Informatik - Forschung und Entwicklung manuscript No.
(will be inserted by the editor)

Florian Kammüller · Sören Preibusch

An Industrial Application of Symbolic Model Checking

The TWIN Elevator Case Study

Eingegangen: date / Angenommen: date

Zusammenfassung Model Checking Techniken liefern anerkanntermaßen zuverlässige und umfassende Ergebnisse. Im Gegensatz zu Testverfahren werden nicht nur Einzelfälle untersucht, sondern der gesamte Zustandsraum fließt in die mathematische Korrektheitsprüfung ein, was jedoch aufgrund schwer handhabbarer Zustandsexplosion als Nachteil angesehen wird. In unserer Industrie-Fallstudie, der Anwendung automatisierter Model Checking Techniken auf das innovative TWIN Aufzugssystem von ThyssenKrupp, beweisen wir die Gültigkeit der Spezifikation bezüglich der Anforderungen; Effizienz wird durch Abstraktion und Nichtdeterminismus erreicht. Die Sicherheitsanforderungen an den Aufzug sind vollständig in Temporallogik ausgedrückt, ebenso wie algorithmische und technische Voraussetzungen, Konsistenzbedingungen und Fairness-Eigenschaften. Unser Fallbeispiel weist nicht nur die Betriebssicherheit eines Produktivsystems nach, sondern unterstreicht die lohnende Anwendbarkeit von Model Checking Techniken im industriellen Maßstab.

Schlüsselwörter Symbolisches Model Checking · Mechanische Verifikation · Industrielle Fallstudie · Zustandsbasierte Systeme · Sicherheit · SMV

Abstract Model checking techniques are recognized to provide reliable and copious results. Instead of examining a few cases only – as it is done in testing – model checking includes the whole state space in mathematical proofs of correctness. Yet, this completeness is seen as a drawback as the state explosion problem is hard to handle. In our industrial case study, we apply automated model checking techniques to an innovative elevator system, the TWIN by ThyssenKrupp. By means of abstraction and nondeterminism, we cope with runtime behaviour and achieve to efficiently prove our specification's validity. The elevator's safety requirements are exhaustively expressed in temporal logic along with real-world and algorithmic prerequisites, consistency properties, and fairness constraints. Beyond verifying system safety for an actual installation, our case study demonstrates the rewarding applicability of model checking at an industrial scale.

Keywords Symbolic Model Checking · Mechanical Verification · Industrial Case Study · State Based Systems · Safety · SMV

CR Subject Classification D.2.4 Program Verification · F.3.1 Specifying and Verifying and Reasoning about Programs · J.7 Computers in Other Systems · C.3 Special-Purpose and Application-Based Systems

Authors are in alphabetical order
[1] Technische Universität Berlin
[2] German Institute for Economic Research (DIW Berlin)
E-Mail: flokam@cs.tu-berlin.de, spreibusch@diw.de

1 Introduction

Standards for safety and security increasingly demand the use of formal development methods and mechanized verification. For example, the Common Criteria are a standard for security jointly defined by an international initiative of security agencies [2]. To achieve an evaluation of the highest level EAL5–EAL7 of these Common Criteria, not only formal specification, but also machine checks of security requirements is prerequisite.

To apply formal methods and mechanical verification in a systematic way, the nature of the system has to be taken into account. A system's characteristics comprise its field, its safety relevance, complexity, response time behaviour, and its interactivity patterns. The presented case study is an example of a typical state-based, reactive system. Hence, it is a well-suited example for the application of model checking ([7], [13], Section 4).

Our contribution is twofold: First we demonstrate the applicability of automated, mechanical model checking techniques to installations at an industrial scale and show exemplarily their benefits in system development; second, for a given safety-critical setup – the TWIN eleva-

tor – we prove the conformance of an implementation to the specification and mandatory safety requirements.

This paper is organized as follows: In Section 2, we outline related work and introduce the SMV model checking system along with the temporal logic constructs that are used in this case study; in Section 3, the TWIN elevator is described, including its transportation logic, followed by a translation of the latter into a formal model in Section 4. Section 5 assembles functional and non-functional system requirements, expresses them in temporal logic, and checks them against the developed specification. We conclude our paper with an outlook and summarize the main lessons learned from the case study.

2 Symbolic Model Checking

Model checking techniques have been applied to industrial applications for some time but scenarios are few and far between [1; 4; 6]. Whilst non-technical analyses exist [10], applications focus on the validation of chip design [3] and protocol verification [9]. The studied systems are characterized by their time-discrete reactivity and the safety relevance. Failure of components may cause major damage; the Royal Academy of Engineering estimates the annual costs caused by software errors at 20 to 25 B euros for UK only. Yet testing and prototyping often fail to circumvent these risks as systems' complexity may not be captured by one-shot tests. Model checking algebraically proves the correctness of these systems by specifying both the structures and their respective requirements in a temporal model and evaluating the formulated assertions on the behaviour.

Model checking is essentially based on the graph of a finite state system that is to be analysed. Each state is a node in the graph represented by a tuple of values representing important characteristics of this system state. The edges in this graph represent the state transitions under which the modelled system evolves over time.

The basis of any logical analysis of this model is an initial assignment of propositional formulas to each state. The definition and assignment of these simple propositions is, after the definition of the states, the second important step of specifying the model.

The input language of a symbolic model checker like SMV enables the description of this model in a language intuitive for programmers: states and their components are defined like data types and variables, the state transition function is given by the `next` construct assigning to each state component the value it will have in the following state depending on the values of the current state.

Properties P we want to certify for such models are now given as a temporal logic expression in linear time logic (LTL) [13]; branching time logic (CTL) may be used as well. The model checking problem is simply defined in finding those states in the model that represent reachable system states for which the property P holds. Temporal logics are particularly well-suited for the specification of reactive, state-based systems, because the change of properties over time can be expressed very naturally.

The model checking procedure is fully automated. If the temporal logic property holds for the model it will just terminate without any error message. If a contradiction (i.e. error) is detected, the path leading toward the property-violating state is output. This path is interpreted as a series of state-changing actions and it represents a counterexample to the property. Accordingly, systems can be developed and tested iteratively.

The major limitation of model checking is the so-called *state explosion problem*: the size of the model scales with the cartesian product over the state components' values. Consequently the effort to check a property over this graph is exponential in the size of the state. Hence, without sufficient abstraction to small states, model checking is not applicable.

Temporal logic extends predicate logic in that it comprises clauses involving temporal operators. Reactive systems (and thus properties describing these systems) evolve over time; different states can be distinguished. The truth value of a formula in temporal logic is dependent upon time [14]. In addition to the traditional boolean operators "and", "or", "not", and "implies" (we use the classical representations $\wedge$,$\vee$,$\neg$, and $\rightarrow$ in this paper), new temporal operators capture relationships in time.

The operator F is used to express a condition that must hold true at some time in the future. The formula Fp is true at a given time if p is true at some later time. On the other hand, Gp (globally p) means that p is true at all times in the future. Usually, we read Fp as "eventually p" and Gp as "henceforth p" [15]. In addition, we have the "until" operator and the "next time" operator. The formula pUq, which is read "p until q" means that q is eventually true, and until then, p must always be true.[1] The formula Xp means that p is true at the next time.

We use the SMV verification system in our case study (the use of alternative verification systems is discussed in Section 6). SMV developed by Ken McMillan as part of his PhD thesis [14], later published as a book [15] in the 1990s, verifies that every possible behaviour of a specified system satisfies the specification. This is in contrast to a simulator, which can only verify the system's behaviour for the particular stimulus provided. SMV's success is due to the fact that McMillan first suggested to check models symbolically using a new representation for the state model and very efficient algorithms to evaluate properties on this model.

A specification for SMV fully captures the system's temporal evolution and englobes a collection of asserti-

[1] The "until" operator U we use is known as the "strong until", as opposed to the "weak until" operator W not requiring that q eventually becomes true in pWq.

ons stating on the system's properties in temporal logic. During model checking, the specifications are automatically formally verified; counterexamples of traces that violate the specified property are produced in case of a conflict.

The state transitions in the SMV specification are formulated in a programming language including temporal assertions (`init`, `next`), conditionals (`if`, `switch`, `case`), and loops (`for`, `chain`). As the resulting program constitutes a logical expressions, multiple contradictory or circular assignments are not allowed. Assertions to be verified during model checking are introduced by the keyword `assert`. By the construct `using ... prove`, assertion hierarchies may be built as to tell the verifier that a given assertion should be assumed to be true when checking the other one.

3 TWIN Elevator System

3.1 System Description

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. In the autumn of 2005, there were a dozen TWIN reference installations with up to 11 TWIN elevators [2], linking up to 39 storeys [3]. The high-speed TWIN is approved for speeds of 8 m/s, allowing the TWIN to be used in very tall buildings such as those predominantly found in North America and East Asia. The optimum configuration is with two or more main access stops.

Each installation is used in combination with at least one conventional elevator. It handles all calls that cannot be processed by one of the TWIN cabins, e.g. routes from the ground to the top floor (see Section 3.3).

The major economic benefits of a TWIN elevator system are space and speed efficiency: Compared with a conventional 4-shaft elevator group, with one car in each shaft, the TWIN system offers either 40% higher passenger capacity or a 25% reduction in building volume. One of the shafts can be freed and put to a different use, e.g. housing equipment such as air conditioning and cables. Floor space is especially valuable in high-rises, and TWIN systems are designed for buildings with a minimum height of 50 meters. Either way, the utility of the building increases significantly [19].

In the TWIN elevator system, two cars are arranged one above the other; they run independently in the same shaft – also at different speeds. A safety distance is kept, depending on the speeds involved. The cars can move in different directions, which means that they can also move toward each other. Each car has its own traction sheave drive and counterweight, but both use the same guide rails.

[2] Federation Tower, Moscow/Russia
[3] Torre Agbar, Madrid/Spain

3.2 Informal Safety Specification

An informal specification of safety requirements is the basis for their formal expression by means of temporal predicates as presented in Section 5. We transform ThyssenKrupp's natural language requirements (e.g. "The cabin door is closed until the cabin is no longer moving") into a formal representation; assertions on the states of object components are concatenated by temporal connectors.

In the following enumeration of the safety levels, we have indicated in parentheses which safety assertions in Section 5 formally translate the requirements, e.g. (D_3). Table 1 summarizes this correspondence by associating the safety levels to the safety assertions.

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 [18]. Four safety levels express an escalating strategy of electronic and mechanical measures targeted to prevent a shaft's two cabins from colliding, cf. Figure 1. The figure is to be read bottom up, in such a way that the safety level indicated in the last column is achieved by a consecutive orchestration of the system components. Temporal logic captures requirements on the software. Mechanical devices such as emergency brakes cannot be analysed by logics, as their functioning relies on laws of nature rather than algorithms. Their reliability has been tested by the German TÜV inspectorate.

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.

	Representation in Specification	System Components	Safety Levels
Software	Temporal Logic	Processor	1
Hardware	Sensors and motor functions represented by their effects on the environment	Controllers PLC	2 3
Mechanics	Passive safety components	Mechanical Brakes	4

Fig. 1 Hardware, Software, and Mechanics: the components' interplay assures the four safety levels

– 0^{th} *level: Generic elevator safety requirements*
Cabin movements are bound by the shaft's limits (B_1). The cabin's door is closed when the cabin is moving (D_2) and only opens after the cabin is stationary (D_3, D_4), and has reached its target level (D_1). To assure timely call processing, a cabin starts moving in the right direction as soon as it gets assigned a call (R_2, P_1, P_3, P_4, P_7, P_8). A given call must be finalised prior to processing the next one (P_2, P_6). To prevent damage from the drives, the direction of travel must not change abruptly (R_1). Call processing is closely related to fairness requirements: a call will be processed and not remain unprocessed for an infinite time (F_1/P_5); once begun, the call will be finished and the passenger will reach his target level (F_2/P_9).
– 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 (S_1) 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 (S_2).
– 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 (S_{3a}, S_{3b}). 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 (S_4). 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.

Fig. 2 Destination Selection Control (DSC).
The photo (©ThyssenKrupp) shows the indication of the elevator to be taken; cf. Figure 3 for a schematic view.

3.3 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 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.

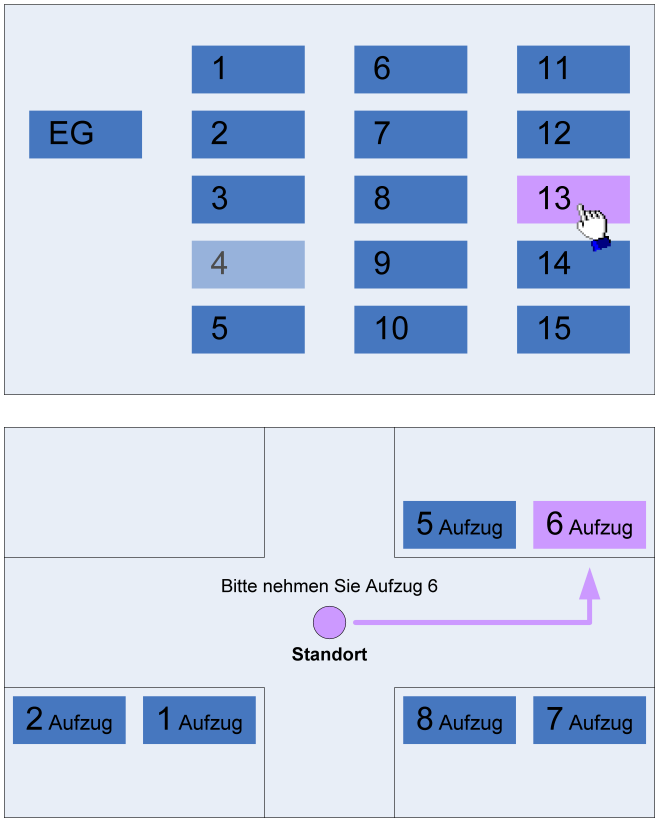

Fig. 3 The user indicates his desired target level **(top)**; promptly the system indicates which door to go to **(bottom)**

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 (cf. Figure 3, top). In contrast to traditional elevator systems, a passenger on a certain level simply enters his desired target level. The system selects the most appropriate elevator; the passenger is informed via the DSC display which door to go to (cf. Figures 2, 3, bottom). 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 4, that explicits the call dispatching. It hereby realizes the 1st 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.

Calls have the properties 'from' and 'to', indicating the originating and the destination level. In our model, the levels are the indices of the `calls[from][to]` matrix. 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 (cf. Assertion (P_6) in Section 5.2): 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.

In our model, a call's status is coded by the matrix entry `calls[from][to]` taking the values `callNone` if no call is waiting, `callWaiting` if a call is waiting but not yet assigned to a cabin, or a combination of a cabin identifier (e.g. `callServedByTWINlower`) with a processing phase identifier (`goingToDepartureLevel`, `goingToArrivalLevel`). If another passenger is requesting the same `from`-`to`-route while `calls[from][to]` = `callWaiting`, the status remains unchanged. When a cabin gets assigned a call, the destination level information is copied to the cabin so that a newly placed call does not perturbate the current call's termination.

4 Specification

In the light of safety relevance, the need for reliable verification techniques is obvious. Model checking is a candidate to verify the safety and requirements adherence of the TWIN elevator system. In addition, since the TWIN is a reactive system, by reacting to the passengers' ride requests, and the transportation logic imposes deterministic, time-discrete state transitions, model checking techniques are the method of choice. This section models the TWIN system in temporal logic along with its requirements.

Our case study will present a twelve-level installation, showing scalability of our approach even above real-world cases like the installation at Stuttgart University, actually an 11-storey building.

4.1 Specification Principles and Specification Structure

We developed our TWIN elevator system specification guided by the following principles: the specification should be as clean and simple as possible, abstracting from unnecessary details as they would hinder intuitive comprehension of the model-building process; on the other hand, the specification should be as complete and concrete as necessary, allowing for all relevant system behaviours to be verified. Especially, all safety relevant components and the four-level safety concept need to be mapped to the temporal logic model.

The specification is structured into three modules Cabin, Control, and DSC, distributing the application logic in the loci of concern. The additional main module instantiates the Control module, hence allowing to model installations with multiple shafts. Inter-module communication is realized via directed `input/output` variables. In reality, communication lines transmit signals between the main computer, the moving cabins, and the DSCs mounted in the storeys.

- The "Cabin"-module represents a single cabin along with all status information associated to it, as also mapped to the global view. The "Cabin"-module also encompasses the cabin moving logic, handling level

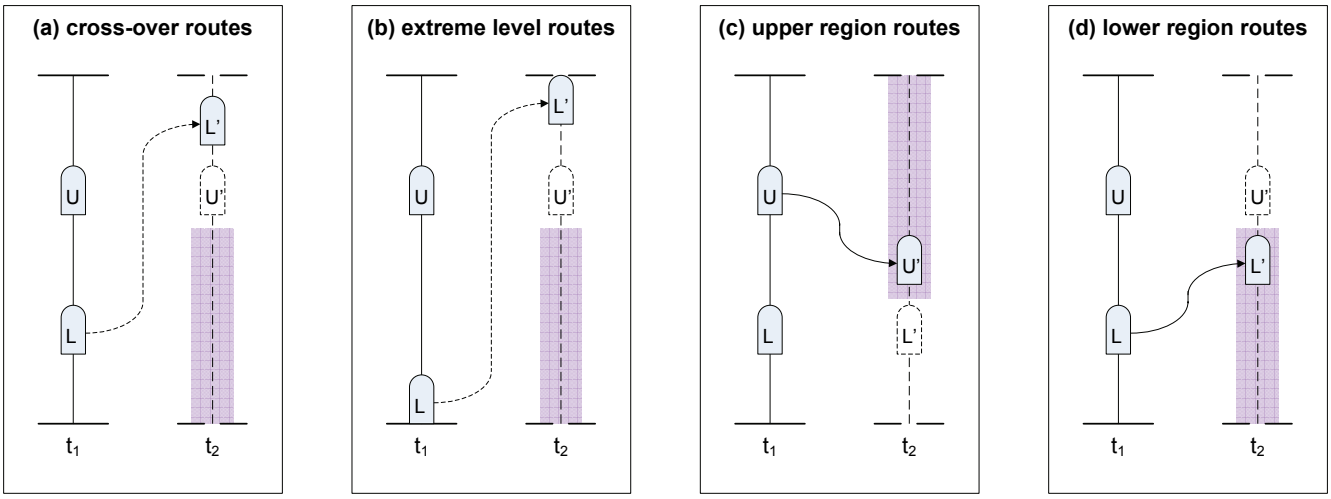

Fig. 4 Call assignment to TWIN cabins. The state transitions for routes (a) and (b) are dotted, as those cannot be served by TWIN cabins: route end-points must be located in the reachable zone (coloured in the figure).

transitions and direction changes of a cabin depending on current level, destination level, and velocity. In case of an emergency brake, triggered by an underflow of the safety distance, the cabin is stopped. Moreover, door closing and opening, and door status monitoring is done on the cabin level.
– The "Control"-module is globally responsible for the elevator logic. A conventional elevator shaft (with a single cabin) and a TWIN shaft (with two cabins) are instantiated. For all three cabins, cabin status information is mirrored: current cabin position, destination level, moving direction and velocity, call processing status, and flags indicating whether the warning distance is underrun or the emergency brake has been triggered. Analogously, all issues involving more than one cabin are handled in the "Control"-module: call assignment to cabins, monitoring of warning, safety, and emergency brake distances.
– The "DSC"-module sends incoming passenger calls to the "Control"-module but does not carry application logic.

We did not incorporate refinement hierarchies in our model, as we considered the module dependencies too tight to carry out sensible abstractions. One straightforward abstraction could be to remove the actual call assignment logic from the specification and allow for multiple implementations. The model does not depend on a given call handling logic, as all safety assertions and system requirements must be fulfilled regardless of a concrete implementation. In an industrial environment, one can think of upgradable service software for which also newer versions have to comply with binding safety requirements.

Our case-study takes the 11-storey TWIN elevator system at Stuttgart University as a representative example. We believe our approach to be scalable to higher buildings. Still, it is noteworthy that TWIN systems are not trivially generalizable in their number of storeys. Whereas a 51-storey case may be treated similarly to a 50-storey case, one cannot conclude from three levels to four levels: security distances do not allow to abstract from the actual distances between the cabins. For a more detailed discussion on the lower bounds for the induction see Section 5.3.

The following three subsections 4.3 to 4.5 depict the modules involved in our case study specification, focusing on the relevant parts. The "main"-module, mandatory root in a SMV program is omitted as it only instantiates the "Control"-module. Symbol declaration parts or programmatically caused redundancies are left out in favour of conciseness. The full specification can be accessed online at [17].

4.2 Introductory SMV Example

In addition to the basic introduction of the SMV syntax at the end of Section 2, we provide a commented example of a state transition and illustrate a simple safety assertion. Both are excerpts of the real SMV code underlying the following sections. The reader familiar with the SMV syntax may skip this subsection and continue directly with Section 4.3.

SMV variables are typed; one can use the basic types integer, integer ranges, and boolean or define new types as sets of enumerated symbolic values. One- and multi-dimensional arrays can be used, for example to construct matrices.

The definition of a variable is provided by combining other variables with arithmetic or logical connectors or by providing the initial state and a state transition relation.

The boolean variable `emergencyBrakedTWIN` is defined as an expression of other variables. The equality holds in the initial state and in all following states.

```
emergencyBrakedTWIN : boolean;
emergencyBrakedTWIN := cabDistance < SafetyDistance;
```

Consider the following SMV code fragment stating on the symbolic variable `door` of the type `DoorState` = {`open, closed`}. The cabin door is closed in the initial state. The following states are determined by the state transition relation where the use of conditionals is allowed. For homogeneity reasons, we will use `switch`-constructs also for choices between only two alternatives where an `if` would suffice.

Here, the door's state depends on whether the cabin has reached its target or not.

```
init(door) := open;
next(door) := switch(reachedTarget)
  { true: open;
    false: closed };
```

Once the system's reactivity is specified by the state transitions, we can verify whether it manifests some particular behaviour. Expected behaviour can be expressed in an assertion, i.e., a formula whose correctness will be verified by the model checker. The assertion combines different variables by means of logical and temporal connectors (cf. Section 2 for an overview over the connectors). In the following example, the assertion named `DoorOpensAfterEmergencyBrake` characterizes the door state: it must hold everytime (globally `G`) that the occurrence of the situation `emergencyBraked` implies that the door is opened in the next (`X`) state.

```
DoorOpensAfterEmergencyBrake : assert                (D4)
  G (emergencyBraked) →
    X (door = open);
```

4.3 Module "Cabin"

The module `Cabin` has a list of parameters instantiated later in the module control to create different instances of the three cabins of a TWIN system: the conventional cabin, and the two cabins of the TWIN shaft.

```
module Cabin
(
  shaft,
  initLevel,
  currLevel,
  currTargetLevel,
  currDirection,
  currServingState,
  emergencyBraked,
  reducedVelocity,
  door
)
```

Some parameters are constant for an instantiation, like `shaft` recording which shaft a cabin is driving in, others describe changeable parts of the state of each cabin, like `currDirection`, a symbolic variable taking the values 'stationary', 'upward', or 'downward' according to its type `Direction`. The types (e.g. `Shaft`, `Level`, `Direction`, etc.) are defined within in the SMV program [17], prior to the modules.

```
  input   shaft :               Shaft;
  input   initLevel :           Level;
  output  currLevel :           Level;
  input   currTargetLevel :     Level;
  output  currDirection :       Direction;
  output  currServingState :    CallServingState;
  input   emergencyBraked :     boolean;
  input   reducedVelocity :     boolean;
  output  door :                DoorState;
```

Using the SMV construct `init`, the initial state is defined. The values of the symbolic variables are defined in the prelude of the SMV encoding (see [17]).

```
  init(currDirection) :=        stationary;
  init(currLevel) :=            initLevel;
  init(currServingState) :=     ready;
  init(door) :=                 open;
```

A cabin has reached its target when the current level is equal to the target level or when there is an emergency break. We also consider the unlikely case of an emergency brake terminating the processing of the call even if the original target level has not been reached. The call abortion sets the variable `reachedTarget` to `true`; the elevator is hereby capable to resume functional behaviour analogously to the normal processing: the call is cleared and the cabin door opens.

```
reachedTarget : boolean;
reachedTarget :=
  (currTargetLevel = currLevel) ∨ emergencyBraked;
```

By the following `switch` statement, we model the cabin's dynamic evolution. The state variable `currLevel` is assigned a new value depending on whether the target level has been reached and on previous values of the current level and the velocity.

A cabin's velocity is the number of passed levels between two state transitions. The variable `currDirection` has the value `stationary`, if the cabin stands still; it indicates the travel direction otherwise (`upward` or `downward`). For each travel direction, the elevator has a fixed nominal speed: `VelocityUp` or `VelocityDown`, decelerated to `VelocityUpSlow` or `VelocityDownSlow` when the cabins are within the warning distance (status variable `reducedVelocity` is true).

Inside the cases of the `switch` we have an example of SMV conditional structure p `?` c_1 `:` c_2 which has to be read as *if* p *then* c_1 *else* c_2.

```
next(currLevel) :=
  switch(currDirection)
  {
  upward :
    (¬ reachedTarget ∧ currLevel ≠ LevelTop ?
      currLevel + (reducedVelocity ?
        VelocityUpSlow : VelocityUp) :
      currLevel);
```

```
  downward :
    (¬ reachedTarget ∧ currLevel ≠ LevelGround ?
      currLevel - (reducedVelocity ?
        VelocityDownSlow : VelocityDown) :
      currLevel);
  stationary :
    currLevel;
  };
```

The current serving state of a cabin reflects whether the cabin is in action or ready to take on an order. The next value of this state component becomes true when the target is reached.

```
next(currServingState) :=
  switch(reachedTarget)
  { true : ready;
    false : busy };
```

The next direction the cabin will take, can be calculated by comparing the current level with the target level: if the former is smaller than the latter the cabin moves upwards, otherwise downwards. We impose some safety constraints on the current direction of cabin motion: if the door is open or the emergency brakes have been activated, the cabin is stationary.

```
next(currDirection) :=
  switch(door)
  { closed :
      case
      { emergencyBraked: stationary;
        currTargetLevel > currLevel: upward;
        currTargetLevel < currLevel: downward;
        default: stationary; };
    open: stationary; };
```

The door opens when the target level is reached. It opens as well in case of an emergency stop.

```
next(door) :=
  switch(reachedTarget)
  { true: open;
    false: closed };
```

In the remainder of the module `Cabin` we can already prove some assertions for the behaviour of a cabin (see Section 5.2).

4.4 Module "Control"

The module containing the control system specification of the TWIN system defines an array `cabs` of cabins indexed by a variable `Cabins`. Any enumeration type can be used for `Cabins`, thus providing a specification instantiatable to larger cabin numbers. We use here the enumeration type {`conv` (for conventional), `twinLower`, `twinUpper`} as index set.

```
module Control() {
cabs : array Cabins;
```

The array `cabs` contains instances of the previous module `Cabin` once we have defined the representation of the cabins from the point of view of the control. The global parameters of the cabin control are just a mirrored version of the parameters of single cabins suitably arranged in an array.

```
cabDirections:      array Cabins of Direction;
cabLevels:          array Cabins of Level;
cabLevelsTarget:    array Cabins of Level;
cabServingState:    array Cabins of CallServingState;
cabEmergencyBraked: array Cabins of boolean;
cabReducedVelocity: array Cabins of boolean;
cabDoor:            array Cabins of DoorState;
```

The global control tracks all calls, i.e. passenger rides *from* a given level *to* a given level in a `Level` × `Level` square matrix `calls` defined as an array of arrays:

```
calls : array Level of array Level of Call;
```

In SMV, an array variable is declared having the type `array` *range* `of` *type* where *range* is an integer range expression and *type* is the content data type. If "`of` *type*" is omitted, a generic array is declared whose elements may have different data types.

The matrix' entry type `Call` is an enumeration datatype whose values indicate the call processing phase and the processing elevator. The state is encoded as the product of prime numbers. For instance, `calls[4][13]` = `callServedByTWINlower` ∗ `goingToDepartureLevel` indicates that the lower TWIN cabin is starting to process a call from a passenger standing at level 4 and heading for level 13. Using the modulo-operator `mod`, it is efficient to check in which processing phase a call is and which elevator is assigned to the call (e.g. assertion (F_1/P_9)).

Every level is equipped with a DSC, referenced in the array `dscTerminals`. Indeterministic instantiation with the calls departing from the given level allows spanning the whole state space without explicit enumeration.

```
dscTerminals : array Level;
∀(e in Level)
 { dscTerminals[e]: DSC( calls[e] ) }
```

All cabins are vacant if the three we consider are ready.

```
CabinsVacant : boolean;
CabinsVacant :=
  cabServingState[conv] =      ready ∧
  cabServingState[twinLower] = ready ∧
  cabServingState[twinUpper] = ready;
```

We initialize the target level with the initial level, so there are no induced calls at the beginning.

```
  init(cabLevelsTarget[conv]) :=      LevelGround;
  init(cabLevelsTarget[twinLower]) := LevelGround;
  init(cabLevelsTarget[twinUpper]) := LevelTop;
```

Now we can determine which cabin will be assigned to handle a given call. The algorithm is too technical to be presented here in full. For the present consideration of the safety specification we only need to know that it determines the `next` values for `cabLevelsTarget` and updates the `calls` if need be.

For the specification of safety properties we define the following abbreviations and predicates.

```
cabDistance : Level;
cabDistance :=
  cabLevels[twinUpper] - cabLevels[twinLower];
emergencyBrakedTWIN : boolean;
reducedVelocityTWIN : boolean;
```

```
emergencyBrakedTWIN := cabDistance < SafetyDistance;
reducedVelocityTWIN := cabDistance < WarningDistance;

cabEmergencyBraked[conv] := false;
cabEmergencyBraked[twinLower] := emergencyBrakedTWIN;
cabEmergencyBraked[twinUpper] := emergencyBrakedTWIN;

cabReducedVelocity[conv] := false;
cabReducedVelocity[twinLower] := reducedVelocityTWIN;
cabReducedVelocity[twinUpper] := reducedVelocityTWIN;
```

The actual array `cabs` is finally created by instantiating the module `Cabin` thrice with the parameters prepared above `cabDirections[.], ..., cabDoor[.]`. The instantiation corresponds to a "call by reference"; a local change of these variables' values inside the module "Cabin" is mirrored. Consequently, the module "Control" can access the current values of these parameters.

```
cabs[conv] :      Cabin( ShaftConv, LevelGround,...);
cabs[twinLower] : Cabin( ShaftTWIN, LevelGround,...);
cabs[twinUpper] : Cabin( ShaftTWIN, LevelTop,...);
```

See Section 5.1 for the assertion we can prove for the control.

4.5 Module "DSC"

The DSC module – representing a crucial component in a TWIN installation – may be left abstract at our safety-oriented point of view. It is sufficient to consider how it participates in the inter-module communication: by an array of calls we need to serve.

```
module DSC ( calls ) {
  input calls : array Level of Call }
```

5 System Requirements and Safety Assertions

In this section we finally present the transformation of the original system description as introduced in Section 3.2 into temporal logic assertions in SMV. The following properties are all verified by the SMV system with respect to the system specification presented in the previous section. For simpler orientation we have arranged the names of the assertions using capitals as summarized in Table 1. We have further organized the assertion according to their residence in the modules. Assertions can be evaluated on the cabin level or the control level. All assertions *may* be evaluated on the control level as the relevant variables are mirrored in the module "Control" (see Sections 4.1 and 4.4). All assertions stating on more than one cabin *must* be evaluated on the control level. This is especially true for the safety assertions (S_x) as cabin distances involve the positions of two cabins. Table 1 also indicates which safety level is captured by the assertions.

5.1 Assertions in Module "Control"

A first illustration of proving properties with SMV is given by the following assertions on constants. In the following formulas there are no temporal operators involved as these basic safety poperties need to hold beyond any time limit.

```
assert WarningDistance ≥ SafetyDistance;
assert SafetyDistance ≥ ForcedBrakeDistance;
assert VelocityDownSlow ≤ VelocityDown;
assert VelocityUpSlow ≤ VelocityUp;
assert VelocityDown ≤ VelocityUp;
```

As an illustration for basic safety properties we prove that a cabin stays in the shaft:

```
cab : Cabin;                                              (B1)
CabinStaysInShaft : assert
  G ( LevelGround ≤ cabLevels[cab] ∧
      cabLevels[cab] ≤ LevelTop );
```

The four-level-safety concept as provided in Section 3.1 is exhaustively expressed by the following five assertions. For the first level we check distance-based dispatching.

```
SafetyLevel1 : assert                                     (S1)
  G (cabLevelsTarget[twinUpper] >
     cabLevelsTarget[twinLower]);
```

For the second level the monitoring of safety distances is verified. We recall that the warning distance is greater than the safety distance. If the two TWIN cabins approach each other, they are slowed down.

```
SafetyLevel2 : assert                                     (S2)
  G ( (cabDistance ≤ WarningDistance) →
      ( reducedVelocityTWIN ) );
```

The third level requires to prove the emergency stop. If the cabin distance reaches or underruns the safety distance, the signal to release an emergency stop is triggered immediately. There is no delay in the detection of the need to brake the cabins (*sensors*, cf. Figure 1). This is reflected in the SMV code by an immediate implication → (no `X`).

```
SafetyLevel3a : assert                                    (S3a)
  G ( (cabDistance ≤ SafetyDistance) →
      ( emergencyBrakedTWIN ) );
```

In contrast, the software-triggered activation (*motor function*) of the brakes in reaction to this signal requires a processing time. Both cabins will be immobile in the `next` step, as expressed by the temporal operator `X`.

$S^{\dagger}$	Safety	1,2,3,4
$B^{\dagger}$	Cabin movements bounded by the shaft	0
P^{*}	Call Processing	0
R^{*}	Directed cabin movements	0
D^{*}	Door	0
$F^{\dagger,*}$	Fairness	0

* evaluated for a given Cabin
† evaluated globally, Control level

Table 1 Assertions overview. The last column indicates the corresponding safety level(s) captured by the assertions.

```
SafetyLevel3b : assert                                    (S3b)
  G ( emergencyBrakedTWIN →
    X ( (cabDirections[twinLower] = stationary) ∧
        (cabDirections[twinUpper] = stationary) ) );
```

To verify the fourth safety level, we check that given that the cab distance violates the safety bounds the two TWIN cabins are stationary. It is significant that the mechanically forced brake takes effect without delay (*mechanics*): the immobility of both cabins is implied in the same moment. In contrast to the software-triggered emergency brake, there is no further lag.

```
SafetyLevel4 : assert                                     (S4)
  G ( (cabDistance ≤ ForcedBrakeDistance) →
      ( (cabDirections[twinLower] = stationary) ∧
        (cabDirections[twinUpper] = stationary) ) );
```

Finally, we show fairness, a property generally interesting for concurrent systems. Here, fairness means that no passenger waits forever. Even if the TWIN elevators may not be able to process a call, the conventional elevator will still be at the passengers' disposal.

```
lFrom, lTo : Level;                                    (F1/P9)
ProcessingBegins : assert
  G (calls[lFrom][lTo] mod callWaiting = 0) →
    F (calls[lFrom][lTo] mod callWaiting ≠ 0);
```

The pattern $Gp_1 \rightarrow Fp_2$ of the previous property is the standard expression of strong fairness [11]. The assertion

```
    F (calls[lFrom][lTo] mod callNone = 0)
```

represents a stronger requirement: Not only the elevator must start processing the calls, but bring them to a termination. This more exigent property also verifies.

5.2 Assertions in Module "Cabin"

In the module Cabin we verify properties related to a single cabin. First there are some basic assertions concerning cabin movement. There are no abrupt changes in cabin direction.

```
DirectionChangeSmooth1 : assert                           (R1)
  G (currDirection = upward →
    X currDirection ≠ downward);
DirectionChangeSmooth2 : assert
  G (currDirection = downward →
    X currDirection ≠ upward);
```

The cabin direction is translated in moving from one level to the next.

```
DirectionOK1 : assert                                     (R2)
  G (currDirection = upward ∧ ¬ reachedTarget) →
    (X currLevel > previousLevel);
DirectionOK2 : assert
  G (currDirection = downward ∧ ¬ reachedTarget) →
    (X currLevel < previousLevel);
```

Next, there are properties verifying the state of the cabins. The cabin is stationary when ready.

```
StationaryWhenReady : assert                              (P1)
  G (currServingState = ready) →
    (currDirection = stationary);
```

The cabin is busy when the target level is not yet reached.

```
CallProcessing : assert                                   (P2)
  G (¬ reachedTarget) →
    (X currServingState = busy);
```

The cabin closes the door, then starts moving until the target level is reached.

```
ProcessingMoves1 : assert                                 (P3)
  G (door = closed) →
    (X (currDirection ≠ stationary U reachedTarget));
ProcessingMoves2 : assert
  G (door = closed) →
    ((¬ reachedTarget) →
     X (currDirection ≠ stationary U reachedTarget));
```

For the well-functioning of the system it is important to prove that the call processing is not delayed.

```
ProcessingStraight : assert                               (P4)
  G ((¬ ready ∧ ¬ reachedTarget ∧ door = open) →
    X (door = closed) ) ∧
  G ((¬ ready ∧ ¬ reachedTarget ∧ door = closed) →
    X (currDirection ≠ stationary) );
```

Similarly, a fairness condition on the control level states that the call will be processed and terminated: the cabin will reach its destination level. This fairness assertion (F_2) stating termination is the counterpart to fairness assertion (F_1) stating commencement.

```
ProcessingTerminates : assert                          (F2/P5)
  G (¬ reachedTarget → F reachedTarget);
```

The current call is finished before a new one is started.

```
NoTargetLevelChangeWhenBusy : assert                      (P6)
  G X ( (currServingState ≠ ready) →
    (currTargetLevel = previousTargetLevel) );
```

The call is processed by moving the cabin in the right direction unless there is an emergency brake.

```
ProcessingCorrect1a : assert                              (P7)
  G (currDirection ≠ stationary) ∧
    (currTargetLevel > currLevel) →
    (X currLevel > previousLevel ∨ emergencyBraked);
ProcessingCorrect2a : assert
  G (currDirection ≠ stationary) ∧
    (currTargetLevel < currLevel) →
    (X currLevel < previousLevel ∨ emergencyBraked);
```

An equivalent formulation of correct processing is given in the following two assertions.

```
ProcessingCorrect1 : assert                               (P8)
  G (¬ reachedTarget ∧ door = closed) ∧
    (currTargetLevel > currLevel) →
    ((X currLevel > previousLevel) ∨ emergencyBraked);
ProcessingCorrect2 : assert
  G (¬ reachedTarget ∧ door = closed) ∧
    (currTargetLevel < currLevel) →
    ((X currLevel < previousLevel) ∨ emergencyBraked);
```

The behaviour of the door of a cabin is crucial for a safe functioning of the elevators. We prove that the door closes, when a call is to be processed.

```
DoorClosesAtStartup : assert                              (D1)
  G (¬ reachedTarget) →
    (X door = closed);
```

Second, the door remains closed when the cabin is moving.

```
DoorClosedWhenMoving : assert                    (D2)
  G (currDirection ≠ stationary) →
    (door = closed);
```

The door is closed until the cabin stands.

```
DoorOpensAfterMoving : assert                    (D3)
  G (door = closed) U
    (currDirection = stationary);
```

Finally, the door opens after an emergency brake.

```
DoorOpensAfterEmergencyBrake : assert            (D4)
  G (emergencyBraked) →
    X (door = open);
```

5.3 Advanced Verification Techniques

All assertions have been successfully verified independently from each other. The proof task can be alleviated by skillfully exploiting assertion dependencies: Once the verifier has proven that a given property holds, it may use this result in future proofs. A domain expert can use the SMV assertion hierarchies (see Section 2) to express relationship between various assertions.

```
using ProcessingCorrect1a prove ProcessingCorrect1;
using ProcessingCorrect2a prove ProcessingCorrect2;

using ProcessingCorrect1 prove DirectionOK1;
using ProcessingCorrect2 prove DirectionOK2;

using ProcessingMoves1, ProcessingMoves2
  prove ProcessingCorrect1;
using ProcessingMoves1, ProcessingMoves2
  prove ProcessingCorrect2;

using DoorClosedWhenMoving
  prove DoorOpensAfterMoving;
using DoorOpensAfterMoving
  prove DoorOpensAfterEmergencyBrake;
```

As mentioned in Section 4.1, carrying out proofs by induction requires special care, as the TWIN elevators' behaviour is not fully generalizable in its number of storeys. For instance, a TWIN installation with three storeys would result in the cabins to be stucked immovable at the ground and the topmost floor; any positive safety distance would prevent them from moving.

A sensible lower bound $\texttt{NoLevels}_{min}$ for the number of levels is the warning distance plus two storeys where the cabins are actually placed. Section 5.1 illustrates the relationship between the three fundamental distances in a TWIN installation; the validity of these inequalities is checked during the model checking process.

$$\begin{aligned}\texttt{NoLevels}_{min}\ \texttt{-}\ 2 &\geq \texttt{WarningDistance}\\ &\geq \texttt{SafetyDistance}\\ &\geq \texttt{ForcedBrakeDistance > 0}\end{aligned}$$

Consequently, a proof's induction basis would be the minimal number of levels $\texttt{NoLevels}_{min}$. Our paper does not focus on the feasibility of proofs by induction; an alternative formal model would be required to allow for their execution.

5.4 Alternative Verification Techniques

An alternative approach to analyse the TWIN case study could either be to recur to different model checking concepts and techniques or to use different tool support. For instance, continuous modelling of real-time and hybrid systems is expressive and well tractable: practically relevant classes of continuous time systems can fully automatically be analysed and verified [12]. Yet, we opted for the more classical approach of LTL formula which allow seemless integration of fairness assertions into the logic – unlike CTL which requires a special "fairness" construct outside the logic. The SMV tool is acknowledged as a classical model checker for both LTL and CTL; newer developments as nuSMV may be used as well [5]. UPPAAL, another tool alternative, is an integrated environment under active development and may be seen as the most current model checking suite [20]. Following this paper's pragmatic impetus, we picked the slim SMV tool whose capabilities cope well with the case study's complexity.

6 Conclusions and Outlook

Software-based systems have spread throughout everydays's life and have replaced formerly used mechanical implementations. Safety-critical functions in hospitals, transport, aviation, and power generation are assured by the well-functioning of software systems, making these systems safety critical in turn. As a result, there is an obvious need for reliable and efficient techniques to verify the correctness of programs. Since testing's coverage is confined to solitary cases, safety-critical behaviour may be overlooked. Model checking can prove system correctness with mathematical precision. But the state explosion problem often hampered model checking techniques to be applied to complex real-world installations.

We successfully applied model checking techniques to prove the conformance of the TWIN's algorithm with its requirements specification. The TWIN elevator system, developed by ThyssenKrupp, is not only particularly well-suited for model checking analyses because of its reactive and time-discrete behaviour; having two cabins running independently in one elevator shaft imposes challenging requirements with regard to the control software.

Using the symbolic model checker SMV, we first modelled the TWIN behaviour with the appropriate granularity. Second, we deduced real-world and algorithmic prerequisites, consistency properties, and fairness constraints and expressed them in temporal logic. Our specification achieves to exhaustively map the TWIN's four-level safety concept. The verification of these properties is performed automatically by the SMV system. In this manner, we demonstrate that all the requirements con-

cerning the software control can be expressed and proven within minutes; we also manifest that well-crafted specifications allow model checking to capture a real-world's system's full complexity.

The algorithm described in Section 3.3 and implemented in SMV [17] represents an abstract control description for the TWIN elevator system. We demonstrated in this paper that this SMV representation conforms to the presented requirements specification. The safety properties of our specification plus algorithm are correct given that the SMV provides a correct implementation of temporal logic. Our code is close enough to a deterministic program to be directly implemented in microcode or higher programming languages. In this sense we have proven our algorithm correct.

We have developed a specification and algorithm topdown. In a bottom-up approach abstraction is used to mechanically model check running systems. Property preservation is assured by additional mechanized abstraction methods that reduce state spaces systematically, e.g. [8].

In summary, our case study portrays the rewarding application of model checking techniques to an innovative real world installation. It hereby surpasses most of the fictional examples currently discussed in literature [16]. We showed that a well-developed specification with an appropriate use of abstraction can make mechanical verification suitable for applications so far assumed to be out of reach. We are convinced our approach to be instructive to both industry and academia — in teaching and in research.

References

1. Bäumler S, Balser M, Dunets A, Reif W, Schmitt J (2006) Verification of Medical Guidelines by Model Checking – A Case Study `http://spinroot.com/spin/Workshops/ws06/027.pdf`
2. Bundesamt für Sicherheit in der Informationsgesellschaft (BSI) (2005) Common Criteria for Information Technology Security Evaluation, Part 3. `http://www.bsi.de/cc/ccpart3v2_3.pdf`
3. Burch J R, Clarke E M, Long D E (1994) Symbolic Model Checking for Sequential Circuit Verification. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 13, pp 401-424
4. Chan W, Anderson R J, Beame P, Burns S, Modugno F, Notkin D, Reese J D (1998) Model checking large software specifications. IEEE Transactions on Software Engineering, vol. 24, issue 7, pp 498–520, July 1998
5. Cimatti A, Clarke E M, Giunchiglia F, Roveri M (2000) NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D. (eds.) International Conference on Computer-Aided Verification (CAV'99), LNCS, vol. 1633, pp 495–499, Springer, Berlin / Heidelberg
6. Cimatti A (2000). Industrial Applications of Model Checking. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) Modeling and Verification of Parallel Processes (MOVEP'00), LNCS, vol. 2067, pp 153–167, Springer, Berlin / Heidelberg
7. Clarke E M, Grumberg O, Peled D A (1999) Model Checking. The MIT Press
8. Helke S, Kammüller F (2005) Property Preserving Abstraction for Statecharts. In: 25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, FORTE 2005, LNCS, vol. 3731, pp 305–319, Springer, Berlin / Heidelberg
9. Holzmann G J (1991) Design and Verification of Computer Protocols. Prentice Hall
10. Janssen W, Mateescu R, Mauw S, Fennema P, van der Stappen P (1999) Model checking for managers In: Proceedings Theoretical and Practical Aspects of SPIN Model Checking, LNCS, vol. 1680, pp 92–107, Springer, Berlin / Heidelberg
11. Lamport L (1994) The temporal logic of actions. ACM Trans. Program. Lang. Syst., vol. 16, pp 872–923, ACM Press, New York `http://doi.acm.org/10.1145/177492.177726`
12. Larsen K G, Steffen B, Weise C (1997) Continuous Modelling of Real Time and Hybrid Systems: From Concepts to Tools. International Journal on Software Tools for Technology Transfer, vol. 1, pp 64–85
13. Manna Z, Pnueli A (1991) The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, New York
14. McMillan K L (1992) Symbolic Model Checking – an Approach to the State Explosion Problem. School of Computer Science, Pittsburgh PA, Carnegie Mellon University
15. McMillan K L (1995) Symbolic Model Checking. Kluwer Academic Publishers
16. nuSMV (1999) NuSMV examples: the collection. `http://nusmv.irst.itc.it/examples/examples.html`
17. Preibusch S (2006) `http://preibusch.de/projects/TWIN/`
18. ThyssenKrupp (2005) Safe distance - Four-level safety concept. `http://twin-elevator.com/Safe_distance.353.0.html?L=1`
19. ThyssenKrupp (2005) Higher performance. `http://twin-elevator.com/New_buildings.368.0.html?L=1`
20. Uppsala University, Department of Information Technology (2006) UPPAAL. `http://www.uppaal.com/`