Softwaretechnik / Software-Engineering

Lecture 6: Formal Methods for Requirements Engineering

2019-05-16

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Requirements Engineering: Content

VL 5	• **Introduction**
	• **Requirements Specification**
	— • Desired Properties
	— • Kinds of Requirements
	— • Analysis Techniques
⋮	• **Documents**
	— • Dictionary, Specification
	• **Specification Languages**
	— • Natural Language
VL 6	— • Decision Tables
	— — • Syntax, Semantics
⋮	— — • Completeness, Consistency, …
VL 7	— • Scenarios
⋮	— — • User Stories, Use Cases
VL 8	— — • Live Sequence Charts
	— — — • Syntax, Semantics
⋮	• Definition: **Software** & **SW Specification**
VL 9	• **Wrap-Up**
⋮	

Vocabulary

Techniques

informal

↕

semi-formal

↕

formal

Content

- **Documents**
  - Dictionary, Specification
- **Requirements Specification Languages**
  - Natural Language
- **(Basic) Decision Tables**
  - Syntax, Semantics
- **…for Requirements Specification**
- **…for Requirements Analysis**
  - Completeness, Useless Rules,
  - Determinism
- **Domain Modelling**
  - Conflict Axiom,
  - Relative Completeness, Vacuous Rules,
  - Conflict Relation
- **Collecting Semantics**
- **Discussion**

Logic

Requirements Documents

Requirements Specification

specification — A document that specifies,

- in a complete, precise, verifiable manner,

the

- requirements, design, behavior,
  or other characteristics of a system or component,

and, often, the procedures for determining whether these provisions have been satisfied. **IEEE 610.12 (1990)**

software requirements specification (SRS) — Documentation of the essential requirements (functions, performance, design constraints, and attributes) of the software and its external interfaces. **IEEE 610.12 (1990)**

IEEE Std 830-1998
(Revision of
IEEE Std 830-1993)

IEEE Recommended Practice for Software Requirements Specifications

Sponsor

Software Engineering Standards Committee
of the
IEEE Computer Society

Approved 25 June 1998

IEEE-SA Standards Board

Abstract: The content and qualities of a good software requirements specification (SRS) are described and several sample SRS outlines are presented. This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. Guidelines for compliance with IEEE/EIA 12207.1-1997 are also provided.
Keywords: contract, customer, prototyping, software requirements specification, supplier, system requirements specifications

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 0-7381-0332-2

Structure of a Requirements Document: Example

!?

Dictionary

(Ludewig and Lichter, 2013) based on (IEEE, 1998)

Dictionary

- Requirements analysis should be based on a **dictionary**.
- A **dictionary** comprises definitions and clarifications of **terms** that are relevant to the project and of which different people (in particular customer and developer) may have different understandings before agreeing on the dictionary.
- Each **entry** in the **dictionary** should provide the following information:
  - **term** and **synonyms** (in the sense of the requirements specification),
  - **meaning** (definition, explanation),
  - **deliminations** (where **not** to use this terms),
  - **validness** (in time, in space, …),
  - **denotation**, unique identifiers, …,
  - **open questions** not yet resolved,
  - **related terms**, cross references.

  Note: entries for terms that **seemed** "crystal clear" at first sight are **not uncommon**.
- All work on requirements should, as far as possible, be done **using terms from the dictionary** consistently and consequently.

  The dictionary should in particular be **negotiated with the customer** and used in communication (if not possible, at least developers should stick to dictionary terms).
- **Note**: do not mix up **real-world/domain** terms with ones only "living" in the software.

Example: Wireless Fire Alarm System

The loss of the ability of the system to transmit a signal from a component to the central unit is

- detected in less than 300 seconds and
- displayed at the central unit within 100 seconds thereafter.

?

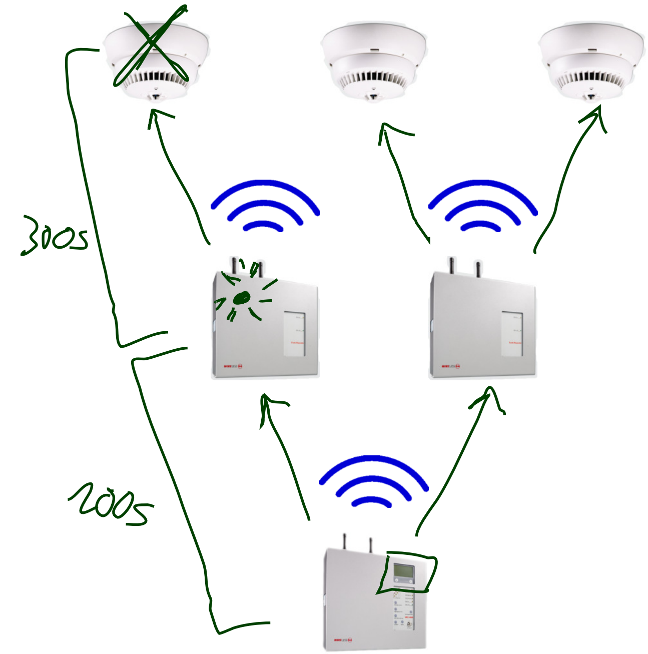
<

<

Dictionary Example

Example: Wireless Fire Alarm System

- During a project on designing a highly reliable, EN-54-25 conforming wireless communication protocol, we had to learn that the relevant components of a fire alarm system are
  - terminal participants (heat/smoke sensors and manual indicators),
  - repeaters (a non-terminal participant),
  - and a central unit (not a participant).
- Repeaters and central unit are technically very similar, but need to be distinguished to understand requirements. The dictionary explains these terms.



(Arenis et al., 2014)

Excerpt from the dictionary (ca. 50 entries in total):

Part A part of a fire alarm system is either a participant or a central unit.

Repeater A repeater is a participant which accepts messages for the central unit from other participants, or messages from the central unit to other participants.

Central Unit A central unit is a part which receives messages from different assigned participants, assesses the messages, and reacts, e.g. by forwarding to persons or optical/acustic signalling devices.

Terminal Participant A terminal participant is a participant which is not a repeater. Each terminal participant consists of exactly one wireless communication module and devices which provide sensor and/or signalling functionality.

Content

- **Documents**
  - Dictionary, Specification
- **Requirements Specification Languages**
  - Natural Language
- **(Basic) Decision Tables**
  - Syntax, Semantics
- **…for Requirements Specification**
- **…for Requirements Analysis**
  - Completeness, Useless Rules,
  - Determinism
- **Domain Modelling**
  - Conflict Axiom,
  - Relative Completeness, Vacuous Rules,
  - Conflict Relation
- **Collecting Semantics**
- **Discussion**

Logic

Requirements Specification Languages

Requirements Specification Language

specification language — A language, often a machine-processible combination of natural and formal language, used to express the requirements, design, behavior, or other characteristics of a system or component.

For example, a design language or requirements specification language. Contrast with: programming language; query language. **IEEE 610.12 (1990)**

requirements specification language — A specification language with special constructs and, sometimes, verification protocols, used to develop, analyze, and document hardware or software requirements. **IEEE 610.12 (1990)**

Natural Language Specification (Ludewig and Lichter, 2013) based on (Rupp and die SOPHISTen, 2009)

	rule	explanation, example
R1	State each requirement in **active voice**.	Name the actors, indicate whether the user or the system does something. Not “the item is deleted”.
R2	Express processes by **full verbs**.	Not “is”, “has”, but “reads”, “creates”; full verbs require information which describe the process more precisely. Not “when data is consistent” but “after program P has checked consistency of the data”.
R3	Discover **incompletely defined verbs**.	In “the component raises an error”, ask whom the message is addressed to.
R4	Discover **incomplete conditions**.	Conditions of the form “if-else” need descriptions of the if- and the then-case.
R5	Discover **universal quantifiers**. $\forall, \exists$	Are sentences with “never”, “always”, “each”, “any”, “all” really universally valid? Are “all” really all or are there exceptions.
R6	Check **nominalisations**.	Nouns like “registration” often hide complex processes that need more detailed descriptions; the verb “register” raises appropriate questions: who, where, for what?
R7	Recognise and refine **unclear substantives**.	Is the substantive used as a generic term or does it denote something specific? Is “user” generic or is a member of a specific classes meant?
R8	Clarify **responsibilities**.	If the specification says that something is “possible”, “impossible”, or “may”, “should”, “must” happen, clarify who is enforcing or prohibiting the behaviour.
R9	Identify **implicit assumptions**.	Terms (“the firewall”) that are not explained further often hint to implicit assumptions (here: there seems to be a firewall).

Natural Language Patterns

Natural language requirements can be (tried to be) written as an instance of the **pattern** "$\langle A \rangle \ \langle B \rangle \ \langle C \rangle \ \langle D \rangle \ \langle E \rangle \ \langle F \rangle$." (German grammar) where

A	clarifies when and under what conditions the activity takes place
B	is MUST (obligation), SHOULD (wish), or WILL (intention); also: MUST NOT (forbidden)
C	is either "the system" or the concrete name of a (sub-)system
D	one of three possibilities: • "does", description of a system activity, • "offers", description of a function offered by the system to somebody, • "is able if", usage of a function offered by a third party, under certain conditions
E	extensions, in particular an object
F	the actual process word (what happens)

(Rupp and die SOPHISTen, 2009)

Example:

After office hours ($= A$), the system ($= C$) should ($= B$) offer to the operator ($= D$) a backup ($= F$) of all new registrations to an external medium ($= E$).

Other Pattern Example: RFC 2119

Network Working Group S. Bradner
Request for Comments: 2119 Harvard University
BCP: 14 March 1997
Category: Best Current Practice

Key words for use in RFCs to Indicate Requirement Levels

Status of this Memo

This document specifies an Internet Best Current Practices for the Internet Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Abstract

In many standards track documents several words are used to signify the requirements in the specification. These words are often capitalized. This document defines these words as they should be interpreted in IETF documents. Authors who follow these guidelines should incorporate this phrase near the beginning of their document:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Note that the force of these words is modified by the requirement level of the document in which they are used.

1. MUST This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

2. MUST NOT This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

3. SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. SHOULD NOT This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

Bradner Best Current Practice [Page 1]

RFC 2119 RFC Key Words

5. MAY This word, or the adjective "OPTIONAL", mean that a
truly optional. One vendor may choose to include the ite
particular marketplace requires it or because the vendor
it enhances the product while another vendor may omit the
An implementation which does not include a particular opt
prepared to interoperate with another implementation whic
include the option, though perhaps with reduced functiona
same vein an implementation which does include a particul
MUST be prepared to interoperate with another implementat
does not include the option (except, of course, for the f
option provides.)

6. Guidance in the use of these Imperatives

Imperatives of the type defined in this memo must be used
and sparingly. In particular, they MUST only be used whe
actually required for interoperation or to limit behavior
potential for causing harm (e.g., limiting retransmissio
example, they must not be used to try to impose a particu
on implementors where the method is not required for
interoperability.

7. Security Considerations

These terms are frequently used to specify behavior with
implications. The effects on security of not implementin
SHOULD, or doing something the specification says MUST NO
NOT be done may be very subtle. Document authors should t
to elaborate the security implications of not following
recommendations or requirements as most implementors will
had the benefit of the experience and discussion that pro
specification.

8. Acknowledgments

The definitions of these terms are an amalgam of definiti
from a number of RFCs. In addition, suggestions have bee
incorporated from a number of people including Robert Ull
Narten, Neal McBurnett, and Robert Elz.

Content

- **Documents**
  - Dictionary, Specification
- **Requirements Specification Languages**
  - Natural Language
- **(Basic) Decision Tables**
  - Syntax, Semantics
- **...for Requirements Specification**
- **...for Requirements Analysis**
  - Completeness, Useless Rules,
  - Determinism
- **Domain Modelling**
  - Conflict Axiom,
  - Relative Completeness, Vacuous Rules,
  - Conflict Relation
- **Collecting Semantics**
- **Discussion**

Logic

Definition. [*Bjørner and Havelund (2014)*]
A method is called **formal method**
if and only if its techniques and tools can be explained in mathematics.

Example:
If a method includes a specification language (as a tool), then that language has

- a **formal syntax**,
- a **formal semantics**, and
- a **formal proof system**.

Formal, Rigorous, or Systematic Development

"The **techniques** of a formal method help

- **construct** a specification, and/or
- **analyse** a specification, and/or
- **transform** (**refine**) one (or more) specification(s) into a **program**.

The **techniques** of a formal method, (besides the specification languages) are typically software packages that help developers use the techniques and other tools.

The aim of developing software, either

- **formally** (all arguments are formal) or
- **rigorously** (some arguments are made and they are formal) or
- **systematically** (some arguments are made on a form that can be made formal)

is to (be able to) **reason in a precise manner about properties** of what is being developed." (Bjørner and Havelund, 2014)

Decision Tables

Decision Tables: Example

opt. description

name

rule

rule name

premise

T		r_1	r_2	r_3
c_1		$\times$	$\times$	$-$
c_2		$\times$	$-$	$*$
c_3		$-$	$\times$	$*$
a_1		$\times$	$-$	$-$
a_2		$-$	$\times$	$-$

conditions

actions

don't care

effect

~~c_1~~	~~$\times$~~	~~$*$~~
~~a_1~~	~~$-$~~	

Decision Table Syntax

- Let C be a set of **conditions** and A be a set of **actions** s.t. $C \cap A = \emptyset$.
- A **decision table** T **over** C **and** A is a labelled $(m+k) \times n$ matrix

T: decision table		r_1	$\cdots$	r_n
c_1	description of condition c_1	$v_{1,1}$	$\cdots$	$v_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_m	description of condition c_m	$v_{m,1}$	$\cdots$	$v_{m,n}$
a_1	description of action a_1	$w_{1,1}$	$\cdots$	$w_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_k	description of action a_k	$w_{k,1}$	$\cdots$	$w_{k,n}$

- where
  - $c_1, \dots, c_m \in C$,
  - $a_1, \dots, a_k \in A$,
  - $v_{1,1}, \dots, v_{m,n} \in \{-, \times, *\}$ and
  - $w_{1,1}, \dots, w_{k,n} \in \{-, \times\}$.

- Columns $(v_{1,i}, \dots, v_{m,i}, w_{1,i}, \dots, w_{k,i})$, $1 \le i \le n$, are called **rules**,
- $r_1, \dots, r_n$ are **rule names**.
- $(v_{1,i}, \dots, v_{m,i})$ is called **premise** of rule r_i,

  $(w_{1,i}, \dots, w_{k,i})$ is called **effect** of r_i.

Decision Table Semantics

Each rule $r \in \{r_1, \dots, r_n\}$ of table T

T: decision table		r_1	$\cdots$	r_n
c_1	description of condition c_1	$v_{1,1}$	$\cdots$	$v_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_m	description of condition c_m	$v_{m,1}$	$\cdots$	$v_{m,n}$
a_1	description of action a_1	$w_{1,1}$	$\cdots$	$w_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_k	description of action a_k	$w_{k,1}$	$\cdots$	$w_{k,n}$

is assigned to a **propositional logical formula** $\mathcal{F}(r)$ over signature $C \,\dot\cup\, A$ as follows:

- Let $(v_1, \dots, v_m)$ and $(w_1, \dots, w_k)$ be premise and effect of r.
- Then

$$\mathcal{F}(r) := \underbrace{F(v_1, c_1) \wedge \dots \wedge F(v_m, c_m)}_{=: \mathcal{F}_{pre}(r)} \wedge \underbrace{F(w_1, a_1) \wedge \dots \wedge F(w_k, a_k)}_{=: \mathcal{F}_{\mathit{eff}}(r)}$$

where

$$F(v, x) = \begin{cases} x & \text{, if } v = \times \\ \neg x & \text{, if } v = - \\ \textit{true} & \text{, if } v = * \end{cases}$$

Decision Table Semantics: Example

$$\mathcal{F}(r) := F(v_1, c_1) \wedge \cdots \wedge F(v_m, c_m) \\ \wedge F(v_1, a_1) \wedge \cdots \wedge F(v_k, a_k)$$

$$F(v, x) = \begin{cases} x & \text{, if } v = \times \\ \neg x & \text{, if } v = - \\ \textit{true} & \text{, if } v = * \end{cases}$$

T		r_1	r_2	r_3
c_1		$\times$	$\times$	$-$
c_2		$\times$	$-$	$*$
c_3		$-$	$\times$	$*$
a_1		$\times$	$-$	$-$
a_2		$-$	$\times$	$-$

- $\mathcal{F}(r_1) = F(\times, c_1) \wedge F(\times, c_2) \wedge F(-, c_3) \wedge F(\times, a_1) \wedge F(-, a_2)$

  $= \underline{c_1 \wedge c_2 \wedge \neg c_3 \wedge a_1 \wedge \neg a_2}$
- $\mathcal{F}(r_2) = c_1 \wedge \neg c_2 \wedge c_3 \wedge \neg a_1 \wedge a_2$
- $\mathcal{F}(r_3) = \neg c_1 \wedge \textit{true} \wedge \textit{true} \wedge \neg a_1 \wedge \neg a_2$

Decision Tables as Requirements Specification

Yes, And?

We can use decision tables to **model** (describe or prescribe) the behaviour of **software**!

Example:
Ventilation system of
lecture hall 101-0-026.

observables

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	$-$
off	ventilation off?	$\times$	$-$	$*$
on	ventilation on?	$-$	$\times$	$*$
go	start ventilation	$\times$	$-$	$-$
stop	stop ventilation	$-$	$\times$	$-$

- We can **observe** whether **button is pressed** and whether room ventilation is **on or off**, and whether (we intend to) **start ventilation** of **stop ventilation**.

  {0,1}, {true, false}

- We can model our observation by a boolean valuation $\sigma : C \cup A \to \mathbb{B}$, e.g., set

$$\sigma(b) := \textit{true}\text{, if button pressed now and } \sigma(b) := \textit{false}\text{, if button not pressed now.}$$

$$\sigma(go) := \textit{true}\text{, we plan to start ventilation and } \sigma(go) := \textit{false}\text{, we plan to stop ventilation.}$$

- A valuation $\sigma : C \cup A \to \mathbb{B}$ can be used to assign a **truth value** to a propositional formula φ over $C \cup A$.
  As usual, we write $\sigma \models \varphi$ iff φ evaluates to *true* under σ (and $\sigma \not\models \varphi$ otherwise).
- Rule formulae $\mathcal{F}(r)$ are propositional formulae over $C \cup A$
  thus, given σ, we have either $\sigma \models \mathcal{F}(r)$ or $\sigma \not\models \mathcal{F}(r)$.

$$\sigma \left[\begin{array}{l} b \mapsto 0 \\ \mathit{off} \mapsto 0 \\ \mathit{on} \mapsto 1 \\ \mathit{start} \mapsto 0 \\ \mathit{stop} \mapsto 0 \end{array} \right. \models \mathcal{F}(r_3)$$

Yes, And?

We can use decision tables to **model** (describe or prescribe) the behaviour of **software**!

Example:
Ventilation system of
lecture hall 101-0-026.

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	$-$
off	ventilation off?	$\times$	$-$	$*$
on	ventilation on?	$-$	$\times$	$*$
go	start ventilation	$\times$	$-$	$-$
stop	stop ventilation	$-$	$\times$	$-$

- We can **observe** whether **button is pressed** and whether room ventilation is **on or off**, and whether (we intend to) **start ventilation** of **stop ventilation**.
- We can model our observation by a boolean valuation $\sigma : C \cup A \to \mathbb{B}$, e.g., set

  $\sigma(b) :=$ *true*, if button pressed now and $\sigma(b) :=$ *false*, if button not pressed now.

  $\sigma(go) :=$ *true*, we plan to start ventilation and $\sigma(go) :=$ *false*, we plan to stop ventilation.

- A valuation $\sigma : C \cup A \to \mathbb{B}$ can be used to assign a **truth value** to a propositional formula φ over $C \cup A$.
  As usual, we write $\sigma \models \varphi$ iff φ evaluates to *true* under σ (and $\sigma \not\models \varphi$ otherwise).
- Rule formulae $\mathcal{F}(r)$ are propositional formulae over $C \cup A$
  thus, given σ, we have either $\sigma \models \mathcal{F}(r)$ or $\sigma \not\models \mathcal{F}(r)$.
- Let σ be a model of an **observation** of C and A.
  We say, σ is **allowed** by **decision table** T if and only if there **exists** a rule r in T such that $\sigma \models \mathcal{F}(r)$.

Example

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	$-$
off	ventilation off?	$\times$	$-$	$*$
on	ventilation on?	$-$	$\times$	$*$
go	start ventilation	$\times$	$-$	$-$
$stop$	stop ventilation	$-$	$\times$	$-$

$$\mathcal{F}(r_1) = b \wedge \mathit{off} \wedge \neg on \wedge go \wedge \neg stop$$
$$\mathcal{F}(r_2) = b \wedge \neg \mathit{off} \wedge on \wedge \neg go \wedge stop$$
$$\mathcal{F}(r_3) = \neg b \wedge \textit{true} \wedge \textit{true} \wedge \neg go \wedge \neg stop$$

(i) **Assume**: button pressed, ventilation off, we (only) plan to start the ventilation.

- Corresponding valuation: $\sigma_1 = \{b \mapsto \textit{true}, \mathit{off} \mapsto \textit{true}, on \mapsto \textit{false}, start \mapsto \textit{true}, stop \mapsto \textit{false}\}$.
- Is our intention (to start the ventilation now) **allowed** by T? **Yes!** (Because $\sigma_1 \models \mathcal{F}(r_1)$)

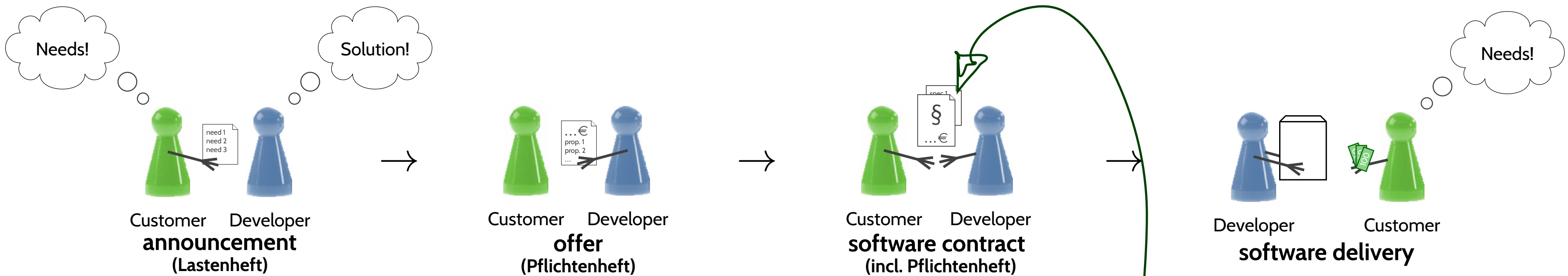
(ii) **Assume**: button pressed, ventilation on, we (only) plan to stop the ventilation.

- Corresponding valuation: $\sigma_2 = \{b \mapsto \textit{true}, \mathit{off} \mapsto \textit{false}, on \mapsto \textit{true}, start \mapsto \textit{false}, stop \mapsto \textit{true}\}$.
- Is our intention (to stop the ventilation now) allowed by T? Yes. (Because $\sigma_2 \models \mathcal{F}(r_2)$)

(iii) **Assume**: button not pressed, ventilation on, we (only) plan to stop the ventilation.

- Corresponding valuation:
- Is our intention (to stop the ventilation now) allowed by T? NO

Decision Tables as Specification Language

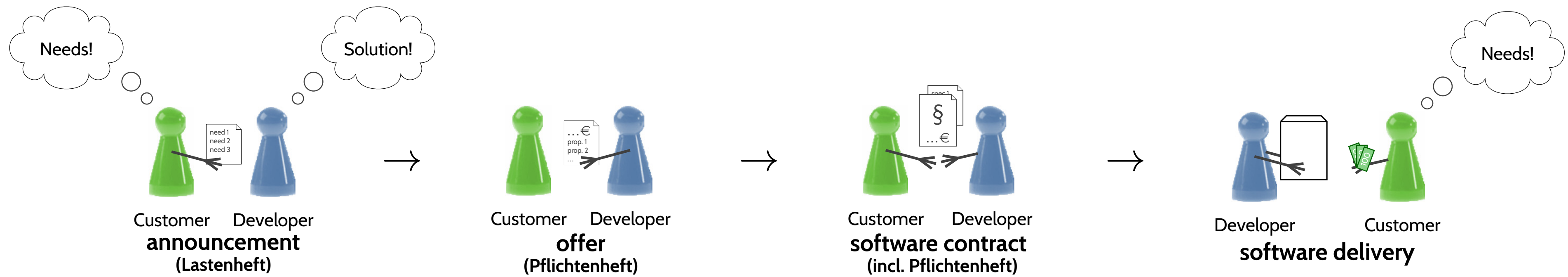


- Decision Tables can be used to **objectively** describe desired software behaviour.

- **Example**: Dear developer, please provide a program such that
  - in each situation (button pressed, ventilation on/off),
  - whatever the software does (action start/stop)
  - is **allowed** by decision table T.

T: room ventilation		r_1	r_2	r_3
b	button pressed?	$\times$	$\times$	$-$
off	ventilation off?	$\times$	$-$	$*$
on	ventilation on?	$-$	$\times$	$*$
go	start ventilation	$\times$	$-$	$-$
stop	stop ventilation	$-$	$\times$	$-$

Decision Tables as Specification Language



- Decision Tables can be used to **objectively** describe desired software behaviour.

- **Another Example**: Customer session at the bank:

$T1$: cash a cheque		r_1	r_2	else
c_1	credit limit exceeded?	$\times$	$\times$	
c_2	payment history ok?	$\times$	$-$	
c_3	overdraft < 500 €?	$-$	$*$	
a_1	cash cheque	$\times$	$-$	$\times$
a_2	do not cash cheque	$-$	$\times$	$-$
a_3	offer new conditions	$\times$	$-$	$-$

(Balzert, 2009)

- clerk checks database state (yields σ for $c_1, \dots, c_3$),
- database says: credit limit exceeded over 500 €, but payment history ok,
- clerk cashes cheque but offers new conditions (according to $T1$).

Requirements on Requirements Specifications

A **requirements specification** should be

- **correct**
  – it correctly represents the wishes/needs of the customer,
- **complete**
  – all requirements (existing in somebody's head, or a document, or …) should be present,
- **relevant**
  – things which are not relevant to the project should not be constrained,
- **consistent**, **free of contradictions**
  – each requirement is compatible with all other requirements; otherwise the requirements are **not realisable**,
- **neutral**, **abstract**
  – a requirements specification does not constrain the realisation more than necessary,
- **traceable**, **comprehensible**
  – the sources of requirements are documented, requirements are uniquely identifiable,
- **testable**, **objective**
  – the final product can **objectively** be checked for satisfying a requirement.

- **Correctness** and **completeness** are defined **relative** to something which is usually only in the customer's head.

  → is is **difficult** (if at all possible) to **be sure of correctness** and **completeness**.

Decision Tables for Requirements Analysis

Recall Once Again

Requirements on Requirements Specifications

A **requirements specification** should be

- **correct**
  – it correctly represents the wishes/needs of the customer,
- **complete**
  – all requirements (existing in somebody's head, or a document, or ...) should be present,
- **relevant**
  – things which are not relevant to the project should not be constrained,
- **consistent**, **free of contradictions**
  – each requirement is compatible with all other requirements; otherwise the requirements are **not realisable**,
- **neutral**, **abstract**
  – a requirements specification does not constrain the realisation more than necessary,
- **traceable**, **comprehensible**
  – the sources of requirements are documented, requirements are uniquely identifiable,
- **testable**, **objective**
  – the final product can **objectively** be checked for satisfying a requirement.

- **Correctness** and **completeness** are defined **relative** to something which is usually only in the customer's head.

  → is is **difficult** (if at all possible) to **be sure of correctness** and **completeness**.

Completeness

Definition. [*Completeness*] A decision table T is called **complete** if and only if the disjunction of all rules' premises is a tautology, i.e. if

$$\models \bigvee_{r \in T} \mathcal{F}_{pre}(r).$$

Tell Them What You've Told Them...

- **Decision Tables**: **one example** for a **formal requirements specification language** with
  - formal syntax,
  - formal semantics.
- Requirements analysts can use **DTs** to
  - **formally** (objectively, precisely)

  describe **their understanding** of requirements.
  Customers may need translations/explanation!
- **DT** properties like
  - (relative) completeness, determinism,
  - uselessness,

  can be used to **analyse** requirements.

  The discussed DT properties are **decidable**,
  there can be **automatic** analysis tools.
- **Domain modelling** formalises assumptions
  on the context of software; for DTs:
  - conflict axioms, conflict relation,

  Note: wrong assumptions can have serious consequences.

→ Mo

References

References

Arenis, S. F., Westphal, B., Dietsch, D., Muñiz, M., and Andisha, A. S. (2014). The wireless fire alarm system: Ensuring conformance to industrial standards through formal verification. In Jones, C. B., Pihlajasaari, P., and Sun, J., editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *LNCS*, pages 658–672. Springer.

Balzert, H. (2009). *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum, 3rd edition.

Bjørner, D. (2006). *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Springer-Verlag.

Bjørner, D. and Havelund, K. (2014). 40 years of formal methods. talk.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

IEEE (1998). *IEEE Recommended Practice for Software Requirements Specifications*. Std 830-1998.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

Rupp, C. and die SOPHISTen (2009). *Requirements-Engineering und -Management*. Hanser, 5th edition.

Wikipedia (2015). Lufthansa flight 2904. id 646105486, Feb., 7th, 2015.