A Method and Tool for Tracing Requirements into Specifications

Stefan Hallerstede[a,*], Michael Jastram[b,*], Lukas Ladenberger[b]

[a]*Aarhus University, Denmark*
[b]*Heinrich-Heine University Düsseldorf, Germany*

Abstract

The creation of a consistent system description is a challenging problem of requirements engineering. Formal and informal reasoning can greatly contribute to meet this challenge. However, this demands that formal and informal reasoning and the system description are connected in such way that the reasoning permits drawing conclusions about the system description.

We describe an incremental approach to requirements modelling and validation that incorporates formal and informal reasoning. Our main contribution is an approach to requirements tracing that delivers the necessary connection that links the reasoning to the system description. Formal refinement is used in order to deal with large and complex system descriptions.

We discuss tool support for our approach of requirements tracing that combines informal requirements modelling with formal modelling and verification while tracing requirements among each other and into the formal model.

Keywords: Requirements; Specification; Traceability; Formal Modelling

1. Introduction

We describe an incremental approach for producing a system description from an initial set of informal requirements. The system description is composed of formal and informal *artefacts* that describe, for instance, assumptions about the environment or requirements proper. The central aspect we are concerned with is reasoning about these artefacts. We allow formal and informal reasoning but demand firm conclusions about satisfaction of the requirements and correctness of the specification to be part of the system description. In order to achieve this we need a method to trace artefacts such that the proofs we carry out formally and informally translate to corresponding validation statements in the system description. The approach must be incremental in the sense that changes to system descriptions and formal models should be considered frequent activities. These changes certainly happen in early development phases

[*] Corresponding authors

of a project but will usually continue during maintenance. This article expands on [25] and continues work that is presented in [21, 26].

Our approach to tracing is based on the WRSPM reference model [12] that identifies common categories of artefacts and specifies soundness conditions that they must satisfy. We extend the categories so that a mixture of formal and informal reasoning can be handled. Using the extended reference model we introduce different notions of traces between artefacts. These traces allow us to relate proofs to involved artefacts and to detect artefacts that are affected by the system description changes or the formal model associated with it. Usually, the elements of the system description are stated in natural language; and some approach for structuring large collections of natural language requirements is used such as [19, 35]. Such structured system descriptions are still understandable by stakeholders like customers and developers but already permit informal reasoning about the requirements. We use Problem Frames [19] in this article but other choices like [35] would be possible too.

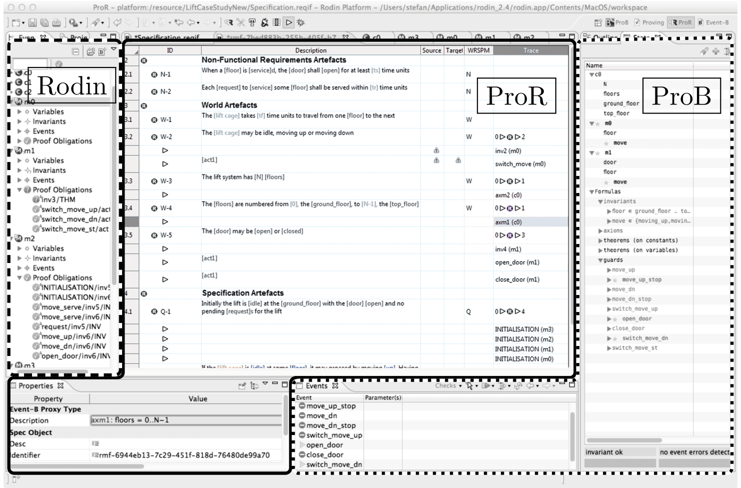
The formal method to be used should compatible with the predicative reasoning style of WRSPM. We consider state-based formalisms such as ASM [7], VDM [27], TLA$^+$ [31] or Event-B [2] particularly suited because they permit straightforward specification of state, state invariants and state transitions for modelling dynamic behaviour. The formal reasoning that we present in this article uses the Event-B method. In particular, we make use of the notion of refinement of Event-B to handle artefacts in small increments. This avoids having to trace a large number of refinements into one monolithic formal model. We not require that all artefacts are modelled formally, but those that are benefit from rigorous reasoning. Once modelled formally, artefacts can be analysed using automated verification by theorem provers or model checkers. For Event-B such tools are available in the form of Rodin [3] and ProB [32], for instance. To deal with natural language requirements and requirements tracing we use the tool ProR [20] that has been customised for use with Rodin and ProB. This permits us to evaluate our approach by tracing informal natural language artefacts among each other and to formal Event-B artefacts.

1.1. Tool Support

Keeping track manually of large sets of requirements and their relationships is not feasible. For this reason it is mandatory that the method for requirements modelling that we suggest be supported by a software tool. The tool ProR can be extended to achieve this. The generic method-independent characteristics of the tool are discussed in [20].

The genericity of ProR is achieved by means of the Eclipse Requirement Modelling Framework (RMF, http://eclipse.org/rmf) [24], which consists of a generic data model for requirements. Despite its genericity, a key objective in the development of ProR has been the support the approach described in this paper.

The generic ProR supports tracing natural language requirements in the form hierarchical tables. A dedicated column of each table summarises the incoming and outgoing traces.


ProR – platform:/resource/LiftCaseStudyNew/Specification.reqif – Rodin Platform – /Users/stefan/Applications/rodin_2.4/rodin.app/Contents/MacOS/workspace
ProB
Proving
ProR
Event-B
Even
Proje
*Specification.reqif
*rmf-7bed883b-255b-405f-b7
c0
m3
m0
m1
m2
Outline
State
Rodin
ProR
ProB
Checks

- c0
- c1
- c2
- m0
  - Variables
  - Invariants
  - Events
  - Proof Obligations
- m1
  - Variables
  - Invariants
  - Events
  - Proof Obligations
    - inv3/THM
    - switch_move_up/act
    - switch_move_dn/act
    - switch_move_st/act
- m2
  - Variables
  - Invariants
  - Events
  - Proof Obligations
    - INITIALISATION/inv5
    - INITIALISATION/inv6
    - move_serve/inv5/IN
    - move_serve/inv6/IN
    - request/inv5/INV
    - move_up/inv6/INV
    - move_dn/inv6/INV
    - open_door/inv6/INV
- m3

		ID	Description	Source	Target	WRSPM	Trace
2	Ⓡ		**Non-Functional Requirements Artefacts**				
2.1		Ⓡ N-1	When a [floor] is [service]d, the [door] shall [open] for at least [ts] time units			N	
2.2		Ⓡ N-2	Each [request] to [service] some [floor] shall be served within [tr] time units			N	
3	Ⓡ		**World Artefacts**				
3.1		Ⓡ W-1	The [lift cage] takes [tf] time units to travel from one [floor] to the next			W	
3.2		Ⓡ W-2	The [lift cage] may be idle, moving up or moving down			W	0 ▷ Ⓡ ▷ 2
		▷		⚠			inv2 (m0)
		▷	[act1]	⚠	⚠		switch_move (m0)
3.3		Ⓡ W-3	The lift system has [N] [floors]			W	0 ▷ Ⓡ ▷ 1
		▷					axm2 (c0)
3.4		Ⓡ W-4	The [floors] are numbered from [0], the [ground_floor], to [N-1], the [top_floor]			W	0 ▷ Ⓡ ▷ 1
		▷					axm1 (c0)
3.5		Ⓡ W-5	The [door] may be [open] or [closed]				0 ▷ Ⓡ ▷ 3
		▷					inv4 (m1)
		▷	[act1]				open_door (m1)
		▷	[act1]				close_door (m1)
4	Ⓡ		**Specification Artefacts**				
4.1		Ⓡ Q-1	Initially the lift is [idle] at the [ground_floor] with the [door] [open] and no pending [request]s for the lift			Q	0 ▷ Ⓡ ▷ 4
		▷					INITIALISATION (m3)
		▷					INITIALISATION (m2)
		▷					INITIALISATION (m1)
		▷					INITIALISATION (m0)
			If the [lift cage] is [idle] at some [floor], it may proceed by moving [up]. Having				

Properties

Property	Value
Event-B Proxy Type	
Description	axm1: floors = 0..N−1
Spec Object	
Desc	
Identifier	rmf-6944eb13-7c29-451f-818d-76480de99a70

Events

Event	Parameter(s)
move_up_stop	
move_dn	
move_dn_stop	
switch_move_up	
open_door	
close_door	
switch_move_dn	

Name

- c0
  - N
  - floors
  - ground_floor
  - top_floor
- ☆ m0
  - floor
  - ☆ move
- ☆ m1
  - door
  - floor
  - ☆ move
- Formulas
  - invariants
    - floor ∈ ground_floor ‥ to…
    - move ∈ {moving_up,movin…
  - axioms
  - theorems (on constants)
  - theorems (on variables)
  - guards
    - move_up
    - ☆ move_up_stop
    - move_dn
    - move_dn_stop
    - switch_move_up
    - ☆ open_door
    - close_door
    - ☆ switch_move_dn
    - switch_move_st

invariant ok	no event errors detect

Figure 1: Integration of ProR with Rodin and ProB

ProR allows the customisation of its meta-model for concrete notions of requirements. Such a customisation may, for example, consist of adding a new type of requirements with a specified number of typed attributes. The display of the requirements models can also be customised, for example, by showing only selected attributes.

Our approach is supported by means of extending ProR. ProR is integrated with the Rodin formal modelling platform [3] and ProB [32]. The tools Rodin and ProB in combination provide support for proof, model checking and animation of formal models specified in the Event-B notation. The integration with these tools provides support the central aspect of our approach: to exploit formal reasoning as much as possible for modelling and analysing informal requirements. A screenshot of the resulting tool is shown in Fig. 1.

1.2. Running Example

Throughout the paper, we use a lift system to demonstrate our approach to requirements modelling and validation. The example is taken from [5], which provides already a structured system description. An important assumption underlying [5] is that all artefacts can be directly represented as logical formulae, in particular, relying on a dedicated logic for real-time modelling. In order to illustrate mixing formal and informal reasoning we do not use such a logic but use only a formalism for discrete modelling. The lift system is described in general terms in [5]:

> "A simple, single lift system allows movement of a single lift cage between a finite number of floors, the starting and stopping of the lift cage and the opening and closing of floor doors — all in response to the pressing of floor call and cage send buttons."

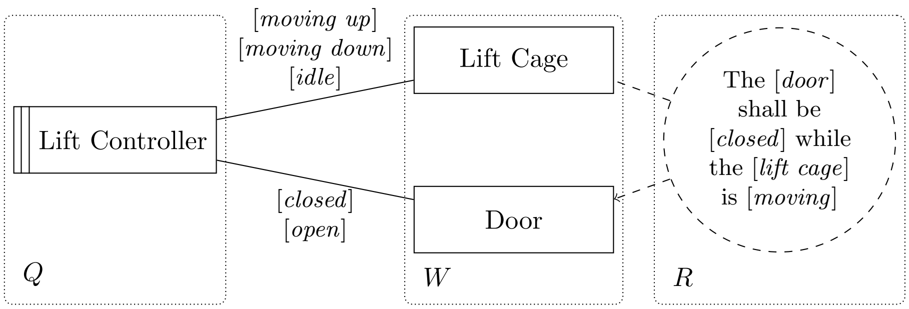


Figure 2: A problem diagram describing door behaviour

We begin by treating the running example informally using Problem Frames [19].

Example 1. Problem Frames offer a way to structure and analyse a system description informally. They can easily be related to the model for requirements tracing that we introduce in Section 2. Although we do not discuss informal structuring and analysis in detail, we find it instructive to show a concrete way to work with the rather abstract model for requirements tracing.

A *problem diagram* for a specific aspect of the lift system is shown in Fig. 2. The diagram shows a *machine domain* called “Lift Controller” that is to be designed and built. The *given domains* “Lift Cage” and “Door” are part of the environment. Their properties are given and cannot be designed. Requirements are drawn as dashed ovals. The requirement shown in this problem diagram says: “The [*door*] shall be [*closed*] while the [*lift cage*] is [*moving*]”. It describes a property of that the lift cage and the door together should satisfy. This is visually indicated by the edges to the corresponding given domains. Edges are decorated with *phenomena* that the connected domains or requirements *share*. For instance, the lift controller and the door share the phenomena [*open*] and [*closed*]. The brackets around the phenomena are our notation. They permit us to trace phenomena and verify their consistent usage. This is explained in Section 2 in terms of the model for requirements tracing that uses the term phenomenon in the same sense. We have partitioned the different domains and the requirements into three distinct parts marked by Q, W and R. Members of these parts are called *artefacts*: Q are the specification elements, W the domain properties and R the requirements. Artefacts are the basis for tracing requirements in our approach. Fig. 2 illustrates how the artefacts can be related to a concrete structured system description.

The concept of a Problem Frames domain does not appear in our approach. Instead, we subsume domains as phenomena in our approach, arguing that they are constructs that group phenomena.

Problem diagrams can grow very large with many requirements and many shared phenomena. In order to keep them as small as possible they can be structured into *sub-problem diagrams*. The sub-problem diagrams are *projections* of a larger diagram. They may describe and constrain overlapping portions of that larger diagram.

Analysis of problem diagrams involves asking whether the right domains are present in order to express the problems to be solved. In fact, the Problem Frames approach has a specific kind of diagram for this sort of analysis, the *context diagram*. Once problem diagrams have been created, we can reason about them by means of *frame concerns*. A frame concern is an informal argument why, given the domain properties, the specification satisfies a requirement. In the Problem Frame approach they take the role of informal proofs.

1.3. Structure of this Paper

Section 2 describes a model for tracing requirements based on WRSPM. In Section 3 we describe the formal method Event-B which we use for formal requirements validation. Section 4 describes a process of how to produce a system description incrementally following our approach of requirements tracing. Finally Sections 5 and 6 discuss related work and draw some conclusions.

2. A Model for Requirements Tracing

Our approach is based on WRSPM by Gunter et. al. [12]. The objective of our approach is to produce a system description of "high quality" by establishing a traceability that allows a systematic validation of the system description and provides robustness with respect to changes in the system description. It further allows the mixing of formal and informal elements, thereby enabling rigorous reasoning where it is desired. WRSPM is a reference model for applying formal methods to the development of user requirements and their reduction to a behavioural system specification. The reference model distinguishes artefacts and phenomena. Phenomena describe the state space and state transitions of an environment and a system, while artefacts describe constraints on the state space and the state transitions. *Artefacts* (A, B) are distinguished into the following categories corresponding to the WRSPM reference model described in [12]:

- *domain properties* (W)
  describe facts about how the world behaves;
- *requirement items* (R) and (N)
  describe how the world shall behave once the system is built;
- *specification elements* (Q)
  describe a system whose implementation satisfies the requirements;
- *implementation elements* (P)
  provide an implementation of the specifications;
- *design decisions* (U)
  describe design options chosen for specification and implementation;

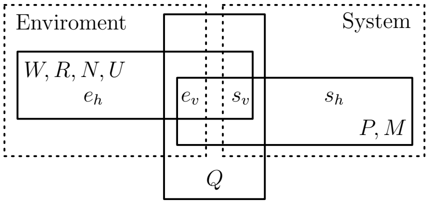


Figure 3: Variant of the WRSPM reference model with design decisions

[*lift cage*] (e_h)	is the cabin carrying the [*passenger*] (e_h)
[*floor*] (e_v)	is the current position of the [*lift cage*] (e_h)
[*open*] (s_v)	is the status of the [*door*] (e_v) being open
[*closed*] (s_v)	is the status of the [*door*] (e_v) being closed
[*request*] (e_v)	is the command by the [*passenger*] (e_h) to service some [*floor*] (e_v)

Figure 4: Excerpt of the lift system glossary

- *platform properties* (M)
  provide an execution environment for the programs.

We have extended the reference model introducing *design decisions* U and an additional class of *requirement items* N as artefacts. We also changed the symbol for specification elements from S (used in WRSPM) to Q, to avoid ambiguities when discussion state based modelling in Section 3. The requirement items N are those requirements that are validated by purely informal reasoning. These will usually include *non-functional requirements* but also requirements that, intentionally, remain informal during modelling. By contrast, requirement items R are intended to be validated by formal reasoning.

Fig. 3 illustrates the different kinds of phenomena and artefacts. Phenomena p are distinguished by whether they are controlled by the system, belonging to set s, or the environment, belonging to set e. They partition the set of all phenomena, that is $s \cap e = \varnothing$ and $s \cup e = p$. Furthermore, phenomena are distinguished by visibility. Environmental phenomena may be visible to the system, belonging to e_v, or hidden from it, belonging to e_h. Similarly, system phenomena belonging to s_v are visible to the environment, while those belonging to s_h are hidden from it. These classes of phenomena are mutually disjoint.

Example 2. All phenomena are listed and described in a glossary. An excerpt of the glossary for the lift system is shown in Fig. 4.

The distinction between environment and system is important; omitting it can lead to misconstrued specifications. The boundary between environment and system is often based on the discretion of the user and may be based on taste or convenience. Nevertheless, it has a profound impact on the problem

R-1	The current [*floor*] shall be between the [*ground floor*] and the [*top floor*]
R-2	If the [*lift cage*] is [*moving up*] or [*moving down*], the [*door*] shall be [*closed*]
N-1	When a [*floor*] is [*service*]d, the [*door*] shall [*open*] for at least [t_s] time units
N-2	Each [*request*] to [*service*] some [*floor*] shall be served within [t_r] time units
W-1	The [*lift cage*] takes [t_f] time units to travel from one [*floor*] to the next
W-2	The [*lift cage*] may be [*idle*], [*moving up*] or [*moving down*]
W-3	The lift system has [N] [*floors*]
W-4	The [*floors*] are numbered from [*0*], the [*ground floor*], to [N], the [*top floor*]

Figure 5: Some requirement items and domain properties of the lift system

analysis and the obligations to be respected between environment and system. The distinction serves to clarify the responsibilities and interfaces between the system and the environment [12].

The artefacts W, R and U may only be expressed referring to phenomena that are visible in the environment, that is $e \cup s_v$. Likewise, P and M may only be expressed referring to phenomena that are visible to the system, that is $s \cup e_v$. Artefacts Q may only be expressed referring to phenomena that are visible to both the system and the environment, that is $e_v \cup s_v$.

Example 3. Fig.. 5 shows some domain properties and requirement items for the lift system. Note that the travel time between floors, W-1, is considered a domain property whereas the the service time for a specific floor, N-1, is considered a requirement item: We cannot influence the travel time but we are supposed to control the service time.

2.1. Reasoning with Artefacts

Once a system is modelled following our approach, a number of properties can be verified with regard to the model, one being *adequacy with respect to* Q:

$$\forall e, s \cdot W \wedge Q \Rightarrow R \wedge U \ . \tag{1}$$

It says that the specification constrains the world such that the requirements and design decisions are realised. Note that if the world is vacuous, that is, $\neg(\exists e, s_v \cdot W)$, the implication would be trivial to satisfy by any specification. However, if the world W is consistent, then we expect the development method to preserve it: a specification Q must not be permitted to falsify the premise $W \wedge Q$. We expect the specification Q to be feasible assuming W. This can be achieved by construction using refinement, e.g., [1].

Example 4. Fig. 6 shows some specification elements for the lift system. We could argue, informally, that Q-2 should contribute to satisfying R-1 given the domain property W-2 about possible lift cage movements.

Q-1	Initially the lift is [*idle*] at the [*ground floor*] with the [*door*] [*open*] and no pending [*request*]s for the lift
Q-2	If the [*lift cage*] is [*idle*] at some [*floor*], it may proceed by [*moving up*]. Having reached the [*top floor*] the [*lift cage*] does not proceed by [*moving up*]
Q-3	If the [*lift cage*] is [*idle*] and at some [*floor*], the [*door*] may be [*open*]ed

Figure 6: Some specification elements of the lift system

U-1	The [*lift cage*] switches from [*moving up*] to [*moving down*] if the [*top floor*] is reached or there are no upwards [*request*]s but downwards [*request*]s
U-2	The [*lift cage*] switches from [*moving down*] to [*moving up*] if the [*ground floor*] is reached or there are no downwards [*request*]s but upwards [*request*]s

Figure 7: Some design decisions of the lift system

Example 5. Fig. 7 shows some design decisions for the lift system. The design decisions U-1 and U-2 contribute to the satisfaction of requirement item N-2. So do W-1 and N-1. We can argue informally that it takes maximally $t_b = (2 * N - 3) * (t_s + t_f)$ time units until the lift cage reaches a specific floor. Hence, if t_r is chosen such that $t_r \geq t_b$, then N-2 is satisfied. Whether t_b is an acceptable bound has to be confirmed with the stakeholders.

The implementation should also satisfy a condition similar to adequacy:

$$\forall e, s \cdot W \wedge M \wedge P \Rightarrow R \wedge U \ . \tag{2}$$

If we have already established adequacy, (2) can be achieved by refinement:

$$\forall e, s \cdot W \wedge M \wedge P \Rightarrow Q \ . \tag{3}$$

The latter formula (3) reflects the refinement condition for relations presented in [15]. An additional side condition provides for feasibility. We use the refinement approach of [2] that permits additionally changing the data-representation. However, for the present purposes we do not make use of this additional concept.

Non-functional requirements depend, in particular, on design decisions. This aspect of non-functional requirements is discussed in [8]. Design decisions introduce architectural concepts or constrain the implementation, for example.

$$\forall e, s \cdot W \wedge R \wedge U \wedge Q \Rightarrow N \ . \tag{4}$$

We assume that often non-function requirements will not be formal. Hence, formula (4) will usually consist of formal and informal artefacts with the conclusion N being informal.

The implications in the formulae (1) to (4) indicate relationships between specific artefacts. For instance, a specific specification element Q' may imply a specific requirement item R'. We can also say that we can trace requirement

R' to specification element Q'. The reference model provides the foundation for our approach of requirement traceability. We also cast the refinement theory of Event-B conceptually into the reference model so that we can trace requirements among formal artefacts, among informal artefacts and across formal and informal artefacts.

2.2. Tracing of Artefacts and Phenomena

In order to trace requirements we need to define relationships between artefacts. Currently, we are not interested in tracing implementation elements P and platform properties M. We focus on the relationship between specification Q and design decision U on one side and requirements items R and N, as well as domain properties W on the other.

We are interested in tracing *justifications* of artefacts, *equivalence* between artefacts, *evolution* of artefacts and tracing of phenomena *used* in artefacts. We discuss the different kinds of tracing in turn.

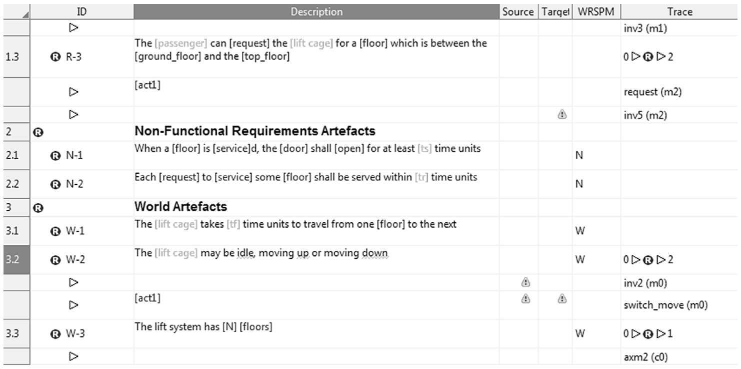
2.2.1. Tracing Artefact Justification

We say B justifies A, or $B \leftarrow A$, if B justifies the *presence* of artefact A. If an artefact appears in the system description, its presence should be justified. It should be there for a reason. If we read implications like (1) from the right to the left we arrive at justifications for the involved artefacts. We say $R \wedge U$ justify $Q \wedge W$. We would like $Q \wedge W$ not to contain more artefacts than necessary in order to establish (1). We call a subset SB of the artefacts $Q \wedge W$ such that $SB \Rightarrow R \wedge U$ a *satisfaction base* [28] for $R \wedge U$. We are particularly interested in small satisfaction bases to obtain as precise justifications as possible. There may not be a minimal unique satisfaction base nor may it be feasible to find it if a minimal set exists. A good estimate is practically sufficient though. Such an estimate can be derived by looking at artefacts mentioned in proofs —formal and informal. Satisfaction bases have been explored in the context of relevance logic [39]. Reading a justification $B \leftarrow A$ in the reverse direction $A \rightarrow B$ we say that A *realises* B. We can rephrase statements about justification using realisation, for instance: each specification element or domain property should be there for a reason. It should realise requirement items or design decisions. The additional notion of realisation provides us with a tracing concept that is similar to implication, unlike justification which is similar to reverse implication. The similarity should not be taken to far, however. In particular, realisation and implication are not the same. Realisation is just a relation between artefacts. A realisation does not state that its left-hand side logically implies its right hand side.

Example 6. The informal proof in Example 5 indicates that there should be a corresponding justification trace.

→

W-1, N-1, U-1, U-2 → N-2



	ID	Description	Source	Target	WRSPM	Trace
	▷					inv3 (m1)
1.3	Ⓡ R-3	The [passenger] can [request] the [lift cage] for a [floor] which is between the [ground_floor] and the [top_floor]				0 ▷ Ⓡ ▷ 2
	▷	[act1]				request (m2)
	▷			⚠		inv5 (m2)
2	Ⓡ	**Non-Functional Requirements Artefacts**				
2.1	Ⓡ N-1	When a [floor] is [service]d, the [door] shall [open] for at least [ts] time units			N	
2.2	Ⓡ N-2	Each [request] to [service] some [floor] shall be served within [tr] time units			N	
3	Ⓡ	**World Artefacts**				
3.1	Ⓡ W-1	The [lift cage] takes [tf] time units to travel from one [floor] to the next			W	
3.2	Ⓡ W-2	The [lift cage] may be idle, moving up or moving down			W	0 ▷ Ⓡ ▷ 2
	▷		⚠			inv2 (m0)
	▷	[act1]	⚠	⚠		switch_move (m0)
3.3	Ⓡ W-3	The lift system has [N] [floors]			W	0 ▷ Ⓡ ▷ 1
	▷					axm2 (c0)

Figure 8: The ProR editor for artefacts and traces

If the informal proof associated with N-2 would mention the artefacts that realise it, then this trace could be generated automatically in ProR. At the moment this is not done. ⌟

Tool Support for Tracing Justification in ProR. Tool support for justification traces needs, in particular, to deal with *creation* and *modification* of traces. We describe briefly how ProR realises this.

- *Creation*
  Traces between artefacts can be created via drag and drop. This includes traces between informal artefacts, as well as informal and formal ones. The tool visualises the resulting traces in the table view that shows the artefacts, as shown in Fig. 8. The column "Trace" summarises the number of incoming and outgoing links, and the trace information itself can be unfolded beneath, marked with a triangle, as shown in the figure as well. Traces can be annotated if additional information is necessary.
- *Modification*
  If the source or target of a trace changes, then the trace is marked as "suspect" by showing a small icon in a dedicated column. This is also shown in Fig. 8. Having validated a trace, double-clicking the corresponding marker removes the "suspect" icon. This is described in more detail in Section 4.1.8.

2.2.2. Tracing Artefact Equivalence

In our approach some but not all artefacts may be formal. Often formal artefacts have informal counterparts. If an artefact A is *formal*, we write A_F. We write B_I if B is *informal*. When formalising informal requirement items we often get direct correspondences between informal items A_I and formal items B_F. We say that these items are *equivalent*, denoted by $A_\mathrm{I} \leftrightarrow B_\mathrm{F}$. Equivalence tracing is particularly useful when dealing with domain properties in formal proofs. Domain properties are on the left hand side of all implications (1) to (4). We take the relation $B\text{F}$ realises $A\text{I}$, $B\text{F} \rightarrow A\text{I}$, to reflect the implication $B\text{F} \Rightarrow A\text{I}$. If $A\text{I}$ and $B\text{F}$ are only related by implication following this correspondence, statements about formal world properties may not hold with respect to the corresponding informal world properties. For this to hold we need either $A\text{I} \rightarrow B\text{F}$, that is the formal assumption about the world are not stronger than the the informal assumptions, or equivalence $A\text{I} \leftrightarrow B\text{F}$. Equivalence tells us that the informal domain properties are not stronger than needed for building the system.

2.2.3. Tracing Artefact Evolution

A system description *evolves* over time. This may happen due to changing requirement items or due to improvements to the description made by modelling and reasoning. Evolution does not follow logical implication. The best we can do is to record approximately how artefacts have changed over time based on differences between different revisions of the system description. We write $A \rightsquigarrow B$ for A evolves into B. Evolution traces are needed for the benefit of various stakeholders to follow original requirement items into current revision of the system description.

2.2.4. Tracing Used Phenomena

The partitioning of the phenomena in Fig. 3 indicates that it is important to trace phenomena into various artefacts. We need to verify that the various artefacts —informal and formal— only refer to allowed phenomena as outline in the figure; the description ought to be syntactically sound. Furthermore, we are interested in verifying that formalised artefacts refer to those phenomena specified in the corresponding informal artefacts be it related by justification or equivalence. We have only little means in our hand to achieve consistency between formal and informal artefacts. This one appears simple and effective, similarly, to type checking or the use of alphabets in UTP [15]. We record references from artefacts to phenomena saying that A uses p, denoted by, $p \in A$. This just means that A makes some statement about p. The management of this relationship can be handled efficiently by proper tool support, as described in Section 1.1.

Example 7. By marking words occurring in artefacts in square brackets they are recognised as phenomena and a corresponding uses-trace is generated. The tool ProR does this automatically. Marking a word p in an artefact A specifies the trace $p \in A$: the artefact A makes some statement about p. For instance, the requirement item R-1 uses the phenomena *floor*, *ground floor* and *top floor*.

floor, ground floor, top floor $\in$ R-1

Uses traces can be useful for informal proof. They allow us to search for related phenomena, a technique that is borrowed from formal proof.

Tool Support for Tracing Used Phenomena in ProR. Tracing of phenomena is supported by highlighting text passages in the natural language artefacts that refer to phenomena, as shown in Figure 8. If a passage is found that is textually identical to a declared phenomenon, ProR underlines it. A phenomenon is declared by including it in a glossary. In order to add a uses-trace for an phenomenon to an artefact the corresponding text passage is put in square brackets. If the phenomena has been declared, the text passage is rendered in blue, otherwise in red, reminding the user that an undeclared phenomenon is used. Underlining text passages that are found in the glossary serves to remind the user that this passage may represent an untraced phenomenon.

3. Formal Modelling and Refinement

Our approach to requirements tracing could be used with a wide range of formal methods for state-based modelling that have an associated notion of refinement. In this paper, we use Event-B which we introduce in Section 3.1. Based on Event-B we also discuss limitations of formalisation: not all requirements can be formalised within the core Event-B formalism. For this reason, formal and informal reasoning need to be combined in a sensible way. The boundary of formalisation in the example is given by temporal and real time properties. We have intentionally chosen a boundary that could be moved by using another formal method or extending Event-B. We think the boundary is not fixed and may vary depending on characteristics of development projects to it may move as a development progresses. We think of modelling and requirements validation as an incremental process: we permit the boundary to be changed as need arises.

We take advantage of the concept of refinement supported by Event-B. Other notions of refinement could be used without changing the approach fundamentally. Our approach of requirements tracing allows us to account for additional requirements at later refinement stages, thereby, providing a structuring mechanism for the introduction of requirements into the formal model. Tracing requirements into and within Event-B models is based on the Event-B proof obligations described in Section 3.2. The approach to tracing requirements in Event-B is described in the subsequent Section 3.3.

3.1. Event-B Models

Formal models in Event-B consist of *contexts* K and *machines* M. Contexts provide their static properties while machines provide behavioural properties. A context K can be *extended* by another context L. We call K an *abstract* context and L a *concrete* context. The behaviour of a machine is expressed in terms of events E. Events model *transitions* between the states of the machine. A dedicated event models the *initialisation* of the machine. An invariant I is specified for a machine that should always hold. This must be proved. In Section 3.2 we describe the corresponding *proof obligations.* Before turning to the proof obligations we describe in the following two Sections 3.1.1 and 3.1.2 the syntax of machines and contexts in more detail. When introducing constructs such as invariants or axioms we indicate in brackets free identifiers that may occur in them.

3.1.1. Event-B Contexts

Contexts K consist of *carrier sets* and *constants.* Carrier sets s are similar to types in other formalisms. Constants c are constrained by axioms $C(s, c)$. Contexts are *seen* by machines. All carrier sets, constants and axioms of a context seen by a machine M are visible in M.

Context K can be *extended* by another context L with carrier sets t, constants d and axioms $D(s, t, c, d)$. The carrier sets s and t as well as the constants c and d must be distinct. All axioms C of K become axioms of L, too.

3.1.2. Event-B Machines

Machines provide behavioural properties of Event-B models. Machines consist of *variables* , *invariants*, *events.* Variables v describe the state of a machine. They are constrained by invariants $I(s, c, v)$ where s and c are carrier sets and constants of a seen context.

Events. Possible *transitions* between states of the machine are described by means of events. Each event is composed of a *guard* $G(p, v)$ and an *action* $A(p, v)$, where p are *parameters* of the event. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by any p when $G(p, v)$ then $A(p, v)$ end in its most general form, or when $G(v)$ then $A(v)$ end if the event does not have parameters, or begin $A(v)$ end if in addition the guard equals *true.* A dedicated event of the third form is used for *initialisation.*

Actions. The action component of an event is composed of *assignments.* Let x be variables, a subset of v. Assignments in Event-B are either deterministic, $x := e$ where $e(p, v)$ are expressions, or non-deterministic: either it is a non-deterministic choice from a set, $x :\in e$, where $e(p, v)$ is an expression, or it is a non-deterministic choice of a x' satisfying a predicate, $x :\mid Q$, where $Q(t, v, x')$ is a predicate. The first two forms of assignment can be expressed in terms of the third: the deterministic assignment $x := e$ is the equivalent of $x :\mid x' = e$ and the non-deterministic assignment $x :\in e$ the equivalent of $v :\mid v' \in e$.

Transitions. The effect of an *assignment* can also be described by a *before-after predicate*: before-after-predicate-of "$x :\mid Q$" $\widehat{=}$ Q. A before-after predicate is used to describe the relationship between the state just before an assignment has occurred (unprimed variable names x within v) and the state just after the assignment has occurred (primed variable names x'). All assignments of an action A occur simultaneously which is expressed by conjoining their before-after predicates, yielding a single predicate P. Variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining P with $y' = y$, yielding the before-after predicate $S(p, v, v')$ of the *action*: $S \mathrel{\widehat{=}} P \wedge y' = y$. Note that the guards are not part of the before-after predicate. In the presentation of the proof obligations below we use events with actions specified in terms of their before-after predicate, that is, events E of the form any p when G then $v :\mid S$ end.

Refinement. A machine M with variables v and invariant $I(v)$ can be *refined* by another machine N with variables w and invariant $J(v, w)$; we also say that M is an *abstraction* of N We call M an *abstract* machine and N a *concrete* machine. Invariant J is called a *gluing invariant.* It relates the states of the abstract machine to those of the concrete machine. A concrete machine N can refine at most one abstract machine M. The reason for this is that the concrete machine N contains the predicate J that links N to the abstract machine M. Invariants are built up incrementally during refinement, preserving invariants from abstractions: the invariant I of the abstract machine M is joined with the invariant J of the concrete machine N in machine N. So the full invariant of N is $I \wedge J$.

Machine N refines M if N behaves similarly to M. This is expressed more precisely by relating abstract and concrete events by *event refinement*: each abstract event $E(v) =$ any p when $G(p, v)$ then $v :\mid S(p, v, v')$ end is *refined* by one or more concrete events $F(w) =$ any q when $H(q, w)$ with $Z(p, q, v, v', w, w')$ then $w :\mid T(q, w, w')$ end. Informally, concrete event F refines abstract event E if the guard H of F is at least as strong as the guard G of E, and the gluing invariant J establishes a simulation of the action of F by the action of E. The concrete event F contains an additional predicate Z following the keyword with, called the *witnesses* for p and v'. Somewhat simplified, they link the abstract parameters p and the abstract variables v' to concrete parameters q and concrete variables and w'. Witnesses describe for each event separately how the refinement is achieved. It is possible to introduce new events in a refinement. They are required to refine the implicit abstract event *skip* $\mathrel{\widehat{=}}$ begin $v := v$ end.

Refinement permits us to verify a large number of properties piecemeal. This is particularly important when dealing with complex systems consisting of many properties to be modelled and verified. We also exploit this for tracing requirements by disentangling intricate properties into simpler ones [13].

Remark on Convergence and Deadlock-Freedom. Event-B also has the notions of *convergence* and *deadlock freedom* [2]. Proving convergence we show that events introduced newly in a refinement do not take control forever. Among other things this can be used to prove loop termination [2]. Deadlock freedom in Event-B means that some concrete event may occur whenever some abstract event could occur. This property ensures that we cannot remove behaviour in a refinement. In our approach this property could be interesting for showing that domain properties modelled by events are not accidentally strengthened in a refinement. For the purpose of this article we do not discuss convergence and deadlock freedom further because they do not contribute new insights.

3.2. Event-B Proof Obligations

We present proof obligations for *machine consistency* in Section 3.2.1 and proof obligations for *machine refinement* in Section 3.2.2. During the presentation of the proof obligations we comment on some issues concerning the reference model discussed in Section 2 before explaining our approach of tracing requirements into formal models in Section 3.3.

3.2.1. Consistency Proof Obligations

Transition Proof Obligations. For an event $E = \mathsf{any}\ p\ \mathsf{where}\ G\ \mathsf{then}\ v :\mid S\ \mathsf{end}$ we have to prove *feasiblity* of the event: $C \wedge I \wedge G \Rightarrow \exists v' \cdot S$. Feasibility expresses that S describes an after state whenever the guard G holds. This means that the guard indeed represents an enabling condition of the event.

Invariants should hold whenever an event possibly changes the values of some variable. The corresponding proof obligation is called *invariant preservation*: $C \wedge I \wedge G \wedge S \Rightarrow I[v := v']$. This proof obligation verifies that if the invariant I holds before the event E occurs, then it also holds after it has occurred. In other words, the invariant I is preserved by event E.

The two consistency proof obligations address the problem of the feasibility of domain properties and specification elements discussed in Section 2.1. For those domain properties modelled by events and invariants we also verify feasibility. There remain those domain properties modelled by the axioms C for which feasibility is not verified. The software tool ProB provides some means to check axioms for consistency and gives feedback, that is, a counterexample, if checking for consistency fails [33].

Example 8. Let us formalise the domain properties W-3 and W-4.

axiom *axm1*: $floors = 0 \mathrel{..} N$
axm2: $\mathrm{card}(\mathit{floors}) = N$

The corresponding traces are: *axm1* ← W-4 and *axm2* ↔ W-3. (W-4 should be split into three different domain properties. We have not done this to save space.) Unfortunately, *axm1* and *axm2* contradict each other. We have to correct the domain properties and decide to number the floors from 0 to $N - 1$. We change W-4 to

W-4	The [*floors*] are numbered from [*0*], the [*ground floor*], to [*N-1*], the [*top floor*]

and keep an evolution trace to the old version of the domain property. The inconsistency between W-3 and W-4 is already contained in [5]. This just shows how tempting it is to formalise a model without tracing artefacts. It is all too easy not to spot problems in informal descriptions and ignore inconsistencies building by mistake the right model but for wrong artefacts.

Initialisation Proof Obligations. For an initialisation we have to prove *feasibility* and *invariant establishment*. These two proof obligations together imply that the state space of the machine is not vacuous provided the axioms C are free of contradictions. The proof obligations of invariant establishment and invariant preservation together manifest an inductive proof that the invariant always holds. So we could also say that the machine M satisfies the temporal property "**always** I".

3.2.2. Refinement Proof Obligations

We only discuss transition proof obligations. Initialisation proof obligations are similar. For abstract event $E = \mathsf{any}\ p\ \mathsf{where}\ G\ \mathsf{then}\ v :\mid S\ \mathsf{end}$ and concrete event $F = \mathsf{any}\ q\ \mathsf{where}\ H\ \mathsf{with}\ Z\ \mathsf{then}\ w :\mid T\ \mathsf{end}$ where F refines E we have to prove *feasibility* of F. It is similar to feasibility of the abstract event except that we additionally assume D and J in the premise.

The proof obligations (5) to (8) verify that the abstract event E can simulate the concrete event F. A consequence of this simulation is that invariant preservation proved for the abstract event also holds for the concrete event.

Guard strengthening establishes that the concrete event cannot occur more often than the abstract event:

$$C \wedge D \wedge I \wedge J \wedge Z \wedge H \wedge T \Rightarrow G \ . \tag{5}$$

Action simulation demonstrates that the action S of the abstract event can simulate the action T of the concrete event:

$$C \wedge D \wedge I \wedge J \wedge Z \wedge H \wedge T \Rightarrow S \ . \tag{6}$$

Invariant preservation verifies that the gluing invariant J is preserved:

$$C \wedge D \wedge I \wedge J \wedge Z \wedge H \wedge T \Rightarrow J[v, w := v', w'] \ . \tag{7}$$

Note that in the formulae (5) and (7) the witness Z constrains w' to satisfy S. The existence of the witness must be established by a separate proof obligation, called, *witness feasibility*:

$$C \wedge D \wedge I \wedge J \wedge H \wedge T \Rightarrow \exists w', q \cdot Z \ . \tag{8}$$

In combination the proof obligations (5) to (8) serve to verify the following property:

$$C \wedge D \wedge I \wedge J \wedge H \wedge T \Rightarrow \exists q, w' \cdot G \wedge S \wedge J[v, w := v', w'] \ . \tag{9}$$

It is a standard proof technique for refinement called *forward simulation* (e.g. [41]). The introduction of witnesses Z has the methodical advantage of explicating the relationship between the abstract and the concrete event. But it makes requirements tracing easier, too. The proof obligations (5) to (8) provide a much better and robust basis for tracing. We can record which premises have been used in the proof per proof obligation. If forward simulation was proved directly by means of (9), used premises would be much more difficult to determine for the constructs occurring the conclusion.

3.3. Tracing Requirements in Event-B

Event-B models consist of formal constructs such as axioms, invariants and events. These should be related to informal artefacts such as domain properties, specification elements and requirements. We assume that non-functional requirements are left to informal reasoning. Hence, when tracing requirements from a formal model we always refer to functional requirements R. In the following we denote by $X\textsc{i}$ an informal artefact and by $X\textsc{f}$ a formal construct. The decorations $\textsc{i}$ and $\textsc{f}$ are only used for emphasis: we keep the letters as employed in the Sections 2.1 and 3.2 above. We also distinguish strictly between informal artefacts and formal constructs for emphasis.

3.3.1. Tracing and Correctness

In the formal model we must not strengthen the assumptions we make by means of the domain properties. That is domain properties $W\textsc{i}$ must realise formal constructs $A\textsc{f}$:

$$W\textsc{i} \rightarrow A\textsc{f} \ . \tag{10}$$

For requirements $R\textsc{i}$ we have the opposite. They must be realised by formal constructs $B\textsc{f}$:

$$B\textsc{f} \rightarrow R\textsc{i} \ . \tag{11}$$

We must not weaken the requirements. Whenever possible (10) and (11) should be equivalence traces, $W\textsc{i} \leftrightarrow A\textsc{f}$ and $B\textsc{f} \leftrightarrow R\textsc{i}$. That is, we do not want to implement more than necessary, either because we have weakened the assumptions or because we have strengthened the requirements. Note, however, that neither weaker assumptions nor stronger requirements will cause a correctness problem. This is a well-known fact in program verification [4].

3.3.2. Tracing Artefacts into Formal Models

The proof obligations for an Event-B machine M are produced by mildly rewriting and composing its constructs (and the constructs of seen contexts). The motivation for this approach is that it should be easy to relate constructs in a formal model with proof obligations [3]. In particular, if proving a proof obligation fails, it should be clear where in the formal model to look for an error. Exploiting the proximity between proof obligations and formal model, we can trace requirements into the proof obligations —as a side effect marking the corresponding constructs in the formal model. In practice, the tracing information is attached to the constructs of the formal model. However, the reference model described in Section 2.1 is formulated predicatively on the the level of the proof obligations.

We focus on specification adequacy (1) first: $\forall e, s \cdot W\textsc{i} \wedge Q\textsc{i} \Rightarrow R\textsc{i} \wedge U\textsc{i}$. It deals with domain properties $W\textsc{i}$, specification elements $Q\textsc{i}$, requirements $R\textsc{i}$ and design decisions $U\textsc{i}$. To show specification adequacy, the domain properties $W\textsc{i}$ and specification elements $Q\textsc{i}$ must realise the constructs $A\textsc{f}$ of the formal model,

$$W\textsc{i} \wedge Q\textsc{i} \rightarrow A\textsc{f} \ , \tag{12}$$

and the constructs $A\textsc{f}$ must realise the requirements $R\textsc{i}$ and design decisions $U\textsc{i}$,

$$A\textsc{f} \rightarrow R\textsc{i} \wedge U\textsc{i}\ . \tag{13}$$

Here we consider those artefacts that can be traced into the formal model. We do not assume, however, that satisfaction of *all* the requirements is proved formally. Some proof or argumentation may be informal involving only informal artefacts as illustrated by Example 5.

Based on (12) we can identify constructs of the formal model that are realised by $W\textsc{i}$ and $Q\textsc{i}$. Assumptions are modelled in Event-B by axioms, hence some *static* domain properties $W\textsc{i}' \subseteq W\textsc{i}$ realise the axioms: $W\textsc{i}' \rightarrow C\textsc{f}$.[1] *Dynamic* domain properties $W\textsc{i}'' \subseteq W\textsc{i}$ realise a subset of the events and invariants of a model: $W\textsc{i}'' \rightarrow \ddot{I}\textsc{f} \wedge \ddot{G}\textsc{f} \wedge \ddot{S}\textsc{f}$. Distinct events and invariants $\dot{I}\textsc{f} \wedge \dot{G}\textsc{f} \wedge \dot{S}\textsc{f}$ are realised by specification elements $Q\textsc{i}' \subseteq Q\textsc{i}$: $Q\textsc{i}' \rightarrow \dot{I}\textsc{f} \wedge \dot{G}\textsc{f} \wedge \dot{S}\textsc{f}$.

Note that there is a qualitative difference between domain properties $W\textsc{i}$ and specification elements $Q\textsc{i}$: the $W\textsc{i}$ are assumed to be given in advance whereas the $Q\textsc{i}$ are produced during the modelling.

Based on (13) we can identify constructs of the formal model that realise $R\textsc{i}$ and $U\textsc{i}$. Both of them are realised by events and invariants: $I\textsc{f} \wedge G\textsc{f} \wedge S\textsc{f} \rightarrow R\textsc{i} \wedge U\textsc{i}$. Note how artefacts that relate to invariants that make a claim of the form "it is always true that ... " correspond to the temporal statement "**always** I". This is not proved directly by the proof obligations for machines but implied by them. This is not formalised in Event-B. So we rely on an informal proof for the claim.

Example 9. In the abstract Event-B model of the lift system we introduce the movement of the lift cage. In order to formalise the current position of the lift cage with respect to requirement item R-1, we introduce a variable *floor* and the invariant *inv1*:

invariant *inv1*: $\mathit{floor} \in \mathit{ground\_floor} \mathrel{..} \mathit{top\_floor}$

We keep track of the relationship between R-1 and *inv1* using an equivalence trace.

↔

inv1 ↔ R-1

Once we have proved preservation of invariant *inv1* by all events, R-1 is formally validated. It remains formally validated in all further refinements. Equivalence is a strong form of justification. Requirement item R-1 justifies the presence of the invariant *inv1* in the formal model but *inv1* is stronger than necessary. ⌟

[1] Informal artefacts can realise more than one formal construct. To simplify the presentation we assume that such artefacts are duplicated.

Example 10. We formalise the domain property W-2 by introducing another variable *move* that describes the direction of the lift cage or whether the lift cage is idle.

invariant *inv2*: $move \in \{up, down, idle\}$

We have the following trace.

$inv2 \;\leftrightarrow\;$ W-2

Some invariants can be traced to requirement items others to domain properties. Tracing artefacts makes this explicit for each invariant.

3.3.3. Tracing Artefacts into Formal Refinements

If a model is developed by formal refinement, a sequence of machines $M\text{F}_1$, $M\text{F}_2, \ldots, M\text{F}_n$ is obtained where $M\text{F}_{i+1}$ refines $M\text{F}_i$. Each $M\text{F}_i$ captures some informal artefacts. For instance, we have a corresponding sequence of subsets of the requirements R that are realised in each model $R\text{I}_1, R\text{I}_2, \ldots, R\text{I}_n$, where $R\text{I}_i \subseteq R\text{I}_{i+1}$. We rephrase the refinement condition (3) of the reference model to deal with series of specifications: $\forall e, s \cdot W\text{I} \wedge Q\text{I}' \Rightarrow Q\text{I}$. This is matched by the refinement notion of Event-B. Formulae (5) and (6) imply:

$$C\text{F} \wedge D\text{F} \wedge I\text{F} \wedge J\text{F} \wedge Z\text{F} \wedge H\text{F} \wedge T\text{F} \Rightarrow G\text{F} \wedge S\text{F}\ . \tag{14}$$

So we can argue that if $G\text{F} \wedge S\text{F}$ realises some requirement, so does $H\text{F} \wedge T\text{F}$. However, $H\text{F} \wedge T\text{F}$ may be stronger. In particular, it may realise more requirements $R\text{I}'$ than $G\text{F} \wedge S\text{F}$ where $R\text{I}' \subseteq R\text{I}_{i+1} \setminus R\text{I}_i$ if $H\text{F} \wedge T\text{F}$ is an event from machine $M\text{F}_{i+1}$. Hence, $H\text{F} \wedge T\text{F}$ realises all requirements that $G\text{F} \wedge S\text{F}$ does plus the requirements $R\text{I}'$. We have already outlined at the end of Section 3.2.2 how invariants are accumulated along a series of formal refinements. A similar argument holds for design decisions $U\text{I}$. Thus formal refinement permits us to introduce and trace requirements gradually, alleviating a major difficulty when dealing with complex requirements.

Additional axioms $D\text{F}$ can be introduced in a refinement. So in the refinement the axioms realise additional domain properties. Concerning events that realise domain properties, proving only (14) is not sufficient because the concrete event may be stronger than the abstract event. But correctness is only maintained if this does not happen. There are several possibilities to deal with this situation. We could argue why the concrete event is still weak enough to realise the domain properties; or a formal proof technique could establish that both events are equivalent.[2] One could also start with the first approach and switching to second when all relevant domain properties have been realised. On

[2] In Event-B this can be achieved by declaring an event as *external*.

the other hand we are allowed to strengthen the formal constructs realising the specification elements $Q\mathrm{I}$. With it, however, we have to strengthen the specification elements of $Q\mathrm{I}$. As discussed above, $Q\mathrm{I}$ is being developed and not assumed to be exclusively under control of the environment. This is a fundamental difference to domain properties we have to observe. For increasing sets of domain properties $W\mathrm{I}_i$ and specification elements $Q\mathrm{I}_i$ where $i \in 1, \ldots, n$ as above, we have thus

$$W\mathrm{I}_{i+1} \wedge Q\mathrm{I}_{i+1} \Rightarrow W\mathrm{I}_i \wedge Q\mathrm{I}_i \ . \tag{15}$$

All in all, respecting the precaution about domain properties of the preceding paragraph, formal refinement preserves (12) and (13). Each refinement step can be used to verify adequacy of the specification gradually:

$$W\mathrm{I}_i \wedge Q\mathrm{I}_i \Rightarrow R\mathrm{I}_i \wedge U\mathrm{I}_i \ . \tag{16}$$

Refinement steps dealing with implementing elements $P\mathrm{I}$ will usually realise fewer additional requirements. The refinement method, however, does not make a particular distinction between the two uses of refinement. Refinement theory guarantees that adequacy validated in earlier refinement steps is preserved. When the end of the series refinements is reached specification adequacy (1) is fully verified.

Example 11. In the first refinement of the Event-B model of the lift system we introduce the door status. Furthermore, we formalise requirement item R-2. It is formally modelled by means of an invariant *inv3*:

```
invariant inv3:   $move \in \{up, down\} \Rightarrow door = closed$
```

We consider this to capture precisely the informal meaning of the requirement. Hence, we create an equivalence trace *inv3* ↔ R-1 from *inv3* to R-2. As a consequence of R-2 specification elements have to be added to enforce that no lift movements occur when the door is open. The formal Event-B model highlights the the needed specification elements by corresponding proof obligations. Often guards like *grd1* in event *switch_move_up* have to be added to events in order to prevent behaviour that would violate the invariants.

```
event switch_move_up
  when
    grd1:  $door = closed$
    grd2:  $move = idle$
  then
    act1:  $move := up$
  end
```

The guards have to be translated into a specification element Q-4, say, and corresponding realisation traces be provided: *grd1* → Q-4 and Q-4 → R-2. ⌟

3.3.4. Informal Proofs about Formal Models

Often requirements can be identified with invariants, event guards or actions. In this case (16) holds trivially for the concerned requirements. Sometimes theorems can be stated that that realise the requirements and are implied by the invariants [13]. However, our approach is not limited to verification by formal proof exclusively. We also permit (and encourage) informal proof. We have already seen the property "**always** I" that is only proved informally. Other temporal properties could be used similarly. However, for this article we content ourselves with a less expressive notation relying only on the core constructs of Event-B. Our aim here is not to formalise everything but to show how formal and informal reasoning can be used together for complex models.

3.3.5. Informal Proofs about Informal Models

We mentioned in Section 3.3.3 that with respect to the formal model "we consider those artefacts that can be traced into the formal model". Those artefacts that are not traced into the formal model can also not be verified formally. There are various ways to verify them informally. We should certainly not simply ignore them if they do not fit into the current formal scheme. To name a few forms of informal verification we mention the following. In Example 5 we have sketched in informal mathematical proof. The Problem Frames approach employs *frame concerns* to argue whether requirements are satisfied. When referring to temporal formulae we may rely on background theory [14] to infer the required temporal properties.

3.3.6. From Informal Premises to Informal Conclusions

At the end of Section 3.3.3 we have only discussed informal conclusions. The reason for this is that our objective is requirements validation and the premise $W_\mathrm{I} \wedge Q_\mathrm{I}$ and conclusion $R_\mathrm{I} \wedge U_\mathrm{I}$ of (1) are both informal. If this is so and we additionally allow for informal proof, what is the formal model with its formal proof worth? This is a question we have to ask with respect to any formal model not just our approach. All we have done is to make the tracing information to informal artefacts explicit. Any formal model that we create will be located between informal domain properties and informal requirements. It is tempting in such formal models to identify formal constructs with domain properties and requirements. In our experience (e.g. [10]) this is not always achievable. For instance, some of the verification is done by specialist engineers and their work is not to be replicated. We can formally prove important system properties. However, we will always begin with an informal premise and end with an informal conclusion. Our approach makes this explicit while giving a dominant role to formal modelling in the requirements process.

4. A Process for Requirements Modelling and Validation

In the requirements engineering process we distinguish the different activities of requirements elicitation, requirements specification, system modelling,

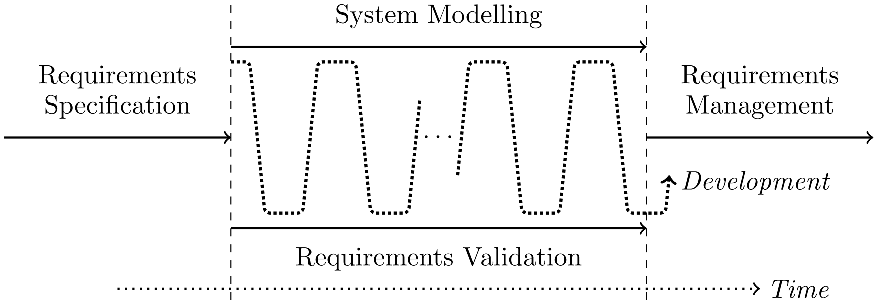


Figure 9: The Iterative Requirements Development Process

requirements validation and requirements management [40]. We focus on modelling and validation. Common approaches of requirements elicitation could be used to gather requirements in early phases during the process. We do not consider this aspect of the requirements process because it has little influence on modelling and validation. Figure 9 shows an overview of the requirements process using iterations of modelling and validation.

Specification. During the *requirements specification* phase requirements and domain properties are first identified. The resulting classification into corresponding artefacts and phenomena (following the reference model of Section 4.1) is the starting point for modelling and validation.

Modelling and Validation. The objective of *system modelling* is the formal modelling of a subset of the system description as well as the elaboration of the specification elements. Artefacts can be incorporated gradually into the formal model using refinement as described in Section 3.

The objective of *requirements validation* is to validate the relationship between informal artefacts and formal constructs and to validate the adequacy of the specification elements. The validation relies on the our model for tracing artefacts and phenomena and its use for tracing artefacts into formal model and refinements.

Usually there will be many iterations of modelling and validation. This has already been observed when using Event-B to produce formal models where dealing with requirements often plays a subordinate role [13]. Note that we do not aim at producing a formal model as opposed to [13] for instance. The aim of the requirements process is to produce a consistent and complete system description. The formal model that is produced on the way is only a tool towards that end. For this reason, constructs of formal models are hardly mentioned in the process description for modelling and validation in Section 4.1 below. Formal models are particularly useful for analysing consistency. Completeness of requirements is more a concern of informal approaches to requirements modelling such as Problem Frames.

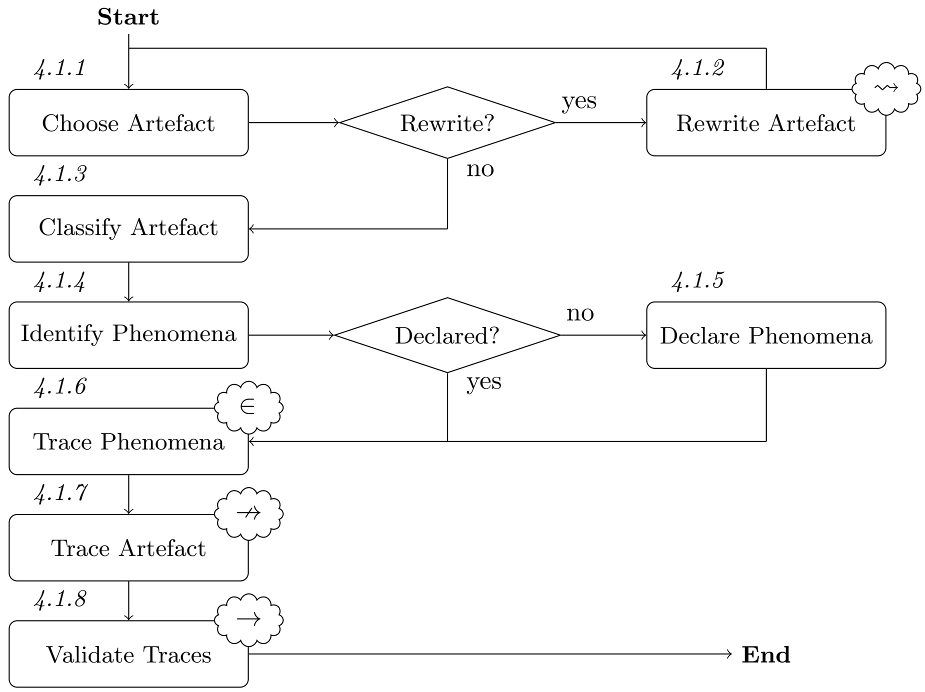


Figure 10: The Requirements Modelling and Validation Process

Management. We see that *requirements management* as a continuation of modelling and validation in a later phase of a project lifecycle. The underlying assumption is that the development of a system description fully finished. The ongoing work includes change management and requirement evolution.

4.1. Incremental Modelling and Validation

The phase of modelling and validation consists usually of many iterations between modelling and validation where the collection of all artefacts is validated incrementally. Figure 10 shows the steps of the modelling and validation process involved for each artefact (based on [26]). The small clouds attached to some of the steps indicate the kind of traces that are modified in the those steps. The modelling and validation process typically starts with a collection of informal artefacts that have been produced during the requirements specification phase. In the following Sections 4.1.1 to 4.1.8 we describe the steps in the process in more detail. The section numbers are also indicated near the corresponding steps in Figure 10.

4.1.1. Choose Artefact

The first step in the process consists in choosing an arbitrary artefact. We do not distinguish "creating a new artefact" but include this for simplicity under

Q-2.1	If the [*lift cage*] is [*idle*] at some [*floor*] below the [*top floor*], it may proceed by [*moving up*]
Q-2.2	If the [*lift cage*] is [*idle*] at the [*top floor*], it does not proceed by [*moving up*]

Figure 11: Specification element Q-2 split into Q-2.1 and Q-2.2

heading "choosing an artefact". Depending on the quality of the artefact, it may have to evolve by being rewritten or split, typically by checking it against quality criteria [16]. Often this rewriting an artefact will also be a consequence of insights gained during formal modelling. New artefacts created during the process will, in particular, be specification elements and design decisions. But also missing domain properties are often found by formal modelling and need then to be added to the system description.

4.1.2. Rewrite Artefact

Changing the artefact results in one or more evolution traces and the process has to start over with an evolved artefact. (The evolved artefacts must be checked with the stakeholders to ensure that they still reflect their needs.)

Example 12. Specification element Q-2 is too complex and should be split into two separate specification elements. The new specification elements Q-2.1 and Q-2.2 to replace Q-2 are shown in Fig. 11.

The evolution trace for Q-2 keeps track of this modification.

Q-2 $\rightsquigarrow$ Q-2.1, Q-2.2

The specification elements Q-2.1 and Q-2.2 are easier to trace into the formal Event-B model.

```
event switch_move_up
  when
    grd1:  door = closed
    grd2:  move = idle
    grd3:  floor < top_floor
  then
    act1:  move := up
  end
```

Specification element Q-2.1 justifies two guards and the action of the event.

grd2,*grd3*,*act1* → Q-2.1

There are more traces to event *switch_move_up* that justify each of the formal constructs like *grd1* → Q-4 discussed in Example 11.

4.1.3. Classify Artefact

The artefact is identified as belonging to one of the classes R, N, W, Q or U. This in turn will determine the kinds of phenomena that are allowed to be used by the artefact: e_h, e_v, s_v or s_h. Here the boundary between the environment and the system as discussed in Section 2 is crucial. The distinction of the artefacts requires the boundary to be fixed.

4.1.4. Identify Phenomena

All phenomena used by the artefact are identified. All phenomena need to be declared in order to achieve consistent usage across the collection of artefacts. The phenomena that that are allowed to occur in the artefact are determined by the class of the artefact. Phenomena that have not yet been declared must be declared before we can continue by tracing the phenomena.

4.1.5. Declare Phenomena

Missing phenomena are declared by introducing a *designation*. This can be as simple as adding the designation to a glossary and classifying it as belonging to one of e_h, e_v or s_v. It may be described further by other artefacts. For formalised phenomena, for instance, a specific artefact with typing information may be necessary.

4.1.6. Trace Phenomena

The association "$\in$" between artefact and the used phenomena is determined. In the ProR integration, this is done by simply surrounding the designation by squared brackets in the informal text that represents the artefact. A phenomenon is considered declared, once a formal element with the same designation has been created. The tool verifies this and provides feedback in the form of colour highlighting. Tracing between phenomena and formal artefacts is done implicitly by building a syntax tree of the formal model.

4.1.7. Trace Artefact

Realisation "$\rightarrow$" or equivalence "$\leftrightarrow$" traces are attached to the artefact. These as well as traces that were attached to the artefact before must be validated afterwards. Until then they are marked as *suspect*, denoted "$\nrightarrow$" or "$\nleftrightarrow$". The traces are constrained by the formulae (1) to (4) (this is currently not verified by the tool). As artefacts and phenomena change, existing traces that are automatically marked as suspect by the tool and must be validated anew.

4.1.8. Validate Traces

All traces of the artefact must be validated eventually. This requires reviewing the artefacts and related formal constructs in order to judge whether the formal construct properly stand in the claimed relationship "$\rightarrow$" or "$\leftrightarrow$". In line with the incremental approach to modelling and validation it is also possible for traces of an artefact to remain suspect, e.g., "$\nrightarrow$" or "$\nleftrightarrow$". For a system description to be consistent, though, all traces must have been validated.

Example 13. If a new trace is added or the source or target of a trace changes the trace is marked as suspect. The ProR tool does this automatically. For instance, if we change *inv3* or R-2, the trace needs to be validated

inv3 $\nleftrightarrow$ R-2 $\nleftrightarrow$

After having reviewed *inv3* and R-2 the trace can be validated and the marking removed. For this task it is important to have not (many) more traces than necessary. The effort of reviewing should be kept as small as possible.

4.2. The Fully Validated System Description

When all traces have been validated and claims in the formal model have been verified the system description is considered consistent with respect to

- the allowed references of phenomena by the different artefacts (Figure 3);
- the use of phenomena across different artefacts;
- the traces respecting formulae (1) to (4)
- the relationship of informal artefacts to formal constructs;
- the verified formal properties of the formal model.

Our approach only classifies artefacts and their relationships. There is no provision for structuring the collection of artefacts as a whole. We acknowledge that the structuring is a relevant issue in practice and rely on approaches complementary to ours to carry this out. For instance, [29] argues that a list of requirements is much easier to understand if they are given a meaningful order. Furthermore, additional structure such as sections or headlines improves readability and scalability of a system description.

A lot of practical advice with respect to structuring is available, e.g., [16, 36, 38]. Some of this advice is manifested in the form of process templates, e.g., [30], or in the form of standards such as IEEE standard 830 [17]. The IEEE standard 830 describes a document-centred approach. It provides standardised document outlines for different types of system descriptions combined with some quality criteria and checklists for completeness. The standard hardly constrains the actual contents. This, however, makes it easy to combine it with our approach, which is primarily concerned with the contents in the form artefacts and their relationships. Finally, we note that the Problem Frames notation [19], which is employed in Example 1, provides some structuring based on a graphical notation (see Figure 2 below).

5. Related Work

Central to this paper is the WRSPM reference model [12]. Earlier work by the authors on expanding the reference model is published in [26] and [25].

The issue of traceability has been analysed in depth by Gotel et. al. [11], according to which our research falls into the area of post-requirements specification traceability. Traceability has been recognised as a difficult problem, e.g. [6].

In this article we use the Event-B formalism [2]. While not directly concerned with requirements traceability, [2] recognises the problem of the transition from informal user requirements to a formal specification. However, there is no mention of how this could be achieved.

There have been successful attempts in applying Problem Frames and Event-B together. In [34], the authors show how these are being applied to an industrial case study. In contrast to our approach, only requirements that were actually modelled formally were included in the specification in the first place.

There are other approaches for requirements traceability between formal and informal artefacts. KAOS [9] is a well-known approach. Rather than allowing informal elements that are omitted from the formal model, it provides so-called "soft-goals" that are broken down into requirements that can still be modelled formally. KAOS does not demand the whole model to be formalised, but does not clearly state the implications of a partial formalisation.

Reveal [37] is an approach that is driven by an industrial company. It is based on Michael Jackson's "World and the Machine" model [18]. There are a lot of similarities to our approach, including the acknowledgement of requirements that are not part of the formal model. Reveal does not define a traceability approach, however: It merely demands that one is defined and followed.

The case study used in this article is described in [5], albeit using a different formalism. In that article a real-time specification method is used and it is assumed that all requirements can be formalised as formulae in that formalism.

This paper is also concerned with tool support. The ProR tool [20] has been used for this purpose, and other integrations for traceability have been realised with it [23] or proposed [22].

6. Conclusion

We have presented an incremental approach for building a system description consisting of formal and informal artefacts. The resulting system description is complemented by a traces between those artefacts that support systematic validation and change management. We have illustrated the approach by means of a small case study.

The main objective of our approach is the validation of the informal system description. Consequently, the formal model is no end in itself, but only serves as a tool for the rigorous validation of the system description or parts thereof. Specifically, any formal model that we create is located between informal domain properties and informal requirements. The aim of the formal modelling is to ensure consistency of the informal system description. The approach of requirements tracing that we have developed plays a crucial role in this.

The role of the formal model in the overall system development process can vary and depends on a number of factors, including the problem to be solved, the formalism that is chosen, the experience of the team, to name a few. In the case study, a state-based Event-B model was build, which resulted in a subset of the artefacts to be formalised. For instance, out of the 8 informal artefacts shown in Table 5, only 5 were formalised.

The formal model was then used to show that the formal counterparts of the requirements are satisfied, by proving that corresponding invariants are preserved.

This work has a strong focus on traceability, and we do not distinguish in principle on whether the artefacts traced are formal or informal. Instead, we focus on classifying the trace as a justification (or its inverse, realisation) and its stronger form, equivalence. This allows us to construct a closed system description that is consistent with respect to the purpose of its artefacts. While such an approach cannot identify missing requirements or assumptions, it can ensure that all recorded requirements are realised. Further, the introduction of phenomena allows the systematic creation and maintenance of a glossary, and allows for some rudimentary consistence checks as well.

Finally, the work described here is supported by a tool integrating requirements modelling, ProR, that is integrated with the tools for formal verification by proof, Rodin, and by model-checking an animation, ProB.

Future Work. The approach introduced here provides little guidance with respect to the macroscopic structure of the system description. We used Problem Frames in the case study to provide some structure, and it would be interesting to develop this further.

The tool chain was helpful and automated a number of tasks required by our approach. But there is more potential for automation, especially in the area of issue reporting. We plan on creating a catalogue of properties that a consistent system description should have, and a reporting tool that lists all violations of those properties. This is already done, to a degree, through the colour highlighting, but could be taken much further.

Acknowledgements. The work in this paper is partly funded by DEPLOY, a European Commission Information and Communication Technologies FP7 project.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modeling in Event-B – System and Software Engineering.* Cambridge University Press, 2010.

[3] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 3rd edition, 2009.

[5] D. Bjørner. Trusted computing systems: the ProCoS experience. In *ICSE*, pages 15–34. ACM, 1992.

[6] D. Bjørner. From domain to requirements. In *Concurrency, Graphs and Models: Essays dedicated to Ugo Montanari on the Occasion of his 65th Birthday*, pages 278–300. Springer, 2008.

[7] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[8] L. Chung and J. C. Sampaio do Prado Leite. On non-functional requirements in software engineering. In A. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. K. Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *LNCS*, pages 363–379. Springer, 2009.

[9] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE*, pages 612–613. ACM, 1997.

[10] R. Gmehlich, K. Grau, S. Hallerstede, M. Leuschel, F. Lösch, and D. Plagge. On fitting a formal method into practice. In S. Qin and Z. Qiu, editors, *ICFEM*, volume 6991 of *LNCS*, pages 195–210. Springer, 2011.

[11] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *ICRE*, pages 94–101. IEEE Computer Society, 1994.

[12] C. A. Gunter, M. Jackson, E. L. Gunter, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17:37–43, 2000.

[13] S. Hallerstede. Incremental system modelling in Event-B. In F. S. de Boer, M. M. Bonsangue, and E. Madelain, editors, *FMCO*, volume 5751 of *LNCS*, pages 139–158. Springer, 2008.

[14] T. S. Hoang and J.-R. Abrial. Reasoning about liveness properties in Event-B. In S. Qin and Z. Qiu, editors, *ICFEM*, volume 6991 of *LNCS*, pages 456–471. Springer, 2011.

[15] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[16] C. Hood and R. Wiebel. *Optimieren von Requirements Management & Engineering mit dem HOOD Capability Model*. Springer, 2005.

[17] IEEE. Recommended practice for software requirements specifications. Technical Report IEEE Std 830-1998, IEEE, 1997.

[18] M. Jackson. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.

[19] M. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley/ACM Press, 2001.

[20] M. Jastram. ProR, an open source platform for requirements engineering based on RIF. *SEISCONF*, 2010.

[21] M. Jastram. *The ProR Approach: Traceability of System Descriptions*. PhD thesis, Heinrich-Heine University of Düsseldorf, 2012. To be submitted.

[22] M. Jastram and A. Graf. Requirement traceability in Topcased with the requirements interchange format (RIF/ReqIF). *First Topcased Days Toulouse*, 2011.

[23] M. Jastram and A. Graf. Requirements, traceability and DSLs in Eclipse with the requirements interchange format (RIF/ReqIF). Technical report, Dagstuhl-Workshop MBEES, 2011.

[24] M. Jastram and A. Graf. ReqIF – the new requirements standard and its open source implementation Eclipse RMF. Technical report, Commercial Vehicle Technology Symposium, 2012.

[25] M. Jastram, S. Hallerstede, and L. Ladenberger. Mixing formal and informal model elements for tracing requirements. *ECEASST*, 46, 2011.

[26] M. Jastram, S. Hallerstede, M. Leuschel, and A. G. Russo. An approach of requirements tracing in formal refinement. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *VSTTE*, volume 6217 of *LNCS*, pages 97–111. Springer, 2010.

[27] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.

[28] E. Kang and D. Jackson. Dependability arguments with trusted bases. In *RE*, pages 262–271. IEEE Computer Society, 2010.

[29] B. L. Kovitz. *Practical software requirements: a manual of content and style*. Manning, 1998.

[30] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 2004.

[31] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[32] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[33] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Asp. Comput*, 23(6):683–709, 2011.

[34] F. Loesch, R. Gmehlich, K. Grau, C. Jones, and M. Mazzara. Report on pilot deployment in the automotive sector. Deliverable D7, 2010.

[35] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[36] K. Pohl. *Requirements Engineering. Grundlagen, Prinzipien, Techniken.* Dpunkt.Verlag, 1. edition, 2007.

[37] Praxis. Reveal – a keystone of modern systems engineering. Technical report, Praxis Critical Systems Limited, 2003.

[38] C. Rupp. *Requirements-Engineering und -Management: professionelle, iterative Anforderungsanalyse für die Praxis.* Hanser, 4. edition, 2007.

[39] N. Tennant. Relevance in reasoning. In Stewart Shapiro, editor, *The Oxford Handbook of Philosophy of Mathematics and Logic*, chapter 23, pages 696–726. Oxford University Press, 2005.

[40] K. E. Wiegers. *Software Requirements: Practical Techniques for Gathering and Managing Requirements throughout the Product Development Cycle.* Microsoft Press, 2nd edition, 2003.

[41] J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof.* Prentice-Hall, 1996.