only control operation is executed in equidistant time points. Before executing the operation, the current measurements of all sensors are read from the corresponding communication channels which is summarized in the process *SensorInputs*. After the operation execution the resulting commands to the actuators are written to the corresponding communication channels which is represented by the process *ActuatorOutputs*. Defining *SensorInputs* and *ActuatorOutputs* as separate processes results in a slightly different syntactic form of the control process as shown in Section 4.3.1.

$$\begin{aligned}
&ControlSystem_{READY} \mathrel{\widehat{=}} \mu\, X \bullet \\
&\quad (SensorInputs; \\
&\quad (ControlOperationExecution \rightarrow Skip); \\
&\quad ActuatorOutputs \\
&\quad \| \\
&\quad Wait\ EXECUTION_INTERVAL);\ X
\end{aligned}$$

The process *SensorInputs* specifies the reading of sensor values before the execution of the control operation. All sensor values are read in parallel from the corresponding communication channels. These values are subsequently written to the channels having the identical names as the inputs of the operation schema.

$$\begin{aligned}
&SensorInputs \mathrel{\widehat{=}} \\
&\quad bank_selector_sensor?bs_status \rightarrow bank_selector!bs_status \rightarrow Skip \\
&\| \\
&\quad request_button_sensor?rq_status \rightarrow request_button!rq_status \rightarrow Skip \\
&\| \\
&\quad reset_button_sensor?rs_status \rightarrow reset_button!rs_status \rightarrow Skip \\
&\| \\
&\quad inhibit_button_sensor?ib_status \rightarrow inhibit_button!ib_status \rightarrow Skip \\
&\| \\
&\quad fire_detector1_sensor?fd1_status \rightarrow fire_detector1!fd1_status \rightarrow Skip
\end{aligned}$$

$$
\begin{array}{l}
\quad \| \\
\qquad fire\_detector2\_sensor?fd2\_status \rightarrow fire\_detector2!fd2\_status \rightarrow Skip \\
\quad \| \\
\qquad gas\_detector\_sensor?gd\_status \rightarrow gas\_detector!gd\_status \rightarrow Skip
\end{array}
$$

The process *ActuatorOutputs* is defined analogously.

$$
\begin{array}{l}
ActuatorOutputs \mathrel{\widehat{=}} \\
\qquad release\_bank\_A?rbA\_status \rightarrow release\_bank\_A\_actuator!rbA\_status \rightarrow Skip \\
\quad \| \\
\qquad release\_bank\_B?rbB\_status \rightarrow release\_bank\_B\_actuator!rbB\_status \rightarrow Skip \\
\quad \| \\
\qquad warning\_light?wl\_status \rightarrow warning\_light\_actuator!wl\_status \rightarrow Skip \\
\quad \| \\
\qquad warning\_beeper?wb\_status \rightarrow warning\_beeper\_actuator!wb\_status \rightarrow Skip \\
\quad \| \\
\qquad mode?m\_status \rightarrow mode\_output!m\_status \rightarrow Skip
\end{array}
$$

The predicate *EnvironmentalAssumption* defines the assumptions concerning the environment. It is supposed that the control operation and the initialization operation may be executed in arbitrary time instants by the controller. It is also assumed that all sensors are always able to send a unique measured value to the controller. Analogously, every actuator must always be able to receive an arbitrary command from the controller.

$$
\begin{array}{l}
EnvironmentalAssumption \mathrel{\widehat{=}} (\forall\, t : [0, \infty) \bullet \\
\qquad ControlOperationExecution \ \mathsf{open}\ t \wedge \\
\qquad InertGasSystemINITExecution \ \mathsf{open}\ t \wedge \\
\qquad (\exists_1\, value : BANK\_SELECTOR\_STATUS \bullet (bank\_selector\_sensor.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\exists_1\, value : BUTTON\_STATUS \bullet (request\_button\_sensor.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\exists_1\, value : BUTTON\_STATUS \bullet (reset\_button\_sensor.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\exists_1\, value : BUTTON\_STATUS \bullet (inhibit\_button\_sensor.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\exists_1\, value : DETECTION\_STATUS \bullet (fire\_detector1\_sensor.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\exists_1\, value : DETECTION\_STATUS \bullet (fire\_detector2\_sensor.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\forall\, value : OPEN\_CLOSED \bullet (release\_bank\_A\_actuator.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\forall\, value : OPEN\_CLOSED \bullet (release\_bank\_B\_actuator.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\forall\, value : LIGHT\_STATUS \bullet (warning\_light\_actuator.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\forall\, value : BEEP\_STATUS \bullet (warning\_beeper\_actuator.value) \ \mathsf{open}\ t) \wedge \\
\qquad (\forall\, value : MODE \bullet (mode\_actuator.value) \ \mathsf{open}\ t))
\end{array}
$$

This example shows that – once a suitable architecture and the necessary operating modes are chosen – the specification can be set up in a fairly routine way. Other case studies performed by the authors confirm this observation.

5.3 Validation of the Specification

The validation consists of three parts. First, the applicable criteria of our checklist [Süh96] are checked. Second, important safety-related properties exhibited by the specification are made explicit. Third, liveness properties following from the specification are stated. We do not perform any mathematical proofs because the claimed properties follow immediately from the specification.

General Criteria. The following properties are of a general nature. They are instances of validation criteria, two examples of which were already given in Section 4.3.1.

1. Feedback control is realized by the *gas_detector* sensor. For the remaining actuators (warning light and beeper) feedback control is not feasible.
2. For each operating mode and each combination of sensor values, there is exactly one successor mode. This means that the system is deterministic and that it treats all possible situations.
3. There are no redundant or unreachable operational modes.

Safety-Related Properties. The specification guarantees the following safety-related properties:

4. Gas leaks are detected and result in an alarm.
5. A gas release can only take place if both of the two smoke sensors detect smoke.
6. If a fire is detected, the persons in the danger area have *WARNING_DURATION* time units to be evacuated before gas is released.
7. If a fire is detected but the release of gas is not successful, this can be noticed by the operator after *CHECK_DURATION* time units.

Liveness Properties. The specification guarantees the following liveness properties:

8. After an unsuccessful gas release, the operator can change the bank selector switch and manually try to release gas.
9. The system can be brought back to normal operation at any time by pressing the reset button.
10. At every time instant the operator is able to by-pass a global inhibit (inhibit button is set) for a certain machine room by pressing the corresponding request button for this room.

To show the properties $1 - 5$ and $8 - 10$, it suffices to consider the Z part of the specification. To show the properties 6 and 7, however, both parts of the specification have to be taken into account.

6 Refinement and Implementation

In the following, we describe how the transition from an abstract requirements specification to an executable program can be performed using formal methods. In a first stage, the specification is refined, where the notion of refinement for specifications in the combined language is based on the notions of refinement as they are defined for Z and real-time CSP. For program development, a program synthesis system developed by the first author can be used to develop a procedure for each system operation defined in Z. At the current stage, the CSP part of a specification has to be hand-coded.

6.1 Refinement

To make stepwise refinement possible for our combined language, we have to define what it means that a specification of a reactive system consisting of a Z part as well as a real-time CSP part is refined by another such combination. To this end, we can make use of the existing refinement notions of Z and real-time CSP.

The essential idea of refinement in Z is that abstract data structures are transformed into more concrete data structures. The relations between abstract and concrete data types have to be formally defined by a so-called *abstraction* relation. For every operation on the abstract system state, a corresponding operation on the concrete system state has to be defined.

The concrete operation must satisfy two conditions: First, if an abstract operation is applicable to an abstract state (i.e. its precondition is fulfilled) then the corresponding concrete operation must be applicable to all concrete states that are related to the abstract state. Second, if the execution of a concrete operation can result in a certain concrete state then there must exist an abstract state which is a possible result of the execution of the corresponding abstract operation and is related to the concrete state.

In real-time CSP every semantic model maps a term of the process syntax to a set of possible observations as described in Section 4.2. A process term is refined by another process term if each possible behavior of the latter is also a possible behavior of the former.

To refine a combined specification, either the Z part or the real-time CSP part is refined separately. For the Z part, however, the notion of refinement must be strengthened. In the definition of refinement in Z as described above, the refining operation can have a weaker precondition than the refined operation, i.e. it can be applicable to a system state to which the latter is not applicable.

If this were admitted, the refining specification would include such behaviors as possible observations that result from the application of the concrete operation to system states where the abstract operation is not applicable. These behaviors would not be observable in the refined specification. To avoid this violation to the notion of refinement, the definition of operation refinement is adapted in the sense that the precondition of the refining operation must be equivalent to the precondition of the refined operation. With this adaptation every isolated refinement of the Z or the real-time CSP part is a refinement of the combined specification.

6.2 Program Synthesis

We first present the synthesis system that is used. Then we describe the conversion of Z schemas into the input format of the system. Finally, we sketch the synthesis of a procedure implementing the operation *OpReleaseSucceed* defined in Section 5.

6.2.1 The Synthesis System

The first author's synthesis system IOSS (Integrated Open Synthesis System) [HSZ95] supports the development of imperative programs using so-called *strategies*, [Hei94]. Strategies describe possible steps during the synthesis process. Their purpose is to find a suitable solution to some *programming problem*. A strategy works by problem reduction. For a given problem, it determines a number of subproblems. From their solutions, it produces a solution to the initial problem. Finally, it checks whether that solution is acceptable. The solutions to subproblems are also obtained by applications of strategies. In general, the subproblems

produced by a strategy are not independent of each other or of the solutions to other subproblems. This restricts the order in which the various subproblems can be set up and solved. A strategy describes how exactly the subproblems are constructed, how the final solution is assembled, and how to check whether this solution is acceptable.

Problems are specifications of programs, expressed as pre- and postconditions that are formulas in first-order predicate logic. To aid focusing on the relevant parts of the task, the postcondition is divided into two parts, *invariant* and *goal*. In addition to these it has to be specified which variables may be changed by the program (result variables), which ones may only be read (input variables), and which variables must not occur in the program (state variables). The latter are used to store the value of variables before execution of the program for reference of this value in its postcondition.

Solutions are programs in an imperative Pascal-like language. Additional components are additional pre- and postconditions, respectively. If the former is not equivalent to *true*, the developed program can only be guaranteed to work if not only the originally specified, but also the additional precondition holds. The additional postcondition gives information about the behavior of the program, i.e. it says *how* the goal is achieved by the program.

A solution is *acceptable* if and only if the program is totally correct with respect to both the original and the additional the pre- and postconditions, does not contain state variables, and does not change input variables. For each developed program a formal proof in dynamic logic [Gol82] is constructed. This is a logic designed to prove properties of imperative programs. The proofs are represented as tree structures that can be inspected at any time during development.

The strategy base of IOSS contains formalized development knowledge in form of strategy modules. A number of interactive, semi-automatic and fully automatic strategies have been implemented. For a more complete description of IOSS, the reader is referred to [HSZ95].

6.2.2 Conversion of Z Schemas into IOSS Problems

The combination of Z and IOSS can be achieved easily: since both formalisms allow for states and have concepts to deal with changing values of variables, Z specifications can mechanically be translated into IOSS programming problems. The translation mechanism as well as the synthesis process resembles the approach of the refinement calculus [Woo91].

Four kinds of variables occurring in a Z schema have to be considered (not to be confused with the variable classification of IOSS problems): *input variables* are the ones decorated with "?". *Output variables* are the variables decorated with "!". *State variables* are the variables of the global state schema. All other variables are *auxiliary variables*. With this classification, the translation of a Z schema into an IOSS programming problem proceeds as follows:

- Each input variable of the Z schema becomes an input variable of the corresponding problem.
- Each output variable of the Z schema becomes a result variable of the problem.
- Each variable x of the Z state schema becomes an input variable if the schema predicate entails $x = x'$.
- Otherwise x becomes a result variable, and a new state variable x_0 is generated for x if x occurs in the schema predicate.
- Each auxiliary variable becomes a result variable.

- The precondition of the IOSS problem is the precondition of the Z schema plus an equation $x = x_0$ for each introduced state variable x_0.
- The invariant of the IOSS problem is the invariant of the Z schema defining the system state.
- The goal of the IOSS problem consists of those conjuncts of the schema predicate that depend on result variables of the IOSS problem, where dashed state variables of the schema have to be replaced by plain variables and plain state variables of the schema have to be replaced by their corresponding state variables of the IOSS problem. Auxiliary variables remain unchanged.

6.2.3 Synthesis of a Procedure for *OpReleaseSucceed*

As an example, we consider the implementation of the internal operation *OpReleaseSucceed*. The programming language of IOSS allows for enumeration types, so that a data refinement is not necessary.

In the mode *RELEASE_SUCCEED*, both timers are zero, the warning light is off, but the beeper is on. The only possible successor modes are *RELEASE_SUCCEED*, *NORMAL* and *INCONSISTENCY*. Hence, only the variables *mode*, *warning_light* and *warning_beeper* of the state schema *InertGasSystem* can change their values. According to the above translation rules, we obtain the following programming problem:

input variables:	$bank\_selector?, \ldots, gas\_detector?,$
	$warning\_timer, release\_check\_timer, release\_bank\_A, release\_bank\_B$
result variables:	$release\_bank\_A!, \ldots, mode!,$
	$mode, warning\_light, warning\_beeper,$
	$fire\_detector, consistency$
state variables:	$mode_0, warning\_light_0, warning\_beeper_0$
precondition:	$mode = mode_0 \wedge warning\_light = warning\_light_0$
	$\wedge\ warning\_beeper = warning\_beeper_0 \wedge mode = RELEASE\_SUCCEED$
invariant:	see *InertGasSystem*
goal:	$(fire\_detector = DETECTION \Leftrightarrow$
	$(fire\_detector1? = DETECTION \wedge fire\_detector2? = DETECTION))$
	$\wedge\ (consistency = NO \Leftrightarrow$
	$(mode_0 \neq RELEASE\_INITIATED \wedge gas\_detector? = DETECTION))$
	$\wedge\ (consistency = NO \Rightarrow mode = INCONSISTENCY)$
	$\wedge\ (consistency = YES \Rightarrow$
	$((reset\_button? = PRESSED \Rightarrow mode = NORMAL)$
	$\wedge\ (reset\_button? = NOT\_PRESSED \Rightarrow mode = RELEASE\_SUCCEED))$
	$\wedge \ldots$see *ACTUATORS*

The goal is a conjunction with 5 top-level conjuncts. We observe that the new values of the result variables can be computed one after another. In this situation, the *disjoint goal* strategy can be applied. This strategy is based on the assumption that a conjunctive goal can be achieved by a compound statement, each part of the compound establishing one conjunct. It can be applied if the goal can be divided into two independent subgoals, i.e. the result variables that need to be changed to achieve one subgoal are disjoint from the result variables that need to be changed to achieve the other one.

For the above problem, we can apply the disjoint goal strategy three times, yielding a program of the form p_1; p_2; p_3; p_4. The statement p_1 determines the new value of *fire_detector*. The new value of *consistency* is set in p_2, and the new values for *mode*, *warning_light* and *warning_beeper* are determined in p_3. Finally, p_4 serves to set the outputs for the actuators.

The program p_4 can be generated automatically using the *automatic assignment* strategy because its goal consists of a conjunction of equations. For the other subprograms, the second strategy to be applied is the *strengthening* strategy. It serves to replace a goal by a stronger (or equivalent) one and is frequently used to incorporate knowledge about the used data types into the synthesis process. In our example, it has to be used to replace equivalences by conjunctions of implications and to replace formulas like $consistency = YES$ and $reset\_button? = NOT\_PRESSED$ by $\neg\ (consistency = NO)$ and $\neg\ (reset\_button? = PRESSED)$, respectively. Moreover, it is necessary to make the implicit predicates of the state schema explicit. For instance, $mode = NORMAL$ has to be strengthened to $mode = NORMAL \wedge warning\_light = OFF \wedge warning\_beeper = NOT\_BEEPING$; similarly for the mode *INCONSISTENCY*.

The third strategy to be applied is called *disjunctive conditional*. It generates a conditional and can be used if the goal is of disjunctive form or equivalent to a disjunction. Each branch of the conditional will establish one disjunct of the goal. In our example, the goal for the variable *consistency* is equivalent to the disjunction $(consistency = NO \wedge mode_0 \neq RELEASE\_INITIATED \wedge gas\_detector? = DETECTION) \vee (consistency = YES \wedge \neg\ (mode_0 \neq RELEASE\_INITIATED \wedge gas\_detector? = DETECTION))$.

The disjunctive conditional strategy can automatically propose a test for the conditional, those parts of the goal consisting solely of variables that cannot be changed by the program and hence cannot be enforced but only be tested. In the above case, the test is $mode \neq RELEASE\_INITIATED \wedge gas\_detector? = DETECTION$ (for inclusion in the program, the state variables are automatically replaced by the corresponding result variables). Since the conclusions of all implications are equations, the *automatic assignment* strategy can automatically generate assignments establishing the equations. The resulting program is shown in Figure 6 where the inputs and outputs of the schema are modeled as parameters, and the other variables are modeled as global variables.

The advantage of using a synthesis system instead of hand-coding this simple procedure is that a correctness proof is generated during the synthesis process.

7 Discussion

We first contrast our work to that of others in the field and then summarize its merits and limitations.

7.1 Related Work

The use of model-based languages like Z or VDM [Jon90] in the area of system safety is not uncommon. Several case studies have been performed using VDM, e.g. the British government regulations for storing explosives [MS93], a railway interlocking system [Han94], and a water-level monitoring system [Wil94]. Mukherjee's and Stavridou's as well as Hansen's work, however, place the focus on the adequate modeling of safety requirements, independently of the fact if software is employed or not. Consequently, they do not discuss issues specific to the construction of safety-critical software.

```
proc  op_release_succeed(bank_selector? : BANK_SELECTOR_STATUS; ...  ;
      gas_detector? : DETECTION_STATUS; ... release_bank_A! : OPEN_CLOSED; ... ;
      mode! : MODE)
do    if fire_detector1?  = DETECTION and fire_detector2?  = DETECTION
      then fire_detector := DETECTION
      else  fire_detector := NO_DETECTION
      fi;
      if mode <> RELEASE_INITIATED and gas_detector?  = DETECTION
      then consistency := NO
      else  consistency := YES
      fi;
      if consistency = NO
      then mode := INCONSISTENCY;
           warning_light := FLASHING;
           warning_beeper := BEEPING
      else  if reset_button?  = PRESSED
           then mode := NORMAL;
                warning_light := OFF;
                warning_beeper := NOT_BEEPING
           fi
      fi;
      release_bank_A! := release_bank_A;
      ...
      mode!  := mode
end
```

Figure 6: Program Synthesized for *OpReleaseSucceed*

Jacky [Jac95] uses Z to define a framework for safety-critical systems that emphasizes safety interlocking. McDermid and Pierce [MP95] define a graphical notation based on a variant of statecharts [Har87] that is translated into Z for the purpose of mechanical validation. This notation is used to specify and develop software for programmable logic controllers. Halang and Krämer [HK94] also focus on programmable logic controllers. They present a development process, from the formalization of requirements to the testing of the constructed program. As formalisms they use the specification language Obj and the Hoare calculus, where their choice is motivated by the tool support available. Both of these formalisms are weaker than the ones we chose. Obj only allows to state conditional equations, and the Hoare calculus is a proper subset of dynamic logic. Heisel [Hei96] describes several phases in the development of safety-critical software where Z is used in the specification phase.

The work presented here is distinguished from these approaches in that it is intended to be used for systems where the exclusive use of model-based or algebraic specification languages does not lead to satisfactory results. The expressive power of these languages does not suffice to specify the behavior of sophisticated real-time systems adequately. Other researchers share our goal to provide more powerful constructs to express behavioral and real-time requirements.

Ravn et al. [RRH93] use the duration calculus to express functional requirements and safety constraints. The duration calculus is a specialized formalism designed to express requirements on the duration of states. These durations are expressed as integrals. In contrast, our approach uses less specialized formalisms that are more easily accessible and more widely used. Weber [Web96] combines Z and statecharts for purposes similar to ours. Since state-

charts are a semi-formal specification technique, the resulting specification is not completely formal. Using a formal language like CSP, however, yields completely formal combined specifications, as shown in Section 4.2.

Like our work, Moser's and Melliar-Smith's approach to the formal verification of safety-critical systems, [MMS90], comprises the specification, design and implementation phases. They use a reliability model for the processors that execute the program. This enables them to take computer failures into account, an aspect not covered by this work. On the other hand, their approach does not cover the validation of the top-level specification, an issue that is of much importance for us.

7.2 Assessment of Our Approach

Limitations. The approach outlined above concentrates on the software aspects of safety-critical systems. Nothing can be guaranteed about the hardware. For instance, our method does not take processor failures into account. This limitation cannot be overcome by means concerning the software alone. Instead, fault tolerance methods like redundancy have to be applied.

Since we can only guarantee that the states before and after execution of an operation are safe, the execution must be sufficiently fast, because in the intermediate states that occur during execution, safety cannot be guaranteed. It is up to the system designers and implementors to judge if this is the case.

Enhancing the Applicability of the Approach. In contrast to hardware or power failure which are beyond our capabilities, the problem that safety cannot be guaranteed in intermediate states can be treated under the condition that sequences of assignments are considered as sufficiently fast. In this case, we can require a "safety invariant" to hold before and after each sequence of assignments. Then the system can be in an unsafe state only for the time that is needed to execute the longest assignment sequence occurring in the implementation. With little effort, IOSS can be extended to deal with such safety invariants.

For relatively small systems, a complete formal treatment certainly can be recommended because the control software is relatively simple. The cost for a formal safety proof would be much less than potential damages. For larger systems, however, a complete formal treatment might not be feasible. In this case, our approach can be applied nevertheless. It is possible to formalize and prove only selected properties of the system and treat the other requirements with traditional techniques (*partial verification*, [Lev91]). When this approach is taken, still all of the software modules have to be considered. To reduce cost further, one might exclude those parts of the software from the verification process that can be guaranteed to be of no importance for safety. Usually, it will be the specifier's responsibility to decide which parts of the software are safety-critical and which are not.

Summary. With the work presented here, we have provided an elaborate methodology for the formal specification of software for safety-critical applications:

- The system model underlying most of these applications is taken into account by explicitly referring to it in the methodology. It provides a suitable structuring and nomenclature to model safety-critical systems.
- Two formal languages are combined according to the needs arising in the development of safety-critical systems. Each of the languages in isolation would not be satisfactory; in

combination, however, they provide adequate constructs for the specification of safety-critical software components. Both languages are well-established.

- The combined language is given a common semantics, making combined specifications completely formal and providing a basis for formal proof.
- A software model for the combined use of the two languages is defined, yielding a general framework for the modeling and specification of control components for safety-critical systems.
- This model is further refined into two reference architectures that capture frequent designs of safety-critical systems. These architectures can be instantiated for concrete systems, thus providing detailed guidance for specifiers.
- Not only for the development of the specification but also for its validation, detailed guidance is given. Besides the use of a checklist that is independent of concrete applications, it is proposed to demonstrate specific safety-related as well as liveness properties that necessarily are application-dependent.
- Not only the specification but also the later phases in software development are supported: a notion of refinement for combined specifications is defined, and a translation of the Z part of a specification into the input format of an existing program synthesis system is provided.
- The feasibility of the approach was illustrated by means of an example.

In the future, we intend to develop a calculus that allows one to perform formal proofs on and refinements of combined specifications and to implement this calculus in order to support the application of our approach by machine.

Acknowledgment. Thanks to Thomas Santen whose comments helped to improve the presentation.

References

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

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* LNCS 130. Springer-Verlag, 1982.

[Han94] Kirsten Mark Hansen. Modelling railway interlocking systems. Available via ftp from ftp.ifad.dk, directory /pub/vdm/examples, 1994.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hei94] Maritta Heisel. A formal notion of strategy for software development. Technical Report 94–28, Technical University of Berlin, 1994.

[Hei96] Maritta Heisel. An approach to develop provably safe software. *High Integrity Systems*, 1996. to appear.

[HK94] Wolfgang Halang and Bernd Krämer. Safety assurance in process control. *IEEE Software*, 11(1):61–67, January 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HS96] Maritta Heisel and Carsten Sühl. Formal specification of safety-critical software with Z and real-time CSP. In E. Schoitsch, editor, *Proceedings 15th International Conference on Computer Safety, Reliability and Security*. to appear, 1996.

[HSZ95] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. Tool support for formal software development: A generic architecture. In W. Schäfer and P. Botella, editors, *Proceedings 5-th European Software Engineering Conference*, LNCS 989, pages 272–293. Springer-Verlag, 1995.

[Jac95] Jonathan Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, February 1995.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[Lev86] Nancy Leveson. Software safety: Why, what, and how. *Computing Surveys*, 18(2):125–163, June 1986.

[Lev91] Nancy Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.

[Lev95] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[MMS90] Louise E. Moser and P.M. Melliar-Smith. Formal verification of safety-critical systems. *Software – Practice and Experience*, 20(8):799–821, August 1990.

[MP95] J.A. McDermid and R.H. Pierce. Accessible formal method support for PLC software development. In G. Rabe, editor, *Proceedings of the 14th International Conference on Computer Safety, Reliablity and Security (SAFECOMP), Belgirate, Italy*, pages 113–127, London, 1995. Springer.

[MS93] Paul Mukherjee and Victoria Stavridou. The formal specification of safety requirements for storing explosives. *Formal Aspects of Computing*, 5:299–336, 1993.

[RRH93] A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, January 1993.

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

[Süh96] Carsten Sühl. Eine Methode für die Entwicklung von Softwarekomponenten zur Steuerung und Kontrolle sichereitsrelevanter Systeme. Master's thesis, Technical University of Berlin, 1996.

[Web96] Matthias Weber. Combining Statecharts and Z for the design of safety-critical systems. In M.-C. Gaudel and J. Woodcock, editors, *FME '96 — Industrial Benefits and Advances in Formal Methods*, LNCS 1051, pages 307–326. Springer Verlag, 1996.

[Wil94] Lloyd Williams. Assessment of safety-critical specifications. *IEEE Software*, pages 51–60, January 1994.

[Woo91] J.C.P. Woodcock. The refinement calculus. In S. Prehm and W.J. Toetenel, editors, *Proc. 4-th International Symposium of VDM Europe, Vol. 2*, LNCS 552, pages 80–95. Springer-Verlag, 1991.

A Real-Time CSP

Prefix: $a \rightarrow P$ first accepts event a and subsequently behaves like process P;

External Choice: $P \,\Box\, Q$ behaves either identical to process P or Q where the environment might influence this choice by accepting a certain initial event;

Channel Input: $c?x \rightarrow P(x)$ first is ready to receive an arbitrary value x from channel c and afterwards behaves like the parameterized process $P(x)$;

Channel Output: $c!v \rightarrow P$ first is ready to write the value v to the channel c and subsequently behaves equal to process P;

Parallel Composition: $P \parallel Q$ has the processes P and Q as parallel subprocesses;

Sequential Composition: P; Q first behaves like process P until its termination and afterwards behaves like process Q;

Atomic Process: *Skip* is only accepting the termination event before releasing control;

Wait: *Wait* t does not accept any event for the first t time units and afterwards is ready to accept the termination event before releasing control.