Applying RT-Z to Develop Safety-Critical Systems

Carsten Sühl

GMD FIRST, Kekuléstraße 7, 12489 Berlin, Germany
suehl@first.gmd.de

Abstract. We present the application of the formal specification language RT-Z, an integration of the model-based specification language Z and the real-time process algebra timed CSP, in the area of safety-critical systems. The characteristics underlying the development of safety-critical systems are identified, and criteria for specification languages to be used in this area are derived. It is demonstrated by means of a case study that RT-Z satisfies these criteria.

1 Introduction

The benefit of using formal methods in the area of safety-critical systems has already been discussed in detail, consult e.g. [2, 7, 13]. Craigen et al. [2, 7] surveyed the use of formal methods in the industrial context, where the majority of the considered projects concerned safety-critical systems. The conclusions drawn were by no means enthusiastic, but revealed the existing deficiencies of formal methods and provided recommendations to overcome the present situation. In [8, 9] we have presented a combination of the formal languages Z and timed CSP including a methodological framework for its application to safety-critical systems. The motivation to use a combined language was that a formalism intended to model safety-critical systems must take into account both the dynamic and the static behaviour of a system. Z [16] is a very successful formal language for specifying data and algorithmic aspects. However, it is not designed to model aspects of control and concurrency. The real-time process algebra timed CSP [3], on the other hand, is a powerful language to specify the dynamic control behaviour, including real-time aspects. However, it does not provide any constructs to model data-oriented aspects. Both formalisms can hence be considered as complementary, covering disjoint aspects of safety-critical systems. Further, their underlying concepts are well suited to each other.

In [15], we presented a successor of the initial approach of combining both languages, called RT-Z. The most relevant improvements of RT-Z concern the following aspects. Most importantly, RT-Z is able to structure a large and complex system specification into a collection of specification units, which interact with each other via well-defined interfaces. Further, it provides abstract and concrete language constructs to adequately express both abstract requirements and concrete designs, and, last but not least, RT-Z is based on a clearer and more

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 51–65, 2000.

strictly defined model of integrating both base languages than its predecessor. A unified denotational semantics serves to map each specification to a unique meaning.

RT-Z is designed to formally specify real-time embedded systems in general; the aim of this paper is to demonstrate the appropriateness of RT-Z in the area of safety-critical systems and to illustrate the merits of RT-Z with respect to its predecessor, especially in the context of safety-critical systems. To facilitate the comparison between both versions, we use a similar case study to discuss RT-Z as was used in [9], namely the specification of a railway network.

The paper is organised as follows. We first give a short overview of the specification language RT-Z in Section 2, which should be sufficient to understand the following case study. In Section 3, we discuss the characteristics of the development of safety-critical systems and the resulting criteria to specification languages used in this area. The railway network case study in Section 4 constitutes the main part of the paper. We then contrast RT-Z with another specification language and compare our case study with a similar one. We complete the paper by drawing conclusions in Section 6.

2 Overview of RT-Z

The aim of this paper is not to introduce RT-Z; a comprehensive description of the integrated formalism RT-Z including an outline of its unified denotational semantics can be found in [15]. Readers not familiar with the base languages Z and timed CSP are referred to [16] and [3], respectively. In this section, we shall only give a short overview of the main principles of RT-Z.

A system specification in RT-Z is composed of a hierarchy of so-called *specification units*. Each specification unit defines the structure and behaviour of a corresponding system component; therefore, a hierarchy of specification units corresponds to a hierarchy of system components constituting the considered system. Specification units interact with each other via well-defined interfaces.

Each specification unit consists of a Z part and a (timed) CSP part. The Z part defines aspects of the overall component behaviour like its data state, transitions thereon, or properties of data values communicated with interactions occurring at the component interface. The CSP part defines aspects of the component behaviour like the structure of its interface and the temporal order of interactions occurring at this interface and their real-time constraints. More precisely, each specification unit consists of several *sections*, where each section serves to define a specific, more restricted fragment of the overall component behaviour and belongs to exactly one part.

Each specification unit may *aggregate* any number of other specification units. Aggregated specification units may be arbitrarily arranged by the aggregating unit by means of CSP operators (e.g., parallel composition). The relationship between aggregating and aggregated specification units induces the mentioned hierarchy.

We distinguish two kinds of specification units: abstract and concrete ones. The former ones are used to abstractly specify properties of a system (software) component in the requirements phases, whereas the latter ones are used to express architectural and implementation aspects in the design phases.

Further details of the formalism are provided when discussing the case study.

3 Developing Safety-Critical Systems

In this section, we discuss the specific characteristics that—from our point of view—underlie the development of safety-critical systems, and we derive criteria for a formalism to be applied in this area. Following Leveson [12, Chapter 9], *safety* is the property of a system to be free from accidents, where *accidents* in turn are defined to be undesired and unplanned events that result in a specified level of loss. These two notions give no indication how to guarantee safety when developing a system that may potentially contribute to accidents; the notion hazard bridges this gap. A *hazard* is defined to be a state of a system that, together with conditions in the environment of this system, will lead inevitably to an accident.

One immediate consequence of these definitions is that software per se cannot be safe or unsafe, because (as an abstraction) it is not able to cause accidents. However, in the context of a system into which it is embedded software may contribute to accidents. Thus, one can only consider the safety of a software component in the context of the specific system within which it is embedded. Moreover, from the perspective of the system design, hazards are the only concept that can be influenced, because accidents (and consequently safety) result from an interaction of conditions inside *and* outside the considered system, where the latter ones cannot be influenced by the system design. As a consequence of these considerations, when we intend to develop "safe software" we have ultimately to aim at *system* safety by designing measures that prevent the whole system from reaching a hazardous state, i.e., one that may contribute to an accident. The identification and unambiguous formulation of these hazardous states should be the starting point of the development of any safety-critical system.

Which are the implications for a formal specification language that is intended to be employed in the development of software embedded in safety-critical systems? Firstly, because safety is intrinsically a system property, the formalism must be able to cope with whole systems rather than only with isolated (software) components. Therefore, the formalism must provide means to structure large and complex artifacts, namely systems. Secondly, it has to be expressive enough to be applied to software components and to general system components, including so different kinds of artifacts like mechanical, electrical, and electronic devices, human operators, etc. Finally, it must be capable of covering a wide range of abstraction layers in order to be applicable in various phases of a system development process, starting at the system requirements acquisition, where the hazardous states are identified and specified, and ending at the software de-

sign, where the identified software components are designed to meet their fixed requirements.

4 Case Study: Railway Network

To demonstrate the applicability of RT-Z to safety-critical systems, we have chosen the formal specification of a railway network for the following reasons. First, it is undoubtedly safety-critical. Second, we had chosen the railroad crossing problem to discuss the predecessor of RT-Z, so that a comparison between both is facilitated. Last but not least, the railway network has the appropriate level of complexity. On the one hand, it is complex enough to demonstrate the decomposition concepts of RT-Z, on the other hand it allows us to find an appropriate abstraction level that is not overwhelming. Our case study is based on the 'Generalized Railroad Crossing' problem presented in [10] and [11]; it is, however, lifted to the specification of a whole railway network in order to be able to better motivate the use of Z.

We first outline how we will proceed in the phases of the development process of the railway network that are supported by the use of RT-Z. The starting point is the system requirements specification, in which the identified system hazards are specified. A crucial task of this phase is to fix the system boundaries. In our case study, the considered system, in terms of which the safety constraints[1] will be formulated, is a railway network located in a certain area, which includes the railway lines running through this area and the railway crossings at the intersections of the railway lines and the traffic lines in this area, see Figure 1. Certainly, there are various hazards that can be identified for such a railway network. For the sake of brevity, we consider one of them as an example: *the state in which a train is in a railway crossing and its gates are not closed,* a state that can lead to a collision between the train and a vehicle crossing the railway line. The purpose of the subsequent system design is to work out an architecture of the considered system that is able to realize the specified requirements and constraints. Such an architecture provides the decomposition into a set of system components and the definition of the channels via which these components interact. The general aim of the following software requirements phase is to abstractly specify the behaviour of the software components identified during the system design, i.e., to formulate requirements to their behaviour at their external interface without restricting their internal realization. It is important to emphasise that in our case study the component to be finally developed is only a part of the system that is considered in toto: the component to be developed is the controller of the railway network, but to express and analyse the conditions of its "safe" operation, we have to take into account the whole railway network within which it is embedded. Finally, in the software design, an architecture of each software component identified in the system design is worked out and fixed, so that the constraints and requirements defined in the software requirements specification

[1] We consider safety constraints to be formalisations of system hazards in a negated form.

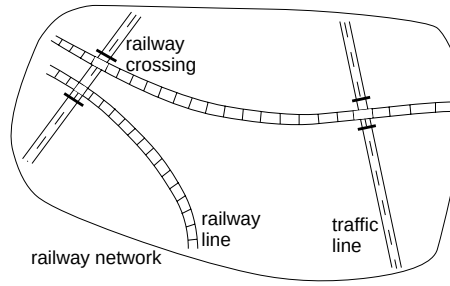

Fig. 1. System under development.

are met. It is one of the essential merits of RT-Z with respect to its predecessor to provide all necessary notational means to express such a software architecture.

4.1 System Requirements

The abstract specification unit *RailwayNetwork*, which fixes the results of the system requirements phase, formally expresses the selected safety constraint on the operation of the railway network. We first encapsulate general definitions within the specification unit *TechnicalPars*.

SPEC. UNIT *TechnicalPars*

We make use of the powerful notation of the base language Z to formally express the technical parameters of operating the railway network, which include relatively complex data-oriented relationships.

TYPES & CONSTANTS

The constituting elements of the railway network are trains (*TRAIN*), railway lines (*RAIL*), and railway crossings (*CROSS*).

$$[TRAIN, RAIL, CROSS]$$

Each railway line can run through any number of railway crossings in the considered area and vice versa, see Figure 1.

$$runs\_through : RAIL \leftrightarrow CROSS$$
$$\mathsf{dom}\ runs\_through = RAIL \land \mathsf{ran}\ runs\_through = CROSS$$

When a train passes through the railway network, we consider four different kinds of events: the train's entry into and exit out of the network and the train's entry into and exit out of the various railway crossings.

The functions ϵ_1 and ϵ_2 model the durations that the fastest and slowest trains need to reach a particular railway crossing after having entered the railway network on a particular railway line. Further, the functions γ_{down} and γ_{up} represent the maximal time needed to completely lower and raise the gates at a particular crossing, respectively.

$$\begin{array}{|l}
\epsilon_1, \epsilon_2 : RAIL \rightarrow (CROSS \nrightarrow \mathbb{T}^+) \\
\gamma_{down}, \gamma_{up} : CROSS \rightarrow \mathbb{T}^+ \\
\hline
(\forall\, r : RAIL \bullet \mathsf{dom}(\epsilon_1\, r) = \mathsf{dom}(\epsilon_2\, r) = runs_through(\!|\{r\}|\!) \wedge \\
\quad (\forall\, c : runs_through(\!|\{r\}|\!) \bullet (\epsilon_1\, r\, c) \leqslant_{\mathbb{R}} (\epsilon_2\, r\, c) \wedge (\epsilon_1\, r\, c) >_{\mathbb{R}} (\gamma_{down}\, c)))
\end{array}$$

The maximal time needed to lower a gate must be less than the time that the fastest train needs to reach the corresponding railway crossing after having entered the railway network.

END UNIT

SPEC. UNIT *RailwayNetwork* **EXTENDS** *TechnicalPars*

The above **EXTENDS** clause causes the import of all definitions made in the previous specification unit.

Because of the chosen system boundaries[2], all relevant events occurring in the railway network are internal. The internal channels of the railway network, on which these internal events occur, are introduced and associated with a type in the **LOCAL** section.

LOCAL

channel *enterN*, *exitN* **of type** $TRAIN \times RAIL$;
channel *enterC*, *exitC* **of type** $TRAIN \times RAIL \times CROSS$;
channel *lowering*, *raising*, *down*, *up* **of type** $CROSS$

Events occurring on the channels *enterN* and *exitN* represent the entry into and the exit out of the railway network on a particular railway line, whereas events on the channels *enterC* and *exitC* represent the entry into and exit out of a railway crossing, respectively. Moreover, events occurring on the channels *down*, *up*, *lowering* and *raising* model that the gates of a particular crossing change into the state of being completely closed, being completely open, going down and going up, respectively.

BEHAVIORAL PROPERTIES

As claimed in Section 3, the most important task of the system requirements phase of the development of safety-critical systems is to identify the relevant system hazards and to formally express them in the form of safety constraints. In RT-Z, this can be achieved in the **BEHAVIORAL PROPERTIES** section, which serves, in general, to abstractly specify the requirements and the constraints on the dynamic and static behaviour. The notational means of this section is the predicate language of the timed failures model of timed CSP [3]. In our integration RT-Z, this predicate language is extended by allowing references to Z schemas in order to enable the compact formulation of properties of both the dynamic and the static behaviour within a single, coherent language.

[2] Trains and railway crossings (including gates) are part of the considered system. Certainly, the railway network is not a closed system from a physical point of view, because trains are entering and leaving the network's area; nevertheless, the corresponding events are internal, since a train is considered to be part of the railway network even in time intervals when it is outside the network's area.

The schema *SafeRailwayCrossing* specifies, for a particular railway crossing *cross* of the railway line *rail*, the required relationship between the last time instants a train has entered and left the railway crossing and the last time instants the corresponding gates have been completely closed and opened, respectively: the gates have to be closed whenever a train has entered but not already left the crossing.

SafeRailwayCrossing

$$rail : RAIL;\ cross : CROSS$$
$$last\_enterC, last\_exitC, last\_down, last\_up, last\_raising : \mathbb{T}$$

$$(last\_enterC >_{\mathbb{R}} last\_exitC \land cross \in runs\_through(\!|\{rail\}|\!))$$
$$\Rightarrow last\_down >_{\mathbb{R}} max_{\mathbb{R}}\{last\_up, last\_raising\}$$

The defined schema is used in order to specify the selected safety constraint, which is expressed by a predicate on timed traces s and timed refusals X, which, according to the timed failures model of timed CSP, constitute the timed observations of the system under consideration. This predicate basically defines how the values with which the Z schema is evaluated are extracted from the timed trace component s. The operators of the predicate language, e.g., $\restriction$, $\downarrow$ and $\lfloor$, are discussed in [15].

$$Behavior(s, X)\ \mathbf{sat}\ \forall\, t : \mathbb{T};\ r : RAIL;\ tr : TRAIN;\ c : CROSS \bullet$$
$$SafeRailwayCrossing(rail == r, cross == c,$$
$$last\_enterC == \mathrm{end}(s \restriction t \downarrow enterC \lfloor \{(tr, r, c)\}),$$
$$last\_exitC == \mathrm{end}(s \restriction t \downarrow exitC \lfloor \{(tr, r, c)\}),$$
$$last\_down == \mathrm{end}(s \restriction t \downarrow down \lfloor \{c\}), \ldots)$$
$$\land \ldots$$

Certainly, the railway network has to guarantee further constraints, e.g., utility constraints ensuring that the gates of railway crossings are not closed when not necessary. For reasons of space, we omit them.

END UNIT

The specification of the system requirements has illustrated that RT-Z is able to cope with system concepts (in addition to software concepts) and to abstractly specify properties that are related to both the dynamic and the static behaviour.

4.2 System Design

To guarantee the safety constraint expressed in the system requirements phase, the railway network is decomposed into three system components. The *train* component and the *gate* component model the assumptions on the behaviour of the trains and gates within the considered network. The task of the third component, *controller*, is to supervise all gates in the network according to the information about the current position of the trains, obtained by sensor reports

via the channels *enterN*, *exitN* and *exitC*. The gates of the network area are instructed by the controller via the channels *lower* and *raise*.

The following concrete specification unit fixes the results of the system design.

SPEC. UNIT *RailwayNetworkDesign* **EXTENDS** *TechnicalPars*

LOCAL

channel $enterN, exitN$ **of type** $TRAIN \times RAIL$;
channel $enterC, exitC$ **of type** $TRAIN \times RAIL \times CROSS$;
channel $lower, raise, lowering, raising, down, up$ **of type** $CROSS$

SUBUNITS

The identified system components are associated with specification units defining their structure and behaviour.

$train(TRAIN)$ **spec. unit** *Train*;
$gate(CROSS)$ **spec. unit** *Gate*;
controller **spec. unit** *Controller*

BEHAVIOR

The *physical* structure of the system components and their physical interaction via channels is specified by the timed CSP process *Behavior*.

$$
\begin{aligned}
Behavior \mathrel{\widehat{=}} \quad & \Big(\Big|\Big|\Big|_{id \in TRAIN} train(id).Behavior \Big) \\
& |[\, \{\!|\, enterN \,|\!\} \cup \{\!|\, exitN \,|\!\} \cup \{\!|\, exitC \,|\!\}\,]| \\
& \qquad controller.Behavior \\
& |[\, \{\!|\, lower \,|\!\} \cup \{\!|\, raise \,|\!\}\,]| \\
& \qquad \Big(\Big|\Big|\Big|_{id \in CROSS} gate(id).Behavior \Big)
\end{aligned}
$$

END UNIT

The above specification unit demonstrates the ability of RT-Z to formally define architectural aspects fixed in the system design.

As already mentioned, the system component to be finally developed is the controller of the railway network. This means that the trains and gates constitute the environment of the component under development. The following specification unit documents the assumptions that underlie the development of the controller with respect to the behaviour of trains.

SPEC. UNIT *Train*

INTERFACE

channel $enterN, exitN$ **of type** $RAIL$;
channel $enterC, exitC$ **of type** $RAIL \times CROSS$

BEHAVIOR

The behaviour of a train, as defined by the process term *Behavior*, is as follows: it enters the railway network on an arbitrary railway line at an arbitrary time

instant, which is expressed by the combination of the internal choice operator ($\sqcap$) and the *Wait* operator parameterised with the interval $\mathbb{T}^+$. It then enters and leaves the different railway crossings of the current railway line, which is modelled by the interleaving operator ($|||$) indexed with the set of crossings of the current railway line. The time distance between the entry into the network and the entry into a particular railway crossing c is restricted to the interval $[\epsilon_1\ r\ c, \epsilon_2\ r\ c]$.

$$\begin{aligned} \mathit{Behavior} \mathrel{\widehat{=}} \mu\, X \bullet \sqcap_{r \in \mathit{RAIL}} \mathit{Wait}\, \mathbb{T}^+;\ \mathit{enterN}!r \rightarrow \\ (|||_{c \in \mathit{runs\_through}(\{r\})} \mathit{Wait}[\epsilon_1\ r\ c, \epsilon_2\ r\ c];\ \mathit{enterC}!(r,c) \rightarrow \\ \mathit{Wait}\, \mathbb{T}^+;\ \mathit{exitC}!(r,c) \rightarrow \mathit{Skip});\ \mathit{Wait}\, \mathbb{T}^+;\ \mathit{exitN}!r \rightarrow X \end{aligned}$$

END UNIT

By treating the trains as components of the considered system, we have succinctly documented all assumptions that the controller can make about their behaviour. Since the specification of the gates' behaviour is analogous, we omit it.

4.3 Software Specification

The aim of the software requirements specification is to abstractly define the constraints on the operation of the railway network controller to be implemented by software.

We first define the interface of the software controller separately.

SPEC. UNIT *SCInterface*

INTERFACE

The interface of the software controller does not contain all channels that we have chosen to express the system requirements and the system design. This is because we have performed the transition from the system into the software phases. The events occurring on the channel *enterC*, for instance, mark relevant points of a system observation, namely the entry of a train into a railway crossing. From the software controller's point of view, however, these events are not visible, because in the system design it has been decided that the entry of trains into the railway crossings is not observed by a sensor.[3] Similar remarks apply to the events on the channels *up*, *down*, *lowering* and *raising*, which label state transitions of the gates that the controller need not know.

channel $\mathit{enterN}, \mathit{exitN}$ **of type** $\mathit{TRAIN} \times \mathit{RAIL}$;
channel exitC **of type** $\mathit{TRAIN} \times \mathit{RAIL} \times \mathit{CROSS}$;
channel $\mathit{lower}, \mathit{raise}$ **of type** CROSS

END UNIT

[3] This is the case, because the controller need not know when a train really enters a crossing. It must simply guarantee that the gates are closed when a train could potentially reach the crossing.

SPEC. UNIT *Controller* **EXTENDS** *TechnicalPars*, *SCInterface*

BEHAVIORAL PROPERTIES

The constraint on the behaviour of the software controller is, like the safety constraint on the behaviour of the whole railway network, expressed by a predicate of the timed failures model. The main difference between both formulations is the set of channels they refer to. Since the interface of the software controller is only a subset of the channels used in the system requirements specification, the occurrence of events outside the controller interface, e.g., the entry of a train into a railway crossing, must be appropriately traced back to the occurrence of events of the controller interface.

The schema *TrainInCrossing* specifies, for a particular railway crossing *crs* of a railway line *rail*, the situation that a train has entered but not already left the crossing.

TrainInCrossing

$$rail : RAIL;\ crs : CROSS$$
$$last\_enterN, last\_exitC : \mathbb{T}$$
$$crs \in runs\_through(\!|\{rail\}|\!) \land last\_enterN +_{\mathbb{R}} (\epsilon_1\ rail\ crs) >_{\mathbb{R}} last\_exitC$$

This schema is used to define the constraint on the behaviour of the software controller.

$$\begin{aligned}
&Behavior(s, X)\ \mathbf{sat}\ \forall\, t : \mathbb{T};\ r : RAIL;\ tr : TRAIN;\ c : CROSS \bullet \\
&\quad \exists_1\, t1 : \mathbb{T} \mid t1 = \mathrm{end}(s \upharpoonright t \downarrow enterN \downharpoonright \{(tr, r)\}) \bullet \\
&\quad TrainInCrossing(rail == r, crs == c, last_enterN == t1, \\
&\qquad\qquad last_exitC == \mathrm{end}(s \upharpoonright t \downarrow exitC \downharpoonright \{(tr, r, c)\})) \\
&\quad \Rightarrow (((lower, c)\ \mathsf{live\ from}\ (t1 +_{\mathbb{R}} (\epsilon_1\ r\ c) -_{\mathbb{R}} (\gamma_{down}\ c)) \\
&\qquad \lor \mathrm{last}(s \upharpoonright (t1 +_{\mathbb{R}} (\epsilon_1\ r\ c) -_{\mathbb{R}} (\gamma_{down}\ c)) \downharpoonright \{(lower, c), (raise, c)\}) = \\
&\qquad\qquad (lower, c)) \\
&\qquad \land\ s \downharpoonright \{(raise, c)\} \upharpoonright [t1 +_{\mathbb{R}} (\epsilon_1\ r\ c) -_{\mathbb{R}} (\gamma_{down}\ c), t] = \langle\rangle)
\end{aligned}$$

If a train tr enters the network at $t1$ on the railway line r, then the gates at the railway crossing c of r have to be closed at $(t1 +_{\mathbb{R}} (\epsilon_1\ r\ c))$, which is the earliest time instant at which the train can reach the crossing. Because the maximal time needed to completely close the gates is $(\gamma_{down}\ c)$, the command to lower the gates must be given at $(t1 +_{\mathbb{R}} (\epsilon_1\ r\ c) -_{\mathbb{R}} (\gamma_{down}\ c))$ at the latest unless the gates are already closed. Further, the gates have to remain closed until the corresponding sensor reports that the train has left railway crossing c.

END UNIT

4.4 Software Design

In this section, we shall discuss how to employ the language constructs offered by RT-Z to specify a safety-oriented software architecture. The software controller is decided to be composed of two components: a general control and a safety

component. The redundant arrangement of the components *gen_control* and *safety*, both responsible for ensuring the specified (safety-related) constraint, is a means to enhance the fault-tolerance of the railway network controller. The only task of the safety component is to enforce satisfaction of the (safety-related) constraint; it is therefore essentially simpler than the general control component, which is additionally cluttered with functional requirements.

SPEC. UNIT *ControllerDesign* **EXTENDS** *TechnicalPars*, *SCInterface*

SUBUNITS

safety **spec. unit** *SafetyComp*;
gen_control **spec. unit** *ControlComp*

BEHAVIOR

The parallel composition operator ($|[\ ]|$) with the chosen synchronisation set is adequate to arrange the general control and safety components. Since the *raise* channel is a member of the synchronisation set, a command to a gate is sent if and only if both components agree on this command simultaneously. The safety component can hence prevent the general control component from issuing a raise command violating the (safety-related) constraint. Further, because the *lower* channel is not part of the synchronisation set, both components are able to send the lower command independently, so that the safety component cannot be prevented by the general control component from lowering gates.

$$Behavior \mathrel{\widehat{=}} safety.Behavior \\ |[\ \{\!|\ raise\ |\!\} \cup \{\!|\ enterN\ |\!\} \cup \{\!|\ exitN\ |\!\} \cup \{\!|\ exitC\ |\!\}\ ]| \\ gen\_control.Behavior$$

END UNIT

Finally, we outline the specification of the safety component, where we concentrate on the discussion of the concrete language constructs of RT-Z for specifying implementation aspects. In the software design, we interpret the software component under consideration as encapsulating a data state and transitions thereon (operations), where both aspects are specified by the Z notation. The definition of the external stimulus-response and the internal control behaviour, i.e., how the software component reacts to external stimuli in terms of causing the execution of operations and sending responses into the environment, is specified by the process term language of timed CSP.

SPEC. UNIT *SafetyComp* **EXTENDS** *TechnicalPars*, *SCInterface*

TYPES & CONSTANTS

Because we are faced with implementation aspects in the software design, we cannot abstract from the time needed to execute operations: accordingly, the constant Δ_{op} represents the duration of executing one of the operations.

$TrainStatus ::= In \mid Out$
$GateStatus ::= Down \mid Up$

$phase, \Delta_{op} : \mathbb{T}^+$
$\Delta_{op} <_{\mathbb{R}} phase$

STATE

To meet its (safety-related) constraint, the safety component has to record some information for each train and gate. For each train in the network, it has to record the current railway line (*train_in_rail*) and the minimal time needed to reach a particular railway crossing (*sched_time*). In this context, the (safety-related) constraint can be formulated as a state invariant: for each train in the network, the minimal time needed to reach a currently open gate must be greater than the maximal time needed to lower this gate.

State

$train\_status : TRAIN \rightarrow TrainStatus$
$train\_in\_rail : TRAIN \nrightarrow RAIL$
$sched\_time : TRAIN \nrightarrow CROSS \nrightarrow \mathbb{T}$
$gate\_status : CROSS \rightarrow GateStatus$

$\mathsf{dom}\ sched\_time = \mathsf{dom}\ train\_in\_rail = \{tr : TRAIN \mid train\_status\ tr = In\}$
$(\forall\, cr : CROSS \mid gate\_status\ cr = Up \bullet$
$(\forall\, tr : TRAIN \mid tr \in \mathsf{dom}\ train\_in\_rail \land cr \in runs\_through(\!|\{train\_in\_rail\ tr\}|\!) \bullet$
$(sched\_time\ tr\ cr) \geqslant_{\mathbb{R}} (\gamma_{down}\ cr)))$

OPERATIONS

We explain only one of the safety component's operations as an example.

The operation *enterN* defines the reaction of the safety component to receiving the sensor report that a train has entered the railway network on a particular railway line.

enterN

$\Delta State$
$id? : TRAIN \times RAIL$

$train\_status(first\ id?) = Out$
$train\_status' = train\_status \oplus \{first\ id? \mapsto In\}$
$train\_in\_rail' = train\_in\_rail \oplus \{id?\}$
$sched\_time' = sched\_time \oplus \{first\ id? \mapsto \epsilon_1(second\ id?)\}$
$gate\_status' = gate\_status$

BEHAVIOR

The process term *Behavior* defines the stimulus-response and the internal control behaviour of the safety component. As a result of the software design, it is decided that the safety component has to pass cyclically through two phases: in *Phase*1 it is ready to receive sensor reports about entering and leaving trains and records this information by executing the corresponding operations. After *phase* time units, it changes to *Phase*2 where it computes the commands to be sent to the

gates according to the received sensor reports. For reasons of space, we show only a fragment of the behaviour definition.

$$Behavior \mathrel{\widehat=} \mu\, Cycle \bullet (Phase1 \swarrow \{phase\}\ Phase2);\ Cycle$$

$$Phase1 \mathrel{\widehat=} \ldots\, TrainCtrl \ldots$$

The process *TrainCtrl* specifies the reaction of the safety component to receiving a sensor report, e.g., that a train has entered the network on a particular railway line. In this case, the execution of the operation *enterN* is caused after a delay of Δ_{op} time units, which is modelled by the communication on the channel $enterN_X$; for a more detailed explanation of the interplay between the Z and timed CSP definitions see [15].

$$\begin{aligned} &TrainCtrl \mathrel{\widehat=} \mu\, X \bullet \\ &\quad enterN?(id : TRAIN \times RAIL) \rightarrow Wait\, \Delta_{op}; \\ &\qquad enterN_X?(par : enterN_{InMap}(id)) \rightarrow X \\ &\quad \Box \ldots \end{aligned}$$

...

END UNIT

The design specification of the safety component has demonstrated the ability of RT-Z to specify implementation aspects of a software component, including real-time aspects.

5 Related Work

Recently, combining formalisms has become a research field of growing interest. The conference on 'Integrated Formal Methods' [1] has been dedicated to this topic, where different combinations/integrations of behavioural and model-based formalisms and their varying underlying principles and concepts have been discussed. In this article, we have discussed the application of one of these integrations, RT-Z, in the area of safety-critical systems.

An overview of the general principles of combining model-based and behavioural formalisms can be found in [6]. Fischer [5] has provided a succinct survey of the more restricted field of combining Z with process algebras. Two formalisms closely related to RT-Z are TCOZ [14] and CSP-OZ [4], which integrate Object-Z with timed CSP and (plain) CSP, respectively.

In [11], the 'Generalized Railroad Crossing' (GRC) has been proposed as a benchmark problem for specifying and verifying real-time systems, to which different formal languages have been applied. As mentioned, we have based our case study on this benchmark problem but have extended the problem definition in order to be able to better motivate the use of Z in our integration RT-Z.

The GRC case study presented in [10] has demonstrated one merit of the timed automaton formalism, namely its means to verify that a design meets its requirements with the help of a more or less detailed (mathematical) argumentation. In our case study, in contrast, we have not addressed these verification

tasks. It is part of further work to develop more powerful verification techniques for RT-Z than currently available, see also Section 6. The drawback of the timed automaton formalism, however, are its rather rudimentary constructs for specifying the data-oriented aspects of a system. The system state is constituted by a plain set of variables without any facility to structure the state by higher-level components, to specify invariant properties these variables have to satisfy, or to introduce new data types with an associated domain theory. In contrast, RT-Z is based on the Z notation, which supports these aspects in a very powerful way. Moreover, the concepts provided by the timed automaton formalism to express designs in terms of state-transition systems are less convenient than RT-Z's notation to define architectures. This is due to the fact that the concepts provided by timed automata only allow one to specify flat structures and that parallel composition is the only means to decompose systems into system components. From our point of view, these are obstacles to cope with complex systems.

6 Conclusion

The main contribution of this paper is the presentation of a formalism, RT-Z, which is able to support the development of software embedded within safety-critical systems in various phases of the development process. The demonstration of the appropriateness of RT-Z in this area has been achieved by means of a case study involving a safety-critical system.

The discussion of the notions related to safety in Section 3 has resulted in the criteria that a formalism must meet in order to be reasonably used to develop "safety-critical software". The presentation of the case study in Section 4 has demonstrated that RT-Z satisfies all these criteria:

- different phases of the development process have been successfully treated, starting with the system requirements specification and ending with the software design specification,
- RT-Z has been applied to an entire system as well as to software components, and
- RT-Z has accomplished to specify abstract requirements as well as concrete designs.

By treating a similar case study as has been used for RT-Z's predecessor in [9], we have demonstrated the merits of RT-Z with respect to this predecessor, which has not been able to cope with large and complex systems, to formally express architectures, and to specify abstract properties in an equally integrated way like RT-Z.

Planned further work includes to provide methodological support for the application of RT-Z to safety-critical systems as presented in [9] for its predecessor. As has already been pointed out in Section 5, a major point of further work concerns the development of an integrated proof system for RT-Z that enables us, e.g., to formally prove an RT-Z design specification correct with respect to an RT-Z requirements specification, i.e., to prove a refinement relationship between both.

Acknowledgement

Many thanks to Kirsten Winter and the anonymous referees for helpful comments and constructive criticism.

References

[1] K. Araki, A. Galloway, and K. Taguchi, editors. *Proceedings of the 1st International Conference on Integrated Formal Methods*. Springer, 1999.

[2] D. Craigen, S. L. Gerhart, and T. J. Ralston. Formal methods reality check: Industrial usage. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 250–267. Springer, 1993.

[3] J. Davies and S. Schneider. Real-time CSP. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*. World Scientific Publishing Company, Inc., Feb. 1995.

[4] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[5] C. Fischer. How to combine Z with a process algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM '98: The Z Formal Specification Notation*, number 1493 in LNCS, pages 5–23. Springer, 1998.

[6] A. Galloway and B. Stoddart. Integrated formal methods. In *Proceedings of INFORSID '97*, 1997.

[7] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–28, Jan. 1994.

[8] M. Heisel and C. Sühl. Combining Z and Real-Time CSP for the development of safety-critical systems. In *Proceedings 15th International Conference on Computer Safety, Reliability and Security*. Springer, 1996.

[9] M. Heisel and C. Sühl. Methodological support for formally specifying safety-critical software. In *Proceedings 16th International Conference on Computer Safety, Reliability and Security*. Springer, 1997.

[10] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time system. Technical Memo MIT/LCS/TM-511, Laboratory for Computer Science, Massachusetts Institute of Technology, 1994.

[11] C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing*. Number 5 in Trends in Software. John Wiley & Sons, 1996.

[12] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[13] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, Sept. 1994.

[14] B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering*, pages 95–104. IEEE Computer Society Press, 1998.

[15] C. Sühl. RT-Z: An integration of Z and timed CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*. Springer, 1999.

[16] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.