**Softwaretechnik / Software-Engineering**

**Lecture 7: Formal Methods for Requirements Engineering**

2018-05-14

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

**requirement**

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

**IEEE 610.12 (1990)**

**requirements analysis**

1. The process of studying user needs to arrive at a definition of system, hardware, or software requirements.
2. The process of studying and refining system, hardware, or software requirements.

**IEEE 610.12 (1990)**
Risks Implied by Bad Requirements Specifications

- **design and implementation**, without specification, programmers may just "ask around" when in doubt, possibly yielding different interpretations
- **negotiation** (with customer, marketing department, or ...)
- **documentation**, e.g., the user's manual.
  - without specification, the user's manual author can only describe what the system *does*, not what it should do ("every observation is a feature")
  - later re-implementations.
    - the new software may need to adhere to requirements of the old software; if not properly specified, the new software needs to be a 1:1 re-implementation of the old
- **preparation of tests,** without a description of allowed outcomes, tests are randomly searching for generic errors (like crashes)
- **acceptance by customer,** resolving later objections or regress claims.
  - without specification, it is unclear at delivery time whether behaviour is an error (developer needs to fix) or correct (customer needs to accept and pay)
    - nasty disputes, additional effort
- **re-use**:
  - without specification, re-use needs to be based on re-reading the code
    - risk of unexpected changes

**Topic Area Requirements Engineering: Content**

- **Introduction**
  - **Requirements Specification**
    - Desired Properties
    - Kinds of Requirements
    - Analysis Techniques
  - **Documents**
    - Dictionary, Specification
  - **Specification Languages**
    - Natural Language
    - Decision Tables
      - Syntax, Semantics
      - Completeness, Consistency, ...
    - Scenarios
      - User Stories, Use Cases
      - Live Sequence Charts
      - Syntax, Semantics
- **Definition:** Software & SW Specification
- **Wrap-Up**
• 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
IEEE Recommended Practice for
Software Requirements
Specifications

Sponsor
Software Engineering Standards Committee
IEEE Computer Society
Approved June 1998
IEEE-SA Standards Board

Abstract: The content and quality of a good software requirements specification (SRS) are de-
scribed and several sample SRS outlines are provided. This recommended practice is aimed at
clarifying requirements of software to be developed but can also be applicable to assist in the selec-
tion of in-house and commercial software products. Students for compliance with IEEE/EIA
12207.1-1997 are also recommended.

Keywords: contract, customer, prototyping, software requirements specification, supplier, system
requirements specifications
**Example Requirement**

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.
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),
- **delimitations** (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.

---

### 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.

**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/acoustic signaling 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 signaling functionality.
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.

**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.

**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)
(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

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)*

<table>
<thead>
<tr>
<th>rule</th>
<th>explanation, example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R1</strong></td>
<td>State each requirement in active voice. Name the actors, indicate whether the user or the system does something. Not “the item is deleted”.</td>
</tr>
<tr>
<td><strong>R2</strong></td>
<td>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.”</td>
</tr>
<tr>
<td><strong>R3</strong></td>
<td>Discover incompletely defined verbs. In “the component raises an error”, ask whom the message is addressed to.</td>
</tr>
<tr>
<td><strong>R4</strong></td>
<td>Discover incomplete conditions. Conditions of the form ‘if-else’ need descriptions of the if- and the then-case.</td>
</tr>
<tr>
<td><strong>R5</strong></td>
<td>Discover universal quantifiers. Are sentences with “never”, “always”, “each”, “any”, “all” really universally valid? Are “all” really all or are there exceptions.</td>
</tr>
<tr>
<td><strong>R6</strong></td>
<td>Check nominalisations. Nouns like “registration” often hide complex processes that need more detailed descriptions; the verb “register” raises appropriate questions: who, where, for what?</td>
</tr>
<tr>
<td><strong>R7</strong></td>
<td>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?</td>
</tr>
<tr>
<td><strong>R8</strong></td>
<td>Clarify responsibilities. If the specification says that something is “possible”, “impossible”, or “may”, “should”, “must” happen, clarify who is enforcing or prohibiting the behaviour.</td>
</tr>
<tr>
<td><strong>R9</strong></td>
<td>Identify implicit assumptions. Terms (“the firewall”) that are not explained further often hint to implicit assumptions (here: there seems to be a firewall).</td>
</tr>
</tbody>
</table>
Natural Language requirements can be (tried to be) written as an instance of the pattern "⟨\ A \⟩ (\ B) (\ C) (\ D) (\ E) (\ F)\." (German grammar) where

<table>
<thead>
<tr>
<th>A</th>
<th>clarifies when and under what conditions the activity takes place</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>is MUST (obligation), SHOULD (wish), or WILL (intention); also: MUST NOT (forbidden)</td>
</tr>
<tr>
<td>C</td>
<td>is either &quot;the system&quot; or the concrete name of a (sub-)system</td>
</tr>
</tbody>
</table>
| 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) |

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

<table>
<thead>
<tr>
<th>RFC 2119</th>
<th>RFC Key Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. MAY This word, or the adjective &quot;OPTIONAL&quot;, mean that a truly optional. One vendor may choose to include the item if it adds to the product while another vendor may omit it. An implementation which does not include the option, though perhaps with reduced functions, same suite as implementation which does include a particular option must be prepared to interoperate with another implementation that does not include the option (except, of course, for the option provided.)</td>
<td></td>
</tr>
</tbody>
</table>
| 6. Guidance in the use of these Imperatives

Imperatives of the type defined in this memo must be used sparingly. In particular, they MUST only be used when the method is actually required for interoperability; or to limit behavior potentially for causing harm e.g., limiting extraneous systems implementation. May not be used to try to impose a particular implementation 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 implementing MUST or SHOULD, or doing something the specification says MUST NOT be done may be very subtle. Document authors should be explicit in elaborating the security implications of not following recommendations or requirements as most implementors will not have the benefit of the experience and discussion that pre specification. |

8. Acknowledgments

The definitions of these terms are an amalgam of definitions taken from a number of RFCs. In addition, suggestions have been incorporated from a number of people including Robert Ullmann, Thomas Narten, Neal McBurnett, and Robert Elz.
Decision Tables

- (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
**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

<table>
<thead>
<tr>
<th>$T$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$\times$</td>
<td>$-$</td>
<td>$\ast$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$-$</td>
<td>$\times$</td>
<td>$\ast$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$\times$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$-$</td>
<td>$\times$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

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

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

- Columns

$$
\begin{array}{cccc}
  & v_{1,1} & \cdots & v_{1,n} \\
  c_1 & & & \\
  \vdots & & & \\
  \vdots & & & \\
  c_m & & & \\
  \vdots & & & \\
  \vdots & & & \\
  a_1 & & & \\
  \vdots & & & \\
  \vdots & & & \\
  a_k & & & \\
\end{array}
$$

- **premise**
- **effect**
Each rule \( r \in \{ r_1, \ldots, r_n \} \) of table \( T \) is assigned to a propositional logical formula \( F(r) \) over signature \( C \cup A \) as follows:

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

\[
F(r) := F(v_1, c_1) \land \cdots \land F(v_m, c_m) \land F(w_1, a_1) \land \cdots \land F(w_k, a_k)
\]

where

\[
F(v, x) = \begin{cases} 
  x, & \text{if } v = \times \\
  \neg x, & \text{if } v = - \\
  \text{true}, & \text{if } v = * 
\end{cases}
\]
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.
References


