# An Approach to Develop Provably Safe Software

Maritta Heisel
Technische Universität Berlin
FB Informatik – FG Softwaretechnik
Franklinstr. 28-29, Sekr. FR 5-6
D-10587 Berlin
heisel@cs.tu-berlin.de
fax: (+49-30) 314-73488

**Abstract**

We present a process model for the development of provably safe software. It is based on well-established tools and techniques to set up formal specifications in the specification language Z and a program synthesis system designed by the author. The model provides a guideline for the specification and implementation of safe software, consisting of a number of steps that are complemented by proof obligations. The parts of the process specific to software safety are given special consideration. The approach is exemplified by the specification and partial implementation of a program controlling the pump of a steam boiler. Finally, we relate software safety to correctness and reliability.

## 1 Why, What and How

Our approach aims at developing software for which certain safety conditions can be guaranteed. For this purpose, formal methods are applied. The idea is to express the safety conditions in a formal specification language. This makes it possible to treat them like functional requirements, i.e. methods and tools designed to ensure software correctness become applicable. With this approach, the specification of the system not only states functional requirements but consists of two parts, functional and safety requirements, and software safety amounts to correctness with respect to the safety requirements.

The fact that tool support is available to a large extent led us to choose the specification language Z [Spi92b] and a simple imperative programming language as the formalisms we want to use. For Z, type checkers and theorem provers are available. To actually develop safe software as specified in Z, the author's program synthesis system IOSS (Integrated Open Synthesis System) can be used. With this system, imperative programs can be developed and proven correct.

Z and imperative programs are a good match because both explicitly deal with states. The same is true for the majority of computerized systems that have to fulfill safety requirements. These facts lead us to believe that our choice is a reasonable one. We are aware that our approach cannot be applied in all cases. For distributed and parallel systems, it is only useful when the design has progressed to such detail that single modules are identified that can be treated with our chosen formalisms. Real-time requirements cannot be treated at all but only be approximated. It is not our aim to replace, but rather to complement other approaches to the development of safe software.

In the specification phase, our proposed process model roughly follows the recommended Z methodology. Differences occur where special safety considerations have to be taken into account. In contrast to correctness, safety requires to consider the environment in which the software operates.

Basically, software safety means that certain *invariants* have to be maintained. These invariants are partially formal, partially informal. The formal invariants characterize the *software* states that are considered safe. They can be guaranteed by proving the system correct with respect to the safety requirements.

The informal invariants, however, require the internal system state to be a faithful representation of the state of the environment. The informal invariants can, of course, neither be formally proven nor be enforced by the software. Nevertheless, they influence the specification and design of the software. There will be sensors whose failure must be detected if ever possible. This can be achieved by hardware redundancy and/or consistency checks performed by the software in order to determine if the sensor values are credible. All these complications do not occur when only correctness is considered. In this case, the sensor values are input to the program, and the program has to react correctly with respect to this input. Where the input comes from and if it is sensible is of no interest here.

This means that the process of specifying and developing *safe* software differs from the process applied to obtain *correct* software, even if in the end the same formalisms are applied. It is the contribution of this work to make these differences explicit and give a guideline how to adequately deal with safety requirements.

Before we present our approach in more detail (Section 3), we briefly describe Z and IOSS (Section 2). Sections 4 and 5 are devoted to a case study, the control software of a pump in a steam boiler. A discussion of the limitations of the approach, its relation to other work and to software correctness and reliability concludes the paper.

## 2 Z and IOSS

We have chosen the specification language Z because it has gained considerable popularity in industry and comes equipped not only with a methodology [PST91] but also with some tool support, e.g. for type checking [Spi92a] and theorem proving [BG94]. Z is designed to specify state-based systems which is in good accordance with the reality of safety-critical systems. An undeniable deficiency of Z is the fact that neither time nor complex control structures can be specified.

The author's synthesis system IOSS [HSZ95b] 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. If, e.g., the specification requires the value of variable $x$ to be increased, the additional postcondition might contain the equation $x = x_0 + 4711$ which means that $x$ is increased by 4711. Such information is needed in the further synthesis process.

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.

Program synthesis with IOSS consists of a loop of strategy applications. The intermediate states of the development are represented by a data structure called *development tree*. Its nodes contain a problem and its solution (once it has been found). Each new strategy application causes the development tree to be extended, if the strategy reduces the problem to a number of subproblems. Otherwise, the problem is solved immediately, and the solution is recorded in the respective node. When all subproblems of a problem have been solved, its solution can be assembled from the solutions to the subproblems. The development is finished when all problems have been solved. Representing the state of development as a data structure makes it possible to obtain an overview of the development at any time. More about IOSS can be found in [HSZ95a].

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 [Woo91b] and is described in more detail in Section 5.

## 3 Six Steps to Safety

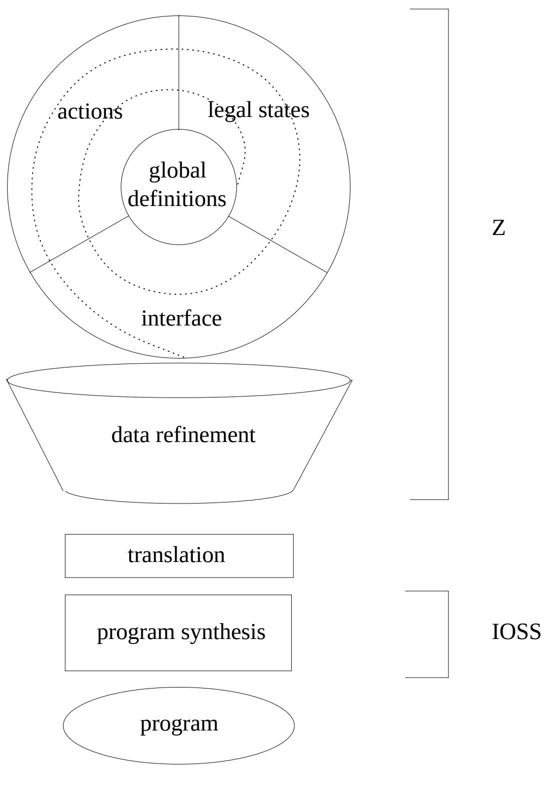

Figure 1: Overview of Approach

In the following, we describe a number of steps that constitute our approach for the development of safe software. Each step comes with some proof obligations. The first three steps give a guideline how to set up the specification of a system, where special attention is devoted to the safety requirements. In general it will not be possible to carry out these steps independently of each other and without iteration. Instead, a process resembling the spiral model of software development will have to be employed. The last three steps describe how to perform the transition from a mere specification to a totally correct program. Figure 1 provides a summary of the procedure.

**Step 1** *Define the legal states of the system.*

This definition must comprise the safety requirements as well as other properties of the legal states. We do not deal with the question how this specification is derived. It can be set up by one party, treating functional as well as safety requirements. Another possibility is to set up two specifications, a functional and a safety specification by different parties and then show that the safety requirements are entailed by the functional specification. The latter approach can be used to double-check the safety requirements, or it may be enforced by certification procedures or safety standards.

Once the legal state is defined, an initial state should be given. This is not only in accordance with the recommended Z methodology but also with other formalisms like finite state machines or statecharts where one has to define start states or default states.

**Proof Obligation 1** *Show that the initial state is legal.*

In this way, it is also shown that the requirements for legal states are satisfiable.

**Step 2** *Define the actions the system can perform.*

These can be triggered either by outside events or by the system itself. In Z, they are defined by operations that may change the system state.

**Proof Obligation 2** *Analyze the conditions under which the actions transform legal states into legal states.*

Technically, this is done by precondition analysis. This analysis yields the condition that must hold if the state reached after execution of the operation is legal, provided the state before execution of the operation is. If the precondition is not trivial, care must be taken that the operation is only executed when its precondition holds.

Analyzing preconditions also helps to detect design errors. If the precondition of an operation turns out to be *false*, the operation cannot be executed at all (or it would lead to an illegal state). This clearly shows that something is wrong with the design of the operation or even the whole system.

So far, we have applied standard Z methodology. The next step deals with the peculiarities of safe software. For software safety, the environment in which the software operates has to be taken into account. This is achieved by modeling the environment using sensors and by performing consistency checks on the sensor values.

**Step 3** *Define the interface between the system and the outside world.*

In this step the sensors must be modeled that enable the system to detect situations to which it must react. It must also be specified how the system reacts to the possible sensor values and/or failures. We advocate to model the system so as to provide exactly one internal operation for each combination of sensor values. This guarantees that each situation is taken into account, and it leads to a clear and comprehensible interface.

**Proof Obligation 3** *Show that the internal system operations are only invoked if their preconditions are satisfied.*

**Proof Obligation 4** *Show that for each combination of sensor values exactly one internal operation is invoked.*

Proof obligation 4 is not necessary to prove the safety of the software. We introduce it to encourage developers to design their systems as clear and simple as possible. These properties can contribute as much to the system's safety as a formal verification can.

**Proof Obligation 5** *Show that – if the sensors work correctly – the system faithfully represents the state of its environment.*

Once this step is performed, it is guaranteed that the internal state of the system always fulfills the safety requirements and that the system state is consistent with the state of the environment, under the condition that failure of sensors can be detected somehow. It follows that (under the same condition) also the "real" system state is safe, provided the implementation of the system is correct.

**Remark concerning proof obligations.** The proofs that have to be carried out are standard and fairly simple. However, there are a lot of them to do. Until now, specialized tool support for this purpose with a sufficient degree of automation is not yet available. Full-fledged first-order theorem provers are not necessary because the proof obligations often have the form of existentially quantified statements, with equations for the existentially quantified variables. We believe that the construction of mostly automatic, specialized provers for the proof obligations occurring in this context poses no severe problems.

The steps presented so far only dealt with the specification of safe software. A *model* of the system has been defined and it has been shown that this model behaves safely. The following steps are concerned with the correct implementation of this model. Since they are not specific to safety they are presented more briefly than the specification steps.

**Step 4** *Refine the operations and data of the specification so that data structures of the target programming language can be used.*

What refinement means and how it is performed is described in the literature, e.g. [Woo91a]. This step is not necessary if the data structures involved are available in the target programming language, as often is the case.

**Step 5** *Transform the specification obtained in Step 4 into a form suitable for the chosen program synthesis system.*

The Z specifications are transformed into IOSS programming problems, as described in Section 2. The translation process is defined in Section 5.1.

**Step 6** *Use the chosen synthesis system to obtain a proven correct implementation of the specified system.*

The last step – which can be performed automatically to a large extent – guarantees that the concrete states of the implementation are always safe, provided the abstract states of the system model are.

## 4 Case Study: A Steam Boiler

To illustrate our approach we consider the specification and implementation of software controlling the pump of a steam boiler. The problem is a simplified version of the problem stated by J.-R. Abrial [Abr94]. The pump is used to keep the water-level in the steam boiler between a minimum *safe_min* and a maximum *safe_max*, see Figure 2. This task is safety-critical because the steam boiler and its environment can be damaged when the water-level is too low or too high. The program can sense the state of its environment with two sensors: one measures the water-level, the other one measures the amount of steam leaving the boiler. To improve fault tolerance, we

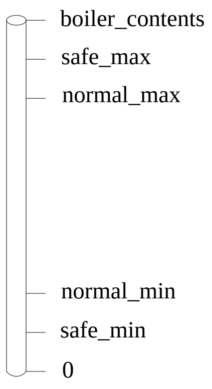

Figure 2: Relation between Boiler Entities

assume two additional sensors monitoring whether the measuring sensors work correctly. Finally, there is a button that can be pressed to initialize the system.

The program can operate in three modes: *normal*, *rescue* and *initializing mode*. In normal mode, the program relies on the measured water-level. If it is below the intervention point *normal_min* it switches on the pump; if it is above *normal_max* it switches off the pump, see Figure 2. The rescue mode is entered if the water-level sensor fails. In this mode the program tries to keep the water-level within the safety limits relying on a computation of the water-level based on the capacity of the pump and the measurement of steam leaving the boiler. When the water-level sensor is repaired the program can switch back to normal mode. In the initializing mode the program can either switch on the pump or open a water valve to establish a safe water-level. In this mode, no steam may be taken from the boiler.

If it happens, however, that the water-level in the boiler is below *safe_min* or above *safe_max* the program must raise an alarm and stop (*emergency mode*). It cannot try to take care of the situation itself because it only controls the pump and not the steam producing device. The safety limits *safe_min* and *safe_max* must be defined in such a way that there is enough time for some backup mechanisms to prevent the boiler (and its environment) from being damaged.

**Control Model.** We assume that the control operation to be defined in Step 3 (Section 4.3) is triggered periodically by a control signal, as is not unusual, see e.g. [HK94]. The rest of the specification relies on this control model in that we assume that the sensor measuring the steam output of the boiler yields the absolute amount of steam leaving the boiler in one such period of time. Similarly, we express the capacity of the pump by giving the absolute amount of water it pumps into the boiler in one interval between two consecutive executions of the control operation. This interval we will call one time unit in the following.

## 4.1 Step 1: Global Definitions and Legal States

First we have to define the entities mentioned above and their relations. This amounts to defining a model of the technical components of the boiler as far as they are relevant for the specification. A pump, for instance, can be characterized by the fact that it is on or off and by the quantity of water it pumps into the boiler per time unit when it is on.

Readers not familiar with Z can find some explanations of the notation in Appendix A.

$$
\begin{array}{|l}
\mathit{boiler\_contents} : \mathbb{N}_1 \\
\mathit{safe\_min} : \mathbb{N}_1 \\
\mathit{safe\_max} : \mathbb{N}_1 \\
\mathit{normal\_min} : \mathbb{N}_1 \\
\mathit{normal\_max} : \mathbb{N}_1 \\
\mathit{pump\_capacity} : \mathbb{N}_1 \\
\mathit{steam\_capacity} : \mathbb{N}_1 \\
\hline
\mathit{safe\_min} < \mathit{normal\_min} < \mathit{normal\_max} < \mathit{safe\_max} < \mathit{boiler\_contents} \\
\mathit{safe\_min} + \mathit{steam\_capacity} < \mathit{normal\_min} \\
\mathit{normal\_max} + \mathit{pump\_capacity} < \mathit{safe\_max} \\
\mathit{pump\_capacity} \geq \mathit{steam\_capacity}
\end{array}
$$

The entity *pump_capacity* states how much water per time unit the pump sends into the steam boiler, whereas *steam_capacity* specifies the maximum amount of steam that can leave the boiler in one time unit. The exact definitions of *safe_min*, *safe_max*, *normal_min* and *normal_max* depend on the technical properties of the pump and on the speed of the backup mechanisms designed to take over when a hazardous situation occurs. The requirements $\mathit{safe\_min} + \mathit{steam\_capacity} < \mathit{normal\_min}$ and $\mathit{normal\_max} + \mathit{pump\_capacity} < \mathit{safe\_max}$ guarantee that the pump can be switched on or off in time to prevent the water-level from leaving the safety limits.

As a consistency check for the measured and computed water-levels, we require that the difference between them must not exceed a certain tolerance. The program always outputs a message, indicating the mode in which it operates.

$$
\begin{array}{|l}
\mathit{tolerance} : \mathbb{N}
\end{array}
$$

$$\mathit{Safe} == \mathit{safe\_min} \mathrel{..} \mathit{safe\_max}$$

$$\mathit{Stable} == \mathit{normal\_min} \mathrel{..} \mathit{normal\_max}$$

$$\mathit{diff} == (\lambda\, x, y : \mathbb{N} \bullet \textbf{if } x \geq y \textbf{ then } x - y \textbf{ else } y - x)$$

$$\mathit{PUMP\_STATE} ::= \mathit{on} \mid \mathit{off}$$

$$\mathit{WATER\_VALVE\_STATE} ::= \mathit{open} \mid \mathit{closed}$$

$$\mathit{BOILER\_MODE} ::= \mathit{initializing} \mid \mathit{normal} \mid \mathit{rescue} \mid \mathit{emergency}$$

$$\mathit{MESSAGE} ::= \mathit{alarm} \mid \mathit{normal\_mode} \mid \mathit{initializing\_mode} \mid \mathit{rescue\_mode}$$

$$\mathit{SENSOR\_STATE} ::= \mathit{working} \mid \mathit{failed}$$

$$\mathit{YesNo} ::= \mathit{yes} \mid \mathit{no}$$

The global system state must be defined in such a way that it is satisfied in all operating modes. Only in the rescue and normal modes, requirements concerning the water-level can be stated. The water valve may only be open in the initializing or emergency mode.

$$
\begin{array}{|l}
\hline
\mathit{SteamBoiler} \\
\hline
\mathit{mode} : \mathit{BOILER\_MODE} \\
\mathit{measured\_waterlevel} : \mathbb{N} \\
\mathit{computed\_waterlevel} : \mathbb{N} \\
\mathit{pump} : \mathit{PUMP\_STATE} \\
\mathit{water\_valve} : \mathit{WATER\_VALVE\_STATE} \\
\hline
(\mathit{mode} = \mathit{normal} \Rightarrow \\
\quad \mathit{measured\_waterlevel} \in \mathit{Safe} \land \\
\quad \mathit{diff}(\mathit{measured\_waterlevel}, \mathit{computed\_waterlevel}) < \mathit{tolerance}) \\
\mathit{mode} = \mathit{rescue} \Rightarrow \mathit{computed\_waterlevel} \in \mathit{Safe} \\
\mathit{water\_valve} = \mathit{open} \Rightarrow \mathit{mode} \in \{\mathit{initializing}, \mathit{emergency}\} \\
\hline
\end{array}
$$

When the steam boiler is installed it will be empty and in initializing mode. The decoration "′" of variable names means that they describe the state *after* an operation is completed. Plain variables describe the state in which an operation is started.

*InitSteamBoiler*

$SteamBoiler'$

---

$mode' = initializing$
$measured\_waterlevel' = 0$
$computed\_waterlevel' = 0$
$pump' = off$
$water\_valve' = closed$

---

The initial state fulfills the state invariant because all of the above implications are vacuously true.

## 4.2 Step 2: Operations of the Steam Boiler Software

We define a schema for each operating mode of the control software. The modes are modeled as internal operations that define what happens in the boiler within one interval between consecutive executions of the control operation. In other words, in each control cycle, exactly one of the operations *NormalMode*, *RescueMode* or *Initialize* is performed. This may involve entering the *EmergencyMode*. For each of these modes we define a schema, give the necessary explanations and informally perform a precondition analysis (proof obligation 2).

We start with the emergency mode because this mode is used by the other ones. It raises an alarm and leaves the water valve as it is. The other components of the state are not specified because the emergency mode cannot only be entered by the overall control procedure (see Step 3) but also from the other internal modes. The notation "$\Delta$*SteamBoiler*" means that the state of the boiler may change. For more details, see Appendix A. By Z convention, inputs are decorated with "?", and outputs are decorated with "!".

*EmergencyMode*

$\Delta SteamBoiler$
$m! : MESSAGE$

---

$mode' = emergency$
$m! = alarm$
$water\_valve' = water\_valve$

---

This schema has precondition *true*. It produces a legal state because in the emergency mode nothing can be guaranteed anyway.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NormalMode}} \\
\Delta \mathit{SteamBoiler} \\
\mathit{steam\_out}? : \mathbb{N} \\
\mathit{water\_measure}? : \mathbb{N} \\
m! : \mathit{MESSAGE} \\
\hline
\mathit{measured\_waterlevel}' = \mathit{water\_measure}? \\
(\mathit{pump} = \mathit{on} \Rightarrow \\
\qquad \mathit{computed\_waterlevel}' = \mathit{measured\_waterlevel} + \mathit{pump\_capacity} - \mathit{steam\_out}?^{1}) \\
\mathit{pump} = \mathit{off} \Rightarrow \mathit{computed\_waterlevel}' = \mathit{measured\_waterlevel} - \mathit{steam\_out}? \\
\mathit{water\_measure}? < \mathit{normal\_min} \Rightarrow \mathit{pump}' = \mathit{on} \\
\mathit{water\_measure}? > \mathit{normal\_max} \Rightarrow \mathit{pump}' = \mathit{off} \\
\mathit{water\_measure}? \in \mathit{Stable} \Rightarrow \mathit{pump}' = \mathit{pump} \\
\mathit{diff}(\mathit{measured\_waterlevel}', \mathit{computed\_waterlevel}') \geq \mathit{tolerance} \vee \\
\qquad \mathit{water\_measure}? \notin \mathit{Safe} \Rightarrow \mathit{EmergencyMode} \\
\mathit{diff}(\mathit{measured\_waterlevel}', \mathit{computed\_waterlevel}') < \mathit{tolerance} \wedge \\
\qquad \mathit{water\_measure}? \in \mathit{Safe} \Rightarrow \mathit{mode}' = \mathit{normal} \wedge m! = \mathit{normal\_mode} \\
\hline
\end{array}$$

The amount of steam taken from the boiler and the measured water-level are inputs to *NormalMode*. The internal variable *measured_waterlevel* is updated accordingly. The new computed water level depends on the state of the pump. It does not use the old computed water-level because this could lead to a divergence between the computed and the measured water-level on the long run (the pump and the sensors will work only with a certain tolerance). Hence, the tolerance is used to detect sudden leakages, a failure of the pump or an undetected failure of the steam sensor. The new pump state is determined according to the measured water-level. If the measured water-level is outside the safety limits or the tolerance is exceeded the emergency mode is entered, i.e. the new mode is *emergency*, the message *alarm* is output, and the water valve is kept as it is. Otherwise, the system stays in normal mode.

Note that in this and the following schema the variable *water_valve* is not mentioned. This is possible because from the global state invariant it follows that the water valve must be closed when the system is in normal or rescue mode.

There is no precondition to the *NormalMode* operation. It always leads to a legal state because the normal mode is only entered or kept ($\mathit{mode}' = \mathit{normal}$) when the corresponding conditions of the state invariant are satisfied. If the emergency mode is entered, every state is legal.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RescueMode}} \\
\Delta \mathit{SteamBoiler} \\
\mathit{steam\_out}? : \mathbb{N} \\
m! : \mathit{MESSAGE} \\
\hline
(\mathit{pump} = \mathit{on} \Rightarrow \\
\qquad \mathit{computed\_waterlevel}' = \mathit{computed\_waterlevel} + \mathit{pump\_capacity} - \mathit{steam\_out}?) \\
\mathit{pump} = \mathit{off} \Rightarrow \mathit{computed\_waterlevel}' = \mathit{computed\_waterlevel} - \mathit{steam\_out}? \\
\mathit{measured\_waterlevel}' = \mathit{computed\_waterlevel}' \\
\mathit{computed\_waterlevel}' < \mathit{normal\_min} \Rightarrow \mathit{pump}' = \mathit{on} \\
\mathit{computed\_waterlevel}' > \mathit{normal\_max} \Rightarrow \mathit{pump}' = \mathit{off} \\
\mathit{computed\_waterlevel}' \in \mathit{Stable} \Rightarrow \mathit{pump}' = \mathit{pump} \\
\mathit{computed\_waterlevel}' \in \mathit{Safe} \Rightarrow \mathit{mode}' = \mathit{rescue} \wedge m! = \mathit{rescue\_mode} \\
\mathit{computed\_waterlevel}' \notin \mathit{Safe} \Rightarrow \mathit{EmergencyMode} \\
\hline
\end{array}$$

Since this mode is only entered when the water-level sensor fails there is no corresponding input, and the value of the variable *measured_waterlevel* is kept equal to *computed_waterlevel*. It

[1]This computation of the new water-level is not completely realistic because one liter of water is not equivalent to one liter of steam and because we assume accumulated values for *steam_out*? and *pump_capacity* instead of using integrals. For a more realistic modeling, an appropriate function should be defined. We refrain from defining this function because the purpose of this paper is primarily to explain our methodology.

is necessary to keep a reasonable value of *measured_waterlevel*, because this value is needed when the water-level sensor is repaired and the system switches back to normal mode.

This operation transforms legal states into legal states because the rescue mode is only entered or kept when the corresponding requirement of the state invariant is fulfilled. Hence, its precondition is *true*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Initialize}} \\
\hline
\Delta \mathit{SteamBoiler} \\
\mathit{steam\_out}? : \mathbb{N} \\
\mathit{water\_measure}? : \mathbb{N} \\
m! : \mathit{MESSAGE} \\
\hline
\mathit{steam\_out}? > 0 \Rightarrow \mathit{EmergencyMode} \\
(\mathit{steam\_out}? = 0 \Rightarrow \\
\quad (\mathit{measured\_waterlevel}' = \mathit{water\_measure}? \land \\
\quad \mathit{computed\_waterlevel}' = \mathit{water\_measure}? \land \\
\quad (\mathit{water\_measure}? > \mathit{safe\_max} \Rightarrow \\
\qquad \mathit{water\_valve}' = \mathit{open} \land \\
\qquad \mathit{pump}' = \mathit{off} \land \\
\qquad \mathit{mode}' = \mathit{initializing} \land \\
\qquad m! = \mathit{initializing\_mode}) \land \\
\quad (\mathit{water\_measure}? < \mathit{safe\_min} \Rightarrow \\
\qquad \mathit{water\_valve}' = \mathit{closed} \land \\
\qquad \mathit{pump}' = \mathit{on} \land \\
\qquad \mathit{mode}' = \mathit{initializing} \land \\
\qquad m! = \mathit{initializing\_mode}) \land \\
\quad (\mathit{water\_measure}? \in \mathit{Safe} \Rightarrow \\
\qquad \mathit{water\_valve}' = \mathit{closed} \land \\
\qquad \mathit{pump}' = \mathit{pump} \land \\
\qquad \mathit{mode}' = \mathit{normal} \land \\
\qquad m! = \mathit{normal\_mode}))) \\
\hline
\end{array}
$$

Initializing the system only works if no steam leaves the boiler. No computed water-level is maintained (i.e. $\mathit{computed\_waterlevel}' = \mathit{water\_measure}?$) because it is not specified how much water leaves the boiler when the water valve is open. We require both sensors to work in initialization mode (see Section 4.3). Thus, it can be guaranteed that *computed_waterlevel* contains a reasonable value.

If the water-level is above the safety limit *safe_max* the water valve must be opened or kept open. If the water-level is below the safety limit *safe_min* the pump must be on. In both cases, the system stays in the initializing mode. As soon as a safe water-level is reached the normal mode is entered. The water valve is only open in the initializing mode, and the normal mode is only entered when the corresponding conditions are fulfilled. Hence, *Initialize* always produces a legal state (precondition *true*).

## 4.3 Step 3: Interface with the Outside World

The schema *Interface* specifies the global control function of the system. According to the operating mode and values of the sensors, the appropriate operation is called. The button that can be pressed in order to initialize the system is modeled by a binary input variable *initialize*?.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Interface}} \\ \hline
\Delta \mathit{SteamBoiler} \\
\mathit{water\_measure}? : \mathbb{N} \\
\mathit{water\_sensor\_state}? : \mathit{SENSOR\_STATE} \\
\mathit{steam\_out}? : \mathbb{N} \\
\mathit{steam\_sensor\_state}? : \mathit{SENSOR\_STATE} \\
\mathit{initialize}? : \mathit{YesNo} \\
m! : \mathit{MESSAGE} \\ \hline
\mathit{mode} \neq \mathit{emergency} \\
\mathit{steam\_out}? > \mathit{steam\_capacity} \lor \mathit{steam\_sensor\_state}? = \mathit{failed} \Rightarrow \mathit{EmergencyMode} \\
(\mathit{steam\_out}? \leq \mathit{steam\_capacity} \land \mathit{steam\_sensor\_state}? = \mathit{working} \Rightarrow \\
\quad (\mathit{initialize}? = \mathit{yes} \lor \mathit{mode} = \mathit{initializing} \Rightarrow \\
\qquad ((\mathit{water\_sensor\_state}? = \mathit{failed} \Rightarrow \mathit{EmergencyMode}) \land \\
\qquad (\mathit{water\_sensor\_state}? = \mathit{working} \Rightarrow \mathit{Initialize})) \land \\
\quad (\mathit{initialize}? = \mathit{no} \land \mathit{mode} \neq \mathit{initializing} \Rightarrow \\
\qquad ((\mathit{water\_sensor\_state}? = \mathit{failed} \Rightarrow \mathit{RescueMode}) \land \\
\qquad (\mathit{water\_sensor\_state}? = \mathit{working} \Rightarrow \mathit{NormalMode})))))) \\ \hline
\end{array}$$

The schema has a precondition: $\mathit{mode} \neq \mathit{emergency}$. This reflects the requirement that the program has to terminate when a hazardous situation occurs.

Proof obligation 3 is trivial: since no operation has a non-trivial precondition, these cannot be violated by *Interface*. For proof obligation 4, we inspect the predicate part of the schema and see that it is a complete case distinction.

Proof obligation 5, however cannot be shown without a further assumption. As stated before, we must require that *Interface* is called exactly once a time unit. Only then do our water-level computations make sense. This requirement cannot be expressed in Z but must be taken care of by the implementation. A possibility to approximate it in Z would be to define time using a constant *zero* and a function *tic* and specify that *Interface* needs exactly one tic.

What we have shown so far is that the program

$$\mathbf{while}\ \mathit{mode} \neq \mathit{emergency}\ \mathbf{do}\ \mathit{Interface}\ \mathbf{od}$$

maintains the state invariant of *SteamBoiler* if all schemas are correctly implemented.

## 5 Synthesizing the Steam Boiler Software

Step 4 is not necessary for the steam boiler because the specification does not make use of any non-trivial data structures.

### 5.1 Step 5: Translation into IOSS Format

The translation of a Z schema into an IOSS programming problem (see Section 2) proceeds as follows:

- Each input variable (decorated with "?") of the Z schema becomes an input variable of the corresponding problem.
- Each output variable (decorated with "!") of the Z schema becomes a result variable.
- 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.

- 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 variables have to be replaced by plain variables and plain variables have to be replaced by their corresponding state variables.

As an example, we sketch this translation for the *RescueMode* schema:

| | |
|---|---|
| input variables: | $steam\_out?, water\_valve$ |
| result variables: | $m!$, $mode$, $measured\_waterlevel$, $computed\_waterlevel$, $pump$ |
| state variables: | $mode_0$, $measured\_waterlevel_0$, $computed\_waterlevel_0$, $pump_0$ |
| precondition: | $mode = mode_0 \wedge measured\_waterlevel = measured\_waterlevel_0$ |
| | $\wedge\ computed\_waterlevel = computed\_waterlevel_0 \wedge pump = pump_0$ |
| invariant: | see *SteamBoiler* |
| goal: | $pump_0 = on \Rightarrow$ |
| | $computed\_waterlevel = computed\_waterlevel_0$ |
| | $+\, pump\_capacity - steam\_out? \ldots$ |

The goal of the programming problem is obtained from the predicate part of *RescueMode*, where each $x$ is replaced by $x_0$ and each $x'$ is replaced by $x$. Moreover, expressions of the form $x \in min \mathinner{..} max$ must be replaced by $min \leq x \wedge x \leq max$, and *EmergencyMode* must be replaced by the equations $mode' = emergency \wedge m! = alarm$. Since from *RescueMode* it follows that the value of $water\_valve$ does not change, $water\_valve$ is an input variable and an equation like $water\_valve = water\_valve_0$ does not need to be included in the goal.

## 5.2 Step 6: Synthesizing a Program for *RescueMode*

The program implementing the schema *RescueMode* can be derived semi-automatically. We assume that the pump can be switched on by setting the variable *pump* to *on* and switched off by setting *pump* to *off*[2]. The basic idea is that the new values of the state 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 $computed\_waterlevel$, establishing the first three lines of the postcondition (the line numbers refer to the predicate part of the *RescueMode* schema). The new value of $measured\_waterlevel$ (line 4) is set in $p_2$, and the new value for *pump* is determined in $p_3$, establishing lines 5 – 7 of the postcondition. Finally, $p_4$ determines the new value of *mode* and the output $m!$, establishing the last two lines of the postcondition.

To develop $p_1$, $p_3$ and $p_4$, we use a strategy called *disjunctive conditional.* This strategy applies 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 first two conjunctions are equivalent to a disjunction $(pump_0 = on \wedge computed\_waterlevel = \ldots) \vee (pump_0 = off \wedge computed\_waterlevel = \ldots)$.

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 our case, the test is $pump = on$ (for inclusion in the

[2]If more sophisticated procedures are needed, the assignments to *pump* can be replaced by procedure calls.

program, the state variables are automatically replaced by the corresponding result variables). Since the conclusions of the implications are equations, the *automatic assignment* strategy can automatically generate assignments establishing the equations. This strategy is also used to generate $p_2$.

The final result of the synthesis process is shown in Figure 3, where the components of the global state are modeled as global variables, and inputs and outputs are modeled as parameters.

```
proc rescue_mode(steam_out? : nat; m! : message)
do  if pump = on
    then computed_waterlevel := computed_waterlevel + pump_capacity - steam_out?
    else computed_waterlevel := computed_waterlevel - steam_out?
    fi;
    measured_waterlevel := computed_waterlevel
    if computed_waterlevel < normal_min
    then pump := on
    else if computed_waterlevel > normal_max
         then pump := off
         fi
    fi;
    if safe_min <= computed_waterlevel and computed_waterlevel <= safe_max
    then mode := rescue;
         m! := rescue_mode
    else mode := emergency;
         m! := alarm
    fi
end
```

Figure 3: Program Synthesized for Rescue Mode

## 6 Discussion

Now that our approach is presented in some detail, we can relate it to other work in the field, compare software safety with correctness and reliability, and finally discuss its merits as well as its drawbacks.

### 6.1 Related Work

Our choice of Z for the specification of safety-critical systems is not completely out of the way, as a look at the literature shows. Several case studies have been performed using the specification language VDM [Jon90], e.g. the British government regulations for storing explosives [MS93], a railway interlocking system [Han94], and a water-level monitoring system similar to the one presented in the present paper, see [Wil94]. VDM and Z are based on similar concepts and have the same expressive power (and weaknesses). 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 safe software.

Williams [Wil94] uses a problem like the one tackled here to assess safety specifications. His conclusions are:

1. " Methods used for the development of safety-critical systems should have well-defined criteria for ensuring the specification's completeness and consistency."
2. "The use of theorem proving is not limited to the verification of refinement steps. ..."
3. "Reviews can be an effective means of detecting errors in formal specifications."
4. "A formal statement of the safety requirements should be a part of the formal system specification. ..."
5. "The use of CASE tools can help eliminate simple syntactic errors in model-based specifications. ..."

Our approach fulfills most of these requirements. The completeness criterion is expressed in proof obligation 4, consistency is taken care of by proof obligations 1 – 3. The proof obligations introduced by our approach exceed the ones occurring in refinement steps. Reviews are not an explicit part of our process model but of course they are encouraged. According to Step 1, the fourth requirement is also fulfilled. Finally, we used the fuzz checker [Spi92a] to check all of the specifications contained in this paper, in order to eliminate simple syntactic errors.

The goals pursued by Halang and Krämer [HK94] are similar to ours. They present a development process, starting with the formalization of requirements and ending with the testing of the constructed program. Their focus is on programmable logic controllers. 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.

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. The transition from an abstract top-level specification to a detailed specification suitable as a basis for program development is done by stepwise refinement. This activity is covered by Step 4 of our approach. Moser and Melliar-Smith use a reliability model for the processors that execute the program. This enables them to take computer failures into account, an aspect not covered in our work. On the other hand, they do not consider the validation of the top-level specification, an issue that is of much importance for us, see the proof obligations of Steps 1–3.

## 6.2 Relation to Correctness and Reliability

In general, safety, correctness and reliability share the goal to make software more dependable. In detail, however, they have to be distinguished carefully.

**Safety vs. Correctness.** One might consider safety a weaker requirement than correctness. Leveson [Lev86] states "We assume that, by definition, the correct states are safe." The example of the steam boiler, however, shows that this is true only after the legal states have been defined with safety requirements in mind. The introduction of the tolerance between the measured and computed water-level or the additional sensors in the *Interface* schema would not be necessary if only the correctness of the program were of interest. As already stated in Section 1, in this case we would only have to guarantee that the input is correctly processed, i.e. that the pump is switched on or off when an intervention point is reached. Whether the value yielded by the sensor is credible would not be checked because correctness is a relation solely between a specification and a program. Hardware failures are of no interest in correctness considerations. Hence, we think that the development of safe software has to proceed differently: hardware failures must explicitly be modeled. This difference is not of a *technical*, but of a *pragmatic* nature.

**Safety vs. Reliability.** The case study shows that reliability and safety can be conflicting goals (see also [Lev86]). Were safety the most important goal for the operation of the steam boiler, something like the rescue mode would not be permitted. The danger that the computed water-level is not accurate is not to be neglected. The only purpose of the rescue mode is to enhance reliability, but this certainly results in a compromise concerning safety.

## 6.3 Assessment of the Approach

We conclude with a summary of the merits and drawbacks or 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, if the sensors yield false values, the system can enter a non-safe state because the software controls the system according to the sensor values. This limitation cannot be overcome by means concerning the software alone. Instead, fault tolerance methods like redundancy have to be applied.

Moreover, it is not possible to deal with absolute time measures in the formalisms we have chosen. If it is, e.g., necessary that a component reacts within 2 ms, then this cannot be guaranteed with our approach. The maximum execution time of the specified operations cannot be specified in Z. Classical complexity analysis and testing should be applied after the program is developed. Finally, our formalisms are not suitable to develop distributed or parallel systems.

As a result, the kind of safety our approach can guarantee is relative. 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. Here, traditional methods like testing are indispensable.

**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. If, however, some aspect of the system does not occur at all in the safety invariant developed in Step 1, it can be guaranteed not to be safety-related. An example can be found in [Hei95].

**Contributions.** Our approach provides a process model for the development of provably safe software. Its contributions are the following:

- A detailed guidance for developers of safe software is provided, complemented by clear and explicit proof obligations.
- The approach can easily be introduced and applied in an organization because it relies on well established techniques and tools.
- The steps of the approach concerned with safety are clearly identified.

- Not only the specification but also the implementation of safety-critical systems is covered.

**Acknowledgment** Many thanks to Thomas Santen and Carsten Sühl for stimulating discussions on the topic and comments on this work.

## A Appendix: Explanation of the Z Notation

The notation used in the paper is explained in the order of appearance.

Global declarations are made by so-called axiomatic boxes. Such a box has a vertical line on the left-hand side. Conditions constraining the values of the declared entities can be stated below a horizontal line.

The natural numbers are denoted $\mathbb{N}$, the positive natural numbers $\mathbb{N}_1$. The symbol $==$ allows one to define abbreviations, e.g. for subtypes of the natural numbers (denoted by "..") or functions, where the arguments of the function are given after the $\lambda$ symbol. Enumeration types can be defined using "::=".

Schemas are used to define the global system state as well as the operations changing this state. The global system state consists of a number of declarations introducing its components and a global system invariant constraining the components and their relations. Unless indicated otherwise (by another connective and parentheses), a new line in the predicate part of a schema or an axiomatic box means that the *conjunction* of the predicates is required.

Operations start in some state and usually terminate in a different state. Since Z is a mathematical notation and not a programming language, assignments are not possible. The state components in the state after completion of the operation must have different names than the components in the state where the operation starts. The convention is that plain variables denote the "before" state, whereas variables decorated with "$'$" denote the "after" state.

It is part of the Z method to give an initial state. This state is defined exclusively with decorated variables because only the state *after* completion of the initialization is defined.

Schemas defining operations usually make use of the delta convention. Writing "$\Delta$*SteamBoiler*" means that all variables of the *SteamBoiler* schema are declared, as well as their decorated versions. Moreover, the state invariant is required to hold for the plain as well as for the decorated versions of the state components. This means, the operation must be started in a legal state and must also end up in a legal state. The delta convention is just an abbreviation for a more complicated schema.

When a schema is imported in the *declaration part* of another schema, this means that all its variables are declared and the predicate part is required to hold. When a schema is imported in the *predicate part* of another schema (like e.g. *EmergencyMode* in *NormalMode*) this means that its variables must already be declared in the importing schema and that the predicate part of the imported schema is required to hold. In this case, the schema is used as a predicate. Again, importing the schema is just an abbreviation.

## References

[Abr94] Jean-Raymond Abrial. Steam-boiler control specification problem. Specification problem suggested to the participants of the Dagstuhl meeting "Methods for Semantics and Specification", June 1995, 1994.

[BG94] J. Bowen and M. Gordon. Z and HOL. In *Z User Workshop*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

[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.

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

[Hei95] Maritta Heisel. Six steps towards provably safe software. In G. Rabe, editor, *Proceedings of the 14th International Conference on Computer Safety, Reliablity and Security (SAFECOMP), Belgirate, Italy*, pages 191–205, London, 1995. Springer.

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

[HSZ95a] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. A generic system architecture of strategy-based software development. Technical Report 95-8, Technical University of Berlin, 1995.

[HSZ95b] 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*, number 989 in Springer LNCS, pages 272–293, 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.

[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.

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

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[Spi92a] J. M. Spivey. The fuzz manual. Computing Science Consultancy, Oxford, 1992.

[Spi92b] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.

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

[Woo91a] J.C.P. Woodcock. An introduction to refinement in Z. In S. Prehm and W.J. Toetenel, editors, *Proc. 4-th International Symposium of VDM Europe, Vol. 2*, LNCS 552, pages 96–117. Springer-Verlag, 1991.

[Woo91b] 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.