Modelchecking Non-Functional Requirements for Interface Specifications

Florian Kammüller[1] and Sören Preibusch[2]

[1] Technische Universität Berlin
Fakultät IV: Elektrotechnik und Informatik
Franklinstraße 28-29, 10587 Berlin
flokam@cs.tu-berlin.de
[2] German Institute for Economic Research
Mohrenstraße 58, 10117 Berlin
spreibusch@diw.de

Abstract. In this paper we present a combination of formal specification and mechanical analysis enabling a simple and flexible development process for interface specifications from requirements. Using the potential of temporal logic for describing non-functional requirements we derive an analysis model from functional requirements. Slightly abusing its original object-oriented incentives we employ the precision and modularity of formal specification in Object-Z for representing interface descriptions. A structure preserving translation of Object-Z specifications to the model checker SMV unifies the temporal logic specification of requirements with the analysis model. The automated verification in SMV supports a feedback loop for a stepwise improvement of the requirement specification and its analysis model. We illustrate this technique on the case study of the safety-critical TWIN elevator system.

1 Introduction

One of the major obstacles for the seemless integration of rigorous specification into the development process of embedded systems is the gap between natural language requirements and formal notations. Engineering starts with requirements elicitation from stakeholders, i.e. customers and development team members. Requirements are continuously refined in iterations of the global development process. For the communication between stakeholders natural language is still the ad hoc standard because not everyone necessarily understands mathematics and specialized formal notations. One cannot deny the importance of informal communication channels (such as coffee breaks or grapevine) that substantially rely on natural language (NL). For precision in system specification mathematics and logics are preferable, in particular in safety-critical applications.

Practically, requirements engineering must be performed in an iterated process in order to guarantee systems adequate to the stakeholders' wishes, possibilities, and needs. An ideal development scenario is rapid prototyping, where

requirements are immediately verified against a prototypical system implementation. However, in industrial contexts rapid prototyping, not unlike other agile techniques, like Extreme Programming, never had a chance to catch on. The feedback loops of such processes stretch out too far over distinct phases in the software development process to scale up to large enterprises. They work well only for very small teams with a tight integration from requirements through to implementation and maintenance. Feedback loops that need to reach far back into the early phases are costly and therefore need to be minimized.

In our view, it is the development of valid interface specifications that is the immediate goal for a formalization of requirements. Interfaces implement the idea of locality and information hiding. Eventually, they allow concurrent engineering in distributed teams. Interfaces allow reliability from early stages on and preserve flexibility in implementation up to late stages. During the transition from requirements to system models in the analysis phase of classical software engineering one of the main results is a first description of system interfaces. In object-oriented software development, usually relying on UML, interfaces are initially described by use-cases and class diagrams. Use-cases capture the externally visible functionality. They provide the interface description towards the user. Class diagrams specify internal interfaces between implementation components.

Only in the design phase, much later in the software development process, we add behavioural descriptions to those interfaces when defining system internal interactions, say by interaction diagrams, or collaborations. However, this is much too late for large scale developments: insufficiencies or inconsistencies arise that could have been easily avoided if simple checks had been used in the initial interface model. As such, good interface specifications are paramount to the success of the software development project.

We strongly believe, based on our interaction with industrial partners, that an early verification of requirements is imperative for good developments. Most requirements analysis tools merely support management and tracing of requirements. It is obviously not possible to verify requirements, unless a system model is available. To this end we suggest the use of Object-Z to prototype analysis models during the transition from requirements to interface design. Object-Z is a suitable language as it is sufficiently abstract, yet offers enough structuring facilities to formalize interfaces of system components. Object-Z augments the formal specification language Z thereby entailing full predicate calculus. For the natural expression of requirements we suggest the use of temporal logics. These logics are most suitable for the expression of many non-functional constraints very often including statements about reactive behaviour: non-functional constraints usually involve the system's environment and its reaction to environmental changes.

The *Quality Feedback Loop* that we present in this paper (Figures 1 and 2) produces an interface description in Object-Z that is additionally adorned with temporal logic specification. The system specification in Object-Z is iteratively improved ("specification progress"). Each of these stepwise improvements relies on an execution of the quality feedback loop. The Object-Z specification, present

at a given time $t + 1$ (i.e. in given version) is translated mechanically into an SMV program, that is checked against the requirements expressed in temporal logics. In case an error is detected in the model checking results, it can be traced back to the original Object-Z specification where this error is then eliminated, leading to an improved version $t + 2$.

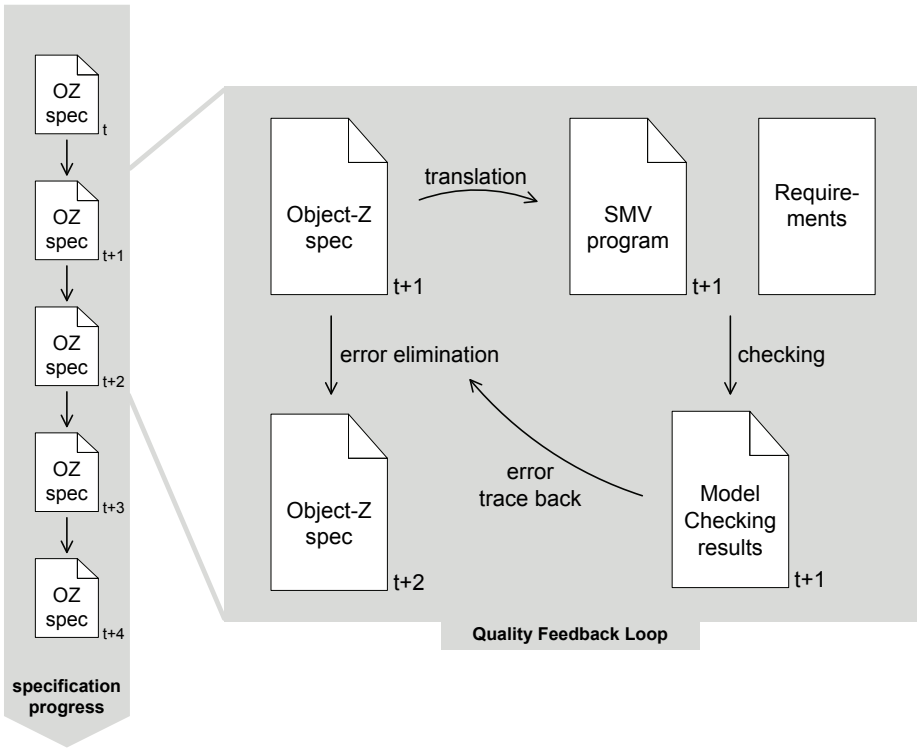

Fig. 1. Quality feedback loop and its integration in the formal model development

The possibility for a cyclic use of our quality feedback loop guarantees that requirements are precise and verified against a formal interface model that is produced as a by-product. The feedback loop is strongly supported by the automated verification with the model checker SMV and the automated translation process between Object-Z and SMV. The latter step is necessary to enable correct verification of requirements on the system model.

The requirements against which the specification is checked derive from a collection and consolidation of formal and natural language requirements (Figure 2). Latter are translated into temporal logic beforehand.

The necessity for a tool-supported quality feedback loop, as presented here, derives from our experiences in our initial TWIN elevator system case study [6].

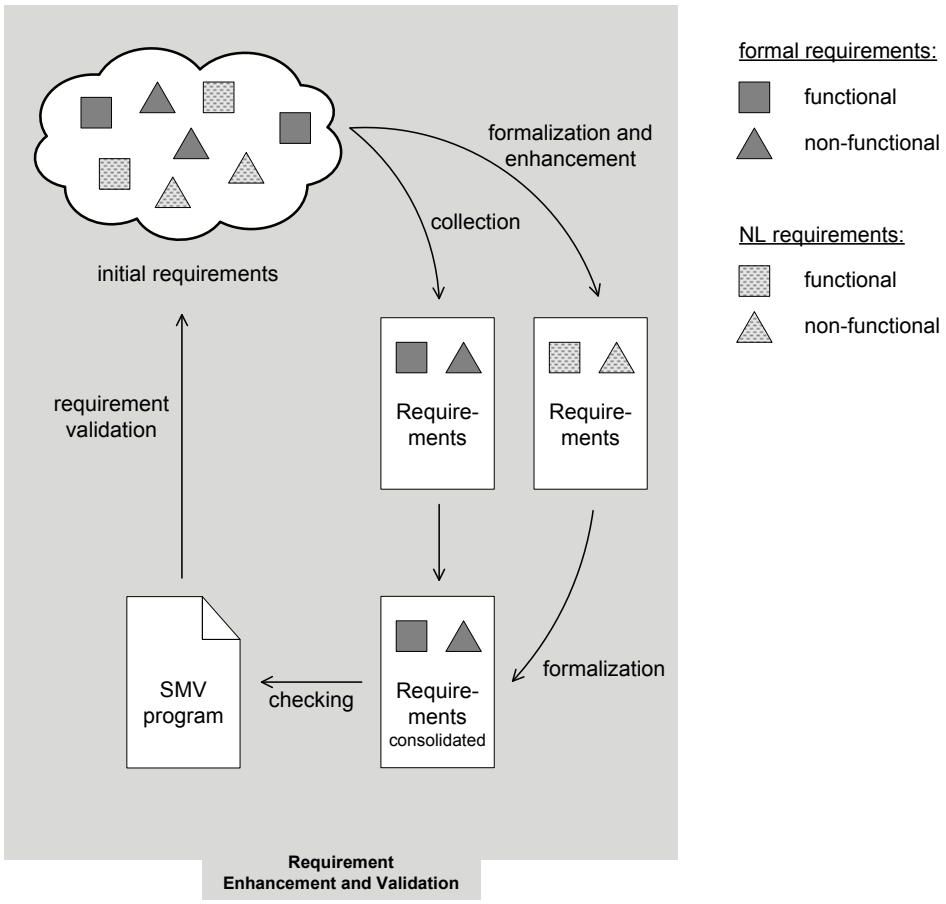

Fig. 2. Quality feedback loop, collection, formalization, and consolidation of requirements.

Though we were able to verify the system's correct behaviour and to demonstrate the scalability of formal verification methods, using SMV as an original specification revealed to be cumbersome and errorprone. The translation from Object-Z to SMV avoids these pitfalls, while the original, manually crafted SMV program remains a benchmark in terms of efficiency.

The idea to modelcheck requirements has already been used by Bharadwaj and Heitmeyer [1]. Their work differs from ours as it is tailored to the software cost reduction model. For the formal specification of interfaces the work by Canal et al. [2] on formal Interface Definition Languages (IDL) using the π-calculus for the definition of component behaviour is relevant. In future work we plan to investigate the alternative use of Canal's formalism in our quality feedback loop.

In this paper we first present an example for a requirements specification of an industrial case study in Section 2. We then summarize the technical setup of our feedback loop by explaining the translation between Object-Z and SMV in Section 3. Finally, we illustrate in Section 4 our quality feedback loop on an excerpt of the TWIN elevator case study.

2 The TWIN Elevator

The idea of having an elevator with two independent cabins operating in the same shaft dates back to the 1930s. However, first attempts to realize this efficient transportation system failed and the engineering of a control system has been an unsolved problem for almost a century. Only in 2002 ThyssenKrupp installed the first TWIN elevator system at Stuttgart University. The history of TWIN proves the importance of computer science as an engineering discipline.

In a TWIN elevator system, two cabins are arranged one above the other; they run independently in the same TWIN shaft – also at different speeds. A safety distance is kept, depending on the speeds involved. The cabins can move in different directions, which means that they can also move towards each other [13]. Because the TWIN cabins cannot sidestep, each TWIN installation comprises at least one conventional shaft to serve routes that would result in a crossing of the TWIN cabins (Fig. 3).

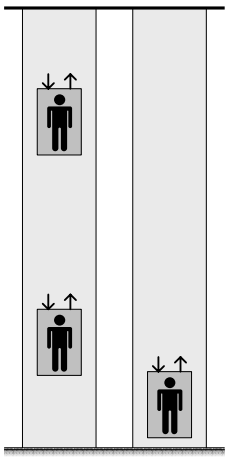

Fig. 3. Minimal TWIN installation (schematic view): a TWIN shaft with two cabins on the left and conventional shaft on the right

A prospective passenger communicates his destination level no longer within the elevator cabin, but instead by Destination Selection Control (DSC) terminals mounted on each floor. The control system then selects one of the cabins capable to serve the call.

The informal specification of safety requirements of ThyssenKrupp is the basis for the formal expression by means of formal specification and model checking.

2.1 Informal Requirements Specification

A system like the TWIN elevator is per se safety critical and requirements are amplified in comparison to traditional elevator systems. Henceforth, a four-level safety concept has been implemented; it constitutes the core of the original safety specification as published by ThyssenKrupp [12]. Four safety levels express an escalating strategy of electronic and mechanical measures targeted to prevent a shaft's two cabins from colliding.

The TWIN-specific requirements are built on top of general safety considerations applying to any multi-storey passenger elevator. We pool latter in the zeroth safety level.

- 0^{th} *level: Generic elevator safety requirements*
  Cabin movements are bound by the shaft's limits. The cabin's door is closed when the cabin is moving and only opens after the cabin is stationary, and has reached its target level. To assure timely call processing, a cabin starts moving in the right direction as soon as it gets assigned a call. A given call must be finalised prior to processing the next one. To prevent damage from the drives, the direction of travel must not change abruptly. Call processing is closely related to fairness requirements: a call will be processed and not remain unprocessed for an infinite time; once begun, the call will be finished and the passenger will reach his target level.
- 1^{st} *level: Distance-based dispatching*
  In the first safety level, calls are allocated in such a way that the two cars of the TWIN cannot hinder each other and a minimum safety distance of one storey is maintained. The safety distance varies depending on system speed: the higher the speed, the greater the safety distance.
- 2^{nd} *level: Monitoring of safety distances*
  The second safety level uses communication software to control the distance between the two elevator control units. Each controller is fed with the location and speed of both cars and uses this information to calculate the distance between them. When the TWIN cars approach each other inadmissibly (warning distance is breached), they are slowed to a speed at which they can be stopped at any time without breaching the required safety distance.
- 3^{rd} *level: Emergency stop*
  The third safety level triggers the emergency stop. If the safety distance is breached, the drives are automatically stopped and the brakes activated. Calculation of the safety distances and activation of the brakes is done by robust safety controllers (PLCs) operating independently of the elevator controllers. Programmable Logic Controllers are components packaged and designed to be functional under hostile conditions. Their functionality is provided by special purpose microprocessors whose well-functioning can be verified independently of the overall system.
- 4^{th} *level: Engagement of mechanical brakes*
  If the three preceeding safety levels fail to slow the cars, the fourth-level safety controller automatically engages the mechanical brakes of both cars. The brake on the upper car works in downward direction and that of the lower car in upward direction. This means that the cars cannot collide even if they are unfavourably loaded or the brake system fails. It is therefore impossible for the cars to collide.

We were able to formalize these natural language requirements into temporal logics for all safety levels.

2.2 Passenger Call Allocation

The allocation of the passenger calls to the cars – essential for the first safety level – is a key requisite for the TWIN system. The Destination Selection Control (DSC) system can optimize traffic flows and help passengers reach their

Fig. 4. Destination Selection Control (DSC). The photo (©ThyssenKrupp) shows the indication of the elevator to be taken.

destinations faster and more safely. In the past, passengers were only able to communicate to the elevator control system that they were waiting on a certain floor to go up or down. Not until they entered the car and pressed the appropriate button could they indicate the desired target level.

The DSC controls are aware of both items of information in advance via a touch screen terminal located at a central point – normally in the hall. Passengers can enter their destination with a single touch. In contrast to traditional elevator systems, a passenger on a certain level simply enters her desired target level. The system selects the most appropriate elevator; the passenger is informed via the DSC display which door to go to (Figure 4). The DSC reduces both the number of stops at other floors and journeys without passengers. The time to destination is shorter, which in turn improves capacity. Allocation of pending calls can be optimized; passengers with similar routes may be grouped and conducted to the same door.

The call dispatching is governed by the impossibility of one cabin overtaking the other cabin in the same shaft. The resulting processing guidelines can be formalized by the following algorithm, also depicted in Figure 5, that explicits the call dispatching. It hereby realizes the 1^{st} safety level. Note that the algorithm is symmetric with regard to the travel direction; that's why we provide the alternative reading in brackets. Sometimes, several cabins may be able to serve a call.

- (a) Cross-over routes, i.e. routes where the lower [upper] TWIN car's target level is above [below] the other car's target level, cannot be served by the TWIN system as the cars would collide.
- (b) Calls involving transits from the downmost to the upmost level or vice versa cannot be handled by the TWIN elevator, as the cars cannot sidestep. These routes are served by a conventional elevator, present in every TWIN installation.
- (c) The upper [lower (d)] TWIN cabin handles all routes whose end-point is above [below] the other car's position.

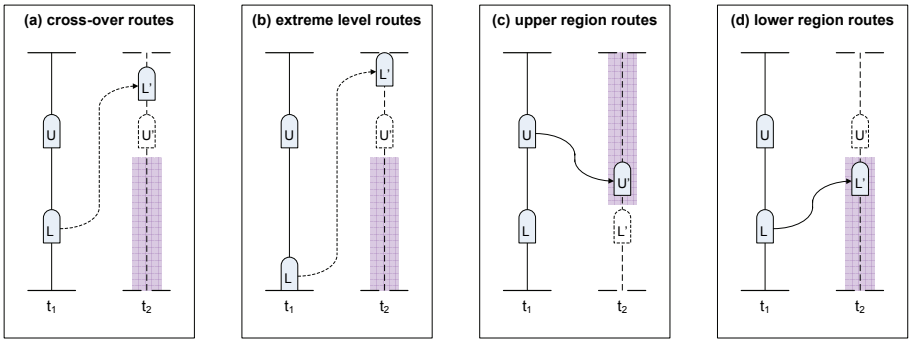

Fig. 5. Call assignment to TWIN cabins. The state transitions for routes (a) and (b) are dotted, as those cannot be served by TWIN cabins: route end-points must be located in the reachable zone (in the figure).

Call assignment to cabins must take into account the future position of the other TWIN cabin in the same shaft. Evaluation of the safety distances occurs at a given time, implying that the present positions are compared to present positions and future positions to future positions, respectively.

Moreover, call processing involves two phases: In the first phase, the assigned car heads for the level where the passenger is waiting to be picked up; in the second phase, the car moves with the passenger to his destination level. As the *same* cabin must be moved in both stages, no reassignment of a given cabin must take place between phase one and two, and neither within each of the phases. Calls may only be assigned to a cabin if the cabin is vacant and not processing another call. Only if the same cabin can execute both phases, it can be assigned the call. Newly placed calls must not perturbate call processing even if all cabins are currently busy.

3 Translating Object-Z to SMV

3.1 Using Object-Z for specifying industrial software

Advantages compared to other specification languages Object-Z [4] is an object-oriented extension of the standardized specification language Z [5]. It has well understood semantics [11] and benefits from tool support [3]. Using Object-Z as a specification language for industrial systems has several advantages compared to alternative approaches, such as pure Z specifications or semi-formal techniques.

Whereas a Z specification defines a single state space, Object-Z's classes with their separate namespaces are especially handy for specifying medium- to large-scale software systems [10]. The object-oriented specification paradigm is well adapted to distributed and embedded systems; communicating objects reflect the spatial separation of different components. Unlike Z, Object-Z supports

specifying concurrent systems. Multiple instantiation of the same class provides for easy scalability where Z would have required a manual enumeration of each instance.

In comparison to semi-formal specification languages such as UML (that can be used complementarily), Object-Z provides formal rigour. Object-Z's number of graphical constructs is manageable and each of them is well understood. Eventually, translating a semi-formal language to a formal language would leave space for interpretation. This is undesirable for reliable and deterministic results.

One of the objections against formal techniques that are articulated in an industrial environment, is that they are intellectually more demanding than semi-formal techniques. Undoubtedly, UML is widespread by now, and therefore it is part of acedemic curricula; considerable secondary literature with numerous guidebooks exists. Yet, adequate tool support can alleviate the cognitive burden and make formal languages like Object-Z more accessible.

Key concepts of Object-Z In Object-Z, graphical *schema* notation enables the concise structuring of state and operation specifications and modularizes them into classes. Any schema consists of a declaration part and a predicate part enabling abstract specification of invariants, pre-conditions and post-conditions. Classes in Object-Z encapsulate a state and an initial schema, as well as operation schemas specifying the methods of an object oriented class. In addition, Object-Z features specific class constructs for visibility, constant declarations, polymorphism, and inheritance.

The idea of instantiation of an object o of a class C is naturally represented by the declaration of a variable $o : C$ where o then denotes the identity of an object. Object-Z has a reference semantics [11] and the common object-oriented dot notation, e.g. $o.m$ to annotate the invocation of an object's feature.

The so-called *schema calculus* comprises operators enabling composition of operations to create new operations, especially in the context of modular systems. In Object-Z operations are composed by conjunction, non-deterministic choice, sequential composition, and parallel composition.

Advantages of translating Object-Z to SMV Translating Object-Z to SMV brings together the advantages from both Object-Z and SMV while surmounting their respective difficulties. On one hand, this is a powerful specification language with inherent structuring concepts; on the other hand, this is an automated model checker. General-purpose model checking tools profit from a larger community and ongoing research resulting in performance enhancements compared to niche tools.

3.2 Translation procedure and characteristics

The translation from Object-Z to SMV is realized by a set of general translation rules that exhaustively cover the industrially relevant constructs of Object-Z. That are, as basic constructs – with the restriction to finite sets – data types,

variables and constant, classes, and instantiation of latter. For objects, state transitions by operations, object communication, and operation composition are covered. The resulting translation rules are intuitively sound and their formal correctness can be proven.

Our translation is *direct* in that it identifies concepts of Object-Z, like propositional logic, basic types, and the class concept, with almost directly corresponding features of SMV. Where appropriate, the missing semantics is added in the translation process using additional definitions, constraints, or other constructions.

The striking advantage of this direct translation is that it is quite obviously *structure preserving*, i.e. the structure of the Object-Z classes and SMV modules correspond one to one and the initial and state schemas of Object-Z have distinct representations in SMV code chunks. Although the granularity of the operations cannot be preserved, one can show that the translation distributes over the constructs of SMV used for operation representation. The translation rules do not flatten the specification.

4 The Quality Feedback Loop in Practice

This section illustrates how our quality feedback loop can be applied rewardingly. Based on an excerpt from the TWIN Elevator case-study [6] (see Section 2), we emphasize on how the feedback loops between the formal specification of interfaces and the requirements verification techniques.

The System Specification in Object-Z. The sub-system we consider is an elevator cabin, as it can be found in a conventional or in the TWIN elevator. For the sake of clarity, we specify the system in a very concise manner.

First, the cabin's specification in Object-Z declares two constants, the lowest (*LevelGround*) and the highest level (*LevelTop*) of the shaft in which the cabin is running. Moreover, the data type *Level* is declared as an integer subrange. These declarations of a global scope are accessible to the environment.

Second, the cabin is specified as an Object-Z class. The cabin's state is represented by the value of its state variable *curr_level* of type *Level*. This state variable is accessible from the environment, as it is included in the visibility list ($\uparrow(curr\_level)$). Initially, as specified in the *INIT*-schema, the cabin's current level is the ground level.

The *Cabin* class comprises an operation *MoveUp*. (A *MoveDown* operation would be included as well, but we can abstain from it here as it does not contribute to the illustration of the feedback loop.) The operation's delta-list ($\Delta(\ldots)$) includes all state variables that will be changed by the operation, here it is only *curr_level*. In the operation's predicate part, the pre-condition and the post-condition are noted. Here, the post-condition is formed by *curr_level*'s value after the execution of the operation. Note the primed notation to reference to the post-state. No restriction is imposed on the pre-state.

$Level ::= (0 \,..\, 200)$

$LevelGround = 1$
$LevelTop = 12$

Cabin

$\lceil (curr\_level)$

$curr\_level : Level$

INIT

$curr\_level = LevelGround$

MoveUp

$\Delta(curr\_level)$

$curr\_level' = curr\_level + 1$

The System Specification translated to SMV. Following the principles and translation rules [7], we are able to translate the specification of the cabin from *Object-Z* to the `SMV input language`.

First, the declared data type *Level* and the two constants *LevelGround* and *LevelTop* are translated. Definitions for the boolean constants are automatically added.

```
typedef Level 0..200;

#define LevelGround 1
#define LevelTop 12

#define true 1
#define false 0
```

Second, the Object-Z class is translated to an SMV module. It contains the type state variable *curr_level*.

```
module Cabin()
{
  curr_level : Level;
```

For each operation in the Object-Z class, two additional boolean variables are introduced: *`operationname`*`_pre` is defined by the expression of the operation's pre-condition. If the latter is missing, the variable's value is true. *`operationname`*`_ stimulus` indicates whether the execution of the operation has been stimulated. According to the semantics of Object-Z, the system offers several operations that may or may not be invoked by the environment. Not each operation is always executed.

```
  /* operation MoveUp */
  MoveUp_pre : boolean;
  MoveUp_pre := true;

  MoveUp_stimulus : boolean;
```

Finally, the state transitions are grouped by state variable. In SMV, the construct `next(`*`variablename`*`) := `*`expression`*`;` assigns *`expression`* to *`variable`* if the state transition happens. We use a conditional construct inside the state transition, that spans over all operations susceptible to change the variable considered. The state transition is guarded by the operation's pre-condition (here: `MoveUp_pre`) and by the operation's stimulus (here: `MoveUp_stimulus`). A fallback case `default:` *`variablename`*`;` is always added providing for the variable value to remain unchanged in case none of the operations is executed.

```
  next(curr_level) := case{
    MoveUp_pre & MoveUp_stimulus : curr_level + 1;
    default : curr_level; };
}
```

SMV requires one `main` module in each program. All top-level modules are instantiated once in this module (here, only the module `Cabin`). Also, the stimulus for all operations not used to construct any other operation inside the Object-Z specification is set to true to assure that a 'running' system is checked. Here, the Object-Z operation *MoveUp* is stimulated: `system.MoveUp_stimulus := true;`.

```
module main()
{
  system : Cabin();
  system.MoveUp_stimulus := true;
}
```

The resulting complete SMV program can be loaded in SMV. In an integrated development environment, SMV can also be called from the command-line.

Formulating Requirements in Temporal Logic. Based on the interface of the system formed by the global scope definitions and exposed state variables (cf. the visibility lists in Object-Z), a stakeholder will naturally formulate system requirements in temporal logic.

As an example, consider the following requirement, that the cabin's movements are restricted by the boundaries of its shaft:

$$LevelGround \leqslant system.curr\_level \leqslant LevelTop$$

The state variable *curr_level* is in the visibility list of the *Cabin* class; the constants *LevelGround* and *LevelTop* are defined system-wide.

After a syntactical rewriting, we have this requirement as a temporal logic formula in the SMV input language, where "assertions" can optionally be named (here: `CabinStaysInShaft`).

```
CabinStaysInShaft: assert
  G (LevelGround <= system.curr_level) & (system.curr_level <= LevelTop);
```

SMV automatically checks the assertion on the specified system. On standard desktop hardware, the verification results are available instantaneously.

In our case, SMV has detected a violation of the formulated assertion. The verification result is "false" (Fig. 6). In an interactive use of SMV, the detected error is reported. A trace leading to the error is calculated and output as a counterexample. During the verification process, SMV also creates a more verbose log, that may be parsed for further analysis.

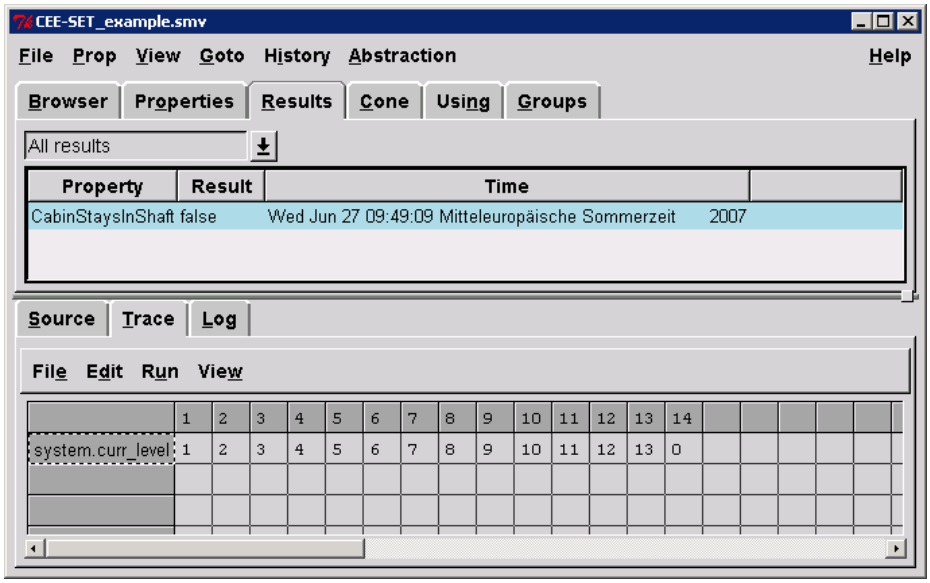

Fig. 6. The property 'CabinStaysInShaft' could not be verified; SMV computes a counter-example and outputs the trace leading to the violation of the property.

Using the generated trace, one is able to discern the state variables, whose changes caused the error. In our example, the state variable `system.curr_level` is changing. Based on this information, the system's operations' delta-lists are examined. This way, we are able to determine those operations in the (Object-Z) specification that are responsible for state transitions in the concerned variables. Here, we locate the operation *MoveUp*.

MoveUp ← name of operation concerned

$\Delta(\mathbf{curr\_level})$ ← delta-list of changed state variables

$curr\_level' = curr\_level + 1$

After a thorough analysis, the system engineer identifies a missing precondition as causing the error. A guard must be added to the operation, so that the cabin is no longer able to move up when it is already on the top-most level:

MoveUp

$\Delta(curr\_level)$

$curr\_level' = curr\_level + 1$

$\mathbf{curr\_level} < \mathbf{LevelTop}$ ← additional pre-condition to be inserted

After having added the precondition, a new quality feedback loop is initiated as a change has occured. This time, verification succeeds and the requirement is fulfilled.

5 Conclusions

We have introduced our quality feedback loop based on a closed iteration looping over requirements and interface models expressed in Object-Z. We presented the requirements specification of the TWIN elevator and the systematic translation process from Object-Z to the model checker SMV. We then illustrated our feedback loop in practice by showing how an insufficiently modelled interface specification is detected: when checking the automatically translated requirement on the system model SMV produces an error message plus a trace indicating the error. The responsible operation in the interface specification can be sytematically located and corrected. In general, this feedback could have produced inconsistencies in the requirements as well.

Using this systematic translation we have translated the entire TWIN requirements specification as presented in Section 2 into Object-Z and sucessfully verified all safety properties on the mechanically generated SMV representation [6, 7].

References

1. Ramesh Bharadwaj and Constance L. Heitmeyer. Model Checking Requirements Specifications Using Abstractions. *Automated Software Engineering* **6**(37–68), Kluwer 1999.
2. C. Canal, L. Fuentes, A. Vallecillo Extending IDLs with p-calculus for Protocol Compatibility *ECOOP'99 Workshop on Object Interoperability Lisbon (Portugal), June 1999* Object Interoperability, Selected Papers, pp. 13-24, Universidad de Málaga, 1999, ISBN: 84-699-0520-1.
3. The Community Z Tools project, 2006. `http://czt.sourceforge.net/`
4. Roger Duke and Gordon Rose. Formal Object-Oriented Specification Using Object-Z. Cornerstones of Computing. MacMillan, 2000.

5. International Organization for Standardization: ISO/IEC 13568:2002: Information technology – Z formal specification notation – Syntax, type system and semantics. http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=21573
6. Florian Kammüller and Sören Preibusch. An Industrial Application of Symbolic Model Checking – The TWIN-Elevator Case Study. Accepted for publication in *Informatik Forschung und Entwicklung*. Springer, 2007.
7. S. Preibusch and F. Kammüller. Checking the TWIN Elevator System by Translating Object-Z to SMV. *Formal Methods for Industrially Critical Systems, FMICS 2007*. Vol. 4916, LNCS, Springer.
8. Sören Preibusch. TWIN Elevator System, Concise Object-Z Specification, 2007 http://preibusch.de/projects/TWIN/Concise_OZ
9. Sören Preibusch. TWIN Elevator System, Concise Object-Z Specification (Translation to SMV), 2007 http://preibusch.de/projects/TWIN/Concise_OZ_Translation_SMV
10. Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods, Kluwer Academic Publishers, 2000.
11. Graeme Smith and Florian Kammüller, Thomas Santen. Encoding Object-Z in Isabelle/HOL. *ZB 2002: Formal Specification and Development in Z and B*Springer LNCS **2272**, 2002.
12. ThyssenKrupp (2005) Safe distance - Four-level safety concept. http://twin-elevator.com/Safe_distance .353.0.html?L=1
13. ThyssenKrupp Elevator. TWIN Report, 2005 http://www.twin.thyssenkrupp-elevator.de/?&L=1