Six Steps Towards 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 an approach to the specification and implementation of provably safe software. It uses well-established tools and techniques that are usually employed to ensure correctness, rather than safety, of software. The approach comprises six steps, each of which is complemented by some proof obligations. For each step, the safety-related aspects are clearly elaborated. Thus, designers of safety-critical systems are given guidance that helps to avoid potentially dangerous gaps in the specification of the system's safety properties.

1 The General Setting

The aim of this work is to support the development of provably safe software. Since a safety proof cannot be obtained by conventional software engineering techniques, we use formal methods to achieve this goal. Formal methods as they are used today mostly have the sole purpose of guaranteeing the *correctness* of software. This means, the software is to implement a certain functionality. Software *safety*, on the other hand, is not so much concerned with the implemented functionality. Instead, it must be guaranteed that certain undesirable states are *not* entered. Moreover, the interaction of the system with its environment plays a crucial role.

Our approach covers the specification as well as the implementation of safety-critical software. As a specification language, we have chosen the model-based language Z [Spi92b]. In Z, system states are modeled explicitly. This is in accordance with the fact that most embedded safety-critical systems have a state. For the implementation of specifications, the program synthesis system IOSS (Integrated Open Synthesis System) designed by the author [HSZ95b] is used. IOSS supports the implementation of imperative programs and thus matches well with Z.

The choice of these formalisms, however, imposes some limitations on our approach: distributed systems, parallelism and real-time requirements cannot

be treated in full generality[1] because Z has no means to express the corresponding notions.

The approach consists of six steps to be performed, each of which comes with some proof obligations. For each step, its safety-related aspects are highlighted. Their description can serve as a checklist, thus providing guidance for the designers of safety-critical systems.

In the next section, Z and IOSS are introduced. Then the steps of our approach are explained in some detail, followed by an example. Finally, related work is discussed and an assessment of the approach is given.

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 supports the development of imperative programs using so-called *strategies*, [Hei94, HSZ95b]. 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 if 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.

Programming problems are basically specifications, expressed as pre- and postconditions of first-order predicate logic. A complete definition is given in Section 4.2.1. Solutions are basically programs in a Pascal-like language. A solution is *acceptable* if and only if the program is totally correct with respect to the specification and additionally fulfills some variable conditions (see Section 4.2.1). For each developed program a formal proof in dynamic logic [Gol82] is constructed. This is a logic designed to prove properties of imperative programs. The proofs are represented as tree structures that can be inspected at any time during development.

Program synthesis with IOSS consists of a loop of strategy applications. The intermediate states of the development are represented by a data structure

[1] Simple time constraints can be modeled with timers and thus be specified in Z, see Section 4.

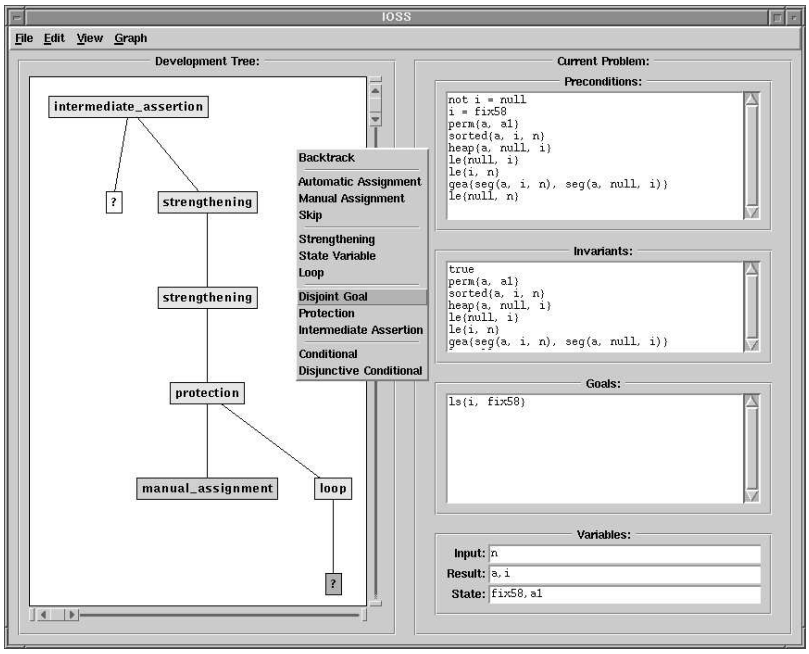

Figure 1: The IOSS interface

called *development tree*. Its nodes contain a problem and its solution (once it has been found). Representing the state of development as a data structure makes it possible to obtain an overview of the development at any time. 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. The result of the development process is the final development tree.

The strategy base of IOSS contains formalized development knowledge in form of strategy modules. A number of interactive, semi-automatic and fully automatic strategies have been implemented. In the current version, they are oriented on programming language constructs. In the near future, higher level strategies, e.g. for the development of divide-and-conquer algorithms or reusable procedures, will be built in. A complete description of the available strategies can be found in [Hei94].

Figure 1 shows the general interface of IOSS. The main window displays the development task, represented by the development tree – on the left-hand side of the window – and the specification of the current problem – on the right-hand side of the window. The tree visualizes the process and the state of development. Each node is labeled with the name of the strategy applied

to it. The state of the node is color coded, showing at a glance whether it is reducible, or solved, etc. The strategy menu is shown in the center of the window. Applications of strategies, inspection of nodes or the proof tree and graph manipulations like scaling are performed via mouse clicks or pull-down menus. For a more complete description of IOSS, the reader is referred to [HSZ95a, HSZ95b].

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 are described in more detail in Section 4.2.

3 The Six Steps in Detail

Table 1 gives an overview of the proposed procedure. 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 correct (and thus safe) program.

Step 1 The definition of the legal states 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 obtained. 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 states are 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. In showing that the initial state is legal, we also demonstrate that the requirements for legal states are satisfiable.

Step 2 The actions of the system 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. The analysis of the conditions under which the actions transform legal states into legal states 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, it must be taken care that the operation is only executed when its precondition holds.

No.	Step	Proof Obligations
1	Define the legal states of the system.	Show that the initial state is legal.
2	Define the actions the system can perform.	Analyze the conditions under which the actions transform legal states into legal states.
3	Define the interface of the system to the outside world.	Show that the internal system operations are only invoked if their preconditions are satisfied. Show that for each combination of sensor values exactly one internal operation is invoked. Show that – if the sensors work correctly – the system faithfully represents the state of its environment.
4	Refine the data and operations of the specification until data and control structures of the target programming language can be used.	Show the correctness of the refinements.
5	Transform the specification obtained in Step 5 into a form suitable for the program synthesis system.	Show the correctness of the algorithm performing this task.
6	Use the synthesis system to obtain a proven correct implementation of the specified system.	Proof obligations are generated by the synthesis system.

Table 1: Steps and Proof Obligations

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 sensor values.

Step 3 In order to define the interface between the system and the outside world, 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 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 care of and yields a clear

and comprehensible interface. It is not strictly necessary to show that for each combination of sensor values exactly one internal operation is invoked. We introduce this proof obligation to encourage developers to design their systems as clear and simple as possible. The other two proof obligations, however, are necessary to ensure the system's safety.

Once step 3 is performed, it is guaranteed that the state internally maintained by the software always fulfills the safety requirements and that this state is consistent with the state of the environment, under the condition that failure of sensors can be detected. It follows that (under the same condition) also the "real" system state is safe, provided the implementation of the software 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. They are not presented in so much detail because they follow a methodology that is common for the application of formal methods.

Step 4 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. On the other hand, it is also possible that several refinement steps are necessary.

Step 5 The Z specifications are transformed into IOSS programming problems, as described in Section 4.2.

Step 6 The program synthesis guarantees that the concrete states of the implementation are always safe, provided the abstract states of the system model are.

4 Example: A Microwave Oven

We exemplify our approach with a simple microwave oven. The description of the oven (which is taken from [SM92]) is as follows:

1. There is a single control button available for the user of the oven. If the oven door is closed and you push the button, the oven will cook (that is, energize the power tube) for 1 minute.

2. If you push the button at any time when the oven is cooking, you get an additional minute of cooking time.
3. Pushing the button when the door is open has no effect.
4. There is a light inside the oven. Any time the oven is cooking, the light must be turned on. Any time the door is open, the light must be on.
5. You can stop the cooking by opening the door.
6. If you close the door, the light goes out. This is the normal configuration when someone has just placed food inside the oven but has not yet pushed the control button.
7. If the oven times out (cooks until the desired preset time), it turns off both the power tube and the light. It then emits a warning beep to tell you that the food is ready.

4.1 Specification

Two hazardous situations can be identified for the microwave oven. (i) If the power tube is on while the door is open, there is a severe risk of human injury. (ii) If the light is off while the power tube is on, a boiling over of food may remain unnoticed and can cause a damage of the oven or even set it on fire. Requirement (i) is certainly more important than (ii). We will come back to this in Section 5.3.

Step 1: Define the legal states of the system. The interesting components of the microwave oven can take on two possible states.

$$MICROWAVE\_STATE ::= energized \mid de\_energized$$
$$LIGHT\_STATE ::= on \mid off$$
$$DOOR\_STATE ::= open \mid closed$$
$$TIMER\_STATE ::= running \mid halted$$
$$BEEPER\_STATE ::= silent \mid beeping$$

The global state of the oven must reflect the safety requirements which are expressed in the first two lines of the state invariant. The other predicates of the following state schema reflect the natural language description given above.

MicrowaveOven

$$power\_tube : MICROWAVE\_STATE$$
$$light : LIGHT\_STATE$$
$$door : DOOR\_STATE$$
$$timer : TIMER\_STATE$$
$$timer\_value : \mathbb{N}$$
$$beeper : BEEPER\_STATE$$

$$door = open \Rightarrow power\_tube = de\_energized$$
$$power\_tube = energized \Rightarrow light = on$$
$$door = open \Rightarrow light = on$$
$$door = closed \land power\_tube = de\_energized \Rightarrow light = off$$
$$power\_tube = energized \Leftrightarrow timer = running$$
$$timer\_value \neq 0 \Rightarrow beeper = silent$$

The initial state describes the microwave oven as you can buy it in a store. It fulfills the state invariant. 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.

MicrowaveOvenInit

$$MicrowaveOven'$$

$$power\_tube' = de\_energized$$
$$light' = off$$
$$door' = closed$$
$$timer' = halted$$
$$timer\_value' = 0$$
$$beeper' = silent$$

Step 2: Define the actions the system can perform. As a user of the oven, you can open and close its door and push the control button. Moreover, there are state-changing operations that are only indirectly invoked by the user. These have to do with the behavior of the timer.

OpenDoor

$$\Delta MicrowaveOven$$

$$door = closed$$
$$power\_tube' = de\_energized$$
$$light' = on$$
$$door' = open$$
$$timer' = halted$$
$$timer\_value' = 0$$
$$beeper' = silent$$

This operation may only be invoked when the door is closed (precondition $door = closed$). It then leads to a legal state[2]. "$\Delta MicrowaveOven$" means that the state of the oven may change. The operation *CloseDoor* is defined analogously, with precondition $door = open$.

For the control button, we have to distinguish whether it is pushed when the door is open or when the door is closed.

[2]This holds for the other operations, too. Therefore, we will not mention this any more in the following.

PressButtonDoorClosed

$\Delta MicrowaveOven$

$door = closed$
$power\_tube' = energized$
$light' = on$
$door' = door$
$timer' = running$
$timer\_value' = timer\_value + 60$
$beeper' = silent$

PressButtonDoorOpen

$\Xi MicrowaveOven$

$door = open$

The schema on the right-hand side specifies that the state of the oven does not change when the button is pushed while the door is open. When the button is pressed, one of the two above operations will be invoked:

$$PressButton \mathrel{\widehat{=}} PressButtonDoorClosed \lor PressButtonDoorOpen$$

The combined operation has the precondition *true*, because the door must be either closed or open, according to the definition of $DOOR\_STATE$.

The timer can either be running or halted or get a timeout. In the first case, just the time value is decreased (precondition: $timer = running \land timer\_value > 0$). In the second case, nothing happens (precondition: $timer = halted$). The third case occurs when the timer is running and reaches the value 0 (precondition: $timer = running \land timer\_value = 0$).

TimerRuns

$\Delta MicrowaveOven$

$timer = running$
$timer\_value > 0$
$power\_tube' = power\_tube$
$light' = light$
$door' = door$
$timer' = timer$
$timer\_value' = timer\_value - 1$
$beeper' = beeper$

TimeOut

$\Delta MicrowaveOven$

$timer = running$
$timer\_value = 0$
$power\_tube' = de\_energized$
$light' = off$
$door' = closed$
$timer' = halted$
$timer\_value' = timer\_value$
$beeper' = beeping$

TimerHalted

$\Xi MicrowaveOven$

$timer = halted$

When the timer is timed out, the beeper starts beeping. In the natural language description, nothing was said about how long the beeper should beep. For simplicity, we decide not to define an extra operation that switches off the beeper but make use of the fact that opening the door does the job.

Again, these cases are combined to form the operation *Timer* with precondition *true*.

$$Timer \mathrel{\widehat{=}} TimerRuns \lor TimeOut \lor TimerHalted$$

The next operation is only needed because not only the correct functioning but also the safety of the microwave oven are of interest: when something unforeseen happens, the oven must enter a safe state. Of course, this operation has no precondition.

EmergencyShutdown

$\Delta MicrowaveOven$

$power\_tube' = de\_energized$
$light' = off$
$door' = door$
$timer' = halted$
$timer\_value' = 0$
$beeper' = silent$

Step 3: Define the interface of the system to the outside world. The connection of the internal system state and the environment is modeled by sensors telling if the door is open or closed and if the button is pressed or not. We assume that a failure of the door sensor is detectable.

$$DOOR\_SENSOR ::= door\_open \mid door\_closed \mid failed$$

$$BUTTON\_SENSOR ::= pressed \mid released$$

The sensor values are connected to internal operations via the following schema that has the sensor values as input parameters:

ExternalEvents

$\Delta MicrowaveOven$
$ds? : DOOR\_SENSOR$
$bs? : BUTTON\_SENSOR$

$ds? = failed \Rightarrow EmergencyShutdown$
$ds? = door\_open \land door = closed \Rightarrow OpenDoor$
$ds? = door\_closed \land door = open \Rightarrow CloseDoor$
$bs? = pressed \land (ds? = door\_open \land door = open$
$\quad \lor ds? = door\_closed \land door = closed) \Rightarrow PressButton$
$bs? = released \land (ds? = door\_open \land door = open$
$\quad \lor ds? = door\_closed \land door = closed) \Rightarrow Timer$

This means that a pressed button is ignored if at the same time the door is moved. A door movement is sensed by comparing the sensor value with the internal variable storing the door state. Only if those two are equal the internal

operation *PressButton* is invoked. If neither the door is moved nor the button is pressed, the *Timer* operation is invoked. This operation is deterministic since the preconditions of the disjuncts exclude each other. Hence *ExternalEvents* is also deterministic. For each constellation of the sensors exactly one internal operation is invoked.

4.2 Implementation

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

4.2.1 Step 5: Translation into IOSS Format.

Problems to be solved with IOSS are specifications of programs, expressed as pre- and postconditions that are formulas of 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.

The translation of a Z schema into an IOSS programming problem 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 consider the implementation of the schema *PressButton-DoorClosed*. The above algorithm yields:

input variables:	$door$
result variables:	$power\_tube$, $light$, $timer$, $timer\_value$, $beeper$
state variables:	$timer\_value_0$
precondition:	$door = closed \wedge timer\_value = timer\_value_0$
invariant:	see *MicrowaveOven*
goal:	$power\_tube = energized \wedge light = on \wedge timer = running \wedge$ $timer\_value = timer\_value_0 + 60 \wedge beeper = silent$

4.2.2 Step 6: Synthesis of a Sample Program.

We assume that the the light, the timer and the beeper can be switched on and off by setting the corresponding variables accordingly[3]. The synthesis of a procedure `press_button_door_closed` can then be performed completely automatically, using the *Automatic Assignment* strategy shown in Figure 1, because for each result variable we have an equation in the goal that can be transformed into an assignment statement. Note that *PressButtonDoorClosed* is embedded in the schema *PressButton*. To implement this schema, one develops a conditional (motivated by the "$\vee$", using the *Disjunctive Conditional* strategy): **if** `door = closed` **then** `press_button_door_closed` **else skip fi**, where **skip** is the program that does nothing.

5 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.

5.1 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 [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] assesses 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."

[3] If more sophisticated procedures are needed, the right-hand sides of the assignments can be replaced by calls to the respective procedures.

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 the proof obligation to show that for each combination of sensor values exactly one internal operation is invoked. Consistency is taken care of by the first three proof obligations shown in Table 1. 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, from the formalization of requirements to 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 by this 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.

5.2 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 microwave oven, however, shows that safety concerns have an influence on what is considered a correct state.

To ensure its safety, we defined the schema *EmergencyShutdown* that switches off the microwave as soon as a failure of the door sensor is detected. This situation is not taken into account when only the correctness of the software is of interest because correctness is a relation solely between a specification and a program. Failures of technical equipment are of no interest in correctness considerations. Hence, we think that the development of safe software has to proceed differently: the environment in which the software operates must explicitly be modeled. This difference is not of a technical, but of a pragmatic nature.

Safety vs. Reliability. Our example study shows that reliability and safety can be conflicting goals (see also [Lev86]). Of the safety requirements for the microwave oven, the requirement that the power tube is de-energized when the door is open is certainly more important than the requirement that the light must be on when the power tube is energized. If the light bulb breaks down, it is a reasonable decision not to invoke the "emergency shutdown" but to sacrifice the less important safety requirement to increase availability (and thus reliability) of the oven.

5.3 Assessment of the Approach

We conclude with a summary of the merits and drawbacks of this work.

Limitations. The approach presented here 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 and consistency checks 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, and we are not aware of any formal methods that allow one to *prove* maximum execution time of programs in higher-level languages[4]. 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 safety cannot be guaranteed in the intermediate states that occur during execution. It is up to the system designers and implementors to judge if this is the case. Here, traditional methods like testing are indispensable.

[4] This is true even for formalisms designed to deal with time, like temporal logic or the duration calculus; again, these limitations come from the fact that the formalisms do not consider the hardware on which the programs are executed.

Enhancing the Applicability of the Approach. In contrast to hardware or power failure which are beyond our capabilities, the problem of unsafe 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 like a microwave oven, 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 further reduce cost, one might exclude those parts of the software from the verification process that can be guaranteed to be of no importance for safety.

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 Jan Peleska for stimulating discussions on the topic and comments on this work.

References

[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.

[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 *Proceedings 5-th European Software Engineering Conference*, Springer LNCS, 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.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles – Modeling the World in States*. Yourdon Press, Englewood Cliffs, 1992.

[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.