TECHNISCHE UNIVERSITÄT DRESDEN

"Friedrich List" Faculty of Transport and Traffic Sciences Institute of Railway Systems and Public Transport

Chair of Transportation Systems Engineering

Diploma Thesis

ON THE DOMAIN-SPECIFIC FORMALIZATION OF REQUIREMENT SPECIFICATIONS – A CASE STUDY OF ETCS*

Moritz Dorka
born on March 18th, 1988 in Kirchen (Sieg)
matr. no. 3472533, moritz.dorka@mailbox.tu-dresden.de

Examined by:
Prof. Dr. rer. nat. Jörg Schütte and Dr.-Ing. Sven Scholz

Supervised by:
Dr.-Ing. Sven Scholz

Submitted on June 19th, 2015

CHANGES TO THIS DOCUMENT

The electronic version of this paper has seen minor editing after official submission and prior to its online publication. These changes are listed below.

- Added the poster (see facing page) which accompanied the defence of this thesis.
- Removed epigraphs preceding each section due to copyright concerns. Specifically those were:

Section 1	A quote on the peculiar approach to software development at NASA.
Section 2	A statement regarding the common confusion over the meaning of *shall* and *will*.
Section 3	An anecdote about allegedly unreadable code.
Section 4	A criticism concerning the lack of standardization in the ETCS domain.
Section 5	An alternative long form of the common Latin abbreviation "q.e.d.".
Appendices	Various humorous key words to indicate legal obligations in specification texts.

  The sources of those texts are given at the positions where the respective epigraphs were originally placed.
- The software "Reqtify" used to be defined as *requirement traceability tool*. This has been changed to the more generic term *requirements management tool*.
- The display of different tick marks such as `` ` `` and `'` in the various Listings of the Appendices (esp. A.2) has been improved.
- The mention of "approx. 14%" has been changed into "approx. 7%" when talking about the `@DomainSpecific` annotation in Section 3.6. Both numbers are, in fact, nothing more than a wild guess. However, 7% makes more sense in the context.
- Minor fixes to spelling in [GKNV93], [IEE98], [MMK01] and [RAC11].
- Fixed a wrong in-text reference to a Listing in Section 2.3.1.

Note that both this page and the following page have been added to the document specifically for the electronic edition of this thesis and are therefore exempted from page numbering.

Moritz Dorka, October 8th, 2015

TECHNISCHE UNIVERSITÄT DRESDEN

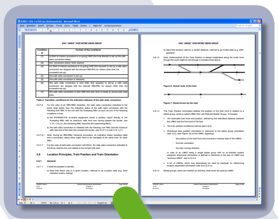
"Friedrich List" Faculty of Transport and Traffic Sciences — Chair of Transportation Systems Engineering

ON THE DOMAIN-SPECIFIC FORMALIZATION OF REQUIREMENT SPECIFICATIONS – A CASE STUDY OF ETCS

sample lifespans: (begin development|commissioning – scrapping of last unit)		
	Airbus A300	1970|1972 – 2050
	Crocodile (French train protection)	1872|1933 – soon…
	ETCS	1998|2001 – ?

Raw requirements

ETCS subset-026, Baseline 3.3.0, 538 pages of *Microsoft Word* documents



1 MOTIVATION

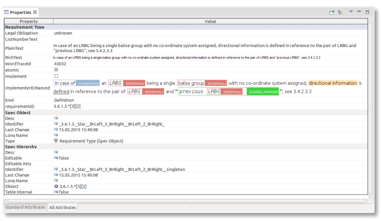
Modern systems are becoming increasingly complex. This has proven to be challenging especially for safety-critical applications where such systems need to remain maintainable for their entire, often decades-long, operating lifetimes.

Since maintainability must be guaranteed irrespective of the original supplier and their individual business decisions (bankruptcy, arbitrary "end of life"), the idea of a so-called "open proofs" approach was born. For today's ever more common software-driven applications this mandates to make both the resulting software program and all auxiliary tools used for its creation available as open-source. Thus, anyone with the respective skills can examine the functioning of the system and take over its maintenance at any point in time.

This thesis shows an implementation of that approach with a tool to enable proper requirements management for the European Train Control System (ETCS).

Single processed requirement

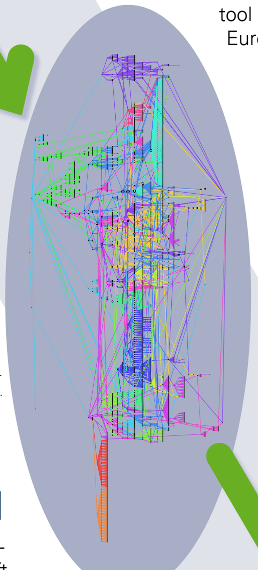
domain-specific metadata of a single requirement as shown in *ProR*



2 OBJECTIVE

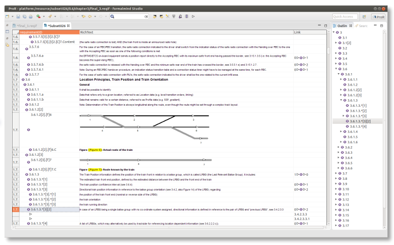
ETCS requirements are provided as *Microsoft Word* files which are neither easily implementable (no metadata, no information on changes between versions) nor sufficiently traceable (as mandated by EN 50128:2011, clause D.58 for SIL 3/4).

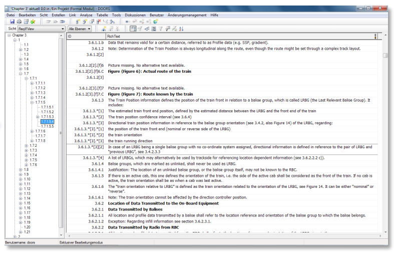
Therefore, this thesis provides a tool which can read these *Word* documents and transform them into an appropriate, openly standardized format which does not suffer from these problems.



Tree of requirements

This visualization shows chapter 3 of subset-026 with approx. 3.500 requirement artifacts connected by 4.200 edges which bear hierarchical or other custom relations.





Processed requirements

Eclipse's *ProR* (above) and *IBM DOORS* (below) displaying an excerpt of the specification

3 IMPLEMENTATION

Although there exist numerous commercial software products for the purpose of requirement import from *Word*, they perform poorly for the core ETCS specification "subset-026" which constituted the focus of the thesis. This is mostly due to the lack of proper structure and the peculiar formatting of those files.

Hence, a novel tool comprising some 16k SLOC of Java was developed to automatically convert those documents into the Requirement Interchange Format (ReqIF). This is a relatively new, but standardized XML-based file format specifically designed for the storage and exchange of requirements.

To do so, the tool first reads the files using a customized, open-source library to access the internal *Word* file structure. It then elicits individual requirements (usually single paragraphs or cells in tables) and assigns them unique, context-sensitive identifiers. Through the employment of regular expressions and Natural Language Processing the textual contents of those requirements are subsequently enhanced for implementation purposes with various metadata. In the next step relations between different requirements are detected and other artifacts (pictures, equations) are handled. The result of these efforts is stored in a tree which eventually gets written to ReqIF. This output can then be consumed by common RM-tools such as *DOORS* or modelled using *SCADE*.

<lots of XML>

software running on a computer for decades

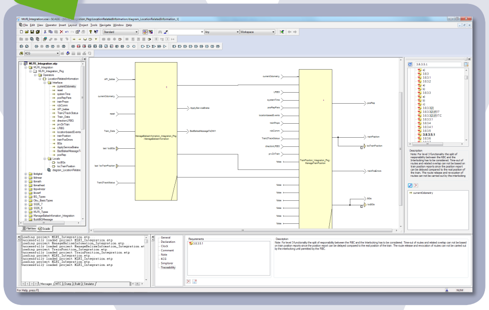
V-model according to EN 50126:1999, p. 26 depicting an entire system's lifecycle (simplified)

4 CONCLUSION

The developed tool is capable of handling all eight chapters of subset-026 and processes them into roughly 22.000 individual requirements.

Its output is used productively within the R&D-project "openETCS" aimed at creating a manufacturer-independent reference implementation covering the software part of ETCS' core onboard component, the European Vital Computer (EVC).

Currently, the tool is tailored to this specific use-case. However, a reconfiguration for other specification documents is also feasible. Thereby it could become an accelerator for the wider adoption of ReqIF, and thus formalized requirements management, in systems engineering.



Turning the requirements into a system

Esterel's *SCADE Suite* displaying a model of a part of the ETCS EVC. The requirements are shown in the upper and lower right panels.

DIPLOMA THESIS

Author:	Moritz Dorka
Examined by:	Prof. Dr. rer. nat. Jörg Schütte and Dr.-Ing. Sven Scholz
Duration:	01/2015 – 06/2015

*German title in conformance with the "Richtlinie für die Anfertigung der Diplom-Arbeit":

Teil-automatisierte Formalisierung von
Lastenheftanforderungen am Beispiel ETCS

Dresden, June 19th, 2015 signature of the student

Bibliografischer Nachweis

DORKA, Moritz:
On the domain-specific formalization of requirement specifications – A case study of ETCS
Diplom-Arbeit, Dresden 2015
Technische Universität Dresden, Fakultät Verkehrswissenschaften „Friedrich List", Institut für Bahnsysteme und Öffentlichen Verkehr, Professur Verkehrssystemtechnik
Studiengang Verkehrsingenieurwesen
99 Seiten, 23 Abbildungen, 5 Tabellen, 6 Programmtexte, 99 Quellenangaben

ABSTRACT

This paper presents a piece of software to automatically extract requirements captured in Microsoft Word files while using domain knowledge. In a subsequent step, these requirements are enhanced for implementation purposes and ultimately saved to ReqIF, an XML-based file format for the exchange of specification documents. ReqIF can be processed by a wide range of industry-standard requirements management tools. By way of this enhancement a formalization of both the document structure and selected elements of its natural language contents is achieved.
In its current version, the software was specifically developed for processing the Subset-026, a conceptually demanding specification document covering the core functionality of the pan-European train protection system ETCS.
Despite this initial focus, the two-part design of this thesis facilitates a generic applicability of its findings: Section 2 presents the fundamental challenges of weakly structured specification documents and devotes a large part to the computation of unique, but human-readable requirement identifiers. Section 3, however, delves into more domain-specific features, the text processing capabilities, and the actual implementation of this novel software.
Due to the open-source nature of the application, an adaption to other use-cases can be achieved with comparably little effort.

Diese Arbeit befasst sich mit einer Software zur automatisierten Extraktion von Anforderungen aus Dokumenten im Microsoft Word Format unter Nutzung von Domänenwissen. In einem nachgelagerten Schritt werden diese Anforderungen für Implementierungszwecke aufgewertet und schließlich als ReqIF, einem XML-basierten Dateiformat zum Austausch von Spezifikationsdokumenten, gespeichert. ReqIF wird von zahlreichen branchenüblichen Anforderungsmanagementwerkzeugen unterstützt. Durch die Aufwertung wird eine Formalisierung der Struktur sowie ausgewählter Teile der natürlichsprachlichen Inhalte des Dokuments erreicht.
Die jetzige Version der Software wurde speziell für die Verarbeitung des Subset-026 entwickelt, eines konzeptionell anspruchsvollen Anforderungsdokuments zur Beschreibung der Kernfunktionalität des europaweiten Zugsicherungssystems ETCS.
Trotz dieser ursprünglichen Intention erlaubt die zweigeteilte Gestaltung der Arbeit eine allgemeine Anwendung der Ergebnisse: Abschnitt 2 zeigt die grundsätzlichen Herausforderungen in Bezug auf schwach strukturierte Anforderungsdokumente auf und widmet sich dabei ausführlich der Ermittlung von eindeutigen, aber dennoch menschenlesbaren Anforderungsidentifikatoren. Abschnitt 3 befasst sich hingegen eingehender mit den domänenspezifischen Eigenschaften, den Textaufbereitungsmöglichkeiten und der konkreten Implementierung der neuen Software.
Da die Software unter open-source Prinzipien entwickelt wurde, ist eine Anpassung an andere Anwendungsfälle mit relativ geringem Aufwand möglich.

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Verkehrswissenschaften „Friedrich List"

Themenblatt zur Diplomarbeit*)

von ~~Frau~~/Herrn cand. Ing. **Moritz Dorka**

Thema (Aufgabenstellung siehe Anlage):

Teil-automatisierte Formalisierung von Lastenheftanforderungen am Beispiel ETCS

Institut für Bahnsysteme und Öffentlichen Verkehr
Professur für Verkehrssystemtechnik

1. Prüfer: Prof. Dr. rer. nat. Jörg Schütte

2. Prüfer/Beisitzer: Dr.-Ing. Sven Scholz

Zur Anfertigung der Diplomarbeit wurde eine dreiseitige Vereinbarung (TUD, Student, Dritter) abgeschlossen: ~~Ja~~ / [✓ Nein] (Zutreffendes unterstreichen, Nichtzutreffendes streichen)

[redacted] J. Schütte
Leiter der Professur

Unterschrift des Prüfers

Dresden, den 05.06.2014

Ausgabetag: 19.01.2015	Bestätigung durch die Fakultät: [redacted]
Abgabetermin: 19.06.2015	
Abgabetag:	Bestätigung durch die Fakultät: ....

Bestätigung durch die Fakultät für eine genehmigte Verlängerung der Bearbeitungszeit: ....

Hiermit bestätige ich den Empfang der Aufgabenstellung für meine Diplomarbeit und erkenne die Festlegungen der Richtlinie für die Anfertigung der Diplomarbeit – insbesondere den Punkt 11 - an:

[redacted]

Unterschrift des Diplomanden

Dresden, den 19. 1. 2015

*) siehe Diplomprüfungsordnung §§ 20 bis 23 und § 25 sowie Studiendokumente 4., Punkt 3 der Regelung für die Ausgabe und Registratur der Studienarbeiten und Diplomarbeiten

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Verkehrswissenschaften „Friedrich List"

Institut für Bahnsysteme und Öffentlicher Verkehr, Professur Verkehrssystemtechnik

Aufgabenstellung zur Diplomarbeit

für Herrn cand. Ing. Moritz Dorka zum Thema

Teil-automatisierte Formalisierung von Lastenheftanforderungen am Beispiel ETCS

ETCS ist ein neues europäisches Zugsicherungssystem für das aktuell noch keine fahrzeugseitigen Komponenten verfügbar sind, die universell auf allen ausgerüsteten Strecken eingesetzt werden können. Um diesem Manko zu begegnen soll in dem EU-finanzierten openETCS Projekt die Software für einen entsprechenden Fahrzeugrechner herstellerunabhängig nach OpenSource-Prinzipien entworfen werden.

Im Rahmen der Arbeit ist ein auf dem Eclipse Modeling Framework aufsetzender Importer für die in Microsoft Word 8.0+ vorliegenden Anforderungsdokumente, zunächst am Beispiel des zentralen Dokuments „Subset-026", zu erstellen, der die einzelnen Anforderungen nachverfolgbar im Requirements Interchange Format (ReqIF) ausgibt. Außerdem sollen vorbereitende Schritte zur Formalisierung der größtenteils natürlichsprachlichen Anforderungstexte unternommen werden. Dabei sind zumindest folgende Teilaufgaben automatisiert zu lösen:

1. Kategorisierung der Anforderungen (z.B. Definition, Beispiel, Anmerkung, optional / verpflichtend, gelöscht, fahrzeugseitig / streckenseitig relevant)
2. Erkennung und maschinenlesbare Ausgabe von in den Texten erwähnten Entitäten, deren Attributen sowie (gegebenenfalls) booleschen Abhängigkeiten
3. Identifizierung von speziellen Schlüsselwörtern / -phrasen, die in anderen Anforderungen referenziert werden (z.B. „Linking information is used" in Anforderung 3.4.4.2.1.1)
4. Markierung von problematischen Passagen aufgrund unspezifischer Ausdrucksweise ("this list is *not exhaustive*" in Anforderung 4.4.12.1.4), Unterspezifikation ("[...] in case of a loss of the safe radio connection, [...] the involved entities shall consider the communication session still established for a *defined time*." in Anforderung 3.5.4.1) oder Widersprüchen ("A level 2/3 MA and track description information *shall* be received from the RBC before the level transition border. *If not*, [...]" in Anforderung 5.10.3.2.3)
5. semantisch korrekte Abbildung von in Tabellen- und Listenform verfassten Anforderungen, eventuell Ideen zur Behandlung von nichttextuellen Anforderungen (Bilder, Grafiken).

Die Arbeit ist in englischer Sprache zu verfassen. Eine leichte Anpassbarkeit der resultierenden Software an mögliche künftige Anwendungsszenarien ist wünschenswert. Bei der Entwicklung sollen die Maßstäbe der EN50128:2011, Abschnitt 6.7 für Werkzeuge der Klasse T1 berücksichtigt werden.

[illegible]

Prof. Dr. rer. nat. Jörg Schütte

Dresden, 5. Juni 2014

THESES

1. The Microsoft Word file format is not especially beneficial to the storage of specification documents due to its low degree of structure and high functional excess.
2. Requirements captured within Microsoft Word files can be automatically extracted and formalized by using domain knowledge.
3. Traceability is an important property of requirements, and Microsoft Word files cannot serve as a decent trace source without dedicated postprocessing.
4. The current representation of Subset-026 does not particularly lend itself as an input to a highly regulated development process in accordance with EN 50128.
5. ReqIF is a file format tailored to the task of requirements management that can represent the contents of specification documents in a manner substantially more formal than Microsoft Word.
6. To derive maximum profit from the efforts on formalization, its representation must be easily consumable by humans as well as by automated means.
7. In a domain-specific context, a regex-based approach to natural language contents is still preferable to NLP / Machine Learning because of its speed and predictability.
8. When processing the raw specification documents of systems of sufficient complexity, using a fully automated tool is indispensable in order to rule out human error.
9. A tool without proper tailoring to the domain will inevitably underperform and may even omit important elements of the specification, with the possible result of fatal outcomes.

CONTENTS

1 INTRODUCTION

deleted epigraph – Peter Norvig, [Sei09, p. 311]

Europe's railway network construction peaked during the Industrial Revolution in the mid 19th and early 20th centuries when the continent was still very much partitioned into small individual countries, all proudly state-owning their respective individual railway operators. Each of those companies quickly started to maintain their very own operational rules and guidances blanketing their knowledge of how they thought a train should safely move from A to B. Since running trains in the early days was in fact merely an A-to-B business and it took quite some time for the underlying networks of each operator to begin converging, nobody felt any urgent need to change this approach. Even more so, *incompatibility*, the obvious outcome of this distinctive sectionalism, was seen as a protective factor against possible invasions by neighbouring countries – Europe's different track gauges still bear visible witness of that tradition [Kle02, p. 17 & pp. 62 f.].
Despite all these intentional hindrances, history has left the continent with the densest railway network on the planet. And today, in a more and more globalized world with bureaucrats from Brussels pressuring, the interests of those individual network operators are steadily shifting towards what is commonly referred to as *interoperability*. Previous efforts in this direction, however, almost exclusively involved such decisive features as standardizing the size, shape and material of toilet seats in passenger carriages [UIC90, Sec. 4.1.4 ff.] (i.e. "non-vital parts") while the aforementioned rules and guidances, the blood-tainted holy grail of train operation, have remained largely untouched.

One major part of those national rules regards train protection systems, technical devices which in essence allow or restrict movements of trains on the basis of trackside inputs. Europe is blessed with a plethora of different such systems, making it difficult for a locomotive, where the system's trainborne part is usually installed, to operate in several countries.
To overcome this drawback the idea of a common protection system was born in the late 1980ies [UNI13] and later became known as European Train Control System (ETCS). By exposing standardized trackside interfaces to the locomotive in every country of operation, it theoretically allows free movement of all equipped rolling stock on all equipped lines. However, in practise there are two main reasons why that is still far from reality. The technical reason refers to additional train parameters unrelated to ETCS which also must be taken into account when speaking about true interoperability, like different loading gauges or traction systems. But more importantly, there is a second not-so-technical reason which essentially boils down to the current implementation of ETCS itself.
As the core ETCS specification, the so-called Subset-026, was written in a joint effort by the Union Industry of Signalling (UNISIG), a consortium of Europe's major railway suppliers, and the ERTMS Users Group (EUG), which represents the European network infrastructure owners, it

comes at no big surprise that they were not always able to agree on a common implementation for a specific functionality. Moreover, some contributors may have even been tempted to add parts of their respective old fashioned, but field-proven national rules[1] and techniques[2] as a supplement to the system, thereby enhancing its complexity. In addition, ETCS allows to implement only a restricted set of its operational modes on a certain track in order to avoid technological overkills on rarely used branch lines and thus helps keeping costs down.

Hence, for the average time- and money-constrained implementer of the trainborne part of ETCS, all of the aspects above lead to the fair conclusion to take into account only parts of the original specification, namely those which are relevant to the specific country and the track a newly built or retrofitted locomotive is supposed to run on. Although this completely contradicts the original aim of interoperability, it is exactly what the market faces today: A variety of on-board computers, so-called European Vital Computers (EVCs), built by different manufacturers for different locomotives, none of which are really interchangeable.
In an attempt to solve this dilemma, DB Netz, the German railway infrastructure operator and driving force behind this thesis, teamed up with several partners from different European countries in the scope of an EU-funded research & development project entitled "openETCS" to try and build the software part of the first *complete EVC* based on open-source methodologies. Not only should this software finally allow for the long awaited interoperability, but the open-source approach also solves the lifecycle issue, which refers to the inherent problem of all long-living devices: At some point support and maintenance may be brought to an end by the original supplier and nobody else will be able to take over. Furthermore, openETCS gives EVC hardware manufacturers the chance to fit their products with a standardized core they can easily extend according to individual needs and thereby significantly decrease time-to-market and development costs.

1.1 Motivation

Creating software from a given set of specifications by using nothing but open-source auxiliary tools, while maintaining compliance with the ambitious Safety Integrity Level (SIL) 4 requirements of the relevant software safety standard EN 50128 [CEN11] at the same time, is not exactly an easy task. One problem that comes up right in the beginning concerns the ETCS specification documents itself. Fortunately, those are publicly available through the European Railway Agency (ERA) which has recently taken over their maintenance from UNISIG. However, their high degree of natural language contents in plain (Brussels-)English together with little attention to any kind of formalization (inconsistent layout, unclear boolean relations, ill-defined formulas, . . .) make them hard to digest as an input to any actual implementation work. Since the current version (3.3.0)[3] of the core specification document, the so-called Subset-026 mentioned earlier, already comprises 538 pages spread across eight individual chapters (and this excludes any supplementary material concerning system tests, special interfaces and the like), it should go without saying that any tool which can help to grasp this enormous amount of content is

[1] *Radio-Infill*, for example, would certainly not be a part of ETCS without the Italians [Sch12, p. 15, CR 742].
[2] The *Euroloop* as a very "German invention", strongly influenced by LZB-technology, should provide a good example here.
[3] During the writing of this thesis an update (3.4.0) was published. However, this will not be considered here.

highly appreciated[4]. In addition, the monolithic nature of the documents with lots of continuous text and lengthy tables does not recommend itself for credible traceability and is not easy to read or work with, either [Mö14, p. 75]. Hence, the requirements captured in those documents need to be extracted and stored in a different format tailored for the task of requirements management, which, from a lifecycle point of view, should be openly documented and standardized. Taking into account the extensive amount of English text in those specifications, it is also desirable to subsequently try and (pre-)process the requirements by automated means as diligently as possible to ease comprehension. This effort towards formalization not only aids the actual implementer but can also be of great help for other steps in the V-Model like the Verification & Validation (V & V) activities [CSN+15, Sec. 6.2.3 f].
Currently, openETCS uses the Safety-Critical Application Development Environment (SCADE) implementation of Reqtify [Est15], a Requirements Management (RM) tool rooted in the aviation industry, in order to import the specification documents and afterwards only manual means of formalization via the usual modelling tools of SCADE Suite / System (see [JPD14] for a more thorough explanation of this primary toolchain). Two closed-source tools based on proprietary file formats, both SCADE and Reqtify contradict the project goal, while the latter even lacks certain features relevant to the ETCS domain, such as proper hierarchy extraction and seamless handling of requirements that are not applicable in the current context. For all these reasons it makes sense to strive for an open-source replacement, as it can be better customized for this particular application.

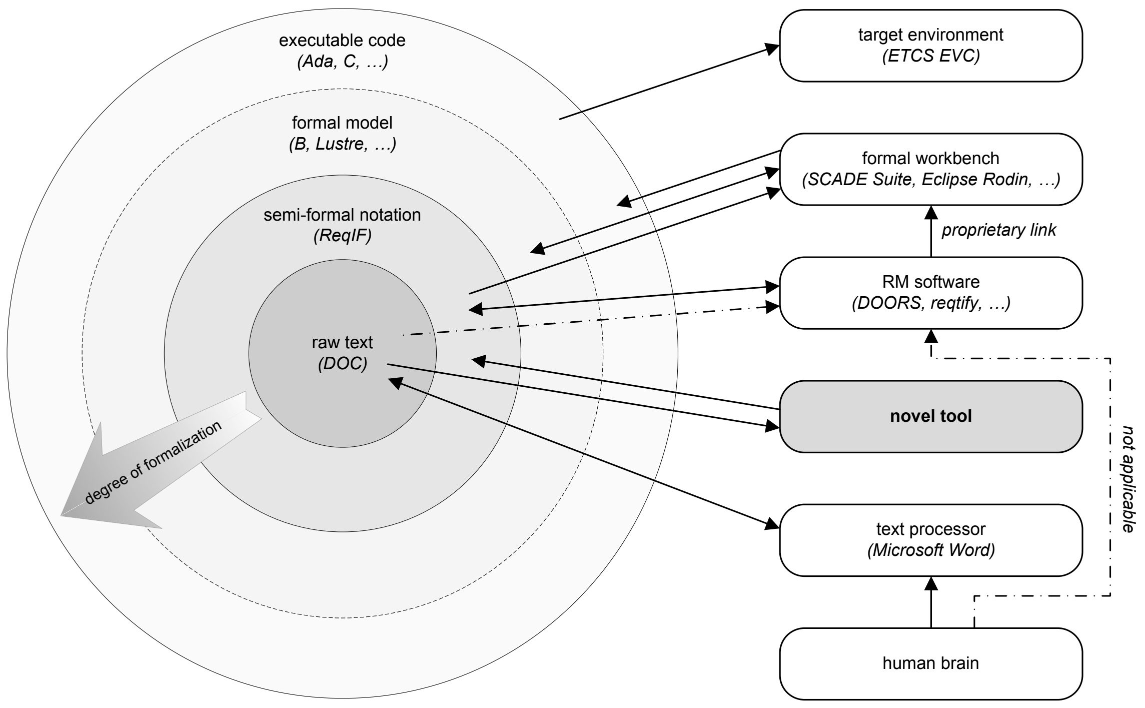
Therefore, the idea is to craft a novel piece of software which can directly import the original specification documents, transform the content into a meaningful hierarchical tree of requirements using domain knowledge, enhance those requirements by attributing computed metadata and eventually write the result into a new file that is easily traceable (both back to the original source as well as to any downstream activities) by automated means [And15]. It also allows for multitenant use and is likely to remain readable for decades to come. This file may then be imported back into Reqtify or other RM-tools and constitute the basis for any actual work further downstream in the V-Model.

Although this thesis is very much motivated by the concrete challenges of the Subset-026, its core ideas regard the automated formalization of weakly structured specification documents and the possibilities of making such input traceable in a user-friendly way. This is why it should be applicable to technical papers in other domains as well.

1.2 Previous formalization attempts

The term "formalization" is used in a rather broad sense in this thesis and essentially describes the process of mining data from unstructured text in order to make it "a little more formal". To be fair, the tool to be introduced in Section 3 will not create any kind of formal language, which could be directly transformed into executable code, but at best a semi-formal representation of the input which renders it partly processable by automated means. Hence, the tool should be

[4]To be fair, the phrase *enormous amount* must be considered in context of the railway domain. Taking a glimpse at other industries reveals that a specification for the software part of a space shuttle running 40 000 pages is nothing exceptional [Fis96].



⟷ = directed data flow

–·→ = data flow involves significant manual labour and/or is of inferior quality

Note: This is not a Venn diagram in a strict sense. Although the information stored in each wider concentric circle is a (more abstract) superset of its predecessor, the layers all represent this information differently.
A dedicated layer for semi-formal UML Modelling *(SCADE System, Eclipse Papyrus, ...)* is not shown here.

Figure 1: Placement of the novel tool in the existing tooling landscape

understood as a domain-specific replacement of the generic specification importers offered by common RM applications such as Reqtify, DOORS, Requisite Pro and friends with a subsequent treatment of the resulting data to allow for the use-cases outlined in Section 2.2. Figure 1 visualizes this context and locates the novel tool.

The ETCS specifications have seen various attempts to actually create fully formal versions from selected parts of their contents. One well known example is that of the *EuRailCheck* tool developed by an Italian research institute on the basis of Requisite Pro [CCM$^+$09], [CRST10], [CCM$^+$10]. Others which involve more manual labour include [ERT15], [Feu12] and [HMBMn11].
The tool described in this thesis differs from those previous works in that it aims for full coverage of the input documents in exchange for a lower degree of formalization while remaining completely automated. Or to put it more bluntly: This tool targets the real world where the plentiful redundancies and contradictions that naturally come with unconstrained input must inevitably be resolved by humans using extrinsic knowledge in a clearly separated, subsequent step.

For a more thorough comparison of the tool with some of its unequal competitors see Section 3.5.

2 PROCESSING SPECIFICATION DOCUMENTS

deleted epigraph – Michael Jackson, [Jac95, pp. 125 f.]

The following Sections will dig into the analysis of specification documents starting on a relatively abstract, macroscopic level, to encourage general applicability of their contents. Especially the input parsing techniques (Section 2.1.1) and the discussion of feasible means of requirement formalizations (Section 2.2) are written in such a way as to minimize their domain-specificity. Only the last Section 2.3 will exhibit a gradual shift towards more concrete examples from the Subset-026. This is mainly to ease comprehension as the fundamental concepts of traceability presented therein are not limited to this specific use-case.
Finally, Section 3 will extend on these abstract foundations and present their concrete implementation for the ETCS domain. Due to its lack of generality, the actual content processing of the Subset-026 will also be postponed to that Section.

2.1 Structural considerations

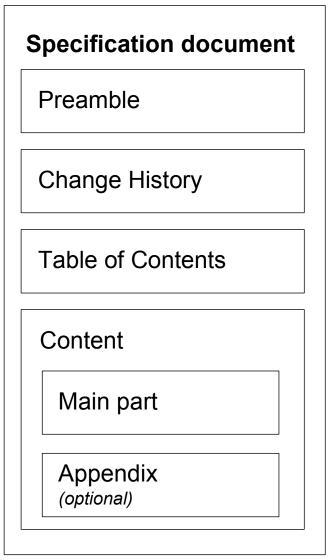


Figure 2: Sequential structure of an ETCS specification document

ETCS specifications are available on the ERA-website in both Portable Document Format (PDF) and Microsoft Word (DOC) formats. Their general structure is depicted in Figure 2. For lengthy specifications which are split across several chapters, such as the Subset-026 (see Section 1.1), each chapter is published separately.
Irrespective of fundamental weaknesses of the specifications discussed in Section 1.1, neither of those two file formats makes a particularly good candidate for automated processing. Although the PDF-format is standardized, fully specified, with special trimmed-down versions for long-term storage existing, it has a strong focus on maintaining the *layout* of a document. This means it is strictly page-oriented, comes with fonts embedded, and all information which is not relevant for viewing or printing is usually stripped from such a file. DOC, in contrast, because it is a format designed for editing rather than viewing, puts much more emphasis on the *structure* of a document. So rather than being page-oriented, it features support for complex tables, hierarchies of lists, logical groupings of characters to assign common properties, designated constructs for recurring elements such as headers and footers and the like. Only upon opening such a file will its contents be rendered onto individual pages using locally available fonts – a process very much different from PDF, where essentially everything is prerendered (so e.g. a table may be represented as a

bunch of vertical and horizontal lines, positioned absolutely, with characters in between instead of a logical construct of rows and columns [HB07]) by the application which originally produced the file.

To sum this up, despite being far from ideal the DOC format is preferable for the task of content extraction as it features a lot more accompanying data which can help to understand the meaning of the extracted elements. Directly using DOC as the input also prevents any errors possibly introduced by the producer of the PDF files (which in fact only constitute a "secondary source" since they were all converted from the original DOC files at some point). However, the downside of this approach is having to deal with quite a complex, historically grown, proprietary file format of which only parts are openly documented ([Mic08a], [Mic14b]) and the only truly complete implementation is by Microsoft itself (i.e. the software "Microsoft Word"). Moreover, this binary format must be considered deprecated by today's standards as Microsoft introduced a new Extensible Markup language (XML)-based variant starting with Word 2007. On the one hand this somewhat limits the use of the described tool for future applications. But on the other hand it once more emphasizes how a proprietary format which comes with a high degree of vendor lock-in makes a poor choice for any specification covering a system of considerable lifetime [Sch14].

2.1.1 The input format: DOC

Figure 3 on the facing page shows a Unified Modeling Language (UML) class diagram of the basic structure of a DOC file. The textual contents of such a file are basically made up of a hierarchy of so-called *Ranges*. The root is a (possibly) very long *Range* covering the entire document with arbitrary (dynamically constructable) children for individual parts of the document all the way down to a single character (confer with the term *granularity* of Section 2.3). By nesting different kinds of those *Ranges* into one another, arbitrarily complex structures can be represented. The various circular dependencies between *Character Run*, *Field*, *Footnote*, *Endnote*, *Table* and *Subpart* (including its self-reference) give an impression of how this is implemented[5]. Requirements (as well as all other textual elements) are trapped within those *Ranges*. So a suitable extraction algorithm needs to be able to map a *Range* (and thus implicitly all its children as well) onto a requirement and then read out the contents of that *Range* including all relevant associated data such as formatting properties, visibility or change tracking information.
As stated in Section 2.1, DOC files are not rendered onto pages until they are opened. Hence, there is no entity named *Page* anywhere in Figure 3. In fact, all the page data are stored consecutively, and for the sole purpose of content extraction there is usually no need to reconstruct their original (paginated) layout[6]. The only true kind of textual segmentation present in a DOC are the completely separate *Ranges* for contents which do not belong to the main part of a document (i.e. headers, footers, textboxes, foot- and endnotes; see Section 2.3.2) and the so-called *Section*. The latter is essentially an artificial structure to associate different page layouts (which may include specific headers and footers) with parts of a document. Specification documents, as they are considered here, may only contain a single *Section*.

[5]Not every constellation which Figure 3 suggests is actually supported by DOC and sometimes there are limits to the number of nesting levels allowed.
[6]For a notable exception see Section 3.3.3

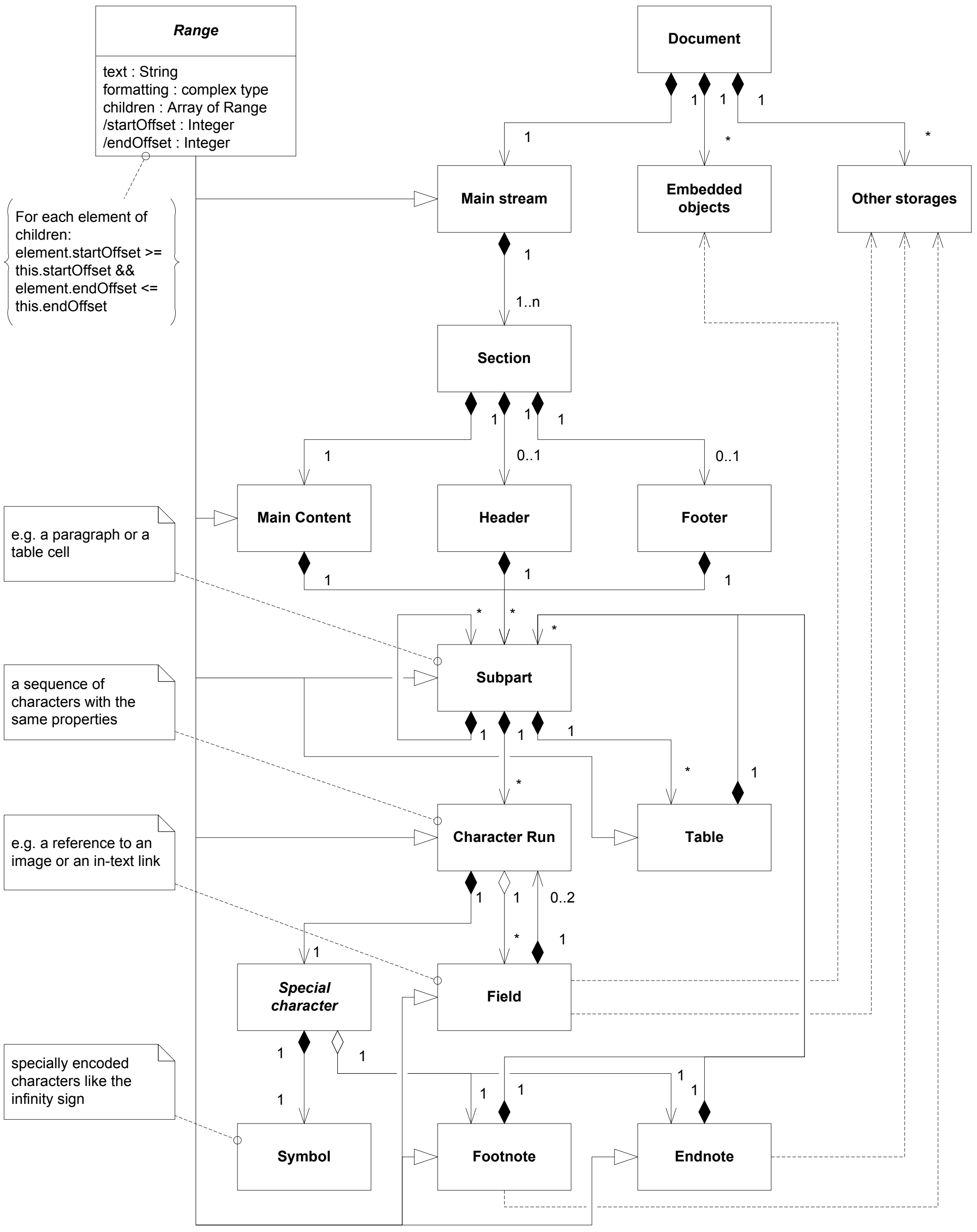

Range
text : String
formatting : complex type
children : Array of Range
/startOffset : Integer
/endOffset : Integer
For each element of children: element.startOffset >= this.startOffset && element.endOffset <= this.endOffset
Document
Main stream
Embedded objects
Other storages
Section
Main Content
Header
Footer
e.g. a paragraph or a table cell
Subpart
a sequence of characters with the same properties
Character Run
Table
e.g. a reference to an image or an in-text link
Special character
Field
specially encoded characters like the infinity sign
Symbol
Footnote
Endnote
1
*
1..n
0..1
0..2

Figure 3: Internal structure of a DOC-file *(simplified)*

Although the presented structure is very flexible regarding the contents which it can represent, it does impose a few restrictions. Among the most notable is the fact that *Ranges* always have some sort of absolute beginning and end (i.e. `startOffset` and `endOffset`). Therefore, they cannot easily represent a floating entity such as a table or figure, which is dynamically placed at an appropriate position within the document by some layout algorithm. This restriction influences the tracestring computation, see Section 2.3.3.

2.1.2 Different parts of a specification document

As Figure 2 on page 17 suggests, the various specification documents always follow the same basic structure. Actual requirements (respectively the *Ranges* containing them) can only be found in the blocks "Main part" and "Appendix". Hence, the rest of the (visual) part of those documents is of little interest to the extraction process. See Figure 7 on page 29 for an example of what a text-heavy page within the "Main part" typically looks like.
However, since list numbering is ubiquitous within those documents and extends across all blocks, the contents before the "Main part" cannot simply be exempted from the extraction process. The iterative computation of the current list number, as it is implemented in the DOC-format (see Section 3.4.1 for a technical explanation), would otherwise fail and thereby falsify the tracestring computation which is based on it (see Section 2.3). Therefore, the goal must be to detect the different blocks, extract as much information as needed from each one of them and eventually store the result in memory for further steps. This includes the preservation of any hierarchy implied by the different levels of the list numbering.
Except for the "Preamble", essentially just a fancy name for the front page and certain structures within the "Main part", to be further discussed in Section 2.3.2, this process turns out to be implementable in a relatively generic way. In other words: The tool needs additional (domain-) knowledge about the layout of the front page and, ideally, about a few more structures, which it cannot obtain from the document itself. As the front page usually contains the chapter number of the specification document and its title, both of which are nice assets to the final output, although neither constitutes a "requirement" in the strictest sense, the tool will refuse to process a file if those contents cannot be found. Hence, the knowledge to correctly parse the front page must be considered mandatory, whereas the presence of detection heuristics for all the other subsequent structures is optional and will only be used to enhance the output quality.
DOC does not contain metadata regarding the history of specific *Ranges* unless the document was continuously edited in "track changes"-mode. So, at first glance it may seem appealing to read out the block "Change history" as well (which technically is just a special kind of a table, see Section 2.3.2) and merge its information into the referenced requirements. Due to the overly broad change descriptions given therein, which usually leave even a human in uncertainty, this does not seem feasible for automated means, though.

2.1.3 The output format: ReqIF

While the choice of the input formats was restricted to those available (i.e. PDF and DOC), the question of an output format initially was a lot more open. A suitable format should provide

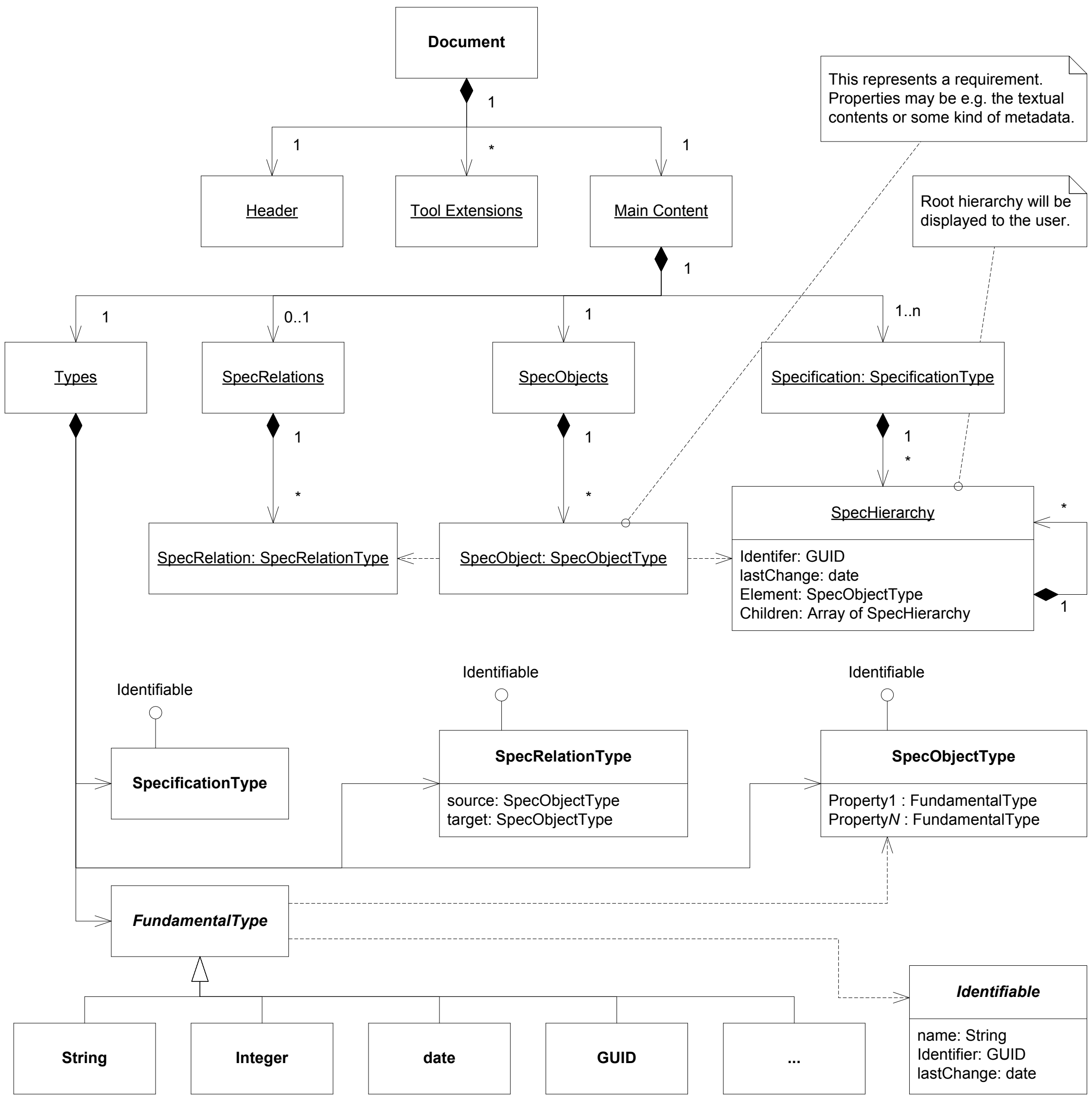


Figure 4: Internal structure of a ReqIF-file *(simplified)*, based on [Obj13, Fig. 10.3]

sufficient means to represent the various different content types of a specification document (namely: text, graphics and equations), keep each requirement separate from its neighbors and ideally be supported as an import source by a wide range of tools possibly used within subsequent steps of the V-Model.
Since practically-minded engineers always lean towards simple solutions, especially when it comes to complex computer-aided work they do not fully understand, **comma-separated values (CSV)** was the first suggestion to pop up. Files of this format are nothing more but plain text with each line constituting an individual entry (requirement). Lines may be separated into different fields by using a special separation character (which originally was a comma [`,`], hence the name – but other characters work as well). So a CSV file is essentially a (not necessarily rectangular) matrix which can store only text. Thus, it fails for graphics and equations and does not offer any specific functionality targeted at requirements management, such as explicit links between requirements, type attribution to its contents or an edit history, either. Above all, CSV implementations are known to have incompatibilities across vendors due to the many subtleties (i.e. how to escape a separation character if it occurs within a field) that may be handled differently [Bur14]. This disqualifies it for any serious use, especially with large specifications in a heterogeneous environment. Nevertheless, thanks to its simplicity the tool will make use of CSV for various accompanying data (see Sections 2.2.1 and 3.1.1).
The next possible candidate is **Microsoft Excel** and its spreadsheet format. It does support embedded media (graphics + equations) and despite its added complexity, incompatible implementations are less likely due to the Excel software as a reference platform. However, for the purpose of requirements management Excel is nothing more than a pimped version of CSV, with the same weaknesses and the additional caveat of being a proprietary, complex file format just like DOC.
A **database**, as used by many RM-tools to internally store their assets, is usually not a file that can be freely passed around. So it does not really qualify for comparison. However, there are two notable exceptions: First, writing the contents of a database into a file. This approach usually leaves the user again with CSV or the Excel file format and is therefore no real advantage. And second, using a dedicated file-based database like SQLite [The15b]. This would actually be a viable solution. However, the binary, query-optimized nature of such files requires a dedicated software library for proper access and thereby adds additional complexity without any substantial win in functionality.

The file format which was eventually chosen is known as **Requirements Interchange Format (ReqIF)**, a relatively new but standardized XML dialect [Obj13]. It was developed mainly by the automobile industry for the purpose of requirements exchange between different suppliers [EJ12]. Its basic structure is depicted in the UML object diagram of Figure 4 on the previous page. The overall design is a lot simpler than DOC by focusing only on features relevant to the domain of requirements management. Requirements are represented as individual entities (called a *SpecObject*) which always come with a unique identifier and may be attributed an arbitrary number of metadata. Like most other entities, both the metadata and the *SpecObject* are typed. Therefore, a specification may contain different kinds of requirements, each with an individual set of different metadata. Due to the way specifications (object: *Specification*) are implemented, they may even contain these requirements several times or not at all. ReqIF files may also comprise more than one specification, which together with so-called *SpecRelation*-

Groups (not shown in Figure 4 on page 21) can be used to model dependencies between e.g. functional and non-functional requirements[7].
A specification in ReqIF is a hierarchical tree, specifically an arborescence, which is a rooted tree with exactly one directed path from the root u to any other vertex v [GM89, Def. 1.1]. In the XML file this is represented by an arbitrarily nested sequence of *SpecHierarchy*-elements. ReqIF is therefore the only format discussed so far that can properly preserve the hierarchical structure of the input documents (see Section 2.1.2). *SpecObjects* are stored separately and only referenced from within this hierarchy. The same applies if two of these *SpecObjects* shall be linked together by means of a *SpecRelation* (which again is typed and in its simplest case represents a cross-reference, see Section 3.2.2).
Because XML is a text-based format, it does not allow for efficient storage of binary contents (BLOBs). Graphics and equations may therefore be referenced from within a *SpecObject* but must[8] be stored externally. However, the resulting ReqIF can be shipped as a ZIP-file bundled with all external objects to both reduce the file size (which is considerably larger than that of an efficient binary format like DOC) and allow for the comfort of a single file [PMS14, Sec. 2.1].
Like DOC, ReqIF files contain a fair amount of logical structure and its contents may therefore be easily edited. The ReqIF standard proposes a special *editable* flag which may be unset for write-protected parts of the file [Obj13, Sec. 10.7]. A second, less pleasant, similarity is the inability to perform cross-file references: Neither can the contents of a DOC file *A* be referenced from a DOC file *B*, nor can an entity in a ReqIF file *C* be referenced from a ReqIF file *D*. The implications of this limitation will be discussed in more detail in Section 3.2.2.

2.2 Enhancing requirement content

An implementation of Section 2.1.2 results in some sort of in-memory tree where each vertex, which shall be called an *artifact* from now on, represents a requirement or otherwise interesting content. As stated in Section 2.1.3, all artifacts must be uniquely identifiable and for obvious reasons they should usually contain some payload (i.e text, graphics, equations or a combination thereof)[9]. Such a tree may therefore be viewed as an associative array which maps identifiers (*key*) onto chunks of the original input document (*value*). Although this may sound like a very basic achievement at first, it should not be underestimated. *Traceability*, which is what this data structure essentially provides, plays a crucial role in any kind of actual implementation of the specified system (ETCS in this example) in a safety-critical context. Only its thorough application to all stages of the V-Model can ensure that an implementation actually matches with the given specification (which is a prerequisite for any verification activities).
If the newly generated ReqIF is regarded the "source" of the system's requirements, this traceability solution – despite its simplicity – already completely fulfils the relevant parts of [CEN11, clause D.58] for this step of the V-Model. Of course, the actual source is not the ReqIF but the DOC files[10]. In order to ensure *backward traceability* [IEE98, Sec. 4.3.8] to them, some value is necessary that can be used to compute the original position where the contents of the current

[7]Subset-026 is (said to be) a purely functional specification, hence these *SpecRelationGroups* are not directly used in the case-study. However, they are discussed in a different context in Section 3.2.2.
[8]Although a base64-encoded file given via an inlined data-URI is also feasible, this comes with a significant overhead in size and will therefore not be considered.
[9]See Section 2.3.3 for an explanation of artificial artifacts which make an exception to this rule.
[10]Which makes another argument against PDF as the input file, see Section 2.1.

artifact originate from. Technically the `startOffset` of the *Range* (see Section 2.1.1) from which the contents had been extracted will be used for this purpose. This yields a number which can be fed back into Microsoft Word to highlight the respective position in the input document. Listing 1 shows how this could be achieved in Visual Basic for Applications (VBA)[11]. A tighter integration directly into a program which consumes the ReqIF files using Office Automation [Mic08b] is also feasible.

```
 1 Option Explicit

 2 Sub findArtifact()
 3   On Error GoTo ErrHandler
 4   Dim wordTraceId As Long
 5   wordTraceId = InputBox("Enter the wordTraceId", "Backward tracing", "")
 6   Dim absoluteTargetOffset As Variant
 7   absoluteTargetOffset = CLng(wordTraceId)
 8   Dim currentStory As Range ' current part of the document which is being examined
 9   Dim absoluteStoryOffset As Long ' startOffset of the currentStory
10   Dim relativeTargetOffset As Long ' targetOffset relative to the startOffset of the
      ↪  current Story
11   absoluteStoryOffset = 0

12   For Each currentStory In ActiveDocument.StoryRanges ' loop over all parts of the
      ↪  document
13     relativeTargetOffset = absoluteTargetOffset - absoluteStoryOffset
14     If (relativeTargetOffset >= currentStory.Start And relativeTargetOffset <=
        ↪  currentStory.End) Then
15       ' currentStory contains the desired artifact
16       ' move the cursor to the beginning of the artifact, scroll to that position and exit
17       Dim targetRange As Range
18       Set targetRange = currentStory
19       targetRange.SetRange Start:=relativeTargetOffset, End:=relativeTargetOffset
20       targetRange.Select
21       ActiveWindow.ScrollIntoView ActiveWindow.Selection.Range, True
22       Exit Sub
23     End If
24     absoluteStoryOffset = absoluteStoryOffset + currentStory.End
25   Next currentStory
26   ' None of the available stories contained the artifact. So the given wordTraceId must be
      ↪  illegal.
27   Err.Raise 513, "Bounds check", "Value out of bounds"

28 ErrHandler:
29   MsgBox "The given wordTraceId is illegal." & vbNewLine & "Reason: " & Err.Description,
      ↪  vbCritical + vbOKOnly, "Error"
30 End Sub
```

Listing 1: Simple VBA procedure to perform backward tracing

Eventually, this offset value makes the first member of a set of metadata which can be assigned to each artifact. A second member will be a human-readable version of the identifier, which is not only unique but fulfils a few additional contracts as well. See Section 2.3 for a discussion of its computation. Other possible metadata could include qualifiers regarding the legal obligation of an artifact, certain status flags (has already been implemented, needs clarification, . . .) or a

[11] In line 19 of Listing 1 the actual `endOffset` of the respective *Range* could have been passed as well. However, this value is not yet processed by the tool.

categorization of the contents (is a Figure / an example / an implementation advice / . . .)[12]. The only constraint on this data is that it must be typed using the *Fundamental Types* of Figure 4 on page 21. For a list of possible types see [Obj13, Sec. 10.6].
Especially for large (legacy) input documents it makes sense to derive as much metadata as possible from the actual contents of the artifact and limit manual assignment to a minimum. If the majority of those contents consists of unconstrained natural language, like with Subset-026, any possible algorithm performing the data extraction will inevitably become domain-specific to some extent. Hence, the discussion of the metadata which can actually be processed by the tool will be postponed until Section 3.2.1.

Not only do individually addressable artifacts allow to store data regarding their hierarchical dependencies, they could also convey information on other relations such as "*A* contradicts *B*", "*A* extends *B*", "*A* is mutually exclusive to *B*" or simply "*A* mentions *B*". See Section 3.3.2 for an application of this concept.

2.2.1 Visualizing dependencies

By combining the artifact's hierarchy with other available relational information, a specification document can also be represented as a graphical tree[13]. However, since the latter group is less constrained, this new tree will no longer be an arborescence but only a generic directed graph (*digraph*). Figure 5 on the following page shows an example (albeit significantly scaled down) rendered by graphviz's dot algorithm [GKNV93] for chapter 3 of Subset-026 in left-to-right-mode (thus: the former root is situated in the very left, centered). This chapter consists of about 3500 artifacts, which are shown as nodes of the tree. Their dependencies are represented by close to 4200 edges in-between.
Different colors and node sizes were subsequently computed by Gephi [BHJ09] on the basis of a clustering algorithm, respectively their betweenness centrality, a measure for how many shortest paths between two nodes pass through the current node. Thus, such a graphical rendering can be of use in the early phases of an implementation project to assign different parts of the specification to different implementers as well as to visualize progress at later stages.

The data used to compute this tree is made available by the tool as CSV-files. Hence, it can also be used to perform other kinds of statistics such as a "graphical delta" between different versions of a file (given the node identifiers remain unchanged, confer with Section 2.3.3) or to form the basis of a taxonomy of related requirements. See Appendix A.1 for an explanation on how to process these files.

2.2.2 Querying for data

Text-heavy specification documents are especially likely to include recurring words, phrases or specific symbols in different artifacts. The tool allows to elicit such entities algorithmically (see

[12]See [Fir05, p. 40] for a long list of further suggestions.
[13]This is somewhat similar to DOORS' "Graphical View" [IBM15] but substantially more powerful.

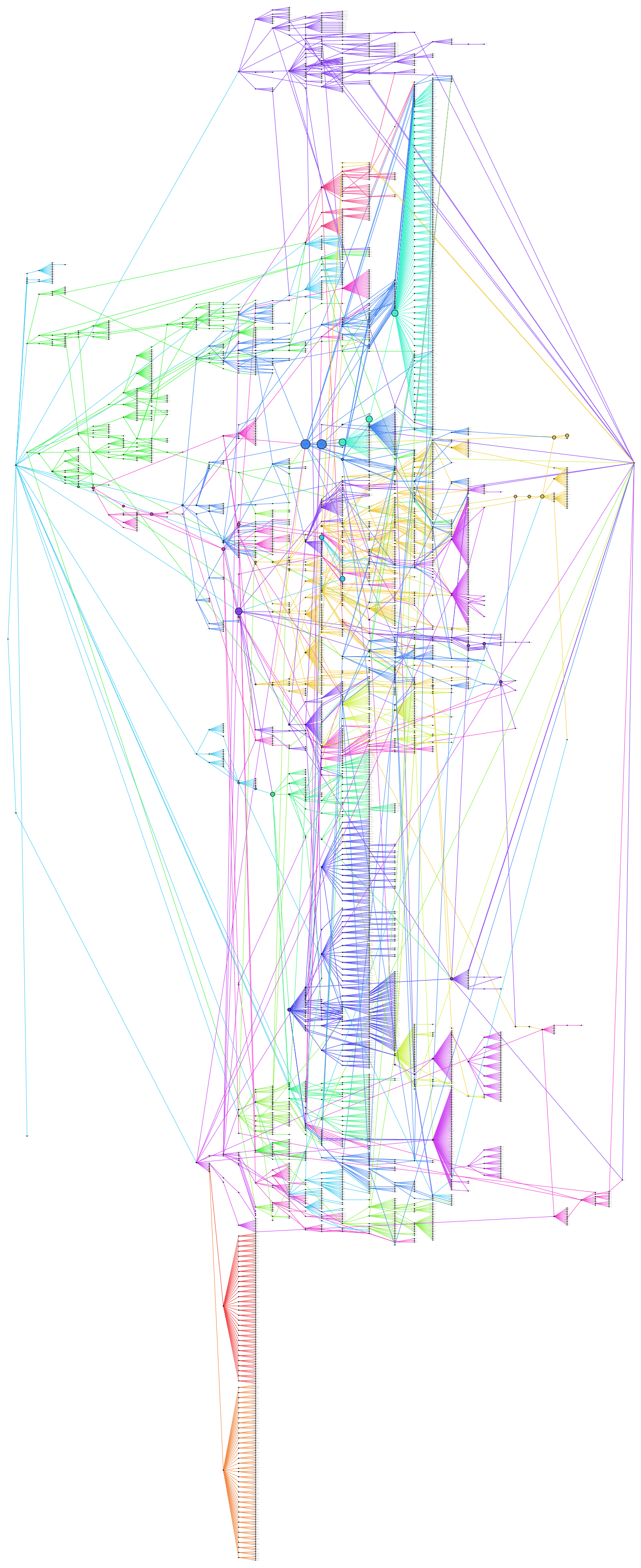


Figure 5: Visual representation of Subset-026, chapter 3

Section 3.2.1) and processes them into a text with lots of colorful “boxes” (see Figure 12 on page 48 for an example) within the resulting ReqIF file. Under the hood this annotated version of the original contents is an embedded Extensible Hypertext Markup Language (XHTML) snippet, a technology commonly used to layout webpages. Each “box” comes with a *class*-attribute matching with the kind of entity which is being annotated. Combining the hierarchical position of the artifact with those attributes makes it possible to formulate queries such as

> “count the number of all entities of kind *A* below artifact *B*”

or

> “return the artifact with the most mentioned entities of kind *C* which is not part of the hierarchy below artifact *D*”.

Likewise, specification documents may contain recurring structures such as tables or lists which always capture data of a similar kind at the same position. Combining their context-aware tracestring (see Section 2.3), which conveys this position, with the actual contents of the respective artifact, enables queries such as

> “sum up all numerical values in column *E* for all tables of kind *F* below artifact *G*”.

This effect carries a high practical potential for chapter 7 of Subset-026 which consists mostly of numerical data structures given as tables.

The technology behind these searches is known as XML Path Language (XPath). Unfortunately, there is currently no RM-tool to support creating queries based on this language through a graphical user interface, and their manual construction is quite cumbersome. See Listing 2 for a trivial query to illustrate.
One major hindrance towards its market adoption is the ReqIF specification [Obj13, 10.8.20], as its current version explicitly forbids the use of *class*-attributes. However, that decision was presumably motivated by ReqIF’s absence of support for the Cascading Style Sheets (CSS) technology they were originally created for. Moreover, this restriction is not enforced by the formal description of the ReqIF file format given as an XML Schema Definition (XSD)[14], which simply references the generic `xhtml.BlkStruct.class` (line 891) for any XHTML content.

```
xmlstarlet sel -N REQIF="http://www.omg.org/spec/ReqIF/20110401/reqif.xsd" -t -v
↪  'count(//REQIF:SPEC-HIERARCHY[@IDENTIFIER="ID_OF_H"]//REQIF:SPEC-HIERARCHY)'
↪  file.reqif
```

Listing 2: XPath query to sum up all artifacts below artifact *H* in `file.reqif` using xmlstarlet [XML14]

A second problem arises when certain annotation patterns cannot be conveyed through XHTML. This technology is based on writing some marker, a so called *tag*, at the beginning and end of

[14]The file assigned to `REQIF` in Listing 2.

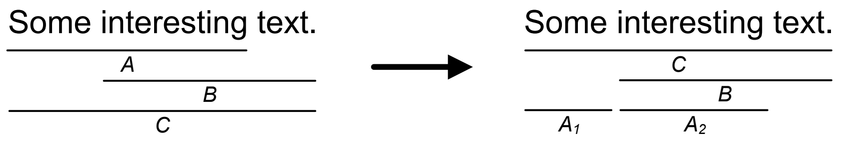


Figure 6: Example of nested annotation patterns (left) which need to be flattened (right) for proper XHTML output

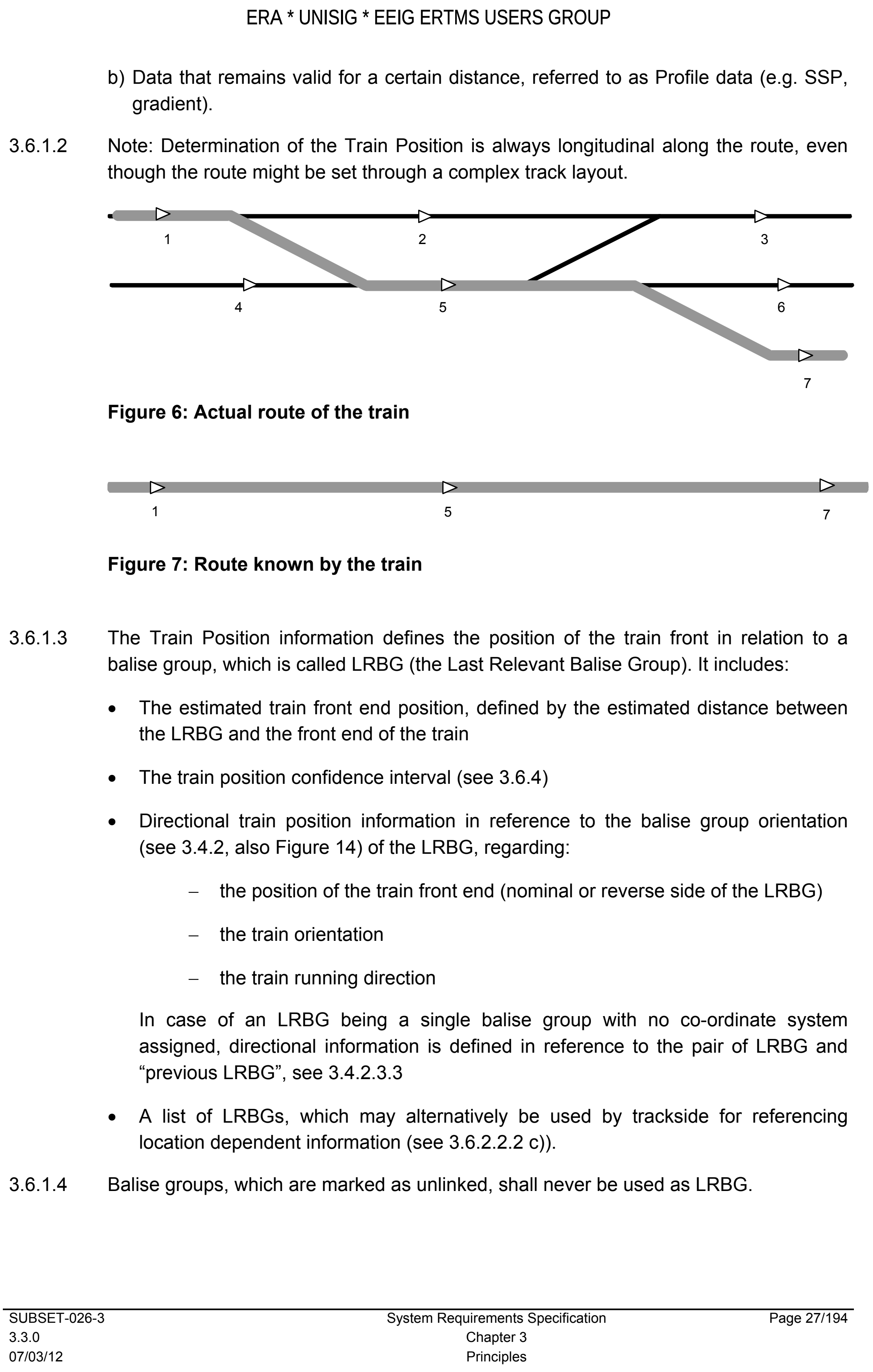
each annotation. By definition the applicability of those *tags* must not overlap, otherwise the resulting snippet is not *well-formed* and therefore illegal. To illustrate this, consider the left part of Figure 6. If the *tags* of the three annotations (*A* through *C*) shown here were written from left to right, the snippet would inevitably become illegal. The tool solves this dilemma by splitting up nested annotations as shown in the right part of the Figure. However, this turns *A* into two annotations (thus: two *class*-attributes) and will therefore alter the result of any queries performing counts.

2.3 Computing requirement identifiers

One fundamental requirement towards a specification document of any system of sufficient complexity is the ability to break down its contents into smaller artifacts of a certain (definable) *granularity*. Generally speaking, artifacts are the building blocks of a specification and may represent anything from a single character of text up to the entire specification itself. Each artifact must be uniquely addressable and may optionally come with a plethora of metadata (see Section 2.2).
While wide-ranging metadata are usually a nice-to-have feature of any multitenant RM-tool (and often the primary reason for its existence), the presence of an identifier is a lot more vital as it builds the foundation for any sort of traceability and thus for a certifiable implementation of the system.

In practise, there are different approaches for generating such an identifier. It may be completely arbitrary (e.g. a hash over the contents of the artifact or its creation time)[15], based on visual properties of the printed specification (e.g. the line- and/or the page number; confer with the layout of [CEN11]), or some sort of running number being defined in the specification itself [GF94, Sec. 3.1]. Each of these approaches comes with its individual downsides, but since the Subset-026 already comprises *many* artifacts with a running number attached, and numerous stakeholders intuitively have been using them for reference purposes since the beginning of time, it seems only natural to *base* any human-readable, unique identifier on this running number.
Coming back to the term *granularity* coined in the beginning of this Section, those existing numbers, however, are neither sufficient to break down the specification into (mostly) atomic, single purpose, artifacts, nor are they guaranteed to be unique. To illustrate this consider the examples in Figures 7 to 8 on pages 29–30. The running numbers turn out to be items, so-called numberTexts, of hierarchical lists. Some are qualified (they include all more significant list levels and can therefore be considered unique), some are not (e.g. bullet points, which are obviously not

[15]In fact, this would be the outcome if only Section 2.2 was to be implemented.



ERA * UNISIG * EEIG ERTMS USERS GROUP

b) Data that remains valid for a certain distance, referred to as Profile data (e.g. SSP, gradient).

3.6.1.2 Note: Determination of the Train Position is always longitudinal along the route, even though the route might be set through a complex track layout.

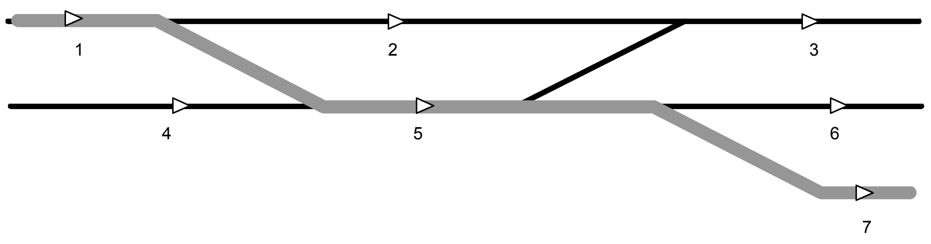


Figure 6: Actual route of the train

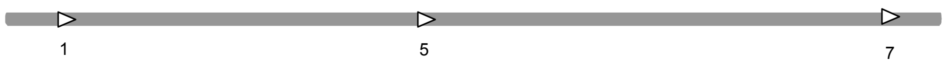


Figure 7: Route known by the train

3.6.1.3 The Train Position information defines the position of the train front in relation to a balise group, which is called LRBG (the Last Relevant Balise Group). It includes:

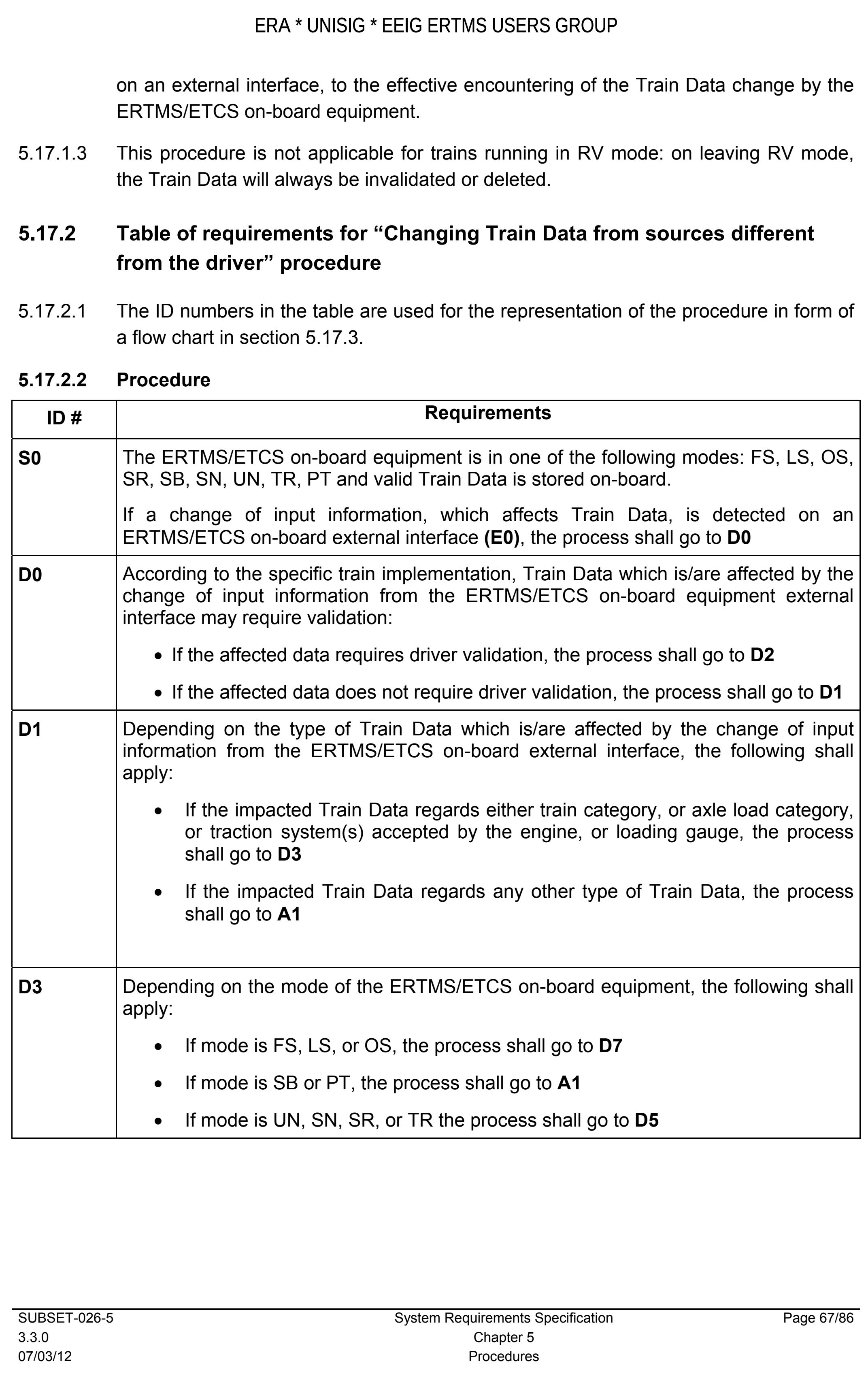
- The estimated train front end position, defined by the estimated distance between the LRBG and the front end of the train
- The train position confidence interval (see 3.6.4)
- Directional train position information in reference to the balise group orientation (see 3.4.2, also Figure 14) of the LRBG, regarding:
  - the position of the train front end (nominal or reverse side of the LRBG)
  - the train orientation
  - the train running direction

  In case of an LRBG being a single balise group with no co-ordinate system assigned, directional information is defined in reference to the pair of LRBG and “previous LRBG”, see 3.4.2.3.3
- A list of LRBGs, which may alternatively be used by trackside for referencing location dependent information (see 3.6.2.2.2 c)).

3.6.1.4 Balise groups, which are marked as unlinked, shall never be used as LRBG.

SUBSET-026-3	System Requirements Specification	Page 27/194
3.3.0	Chapter 3	
07/03/12	Principles	

Figure 7: Example from chapter 3: Paragraphs with and without running numbers



ERA * UNISIG * EEIG ERTMS USERS GROUP

on an external interface, to the effective encountering of the Train Data change by the ERTMS/ETCS on-board equipment.

5.17.1.3 This procedure is not applicable for trains running in RV mode: on leaving RV mode, the Train Data will always be invalidated or deleted.

5.17.2 Table of requirements for “Changing Train Data from sources different from the driver” procedure

5.17.2.1 The ID numbers in the table are used for the representation of the procedure in form of a flow chart in section 5.17.3.

5.17.2.2 Procedure

ID #	Requirements
S0	The ERTMS/ETCS on-board equipment is in one of the following modes: FS, LS, OS, SR, SB, SN, UN, TR, PT and valid Train Data is stored on-board. If a change of input information, which affects Train Data, is detected on an ERTMS/ETCS on-board external interface **(E0)**, the process shall go to **D0**
D0	According to the specific train implementation, Train Data which is/are affected by the change of input information from the ERTMS/ETCS on-board equipment external interface may require validation: • If the affected data requires driver validation, the process shall go to **D2** • If the affected data does not require driver validation, the process shall go to **D1**
D1	Depending on the type of Train Data which is/are affected by the change of input information from the ERTMS/ETCS on-board external interface, the following shall apply: • If the impacted Train Data regards either train category, or axle load category, or traction system(s) accepted by the engine, or loading gauge, the process shall go to **D3** • If the impacted Train Data regards any other type of Train Data, the process shall go to **A1**
D3	Depending on the mode of the ERTMS/ETCS on-board equipment, the following shall apply: • If mode is FS, LS, or OS, the process shall go to **D7** • If mode is SB or PT, the process shall go to **A1** • If mode is UN, SN, SR, or TR the process shall go to **D5**

SUBSET-026-5 System Requirements Specification Page 67/86
3.3.0 Chapter 5
07/03/12 Procedures

Figure 8: Example from chapter 5: Bullet points within a table

unique on their own)[16], and a few others belong to neither of those categories as they contain numerical references to some but not all of their ancestors (see the end of this Section).

If *granularity* is defined in such a way that it shall be possible to individually address each paragraph ($\in$ artifact) of text in the specification document (that is: each consecutive, non-empty array of printable characters, of which at least one must be different from a whitespace, terminated by a newline + carriage-return, no matter where it occurs), it becomes obvious that this is not possible with just the existing running numbers. So the goal is to develop an addressing scheme which maps as many artifacts to existing numbers as possible (thus ensuring "backwards compatibility" with any previously produced works) and allows for finer *granularity* while maintaining uniqueness of the generated addresses.
Such an address shall subsequently be named *tracestring* and is defined as follows:

Tracestring

A document-wide unique, human-readable identifier which is attributed to any traceable artifact and based on the running number of the last preceding (including itself) numbered paragraph.

For a concrete example suppose the paragraph underneath 3.6.1.3 in Figure 7 on page 29 starting with "In case of an LRBG..." shall have a tracestring attributed. A basic version of the algorithm to determine this string may look like this:

1. Let `traceString` be an empty string.
2. Let `paragraph` be the current paragraph.
3. As long as `paragraph` is not a member of the baseList do the following:
   (for our example the closest member of the baseList is the paragraph prefixed by "3.6.1.3")
   (a) Let `paragraphCounter` be 1.
   (b) As long as `paragraph` is not a member of a list do the following:
      i. Increment `paragraphCounter` by 1.
      ii. Set `paragraph` to the closest preceding paragraph of equal significance or break if there is no such paragraph.
   (c) If `paragraphCounter` is greater than 1:
      wrap it in square brackets and prefix the `traceString` with it *(yields "[2]")*
   (d) Let `bulletCounter` be 0.
   (e) As long as `paragraph` is a member of a bulleted list do the following:
      i. Increment `bulletCounter` by 1.
      ii. Set `paragraph` to the next preceding paragraph belonging to the same list.

[16]Technically a bullet point and a numbered list item are the same thing, apart from applying different formatting. Hence, they shall be regarded as such, despite the common usage where a bullet point is not exactly a running number.

(f) If `bulletCounter` is greater than 0:
prefix `traceString` with `bulletCounter` in brackets preceded by "`*`".
(yields "[3][2]")*

(g) If `paragraph` is a member of any non-bulleted list:
remove any unnecessary characters from the current list value (e.g. trailing braces) and prefix `traceString` with this value

(h) Prefix `traceString` with a dot. *(yields ".*[3][2]")*

(i) Set `paragraph` to the closest preceding paragraph of higher significance

4. prefix `traceString` with the current value of the baseList *(yields "3.6.1.3.*[3][2]")*

5. `traceString` is now the fully qualified identifier.

Although this algorithm is greatly simplified (a more technical discussion of the actual implementation is given in Section 3.4.1), it shows the basic concept of how the individual parts of a tracestring are constructed: There is a base, which covers all the way from the chapter number to the last level of the last preceding numbered paragraph, some annex to describe non-qualified sublists, and eventually a counter to unambiguously reference a certain unnumbered paragraph underneath the last list item.

The term *significance* plays an important role in this process as the running numbers are of hierarchical nature. To maintain this hierarchy not only throughout the baseList (where it is obvious due to the dots separating different list levels) but also for any non-qualified paragraphs, their individual relationships have to be computed. This is mostly done on the basis of their relative left indentation (with a few exceptions for non-indented and equally indented paragraphs; confer with Listing 6 on page 68).
In the example above there are three such hierarchical levels: The most significant one to which the paragraph "3.6.1.3" belongs, a second intermediate one formed by the list with the ordinary (round) bullets plus the example paragraph, and a third one to which the dashed list belongs. Since the dashed list is less significant than our example paragraph, it remains invisible to the above algorithm.
Dots (`.`) are universally used as separators for hierarchical levels regardless of their presence in the original input document. This is different from how the original specification authors reference these paragraphs, see Section 3.2.2.
Single alphabetic characters in square brackets following a dot generally represent some sort of typing information which applies to the current level (e.g. `.[t]21` states that this level represents "table number 21" and not a "list level 21")[17]. Alphanumeric strings in square brackets at arbitrary positions (this includes the previous case) indicate this information was added (and thus was not visually present in the input file; i.e `.3[2]` to describe the second unnumbered paragraph underneath some artifact called `.3`).
All hierarchical levels of a tracestring must exist (i.e. the string may be cropped just before any dot and will always point to an existing artifact). Hence, it is sometimes necessary to introduce placeholders at non-existing levels. So any hierarchical structure like the one printed below on

[17]A full list of the currently implemented types can be found in `helper.Constants.Traceability` of the tool's source. The most important ones are mentioned in the two following Sections 2.3.1 and 2.3.2.

the left will be turned into the one printed on the right. If this contract does not hold, uniqueness of the generated tracestrings cannot be guaranteed under all circumstances.

`1.` Some text
`1.1.1` Some more text

⇒

`1.` Some text
`1.1` *Placeholder*
`1.1.1` Some more text

2.3.1 Unwinding complex structures: Tables

Using the algorithm above, a table, like the one shown in Figure 8 on page 30, can only be processed as a monolithic artifact by considering it as one (potentially very big) paragraph. If individual cells are to be identifiable as well, a tracing methodology will have to be invented that can properly disassemble a table and generate table-wide unique identifiers on this basis. As a result, these identifiers may become document-wide unique by prefixing them with the identifier of the closest previous paragraph, as stated earlier.
Commercial RM-tools usually either have no specific support for table handling (leaving the user with a monolithic paragraph), or use a generic disassembly approach (e.g. "DOORS Tables" [PMS14, Sec. 2.4]), irrespective of the structure of the concrete table.
Since the Subset-026 contains quite a few tables for all sorts of purposes ranging from simple layout grids to complex matrix-like structures, it was decided to craft a novel algorithm that can adapt to these manifold inputs. Its goals were:

1. to handle arbitrarily shaped merged cells
2. to allow for intra-cell requirements (see Section 3.3.3)
3. to process as few cells as possible to minimize the number of artifacts not representing any actual requirement
4. to create meaningful tracestrings based on the current row, column and/or cell contents where possible

All of the aforementioned points were motivated by the specific structure of the specification documents. The first two items are simply stringent conditions necessary to process the tables without informational loss. The latter two focus on enhancing the user experience as fewer artifacts mean less work (no need to justify why some empty cell has not been implemented in any downstream activities), and meaningful tracestrings can greatly improve the overall usability of the resulting artifacts especially within lengthy tables.

The vast majority of tables within the Subset-026 follow a "row-first, column-second"-approach (that is: each row represents some kind of coherent information). Hence, it was decided to generate tracestrings on this basis. Each table always consists of exactly one table-artifact, possibly a caption and a number of row- and cell-artifacts. Their simple hierarchy is depicted in Figure 9 on the next page. Depending on the structure of the concrete table, the number of cell-artifacts may vary across rows (merged cells and/or user configuration). Row-artifacts that do not contain any cells are omitted.
A slightly abridged version of the algorithm to create the tracestrings for the respective artifacts (= entities of Figure 9 on the following page) looks like this:

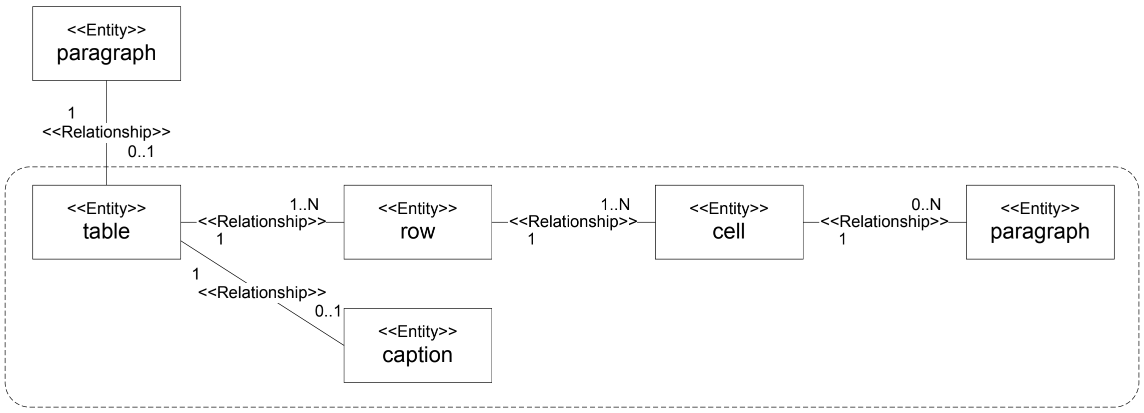


Figure 9: Relations of the different artifacts of a table hierarchy

1. Let `traceString` be the tracestring of the current paragraph
2. Upon the detection of a table find the next subsequent non-table paragraph and check if it matches the heuristic of a caption (see Section 3.3.1).
3. If so:
   (a) Let `extractedNumber` be the extracted number of the table from the caption.
   (b) Suffix `traceString` with ".[t]`extractedNumber`"
   (c) Set the tracestring of the caption-artifact to `traceString` suffixed by ".C".
4. If not: Suffix `traceString` with ".[t]*"
5. Let `rowCounter` be 0
6. For each row:
   (a) increment `rowCounter`
   (b) If this row is not of importance, continue with the next iteration of this loop
   (c) Copy `traceString` to `traceStringRow`
   (d) Suffix `traceStringRow` with ".[r][`rowCounter`]" or a user-defined value and let this be the tracestring of the row-artifact
   (e) Let `columnCounter` be 0
   (f) For each column within this row:
      i. increment `columnCounter`
      ii. If this column is not of importance, continue with the next iteration of this loop
      iii. Copy `traceStringRow` to `traceStringColumn`
      iv. Suffix `traceStringColumn` with ".[c][`columnCounter`]" or a user-defined value and let this be the tracestring of the cell-artifact
7. skip any caption (since it has already been processed above)

By applying this algorithm to Figure 8 on page 30, its table may now possess a generic hierarchy like that of Table 1 on the facing page.

`5.17.2.2`	Procedure
`5.17.2.2[2]`	
`5.17.2.2[2].t[*]`	*Entire table, formatted*
`5.17.2.2[2].[t]*.[r][1]`	
`5.17.2.2[2].[t]*.[r][1].[c][1]`	ID#
`5.17.2.2[2].[t]*.[r][1].[c][2]`	Requirements
`5.17.2.2[2].[t]*.[r][2]`	
`5.17.2.2[2].[t]*.[r][2].[c][1]`	S0
`5.17.2.2[2].[t]*.[r][2].[c][2]`	The ERTMS/ETCS . . .
	If a change . . .
`5.17.2.2[2].[t]*.[r][3]`	
`5.17.2.2[2].[t]*.[r][3].[c][1]`	D0
`5.17.2.2[2].[t]*.[r][3].[c][2]`	According to . . .
	If the affected data requires . . .
	If the affected data does . . .
⋮	⋮

Table 1: Generic tracestring attribution for the table in Figure 8 on page 30

The downsides of this simple approach, which the tool in fact only uses as a fallback-solution, are fairly obvious: Too much detail in the beginning (no one really cares about a specific tracestring for the header of a table) and too little detail for the actual payload (all paragraphs of a cell are merged and assigned a single tracestring).

A much improved version, which is actually used in the ReqIF output, can be seen in Table 2. In here headers are omitted (by using a special flag in the Word file), irrelevant cells are removed (in this example: all cells belonging to the first column), the contents of individual cells are split up and the individual artifacts have meaningful (i.e. user-defined) tracestrings attributed.
This context-aware tracestring generation is implemented through an abstract table definition consisting of invariants (contents and formatting of specific cells) against which this concrete table is matched. Currently, the tool knows of 29 such definitions which cover all major recurring table-structures in the Subset-026, leaving only a very few (usually very short) tables exposed to the fallback version explained earlier.
For the example table of Figure 8 on page 30 the respective matcher definition is given in lines 2–6 of Listing 3 on page 37. Each of these lines states an expectation which must be met in order to qualify the concrete table as a "match". Expectations are cell specific (the first two parameters of `addData()` method are row- respectively column-numbers[18]) and define visual properties of the contents of the entire cell. Namely those are (given in the order of appearance in the static constructor `MatchingData.newMatchingData()`):

1. The formatting of all textual contents of this cell
   allowed values: ***`NORMAL, BOLD, INCONSISTENT`***[19]
2. The horizontal alignment of the content within this cell
   allowed values: ***`LEFT, CENTER, RIGHT, LEFTORJUSTIFY, INCONSISTENT`***
3. A regular expression describing the content of this cell[20]

[18]Contrary to the tracestrings in Table 1 these numbers are 0-based.
[19]Italic formatting is not commonly used for tables in the Subset-026.
[20]Note the Java-specific escapes. E.g. line 5 actually reads "`\S.+`" and therefore matches any non-space character followed by at least one (arbitrary) character.

`5.17.2.2`	Procedure
`5.17.2.2[2]`	
`5.17.2.2[2].t[*]`	*Entire table, formatted*
`5.17.2.2[2].[t]*.[I]S0`	
`5.17.2.2[2].[t]*.[I]S0.Content`	*Split up indicator*
`5.17.2.2[2].[t]*.[I]S0.Content.[1]`	The ERTMS/ETCS . . .
`5.17.2.2[2].[t]*.[I]S0.Content.[2]`	If a change . . .
`5.17.2.2[2].[t]*.[I]D0`	
`5.17.2.2[2].[t]*.[I]D0.Content`	*Split up indicator*
`5.17.2.2[2].[t]*.[I]D0.Content.[1]`	According to . . .
`5.17.2.2[2].[t]*.[I]D0.Content.[1].*[1]`	If the affected data requires . . .
`5.17.2.2[2].[t]*.[I]D0.Content.[1].*[2]`	If the affected data does . . .
⋮	⋮

Table 2: Improved context-aware tracestring attribution for the table in Figure 8 on page 30

Eventually, line 6 will cause all matching data applicable to row 1 (thus: lines 4 and 5) to apply to all remaining rows of the concrete table as well.

If the concrete table turns out to be a "match", lines 7 and 8 come into play. In here the actual tracing methodology is defined (line 7) and then copied onto all remaining rows, just as before (line 8). Unlike `MatchingData`, `TracingData` defines a whole range of different (static) constructors for various use-cases. `TracingData.newTracingDataRowIdFromCell()`, which is used here, generates a tracestring well suited for simple lists of conditions like those of the example table. For the actual assembly of the tracestring (line 7) a relative source cell one column left of the current cell (first parameter) is defined from which some content (regex group in parameter 2) is extracted, then prefixed by a given `RowLevelPrefix` (parameter 3) and ultimately suffixed by some static string (parameter 4). What makes this constructor special is that everything except parameter 4 will be inferred by the parental row-artifact (which would otherwise remain inaccessible). The result of this rather cryptic statement can be seen in all artifacts of Table 2 which are children of the table-artifact (lines 4 ff.).
The matcher code of Listing 3 on the facing page (targeting a so-called *Procedures2ColumnTable*) as well as extensive Javadoc documentation for all the other tracestring generation techniques implemented by `TracingData` can be found in the package `helper.subset26.tables` of the tool's source.

For performance reasons the entire matching process runs in parallel for all available abstract table definitions. Hence, the user must make sure to avoid setting up several definitions that can potentially match the same concrete table. Otherwise, a race-condition is provoked which ultimately leads to non-deterministic behaviour.

What can also be specified on a per-cell basis for a matching table (but has been left with the global default in Listing 3) is the handling of complex structures within table cells. The DOC file format not only allows nested tables but just about any conceivable structure within a table cell (see Figure 3 on page 19). Thus, the algorithm needs to be recursive if the user decides to break up such a structure in the same way as if it was encountered outside a table. The result of this process can be seen with any children of the various `Content` cell-artifacts in Table 2.

```java
final String idRegex = "[A-Z][0-9]+";

addData(0, 0, MatchingData.newMatchingData(ContentFormatting.BOLD,
 ↪ ContentAlignment.CENTER, "ID #"));
addData(0, 1, MatchingData.newMatchingData(ContentFormatting.BOLD,
 ↪ ContentAlignment.CENTER, "Requirements"));
addData(1, 0, MatchingData.newMatchingData(ContentFormatting.BOLD,
 ↪ ContentAlignment.LEFTORJUSTIFY, idRegex));
addData(1, 1, MatchingData.newMatchingData(ContentFormatting.INCONSISTENT,
 ↪ ContentAlignment.LEFTORJUSTIFY, "\\S.+"));
setRepeatingRowMatchingData(1);

addData(1, 1, TracingData.newTracingDataRowIdFromCell(-1, '(' + idRegex + ')',
 ↪ RowLevelPrefix.ID, "Content", false));
setRepeatingRowTracingData(1);
```

Listing 3: Java source to set up the tracestrings shown in Table 2 on the facing page

2.3.2 Unwinding complex structures: Other structures

Besides tables the Subset-026 comprises a number of other, less complex structures which also require special handling. Namely:

Images and Equations Those are by definition inline elements, meaning they can only occur within a line of a paragraph. Currently, those artifacts are attributed a tracestring which equals that of its parent (the containing paragraph) suffixed by either `.I` (Images) or `.E` (Equations) and an optional running number in square brackets if there is more than one such artifact within a paragraph[21]. However, this tracestring is not displayed to the user but only used as the basis of the filename under which the respective artifact will be saved to disk.
Both Equations and Images are always flattened to a bitmap image in PNG-format irrespective of their original representation within the input document and can therefore only be processed as a whole (which implies that `.I` and `.E` always constitute an endpoint of a tracestring, confer with Listing 4 on page 39). However, in most of all cases Microsoft Word also saves the original data of the source application that was used to create the respective artifact along with / instead of a bitmap representation in the DOC-file. Hence, for equations embedded as OLE-data, MTEF-BLOBs are available [Des99]. As for Images, which can be anything from Office's own internal Shape-format [Mic14b] to embedded Visio- or even Word-files (think: recursion), their respective original source-file can be extracted from the internal FAT-filesystem of the DOC-file. Given infinite resources to implement each single file format present in the DOC, it would thus be possible to break up those structures as well and trace into them. See also Section 3.1.1.

Foot– and Endnotes Those artifacts are attributed a tracestring which equals that of its parent (the containing paragraph) suffixed by either `.[N]` (Footnotes) or `.[n]` (Endnotes) and a document-wide, type-specific running number (1-based) in square brackets.

[21] See artifacts 3.13.9.3.5.5[3] and 3.13.9.3.5.6[3] in Figure 23 on page 75 for an example of this.

Both note types can contain arbitrarily complex structures (confer with Figure 3 on page 19). Hence, they are internally processed very much like table cells (see above). In terms of hierarchy, the note will become a child of the containing paragraph.

Figures Technically a figure is an image on a separate line of text followed by a caption (which, unlike that of a table, is mandatory). So the processing is essentially a combination of that of a table (heuristic caption detection) and that of the image extraction outlined above.
A figure is attributed a tracestring which equals that of its parent (the containing paragraph) suffixed by `.[f]` followed by its running number (extracted from the caption). The caption itself inherits this string suffixed by `.C`, just like for a table.

Table of Contents Each chapter of the Subset-026 begins with a Table of Contents. The tool contains a special (non domain-specific) detector for such lists and deliberately skips them, since the information therein is redundant and the given page numbers are not of interest[22].
The version history that precedes the table of contents is skipped likewise. However, it comes embedded in a table and thus needs no special handling other than a dedicated table-matcher that does not attribute any tracestrings. See Section 2.1.2.

Headers, Footers and Textboxes Although the Subset-026 contains such elements they are currently not processed due to the challenges involved in anchoring them in the tracestring hierarchy in a meaningful way, and the lack of proper extraction methods in Apache POI (see Section 3.4). However, upon detection of any of these elements a warning is displayed to the user.

2.3.3 Summary

Unfortunately, the computation of a tracestring is not a trivial task, especially given the constraint to stay backwards compatible to previous, less sophisticated attempts to reference artifacts within the specification documents.
Although the presented methodology introduces a few redundancies (namely for tables, figures and foot-/endnotes which are already unique by their respective running number, but are nevertheless prefixed by the tracestring of the last processed paragraph in order to properly anchor them in the tracestring hierarchy) and sometimes requires artificial elements to be introduced (placeholders for skipped levels, split-up indicators), it is certainly a workable solution to ensure traceability not only for the Subset-026 but for a wide range of other hierarchically structured requirements documents as well. They can all benefit from improved *granularity* over that offered by any previously existing identifiers.
Listing 4 on the next page shows a regular expression to capture a tracestring with all its bells and whistles. Except for the `globalprefix` (line 2), simply a static text each tracestring may be prefixed with, the meaning of each named capturing group has been discussed in the Sections above. It can be easily spotted that the possible recursion inside table cells and foot-/endnotes

[22]Neither for the tracestrings nor in any other part of the processing does the page number ever play a role, which is why the tool does not compute these numbers in the first place (see Section 2.1.1). Due to missing fonts, different output resolutions, page sizes etc. page numbers in DOC files in fact make a poor choice for reference purposes (confer with Section 2.1).

```
(?x)^
(?<globalprefix>[a-z][^\.]*\.)?
(?<chapternumber>[AR1-8])
(?<nestingLevel>
  (?<levelnumber>
    (?:
      \.
      (?:
        \d+ # ordinary number
        |[a-z] # lettered sublist
        |\*\[[1-9]\d*\] # bullet point
      )
      (?:
        \[
        (?:[2-9]|[1-9]\d+)*
        \]
      )? # paragraph counter
    )*
  )
  (?<floatingobject>
    \.
    (?:
      \[f\][1-9]\d*[a-z]? # figure
      | \[t\](?:[1-9]\d*[a-z]?|\*) # table
    )
    (?<row>
      \.
      (?:\[r\]\[[1-9]\d*\]|\[[A-Z]\]\w+)
    )?
    (?<column>
      \.
      (?:C|\[c\]\[[1-9]\d*\]|\w+) # caption or column
      (?<nestingLevelColumn>
        (?:\.\[[1-9]\d*\])? # non-prefixed paragraph count
        (?&nestingLevel)
      )?
    )?
  )?
  (?<paragraphelement>
    \.
    (?:
      [IE] # inline image or equation
      (?:\[[1-9]\d*\])?|
      \[[nN]\][1-9]\d* # footnote or endnote
      (?<nestingLevelNote>
        (?:\.\[[1-9]\d*\])? # non-prefixed paragraph count
        (?&nestingLevel)
      )?
    )
  )?
)
$
```

Listing 4: PCRE-compliant regex to match a tracestring

(lines 35 & 47) which, together with the user-definable row- and column-identifiers (see Section 2.3.1), renders the underlying grammar of the tracestring not context-free (formally speaking: it does not ensure terminals to be always disjoint from variables, see [Sip06, p. 104, Def. 2.2]). Also, this expression clearly defines all legal sequences of tracestring elements. For example, a column-identifier only makes sense inside a table. Hence, lines 30–37 can only trigger if lines 22–29 are active as well.

Because the running numbers of paragraphs (lines 8–12) are mostly implemented as numberTexts of list items and are therefore automatically numbered by Microsoft Word, the tracestring of this paragraph (as well as that of its children and possible successors) can only be regarded as document-wide unique for a given revision (read: *Baseline* in ETCS' parlance) of this document. Adding and removing list items (as well as certain kinds of formatting) anywhere ahead of the current paragraph results in a shift of the numberText of the list item surrounding that paragraph. So a hierarchy like the one printed below on the left may turn into the one on the right by simply adding a new element after the first item.

`1.`	1st item		`1.`	1st item
			`2.`	new 2nd item
`2.`	2nd item	⇒	`3.`	old 2nd item
`2.1`	3rd item		`3.1`	old 3rd item
`2.1.1`	4th item		`3.1.1`	old 4th item

This behaviour makes tracestrings unsuitable for any cross-document comparisons and forbids any attempts to calculate a delta between two revisions by simply comparing each artifact with the same tracestring attributed. This is a fundamental flaw (fixing it would imply a totally different tracestring methodology and thus break backwards compatibility) and the requirement authors are suffering from it as well [Sma12].
However, what will remain unchanged across different document revisions is the *granularity* used by the tracestring attribution algorithms to elicit artifacts within the document. Matching the actual contents of the individual traceable artifacts of two revisions is therefore a legitimate, albeit computationally expensive, way of computing a delta that is still far superior to any DOC- or PDF-based comparison algorithms, as the latter do not know anything about the document's structure.

As a final note, it should be clear that the tracestring itself is actually nothing more than a nice-looking, unique ID for a human user and will never be employed for internal referencing by any algorithm. Instead, the tracestring only forms the basis to calculate a Globally Unique Identifier (GUID). This is exactly what ReqIF (and XML-based formats in general) effectively uses to uniquely identify an element. GUIDs must be `xsd:ID`-compliant [W3C04, clause 3.3.8], which would greatly reduce the readability of the tracestring.

3 THE TOOL

deleted epigraph – Alfred V. Aho on AWK, [War09, p. 103]

The following Sections will offer a more detailed description of the "tool" which was created for this thesis and, unfortunately, still comes without a proper name. In a nutshell, this tool is a self-contained piece of software which processes a DOC file into ReqIF and meanwhile enhances its contents. Written in the Java programming language, the tool currently consists of about 16 000 source lines of code (SLOC), 3 000 of which are unit and integration tests.
The novel idea that drove its development is the positioning at the interface between the author and the implementer of a system (confer with Figure 1 on page 16). So this software neither constitutes a conventional quality checker for use during the writing of requirements specifications, nor a full-fledged RM solution (Section 3.5). Instead, it enables a seamless transition from a weakly structured input file to a data format of a substantially higher formalization degree (Section 1.1). This is necessary not only to facilitate the later administration of the individual requirements with an RM application and to allow for proper traceability. But also to increase the comprehensibility, and thus quality, of those contents both for a human as well as for automated means (Section 3.2).

The tool is completely open-sourced, including the algorithms to read DOC files (Section 3.4). Its sourcecode can be obtained from [Dor15]. All it requires is a recent Java runtime: At least Java SE 7 without Natural Language Processing (NLP), respectively Java SE 8 with NLP (see Section 3.4.2). There are no external dependencies other than to a library for low-level DOC processing and (optionally) to a second one for NLP.

3.1 Basic usage

Currently, the tool features only a simplistic command-line interface. Hence, it is a very straightforward action to process a given specification document but there is no way to tweak any settings of the conversion (see Section 3.6). A typical call on a command line (OS-agnostic) looks like this:

```
java -jar tool.jar Subset026 input.doc output.reqif
```

The first parameter (in the example: `Subset026`) is a text used as a prefix for the filenames of resulting media artifacts (images, equations, . . .). The other two parameters should be self-explanatory. After stout-heartedly pressing *<ENTER>*, the tool will start operating and fill both Standard Out and Standard Error with some meaningful messages.
Listing 5 on the next page shows an excerpt of the combined output for a run with chapter 3 of Subset-026, which can be decomposed into the following different phases:

```
------------------------------------------------------------------------------
Subset26 reader Version 0.6

INPUT
Input filename      : /tmp/chap3.doc
Document title      : Title to be introduced in the properties
Document subject    : issue to be introduced in the properties
Lines omitted
Creation tool       : Microsoft Office Word
Operating System    : Windows 7 or Windows Server 2008 R2
Lines omitted

OUTPUT
Output filename     : /tmp/out.reqif
Media output dir    : media
Prefix              : Subset026-
------------------------------------------------------------------------------
Mai 08, 2015 1:13:36 PM docreader.ReaderData checkDocumentAssumptions
INFORMATION: This document contains textboxes. Will skip them.
  21/4220                          3.1: Modification History
  22/4220                       3.1[2]:
 197/4220                          3.2: Table of Contents
 317/4220                          3.3: Introduction
 318/4220                        3.3.1: Scope and purpose
 319/4220                      3.3.1.1: The chapter 3, Principles, specifies th
Lines omitted
 354/4220                  3.4.2.3.3.4: If a new single balise group (BG2), dif
Mai 08, 2015 1:13:45 PM
 ↪ docreader.range.paragraph.characterRun.FieldReader$FieldHandler$7 process
INFORMATION: Got a shape with an OfficeDrawing here. Will try obtain an image
 ↪ representation.
 355/4220               3.4.2.3.3.4[2]:
Lines omitted
3313/4220                     3.20.1.9: Intentionally deleted.
3314/4220                          A.3: Appendix to Chapter 3
3315/4220                        A.3.1: List of Fixed Value Data
Lines omitted
4219/4220                     A.3.10.6: Note: If feedback has started but T_bs1
Performing second pass of generated document hierarchy.
NLP is active. Processing may take a while...
 2162 NLP-jobs remaining.
Lines omitted
    1 NLP-jobs remaining.
Starting XML serialization
DONE
Processed 3757 traceable artifacts.

Media summary:
291 images. Please process /tmp/media/images.csv
3 shapes. Please process /tmp/media/shapes.csv

Running time: 6 min, 52 sec
```

Listing 5: Tool output for a run with chapter 3 of Subset-026

Phase 1, Lines 1–15 In the beginning some statistical information is given about the input file and the options specified by the user.

Phase 2, Lines 16–34 After that a sequential processing of the input file takes place whose progress is visualized by a status line of each processed top-level paragraph with its attributed tracestring (Section 2.3) and the first characters of its contents. This output is occasionally interrupted by some information (Lines 16/17 and 26/27) or perhaps a warning, if suspicious elements like hidden text or broken links were found (not shown here). The sequence strictly follows the one outlined in Figure 2 on page 17. Line 18 corresponds with the *Change History*, Line 20 with the *Table of Contents*, Lines 21–30 represent the *Main part*, and Lines 31–34 the *Appendix*. Only the title page is not shown in the output.
This process yields the in-memory tree of all document artifacts already mentioned in Section 2.2.

Phase 3, Lines 35–39 In the next step this tree is visited[23] again and metadata refinements are performed which require knowledge of the hierarchical position of the visited artifact. A heading detection algorithm, for instance, greatly benefits from knowing whether the current artifact does or does not have children (the latter being an exclusion criterion for headings, see Section 3.3.1). NLP is also performed within this step (Section 3.4.2).

Phase 4, Line 40 This line represents another visit of the tree, this time with the intention to serialize[24] its contents into the output file.

Phase 5, Lines 41–46 All remaining lines again give statistical information about what has been done. Lines 44–45 reference external CSV files which must be fed to helper tools (see Section 3.1.1) in order to properly extract and convert embedded media objects.
The most influential parts of the running time (Line 46) are the NLP-related algorithms of Phase 3. If those are disabled, the time is cut down to only eight seconds for this chapter on the same machine.

After the tool finishes with a return value of `0` (i.e. no error), the user is left with a ReqIF file, a `media` and a `statistics` subfolder. The latter serves as the input to statistics processing as discussed in Section 2.2.1. If the `media` folder is non-empty, the contained files need to undergo an additional treatment outlined in the next Section. Otherwise, this step can be skipped and the ReqIF file may be directly imported by an RM-tool of choice.

3.1.1 Dealing with embedded media

The ReqIF file format allows to embed external media into rich text content (see Section 2.1.3). In order to maximize compatibility across different RM-tools, ReqIF contains different layers of content for each media-artifact. On the lowest level each such artifact is represented by an XHTML-formatted String which is expected to be digestible by all conceivable RM-tools (the embedded pictures in Figure 10 on page 47 are displayed this way). The next level is always a PNG-image and the last, optional level is a file of arbitrary format. While rendering, RM-tools are

[23]In the computer-science denotation of the term. I.e. some action is being performed on each node of the tree.
[24]read: “write in correct order”

required to start with the highest available layer, but may fall back onto the preceding one if they fail to handle it. The entire process is described in more detail in [Obj13, clause 10.8.20, point 2].

Currently, the tool only writes the first two layers. This implies that all embedded media which is not already in PNG format needs to receive special treatment. For this purpose the `media` subfolder contains two CSV-files:

images.csv deals with all graphical objects which can be extracted as a *separate file* from the DOC input. Those raw (unconverted) files are saved alongside the CSV and are usually of Windows Metafile (WMF) or Windows Enhanced Metafile (EMF) format.
Each line in `images.csv` represents a reference to an individual object and stores the target dimensions (width and height) along with it. By feeding this file to a dedicated macro designed for Microsoft Visio, all those objects can be batch-converted into PNG.
The tool can also be reconfigured to use different conversion approaches which do not rely on proprietary software from the Microsoft Office family. However, those alternatives (namely: ImageMagick's `convert` on Windows with GDI-support and `libwmf` on Unix, both of which are open-source) do not provide comparable quality.

shapes.csv deals with shapes in the so-called "Office Drawing Binary Format" as specified in [Mic14b]. These are commonly created through the drawing tools natively provided by Microsoft Word. Such shapes cannot exist in isolation (i.e. they cannot be extracted and legally saved into a separate file) [Mic11]. Thus, `shapes.csv` only states offsets (similar to the `startOffset` used for backward tracing in Section 2.2) of those objects in the original input DOC together with the filename where the resulting PNG is expected to go. The actual extraction is performed by another macro, which requires both the original DOC-file and `shapes.csv` as its input. Although this macro runs inside Microsoft Word, it needs Microsoft Visio to be present as well.
There is no viable alternative[25] for the handling of such content, except for one special kind of drawings (see Section 3.3.3). Formalized directly by the tool, they use a very limited subset of the drawing format discussed above and are therefore exempted from the file `shapes.csv`. Hence, this is the only time when the tool must rely on external proprietary software.

In the example of Listing 5 both CSV-files are explicitly referenced (Lines 44–45). If the input file happens to contain only one kind of media or no media at all, the non-applicable lines are omitted and the CSV will not be present, either.

As stated in Section 2.3.2, the input documents contain a fair amount of OLE-data. Using the approach outlined above, these data will always be flattened to WMF or EMF[26]. By utilizing ReqIF's third content layer which can hold arbitrary data, one could also link these original OLE-BLOBs to the ReqIF output file. However, only a few RM-tools can actually take advantage of this option. Besides, the focus of the tool was primarily on providing a decent input to implementers of a system, rather than to authors of a specification willing to alter the embedded

[25]Although LibreOffice/OpenOffice, respectively their headless variant `unoconv`, claim to support such drawings, they, in fact, fail miserably with those embedded in the Subset-026.

[26]In fact, this is performed by Microsoft Word automatically in order to display something meaningful in case the application which originally created the respective files is not available on the user's computer.

graphics (which is why one would embed the original OLE-data in the first place). Lastly, this approach will not work for the non-independent data referenced by `shapes.csv` unless it is embedded into an artificial wrapper document[27], which is quite an onerous task.
Extracting the original MTEF-representation of equations is also unlikely to be worthwhile since edits can only be performed using Microsoft's own Equation editor for as long as those objects are embedded in an Office document. Alternatively, Design Science's MathType software, which the Microsoft editor derives from, may still be used even after they have been extracted. However, that is a rather exotic piece of proprietary software without any significant market penetration. Alas, a truly useful formalization of such equations as $\TeX$- or MathML-markup is hard to obtain because only a limited open-source implementation of the MTEF file format is available [SP12] and ReqIF lacks support for any of the aforementioned markups. Fortunately, this situation has somewhat improved with the XML-based successor of the DOC file format (`*.docx`) where equations are stored in the openly documented Office MathML (OMML) format, a competitor to MathML [Mur06].

3.2 ReqIF output

As stated in Section 2.1.3 earlier, ReqIF files may come in two flavours: Plain, uncompressed `*.reqif`, which requires any referenced media to be shipped separately. And `*.reqifz`, which is simply a ZIP-archive[28] of all those files. Currently, the tool only writes the uncompressed variant, because the handling of media objects requires manual intervention as outlined in the preceding Section. However, `*.reqifz` may be easily generated from the postprocessed output using any third-party ZIP archiving tool. Doing so is indeed advisable, for two reasons: Some of the chapters of Subset-026 require more than 30 Megabytes of space in their uncompressed form, and the high degree of redundancy coming with XML lends itself to a good compression ratio thanks to the Huffman coding used inside the ZIP algorithm.

Regardless of its format, the file may be opened by one of the numerous RM-tools supporting the ReqIF standard. Two example renderings of the identical, generated ReqIF-file by *formalmind Studio*, a somewhat extended version of ProR commonly regarded as the ReqIF reference renderer, and DOORS are shown in Figures 10, respectively 11 on page 47. Both screenshots display the same specification excerpt as the one given in Figure 7 on page 29.

Although ReqIF allows several specifications within one file, this functionality has not been used for the Subset-026, whose individually available chapters (see Section 2.1) could have been represented in this way. This is mostly due to the nature of the tool which processes each DOC file separately. But this also comes with the advantage of manageable file sizes (see above) and the theoretically feasible option to use file-based locks to prevent rivalling edits to those files, when more granular artifact-based locking (as multitenant RM-tools usually provide it) is not available. The downside of this approach is the necessity to use proxy elements for external references, see Section 3.2.2.

[27]ReqIF's implementation guide suggests to use Rich Text Format (RTF) for this purpose [PMS14, Sec. 2.5.3, subclause 8]

[28]Note: `zip`, not `compress` as the simple `z` suffix may confusingly suggest to anyone familiar with Unix.

However, this multi-specification functionality may still be employed by a user of the ReqIF-file in the sense of a *view* onto certain artifacts. In this concept the standard *view* provided by the tool happens to closely resemble the one Microsoft Word would show upon opening the source DOC (same artifacts, same sequence, similar hierarchy). A different *view* may instead focus only on a certain set of artifacts, perhaps those which are only relevant to the trackside part of ETCS. Artifacts referenced from within this new *view* can have a different hierarchy and/or a different sequence. Thanks to the unique tracestrings and the derived internal identifiers, which are both based on the standard *view* (see Section 2.3), this new *view* is equally valid for traceability- and other RM-purposes.

3.2.1 Data associated with a requirement artifact

One major goal of the tool is to split continuous text into smaller chunks (artifacts) which convey a coherent piece of information. This is not only done to allow for traceability as explained in Section 2.3, but also to be able to attribute individual metadata to these artifacts.
For use with ReqIF such artifacts are mapped onto *SpecObjects* (Section 2.1.3). In Figures 10 and 11 each of these *SpecObjects* is represented by a single line within the central grid. In order to avoid cluttering this grid with too many columns holding all this metadata, ProR features a separate `Properties`-window for this purpose. A screenshot of this window showing the contents of 3.6.1.3.*[3][2], an artifact already discussed in Section 2.3, is depicted in Figure 12 on page 48. The properties are grouped into three segments, each with an individual set of fields to capture a piece of data. The meaning of those segments is as follows:

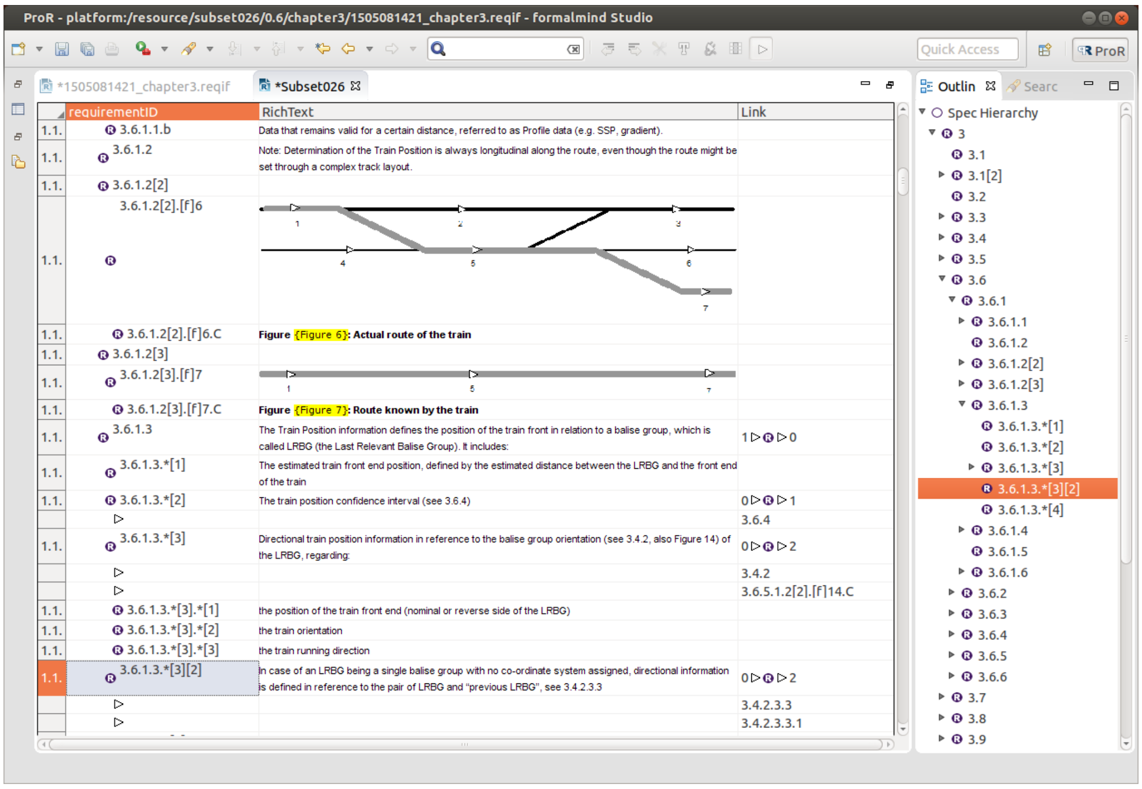
Requirement Type This holds the actual payload: tool-computed metadata, specific to the concrete ETCS use-case. Standard fields according to the ReqIF naming conventions (see Section 3.2.3) would also appear here.
As the name suggests, all the fields in this segment are specific to the type of the current artifact (see Figure 4 on page 21). However, since the tool always writes the same type for all kinds of artifacts except proxies (see Section 3.2.2), this set of fields is always identical.

Spec Object A set of internal fields. Among others this shows the `xsd:id`-compliant internal identifier of this artifact which is derived from the tracestring (Section 2.3.3).
As with the fields of `Requirement Type`, this set is also associated with the current type of the artifact.

Spec Hierarchy As discussed in Section 2.1.3, *SpecObjects* can be referenced from multiple specifications. This group displays the data associated with the instantiation of the *SpecObject* within the current specification. So all these fields originate from the *Specification* part of the ReqIF file rather than the *SpecObject* itself (Figure 4 on page 21).
Because the tool only writes a single specification, the identifier in here is equal to that inside the `Spec Object`-group above, suffixed by `_singleton`[29].

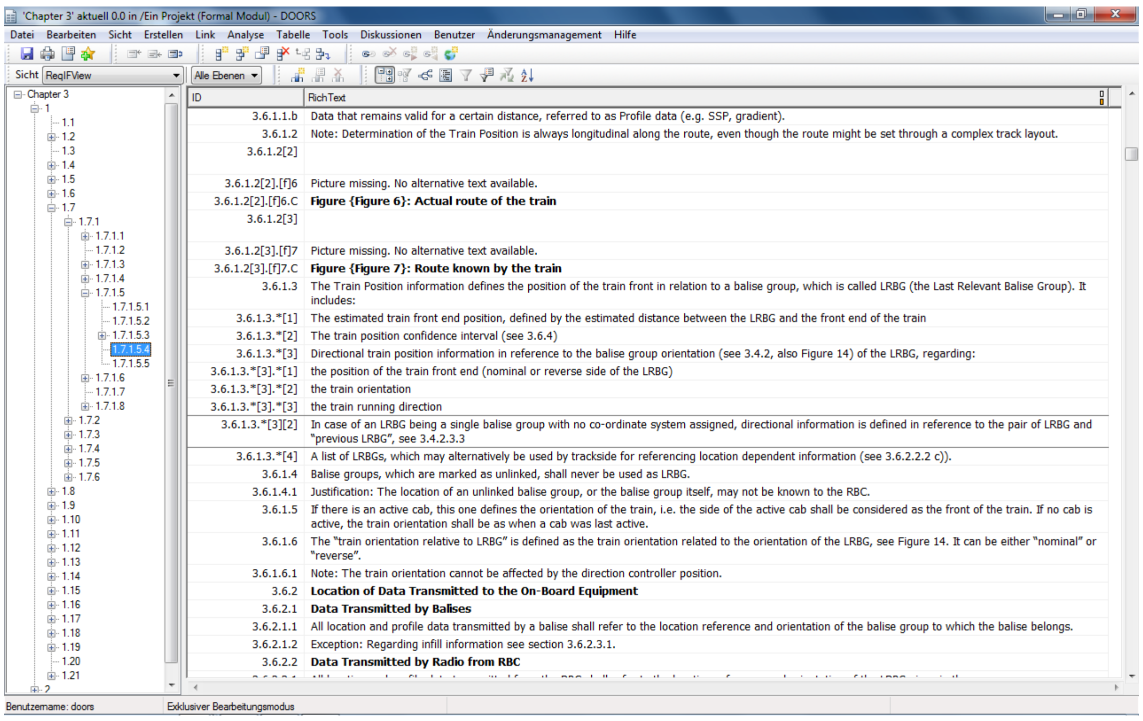
The `Spec Object`- and `Spec Hierarchy`-segments are mostly of internal use for ReqIF rendering tools. Therefore, they are often omitted from user interfaces or only displayed in a dedicated

[29]See the explanation of singleton in the glossary.



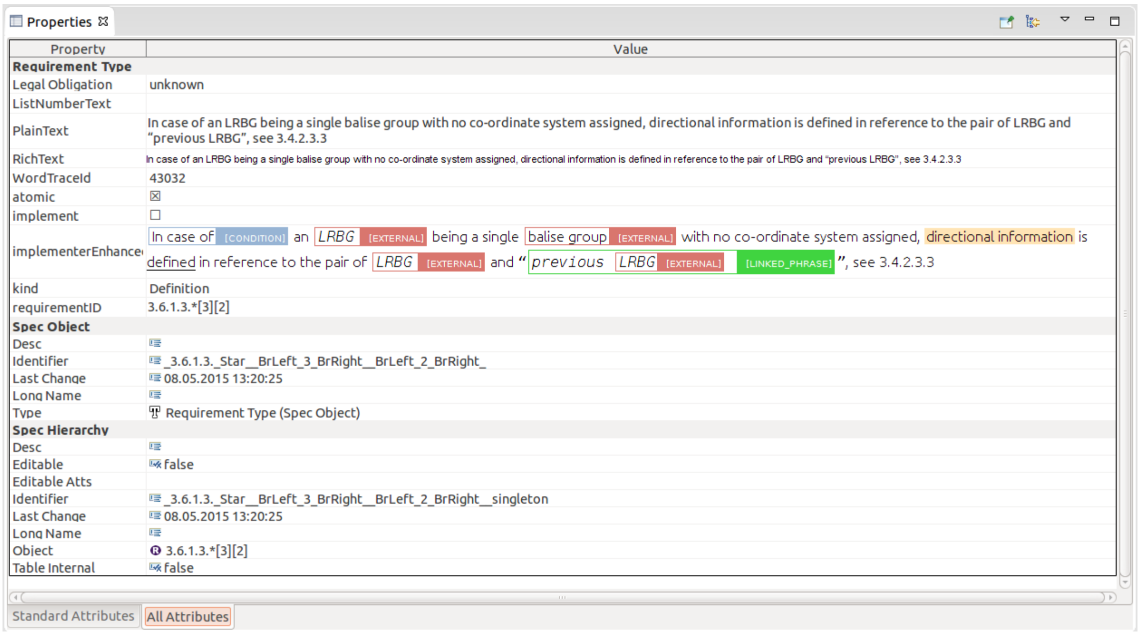
	requirementID	RichText	Link
1.1.	3.6.1.1.b	Data that remains valid for a certain distance, referred to as Profile data (e.g. SSP, gradient).	
1.1.	3.6.1.2	Note: Determination of the Train Position is always longitudinal along the route, even though the route might be set through a complex track layout.	
1.1.	3.6.1.2[2]		
1.1.	3.6.1.2[2].[f]6		
1.1.	3.6.1.2[2].[f]6.C	**Figure {Figure 6}: Actual route of the train**	
1.1.	3.6.1.2[3]		
1.1.	3.6.1.2[3].[f]7		
1.1.	3.6.1.2[3].[f]7.C	**Figure {Figure 7}: Route known by the train**	
1.1.	3.6.1.3	The Train Position information defines the position of the train front in relation to a balise group, which is called LRBG (the Last Relevant Balise Group). It includes:	1 ▷ Ⓡ ▷ 0
1.1.	3.6.1.3.*[1]	The estimated train front end position, defined by the estimated distance between the LRBG and the front end of the train	
1.1.	3.6.1.3.*[2]	The train position confidence interval (see 3.6.4)	0 ▷ Ⓡ ▷ 1
	▷		3.6.4
1.1.	3.6.1.3.*[3]	Directional train position information in reference to the balise group orientation (see 3.4.2, also Figure 14) of the LRBG, regarding:	0 ▷ Ⓡ ▷ 2
	▷		3.4.2
	▷		3.6.5.1.2[2].[f]14.C
1.1.	3.6.1.3.*[3].*[1]	the position of the train front end (nominal or reverse side of the LRBG)	
1.1.	3.6.1.3.*[3].*[2]	the train orientation	
1.1.	3.6.1.3.*[3].*[3]	the train running direction	
1.1.	3.6.1.3.*[3][2]	In case of an LRBG being a single balise group with no co-ordinate system assigned, directional information is defined in reference to the pair of LRBG and "previous LRBG", see 3.4.2.3.3	0 ▷ Ⓡ ▷ 2
	▷		3.4.2.3.3
	▷		3.4.2.3.3.1

Figure 10: Example rendering of chapter 3 by ProR



ID	RichText
3.6.1.1.b	Data that remains valid for a certain distance, referred to as Profile data (e.g. SSP, gradient).
3.6.1.2	Note: Determination of the Train Position is always longitudinal along the route, even though the route might be set through a complex track layout.
3.6.1.2[2]	
3.6.1.2[2].[f]6	Picture missing. No alternative text available.
3.6.1.2[2].[f]6.C	**Figure {Figure 6}: Actual route of the train**
3.6.1.2[3]	
3.6.1.2[3].[f]7	Picture missing. No alternative text available.
3.6.1.2[3].[f]7.C	**Figure {Figure 7}: Route known by the train**
3.6.1.3	The Train Position information defines the position of the train front in relation to a balise group, which is called LRBG (the Last Relevant Balise Group). It includes:
3.6.1.3.*[1]	The estimated train front end position, defined by the estimated distance between the LRBG and the front end of the train
3.6.1.3.*[2]	The train position confidence interval (see 3.6.4)
3.6.1.3.*[3]	Directional train position information in reference to the balise group orientation (see 3.4.2, also Figure 14) of the LRBG, regarding:
3.6.1.3.*[3].*[1]	the position of the train front end (nominal or reverse side of the LRBG)
3.6.1.3.*[3].*[2]	the train orientation
3.6.1.3.*[3].*[3]	the train running direction
3.6.1.3.*[3][2]	In case of an LRBG being a single balise group with no co-ordinate system assigned, directional information is defined in reference to the pair of LRBG and "previous LRBG", see 3.4.2.3.3
3.6.1.3.*[4]	A list of LRBGs, which may alternatively be used by trackside for referencing location dependent information (see 3.6.2.2.2 c)).
3.6.1.4	Balise groups, which are marked as unlinked, shall never be used as LRBG.
3.6.1.4.1	Justification: The location of an unlinked balise group, or the balise group itself, may not be known to the RBC.
3.6.1.5	If there is an active cab, this one defines the orientation of the train, i.e. the side of the active cab shall be considered as the front of the train. If no cab is active, the train orientation shall be as when a cab was last active.
3.6.1.6	The "train orientation relative to LRBG" is defined as the train orientation related to the orientation of the LRBG, see Figure 14. It can be either "nominal" or "reverse".
3.6.1.6.1	Note: The train orientation cannot be affected by the direction controller position.
3.6.2	**Location of Data Transmitted to the On-Board Equipment**
3.6.2.1	**Data Transmitted by Balises**
3.6.2.1.1	All location and profile data transmitted by a balise shall refer to the location reference and orientation of the balise group to which the balise belongs.
3.6.2.1.2	Exception: Regarding infill information see section 3.6.2.3.1.
3.6.2.2	**Data Transmitted by Radio from RBC**

Figure 11: Example rendering of chapter 3 by DOORS


Properties

Property	Value
Requirement Type	
Legal Obligation	unknown
ListNumberText	
PlainText	In case of an LRBG being a single balise group with no co-ordinate system assigned, directional information is defined in reference to the pair of LRBG and “previous LRBG”, see 3.4.2.3.3
RichText	In case of an LRBG being a single balise group with no co-ordinate system assigned, directional information is defined in reference to the pair of LRBG and “previous LRBG”, see 3.4.2.3.3
WordTraceId	43032
atomic	☒
implement	☐
implementerEnhance	In case of [CONDITION] an LRBG [EXTERNAL] being a single balise group [EXTERNAL] with no co-ordinate system assigned, directional information is defined in reference to the pair of LRBG [EXTERNAL] and “previous LRBG [EXTERNAL] [LINKED_PHRASE]”, see 3.4.2.3.3
kind	Definition
requirementID	3.6.1.3.*[3][2]
Spec Object	
Desc	
Identifier	_3.6.1.3._Star__BrLeft_3_BrRight__BrLeft_2_BrRight_
Last Change	08.05.2015 13:20:25
Long Name	
Type	Requirement Type (Spec Object)
Spec Hierarchy	
Desc	
Editable	false
Editable Atts	
Identifier	_3.6.1.3._Star__BrLeft_3_BrRight__BrLeft_2_BrRight__singleton
Last Change	08.05.2015 13:20:25
Long Name	
Object	3.6.1.3.*[3][2]
Table Internal	false

Figure 12: Properties view of ProR showing artifact 3.6.1.3.*[3][2]

enhanced view. The latter is the case with ProR, where these data are only shown when "All Attributes" (right tab in the bottom of Figure 12) is explicitly selected. In contrast, "Standard Attributes" (left tab) focuses on the data of the first segment `Requirement Type`, which is what the user is most interested in.
All the fields of this segment bear custom, domain-specific names and their associated data represents the output of various different algorithms of the tool predominantly targeted at enhancing the user experience for implementers of ETCS. However, these data may also be used as an input to further automated processing means, such as those outlined in Sections 2.2.1 and 2.2.2. The meaning of the fields is as follows:

Legal Obligation A list of all possible values defining the degree of obligation inherent in the current artifact:

`mandatory (>= 1)`	At least one of the artifacts of this sublist must be implemented. *(only applicable to sublist items which are related to each other)*
`mandatory (== 1)`	Exactly one of the artifacts of this sublist must be implemented. *(only applicable to sublist items which are related to each other)*
`mandatory`	This artifact must be implemented.
`optional (== 1)`	At most one of the artifacts of this sublist may be implemented. *(only applicable to sublist items which are related to each other)*
`optional`	This artifact may be implemented.
`mixed`	Parts of this artifact must be implemented, other parts may be implemented. This implies `atomic` (see below) is `false`.
`unknown`	The obligation of this artifact could not be determined.
`not applicable`	This artifact does not have a legal obligation (e.g. it is marked as deleted).

Annotation *(Example)*	***class*-attribute**	**Explanation**
`ARROW TO: [r][4].EOLMValid`	`arrow`	Formalized arrow inside of a table. See Section 3.3.3.
`[r][4].EOLMValid`	`hrMetadata`	Visual helper to quickly find the correct cell-artifact associated with a particular cell in a table layout. Its content equals the tracestring of the respective artifact starting from the row-level (which constitutes a table-wide unique identifier). See Section 2.3.1 for an explanation of the underlying concept and Figure 16 for an exemplary application.
Table 2	`field`	Cross-reference pointing to a different part of the same document. See Section 3.2.2.
Table 2	`field`	Cross-reference as above whose target could not be found. In such cases a warning (Listing 5) will be printed on the screen during conversion as well.
`{Figure 6}`	`field`	So-called *SEQUENCE* field [ECM06, clause 2.16.5.63]. Such a field contains a type (in the example it is bound to *Figures*) and a type-specific sequential number generated by Microsoft Word.
some text `[N]1`	`note`	Footnote. Its actual contents are given in a child of this artifact. A lowercase `[n]` designates an endnote, instead. The number is tool-computed. Emits a warning (Listing 5) if the author of the DOC requested a non-arabic numbering scheme, since this does not play well with the tracestring. See Section 2.3.2.
~~Section 3.6.2.4 shall not apply.~~		So-called *Vanished Text* [ISO12, clause 17.3.2.41], which, depending on user configuration, may or may not display on screen / be printed. Emits a warning (Listing 5) as well.
`PLACEHOLDER REQUIREMENT - DO NOT TRACE`		Marks an artifact whose only purpose is to maintain a proper hierarchy. No other content is allowed in such cases. This also implies the artifact's `kind`-field equals `Placeholder`.
`CONTENTS HAVE BEEN SPLIT UP - SEE CHILDREN`		Marks an artifact with complex content (e.g. a table cell with a list as in Figure 8). See Section 2.3.1.

Table 3: Different annotations of the `RichText`-field

Annotation *(Example)*	***class*-attribute**	**Explanation**
shall	`LegalObligation`	Legal obligation of the current artifact according to the list in Section A.2.1
can	`LegalObligationUnknown`	Words which may indicate a legal obligation but are nowhere defined as such. According to the list in Section A.2.1
indicates	`Predicate`	Predicate of a sentence (based on NLP)
possible	`Predicate`	Phrases which should take the role of a predicate but were not detected as verbs (based on NLP)
The LRBG	`Headphrase`	Subject of a sentence and its modifiers (based on NLP)
immediately [weak]	`weak`	Weak words according to the lists given in Section A.2.2
if [Condition]	`Condition`	A condition according to the list given in Section A.2.3
for each [Loop]	`Loop`	A repetition according to the list given in Section A.2.3
re-evaluate [Again]	`Again`	A repeated action according to the list given in Section A.2.3
as long as [Time]	`Time`	A time-reference according to the list given in Section A.2.3
signalman [External]	`External`	A known external entity according to the list given in Section A.2.3
on-board unit [Self]	`Self`	The specified system according to the list given in Section A.2.3
previous LRBG [Linked_Phrase]	`Linked_Phrase`	Phrases which have been seen previously in other artifacts. A link to the artifact where this phrase was first seen will be added in such cases as well.
"previous LRBG"	`NamedEntity`	Named entities. These are either all-caps or come enclosed in quotation marks.
(some text)	`Embraced`	Embraced text
Note:	`NoteIdentifier`	A Type prefix for a *Note*
Deleted	`DeletedIdentifier`	A qualifier for a deleted artifact
Justification:	`JustificationIdentifier`	A type prefix for a *Justification*
Exception:	`ExceptionIdentifier`	A type prefix for an *Exception*
even if [Condition] [weak]	Annotations may also be nested, confer with Section 2.2.2	

Table 4: Different annotations of the `implementerEnhanced`-field

The ReqIF datatype (see Figure 4 on page 21) of this list is a single-valued enumeration. Section 3.3.2 further discusses the items classified as belonging to a "related sublist".

ListNumberText Original numberText which Microsoft Word applied to this artifact or an empty String if this artifact is not part of a list.

PlainText Textual contents of this artifact without any formatting. Or an empty String if such a representation does not make sense (e.g. the artifact holds an image).

RichText Contents (text, image, equation, . . .) of this artifact with formatting applied. Some special annotations may appear here as well, their meaning is explained in Table 3[30].
The value of this field is also displayed in the second column of ProR's grid (see Figure 10).

WordTraceId Running number used for backward tracing to Microsoft Word (see Section 2.2).

atomic Boolean qualifier which is `true` if and only if this artifact describes a single thing. The value is computed based on the presence of conjunctions ("and", "or") combined with a sentence count.

implement Boolean qualifier which is `true` if and only if this artifact needs to be implemented (i.e. this artifact is a *real* requirement). Currently, the tool defaults this value to `true` unless there are clear hints indicating the opposite (e.g. those mentioned in the `kind`-field below).

implementerEnhanced Representation of the textual contents of this artifact based on `PlainText` (see above) and enhanced for implementation purposes. See Table 4 for a description of all the colorful annotations. The class-attribute of those annotations can be used for XPath-queries as outlined in Section 2.2.2.

kind A list of different kinds of content which can be represented by this artifact:

`ordinary`	Ordinary text without any specifics.
`Table`	Table, the row of a table or its caption.
`Figure`	Figure or its caption.
`Note`	Note which does not need implementation.
`Example`	Example which does not need implementation.
`Justification`	Justification for another artifact. Does not need implementation.
`Heading`	Heading to other artifacts. Does not need implementation.
`Placeholder`	Artifact only exists to ensure a proper hierarchy or is marked as deleted. Does not need implementation. See Section 2.3.
`Definition`	Artifact defines something. Does not need implementation. The defined term, so-called *definiendum* [WP06, Sec. 3], is not yet explicitly marked.

The ReqIF datatype of this list is a single-valued enumeration (i.e. one artifact is always of exactly one `kind`). However, the user may change it into a multi-valued one if that proves more useful (think: lots of artifacts with a `Legal Obligation` of `mixed` and/or an unset `atomic` field).

[30]This Table cites standards relevant to the XML-based successor of DOC. This is due to the lack of equally comprehensive documentation for DOC.

requirementID The tracestring of this artifact as defined in Section 2.3. The value of this field is also displayed in the first column of ProR's grid (see Figure 10).

Admittedly, the selection of all these fields is arbitrary. However, they represent a best-effort attempt to extract as much meaningful data as possible from the natural language contents of the ETCS specification documents. Except for the annotations inside the `Rich Text`- and `implementerEnhanced`-fields (Tables 3 and 4), the contents of all those fields may also be easily amended by a human if the algorithms computing their initial values did not deliver satisfactory results.
In other words: All these fields are meant to assist anyone involved with the ETCS specification. But there is no obligation to actually use this data for any particular purpose.

The actual implementation of the algorithms used to compute the initial values of the fields has only been briefly discussed here. Section 3.4.2 will extend on the technology behind the `implementerEnhanced`-field. For the remainder see the Javadoc-documentation of the code in `helper.subset26.MetadataDeterminer`, `helper.subset26.MetadataDeterminerSecondPass` and their respective subclasses in the tool's source.

3.2.2 Links between requirement artifacts

Technical papers on complex systems – and the Subset-026 makes a particularly striking example here – tend to show a high degree of cohesion, while their degree of contiguity is much lower. So concepts are often intertwined (cohesion) but their individual parts are spread across a wide range of pages (little contiguity). Although this is certainly not a favorable text property, it can hardly be avoided in all cases, especially if the respective document stays on an abstract level, where intentionally only few interfaces, packages or other means of functional grouping are present. They would indeed predetermine a certain way of implementation which is not desired in these early stages of the systems development life cycle (SDLC).
For the Subset-026 this situation led to a specification sprinkled with all sorts of references to tie its otherwise unrelated blocks of information together. Structurally speaking, those references can be grouped as follows:

1. **Cross-references**: Any kind of in-text mention of another named part of the document.

   (a) *Explicitly set through Microsoft Word*: This is a cross-reference explicitly created by the author of the input DOC[31]. Microsoft Word stores both source and target in a standardized way so they can be automatically extracted by the tool. ProR renders this kind of reference as shown in Table 3 on page 49.

   EXAMPLES: refer to chapter 4.7 "DMI depending on modes"). *Source: 4.4.9.1.6*
   described in Table 1 *Source: 3.5.7.5*
   8 *Source: 7.4.2.0[2].[t]*.[r][4].Length*

[31] E.g. in Microsoft Word 2003 this is performed by clicking on `Insert` → `References` → `Cross-reference`.

(b) *Fully-qualified implicit reference*: A simple in-text mention of a some other entity. The tool employs heuristic approaches to identify and normalize such mentions.

EXAMPLES:	according to 3.6.2.2.2c	*Source: 3.4.2.3.3.8.1*
	according to 3.5.3.4 f).	*Source: 3.5.4.3.1*
	see items b), c), d), e) in 3.5.3.4	*Source: 3.5.7.3.a*
	(see 3.18.3.2 items b) c) and d))	*Source: 3.13.2.2.1.2*
	Concerning a) and b) of clause 3.16.2.5.1	*Source: 3.16.2.5.1.1*

(c) *Unambiguous implicit reference*: This is conceptually similar to the case above, except the mention does not use the numberText of a list but some other unambiguously resolvable numbering scheme.

EXAMPLES:	(see Figure 25a)	*Source: 3.9.3.12.2*
	referring to figure 22c	*Source: 3.8.3.4.1*

(d) *relative reference*: A possibly ambiguous reference to some preceding or following entity. Currently, the tool only features rudimentary support for the handling of the former case.

EXAMPLES:	Exception to a):	*Source: 3.6.2.2.2.1*
	Regarding c):	*Source: 3.6.2.2.2.3*

2. **Recurring phrases**: Any kind of in-text mention of a phrase which has been previously used elsewhere.
At the moment, the tool does not feature any generic algorithm to detect such phrases, but focuses only on those whose first mention is enclosed in quotation marks – or typographically questionable variations thereof (see Section 4 for a more generic approach). If such a phrase, which must consist of at least two words to reduce false-positives, is found again somewhere later in the document, a reference to the first mention will be generated. Thus, for better illustration the following examples feature possible targets (i.e. first mentions) of references instead of their sources as in all the cases above.

EXAMPLES:	"Linking information is used" shall be interpreted	*Source: 3.4.4.2.1.1*
	requirements for “Start of Mission” procedure	*Source: 5.4.3*
	the information "default balise/loop/RIU information".	*Source: 3.16.2.4.8*

For the very last case the tool saves all possible interpretations as potential reference targets. That is:

- `default balise/loop/RIU information`
- `default balise information`
- `default loop information`
- `default RIU information`

In ReqIF all these references are turned into so-called *SpecRelations* (Section 2.1.3) with all cases below item 1 of the above list having the type `CrossRefLink` attributed and cases belonging to item 2 being typed as `KnownTermLink`.
Such a *SpecRelation* always has exactly one source and one target, both of which must be existing artifacts within the current file. For the examples above this means the respective artifacts need to be resolved or created. Resolving depends heavily on the granularity used while

creating the artifacts. If, for instance, a certain cell within a table has been configured as superfluous by a table matcher (Section 2.3.1), but is nonetheless targeted by a reference, a fallback to the parent (in this case: row-) artifact must take place. Artifact creation is necessary if an entity outside the scope of the current document is referenced, as it is sometimes the case for item 1b. This is achieved by writing an artificial proxy *SpecObject* of a special type called `Proxy Type`, with only an identifier but no further content, into the ReqIF and then referencing this in the same way as any other ordinary *SpecObject*. Since the detection algorithm behind item 1b still has flaws, it does sometimes create false-positive references (see the jUnit test cases in `docreader.range.paragraph.characterRun.FakeFieldHandlerTest` for examples). These are particularly disturbing for statistical postprocessing as outlined in Section 2.2.1. Appendix A.1.1 shows how to manually correct such mistakes.
Entirely ruling out these false-positives is quite challenging because there is no way to verify if a target represented by a proxy actually does exist somewhere in an external file. Matters are further complicated by inaccurate mentions of external, unnumbered tables, consisting only of a reference to the immediately preceding numberText, whose associated paragraph may potentially encompass more than one table. The latter frequently happens in the headings of chapter 6 of Subset-026.
Some references such as those involving images or OLE-objects currently cannot be processed at all (see for instance the references in 5.11.2.2[2].[t]* targeting the image in 5.11.3.1[2].[f]8).

By design, a *SpecRelation* always applies to an entire *SpecObject* (read: artifact). Combined with the philosophy of the tool to only turn each paragraph into such an artifact (Section 2.3), those relations therefore often become less granular than they were in the input DOC. In other words: A few characters of an artifact *A* which constitute the source of a relation to an artifact *B* cannot be directly linked to *B*. Instead, only artifact *A* in its entirety can take the role of the source.
This weakness could be mitigated by either creating a single artifact for all those "few characters" which make up a link, thereby effectively equalling out the granularities of an in-text mention and a *SpecObject*. Or by creating a proprietary ReqIF-extension that attributes metadata to each *SpecRelation*. However, this data can at most convey the position of the "few characters" within the respective artifact and would thus need a compatible renderer to make use of it. In practise, neither of those two approaches was followed as the current situation seldom leads to ambiguities, especially because the phrases of item 2 are also annotated within the `implementerEnhanced`-field (green `[Linked_Phrase]`-box of Table 4 on page 50). For a truly formalized specification this may be too weak, though.
If the same artifact *B* is referenced multiple times from within *A*, as it is the case for the first example in item 1a, only one *SpecRelation* will be created.

A second property of *SpecRelations* which can be viewed both as a bug and a feature is the fact that they are not necessarily "clickable". See the right column entitled `Link` in Figure 11 on page 47 for how ProR renders such relations in its grid. One might expect that a click on any of the artifacts (targets) mentioned in this column will cause the grid to scroll to the respective position of that artifact. However, since *SpecObjects* which act as those targets may be instantiated multiple times even within the same specification (Section 3.2.1), it is not always clear where to scroll to. This becomes particularly apparent if *views* as suggested in Section 3.2 are employed.

The multiple specifications that come with *views*, however, offer one more treat of the ReqIF file format: *SpecRelationGroups* can be used to clump *SpecRelations* by the different specifications they connect [Obj13, Fig. 10.7]. For instance, after creating one *view* for all trackside requirements and another one for all those which belong to the train, such a group could be utilised to tag relations that define interfaces between the two.

Figure 5 on page 26, which has been previously discussed in Section 2.2.1, shows a graphical representation of the roughly 400 relations of type `CrossRefLink` (as well as the ordinary hierarchical connections) within chapter 3 of Subset-026. The 90 `KnownTermLink`-relations of this chapter have been omitted.

3.2.3 Issues with IBM DOORS

Obviously, the rendering by DOORS in Figure 11 on page 47 is a little less pleasing compared to that of ProR in Figure 10. The reasons for this nicely indicate the challenges any standardized interchange format commonly faces:

1. DOORS has generally poor XHTML-processing capabilities. For instance, it fails to handle any kind of table rendering such as those produced for reasons of visual aid by the tracestring-algorithm outlined in Section 2.3.1. Since IBM's proprietary concept of "DOORS Tables", which can be triggered by a special "Table Internal" flag inside the ReqIF file (shown in Figure 12 on page 48 in the very bottom), cannot really cope with the complexity of the tables of Subset-026[32], this is quite an unpleasant situation.
2. ReqIF does not provide a generic way of specifying its visual appearance. Therefore, the generated files include *tool extensions* (confer with Figure 4 on page 21) to convey this information which can only be parsed by ProR. Such extensions are currently not generated for DOORS.
3. The files intentionally do not comply with the recommendatory ReqIF naming conventions [PMS14, Sec. 2.2] intended to map common properties of a requirement to the respective fields of different RM-tools.

To justify why this naming contract has been broken, consider Figure 12 again. The textual content of the artifact shown there is always trapped within a field entitled *RichText*. A second field called *kind* holds a list of all possible kinds of such an artifact (see Section 3.2.1 for a more thorough explanation). The value of *kind* for this artifact is precomputed by the tool based on heuristics. If those heuristics fail for any reason, this value may be later amended by the user. So this approach combines a high degree of automatization with maximum flexibility.
On the other hand, ReqIF's naming conventions mandate the textual contents of an artifact to go into three different fields depending on what kind of information they convey. A *heading* to many artifacts must go into `ReqIF.ChapterName`, unless it has the character of a *title* to a single artifact (the subtle differentiation between these two cases is left as an exercise for the reader) in which case `ReqIF.Name` is the field of choice. For all other content `ReqIF.Text` must be used.

[32]Specifically it neither supports merged nor omitted cells very well.

Owing to the different philosophies of the tool vendors who designed the ReqIF standard, such arbitrary distinctions make it substantially harder for a user to correct any wrong assignments by the heuristic algorithms. Hence, it was decided not to follow these conventions. This comes at the cost of DOORS not recognizing any fields in those files out of the box along with using arbitrary numbering for the tree[33] shown in the left part of Figure 11. However, since the field names can be easily changed, this can be fixed by renaming `requirementID` into the slightly confusing term `ReqIF.ChapterNumber` prior to importing the file into DOORS.

The fact that pictures are neglected by DOORS (Figure 11 only shows the least-significant, text-only content layer for the Figures in 3.6.1.2[2].[f]6 and 3.6.1.2[3].[f]7) is a bug in DOORS which is currently being investigated by IBM[34].

3.3 Content formalization

A fair amount of technical documentation is based on repetition. This does not necessarily mean such papers are full of redundancies, but rather that a reader will often come across conceptually similar passages which are always presented in the same way. In contrast to poetry where the author may be tempted to describe similar events with different sets of words from a rich and metaphorical vocabulary, technical writers strive for the exact opposite: well-tended boredom.
So, "formalization" in this context means to detect such similar passages and output them in a uniform way. What makes this challenging is that computers are rather averse to "similarity" and instead much prefer "equality". Hence, the involved algorithms must allow for a certain degree of fuzziness in their input to be able to correctly classify *mostly similar* still as *similar*. How this can be achieved for the case of tables has already been shown in Section 2.3.1. The following Sections will therefore put more emphasis on the remaining formalizable elements of Subset-026.

3.3.1 Detection of recurring elements

Tables (Section 2.3.1) make up the largest part of recurring elements in the Subset-026 and they can often be correctly detected simply by means of their visual structure (e.g. n columns and m rows, of which the first x are part of the table header, makes a table of type y). Their actual contents are only a secondary measure for categorization if the structure alone is not stark enough. For other elements structural information is not available to such an extent, so the detection must focus a lot more on their contents. Two kinds of these elements shall be discussed in more detail:

figure- and table-captions Throughout the Subset-026 captions are always made up of ordinary text following the element they describe. So DOC's dedicated features for typesetting them are not used. The detection heuristic must therefore be able to elicit such cap-

[33] *Module Explorer* in DOORS' terminology.
[34] According to an email from IBM United Kingdom received on May 21, 2015

tions from other textual contents, make sure those texts are not mistakenly read in again as some other kind of artifact at a later point, and then handle them in a special way. This is done slightly differently for the captions of figures and tables, since they are an essential qualifier to differentiate between an ordinary image and a figure in the former case, but only an optional part in the latter case (Section 2.3.2).
The algorithm responsible for this process has grown quite complex over time due to the number of different caption layouts present in the input files. The only invariant (common property) it can count on is the caption's two-part design: There is always some identifier with a running number (i.e. "Table 3") followed by some descriptive text. Both parts are separated by a colon (`:`). Usually such a caption lives in a separate paragraph immediately following that of the described element. But for certain edge cases there are empty paragraphs in-between or the caption is only separated by a line break. The alignment (left aligned, centered) varies as well. In addition to that, the actual content of a caption may either contain a SEQUENCE-field for autonumbering its running number (confer with Table 3 on page 49). This field may or may not be suffixed by a single, manually entered alphabetic character (i.e. something like "Figure 20 b"; with the grey part being a field). Or the number in its entirety was manually entered, in which case special care must be taken to work out if this number is actually unique (confer with item 1c of Section 3.2.2). A conceptual relationship between the caption identifier and the described element cannot be safely assumed, as chapter 4 of Subset-026 does contain tables whose captions confusingly call it a "Figure" (e.g. artifact 4.5.2.1[2].[t]1).
The class `docreader.range.paragraph.CaptionReader` of the tool's source contains the implementations of the heuristics discussed above.

Once the caption has been identified, it always becomes the first child of the element it describes and inherits its tracestring suffixed by `.C` (Listing 4 on page 39, line 32). Embedding the caption right into the `RichText` data of the artifact representing the described element is not possible because ReqIF's XHTML subset allows to use the relevant tag only for a caption to tables (i.e. `<caption>`) but not to figures (`<figcaption>`).
Eventually, the running number of the caption is extracted and used for the tracestring computation (see Section 2.3.1).

headings The detection of headings is a two-step process: During phase 2 of a tool run (see Section 3.1) possible candidates are selected on the basis of their visual appearance. They must constitute a single paragraph without any line breaks, certain characters like a dot (`.`) are banned, and all remaining ones must have **bold** or SMALL CAPS formatting applied. Moreover, there is an upper limit to the number of allowed characters and the name of the style attached to them must contain "Heading", "Überschrift" or "Titre" (the latter two being the German, respectively French equivalent of "Heading"). All of this happens in the method `determineRequirementKind()` of `helper.subset26.MetadataDeterminer`. Unfortunately, the outline level, which is a hierarchical number attached to each paragraph of a DOC file to mark a heading, is completely messed up in Subset-026 and therefore not respected[35].

In a second step, which takes place during phase 3 of the tool run, these candidates are checked again taking into account their hierarchical position. Would-be headings without

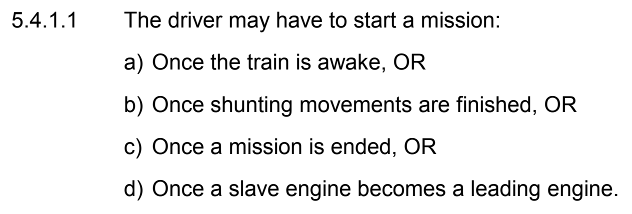
[35]The import facilities of other tools like DOORS heavily rely on the correctness of this number. This makes one contributing factor why they fail so miserably for the Subset-026. See Section 3.5.

any children are reset and ordinary artifacts preceding a sublist (see Section 3.3.2) are added. Eventually, all resulting headings receive their `kind`-field set to the value `Heading` (Section 3.2.1).
The method `processRequirement()` in `helper.subset26.MetadataDeterminerSecondPass` contains the implementation of this second step.

There are a few more recurring structures in the Subset-026 which are given special treatment by the tool. However, their algorithms are comparably simple and therefore skipped for reasons of brevity. The interested reader may refer to the Javadoc in `helper.subset26.MetadataDeterminer` (especially methods such as `isDefinition()`) and `helper.subset26.LegalObligationDeterminer`.

3.3.2 Sublist dependencies

The Subset-026 contains a number of sublists whose items share a certain boolean relation. A sublist, in this context, is a list where all items have numberTexts attributed which are not unique on their own (confer with the beginning of Section 2.3). Furthermore, the paragraph preceding the list must contain a condition and a keyword indicating a forward reference[36] such as "following" (e.g. "If the following...") or end with a colon (`:`)[37]. For formalization purposes, this boolean relation is to be extracted and attributed to the individual sublist items through their `Legal Obligation` field (see the respectively tagged values of this field in Section 3.2.1).
There are essentially two corner cases for such sublists and a variety of blends in-between:


5.4.1.1 The driver may have to start a mission:

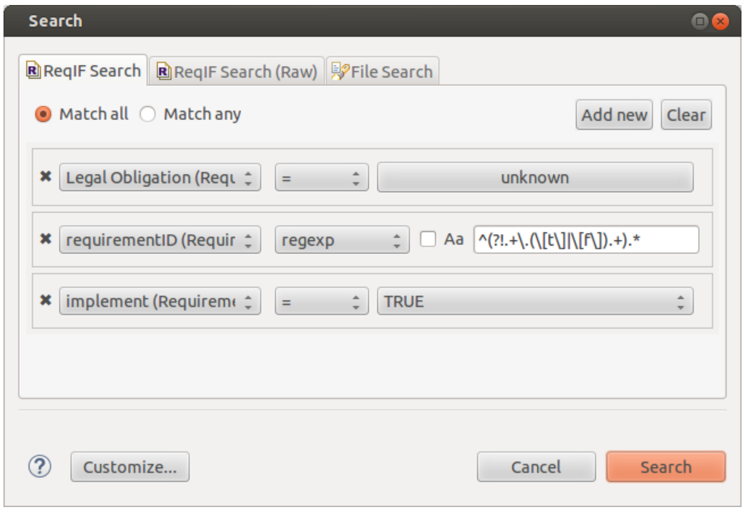
a) Once the train is awake, OR
b) Once shunting movements are finished, OR
c) Once a mission is ended, OR
d) Once a slave engine becomes a leading engine.

Figure 13: Artifact 5.4.1.1 as shown by Microsoft Word

1. In the first case the preceding paragraph completely defines the relation of the sublist items. A typical example would be artifact 3.13.2.2.4.1 *"The brake position shall be set to one of the following three values:"*, with those "values" listed in the children of the artifact.
   Here, the algorithm extracts the legal obligation (in this case `mandatory (==1)`) from the given parental artifact, bequeaths it to the children and attributes the parent either a `Heading` or `ordinary` kind, depending on its exact wording (in this case `ordinary`). Its legal obligation is reset to the value that would have been attributed without the presence of a sublist (in this case `mandatory`).

2. The second case defines the boolean relation only on the basis of the sublist items. Unfortunately, the Subset-026 does not contain a really good example of this case in its purest form, so consider the following artificial structure, instead:

[36] Literature sometimes refers to this as a "continuance" [WRH97, p. 169].
[37] There are a few exceptions to this generic rule which will not be discussed here. See the tool's source for details.



[● Match all] ○ Match any

Legal Obligation (Requ	=	unknown
requirementID (Requir	regexp	☐ Aa ^(?!.+\.(\[t\]|\[f\]).+).*
implement (Requireme	=	TRUE

Figure 14: Search for ill-categorized sublists using ProR

1.2.3 To perform a system reset:
- The driver shall press the system's reset button, OR
- The driver shall pull the emergency brake, OR
- The driver shall power cycle the entire on-board unit.

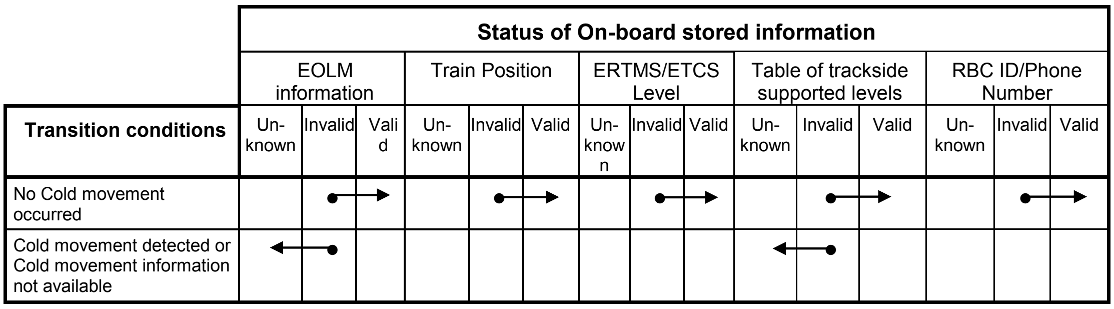
Each child of "1.2.3" (except the very last) ends with "OR". Combining this information with the keyword "shall" (see Appendix A.2.1) contained in each item's text yields a legal obligation of `mandatory (>= 1)` for all of them.
Similar hierarchies within the Subset-026 would come without the redundant "The driver shall" prefix in the sublist items. Instead, this phrase would appear only once at the end of the parental artifact (see e.g. artifact 3.9.3.12).

Figure 13 on the facing page shows an example of the many blends between the two aforementioned cases. In here all the sublist items are attributed the legal obligation `optional (==1)` (i.e. they are mutually exclusive)[38] based both on the preceding paragraph and on their "OR"-suffix.
The class `helper.subset26.MetadataDeterminerSecondPass` contains the implementation of the different sublist-related algorithms. Due to the highly heterogeneous wording used for the paragraphs preceding the sublists, it is infeasible for the code to automatically detect all their possible manifestations. This is a result of the regex foundations of the employed heuristics which do not perform well in such cases (Section 3.4.2). However, by using the search facilities of an RM-tool the artifacts in need for manual post-treatment can be easily identified. A possible query for chapter 3 of Subset-026 using ProR is depicted in Figure 14.

3.3.3 Intra-cell requirements

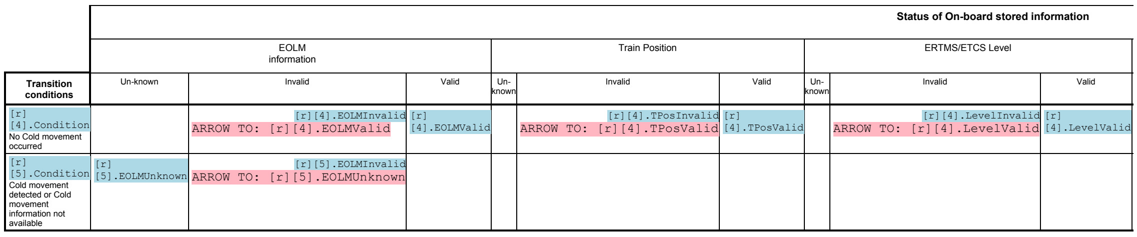
Figure 15 on the following page depicts a short example of a so-called "transition table". As its name implies, such a table is intended to visualize permissible transitions from one system

[38] Formally speaking, the "OR" at the end of each sublist item must actually be interpreted as an "XOR". It remains unclear whether this is what the requirement author actually intended.



	Status of On-board stored information														
	EOLM information			Train Position			ERTMS/ETCS Level			Table of trackside supported levels			RBC ID/Phone Number		
Transition conditions	Un-known	Invalid	Valid	Un-known	Invalid	Valid	Un-known	Invalid	Valid	Un-known	Invalid	Valid	Un-known	Invalid	Valid
No Cold movement occurred		●	▶		●	▶		●	▶		●	▶		●	▶
Cold movement detected or Cold movement information not available	◀	●								◀	●				

Figure 15: Table with domain-specific formatting in 4.11.1.1[2].[t]*



	Status of On-board stored information								
	EOLM information			Train Position			ERTMS/ETCS Level		
Transition conditions	Un-known	Invalid	Valid	Un-known	Invalid	Valid	Un-known	Invalid	Valid
[r] [4].Condition No Cold movement occurred		[r][4].EOLMInvalid ARROW TO: [r][4].EOLMValid	[r] [4].EOLMValid		[r][4].TPosInvalid ARROW TO: [r][4].TPosValid	[r] [4].TPosValid		[r][4].LevelInvalid ARROW TO: [r][4].LevelValid	[r] [4].LevelValid
[r] [5].Condition Cold movement detected or Cold movement information not available	[r] [5].EOLMUnknown	[r][5].EOLMInvalid ARROW TO: [r][5].EOLMUnknown							

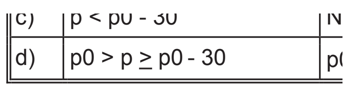
Figure 16: Left half of the table of Figure 15 after processing by the tool

state to another by means of an arrow connecting two table cells. Technically the table consists of a bit of text in the left and upper parts, which serves as the heading of the respective row or column. However, except for a few bullets representing arrow tails, the actual content area is mostly empty. The arrows themselves are floating objects anchored somewhere in the table, but not necessarily at the position where they are finally displayed by Microsoft Word.
As outlined in Section 2.3.1, the default granularity level for the formalization of table contents is the cell. Hence, those arrows must be converted into a representation that conveys the same piece of information as before but fits into a single such cell. Since the positions of those arrows are only known relative to their anchor, the first step to achieve this is inevitably to lay out the entire table mimicking the behavior of Microsoft Word as closely as possible. This is a rather complex and computationally expensive task because row heights are mostly not given explicitly but must be inferred from the cell containing the longest text. Rendering out this text in a headless mode[39] while taking into account a plethora of subtleties like kerning, line pitch and hyphenation can be at most a best-effort attempt (e.g. correct emulation of the hyphenation behavior is virtually impossible without reimplementing Microsoft Word's typesetting engine from the ground up). Once this is done, a special `TableContentOverrideManager` (the class may be found in `docreader.range.table`) comes into play. It scans all cells of the table for the aforementioned anchors and assigns them to the cell where their starting point will effectively be displayed. All other content items are stripped, namely the bullets which do not really convey any information. In a subsequent step an `OfficeDrawingReader` takes over (its code may be found in `docreader.range.paragraph.characterRun`). The methods of this class identify the arrow, check if it is rotated or flipped and then (re-)assign the final cell where the visible arrow tail is situated. Finally, a standardized "ARROW TO: <*TARGET*>" text will be appended to the contents of this cell (see Table 3 on page 49 for details) and both the former target- and this source-cell will get a tracestring attributed. Figure 16 shows the result of this tremendous effort for the case of

[39]That is: without displaying it to the user

the example table of Figure 15.
Since the implementations of all the involved algorithms are nowhere near perfect, a subsequent manual check of the results is greatly advisable. However, they seem to perform quite well for the current version of the Subset-026. Ultimately, the numerous tables which contain such arrows (e.g. artifact 5.4.3.3[2].[t]* covering a multitude of transitions between different variable states) ought to justify why so much work has been put into this project.

3.3.4 Unformalizable elements



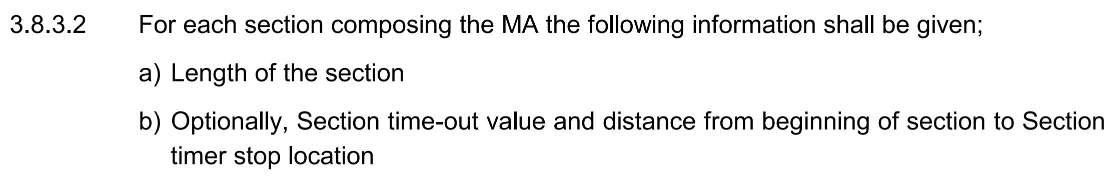
c)	p < p0 - 30	N
d)	p0 > p ≥ p0 - 30	p(

Figure 17: Artifact A.3.10.4[49].[t]*.[r][5].[c][2] as shown by Microsoft Word

There are certain elements within a specification which simply cannot be reasonably tackled by a computer. And often enough they raise questions for a human reader as well. Consider Figure 17 for a tiny, but drastic example of such an element. At the risk of stating the obvious, the problem with this Figure lies in the interpretation of the symbol $\geq$. In the DOC file this is encoded as a greater-than sign ($>$) with underlining applied. Such a construct is highly ambiguous because, without any context, a human reader (and a computer will not perform any better at this) can only speculate if this underline was intended as a way of putting emphasis on the greater-than sign or if the author was simply unable to locate the greater-than-equals sign ($\geq$) on his/her keyboard (i.e. the entity constitutes a special kind of digraph). As nit-picky as this may seem, a similar, seemingly innocent issue with messed up equations, which intentionally were not checked for overflows, effectively constituted the root cause for the loss of the Ariane 501 rocket in 1996 [Lio96, p. 5]. This is widely regarded as the most costly single accident caused by a software failure in human history.
The tool handles the above case by showing the equation (which technically is not an OLE-object as described in Section 2.3.2 but only an ordinary sequence of characters) with a similar rendering like that of Microsoft Word in its `richText`-field (Section 3.2.1). However, the `plainText` only contains $>$ at the position of this dubious symbol.

A second example of barely formalizable data are text-heavy images (within the Subset-026 this often means: statecharts). Those which are also available as OLE-data may be treated in the same way as similarly wrapped equations (i.e. implement their original file format and unwind the contents; see the end of Section 3.1.1). Those which can only be obtained in a rasterized format effectively mark a dead end unless Optical Character Recognition (OCR) was to be employed. Both approaches are extremely difficult to implement while the immediate benefit may be comparably low. However, they constitute the only viable way to automatically resolve references to named parts of their contents as outlined in Section 3.2.2.

Eventually, there are also various less severe cases of formalization-reluctant elements which must be taken care of manually due to some sort of underspecification: A sublist as discussed in Section 3.3.2 earlier is depicted Figure 18 on the next page. Items a) and b) inherit their legal obligation from the parental element which contains the keyword “shall” indicating a value of `mandatory` for this field (Section 3.2.1). However, item b) also contains the adverb “optionally” suggesting a more lenient legal obligation of `optional`. Since “optionally” is not defined


3.8.3.2 For each section composing the MA the following information shall be given;

a) Length of the section

b) Optionally, Section time-out value and distance from beginning of section to Section timer stop location

Figure 18: Artifact 3.8.3.2 together with its sublist as shown by Microsoft Word

as a keyword for the Subset-026, the tool will only highlight this word as *unknown* within the `implementerEnhanced` field (see Table 4 on page 50) but refrain from altering the legal obligation.

3.4 Inner workings

Most of this thesis is written in a bird's-eye manner oriented primarily towards the final output and the new opportunities that come with it. The following Sections will deviate slightly from this path to try and give an impression of the Java code driving the tool behind the scenes. Two examples are chosen for this purpose: The hierarchy extraction algorithm that assigns parent/child-relations to individual artifacts, and the techniques used for the much higher level text analysis. Before going into the details, however, a short summary of the tool's overall structure will be given.

Figure 19 on the next page shows a graphical representation of the tool without any unit tests, integration tests and dependencies on external libraries. `subset026reader` serves as the application's entry point. Its `main()`-method handles the input parameters (see the beginning of Section 3.1) and passes them on to a `DocumentReader` which lives inside `docreader`. In this class the different phases discussed in Section 3.1 are effectively carried out. Its child packages below `range` resemble the relevant parts of DOC's range philosophy (see Figure 3 on page 19).
As outlined in Section 2.2, the data gathered by `DocumentReader` will be stored in a tree of artifacts. These artifacts are modelled by the package `requirement` which houses a hierarchy of classes representing different artifact grades. In this hierarchy a proxy artifact, for instance, is represented by a class of higher genericity than a full-fledged textual requirement with plentiful metadata. Objects of theses classes store references to their individual children, thereby constituting the aforementioned tree. The `data` and `metadata` subpackages of `requirement` are responsible for artifact-related information such as links (Section 3.2.2), respectively various kinds of metadata (Section 3.2.1).
The output serialization to the ReqIF file format is handled by `reqifwriter`. The purposes of `ReqifDataType` and `ReqifField` are to manage the *FundamentalTypes* of the output document (Figure 4 on page 21), respectively the individual fields associated with an artifact (Section 3.2.1). `SpecObjectMapper` is a supplementary enum-class to take care of the particularities regarding proxies (Section 3.2.2). Eventually, `DocumentWriter` contains the entry point and the glue code to link all the involved classes.
There is also an elaborate collection of packages below `helper`. These were originally intended to encapsulate various kinds of static information, such as the different annotation patterns for the `RichText`- and `implementerEnhanced`-fields (Tables 3 to 4 on pages 49–50) found in

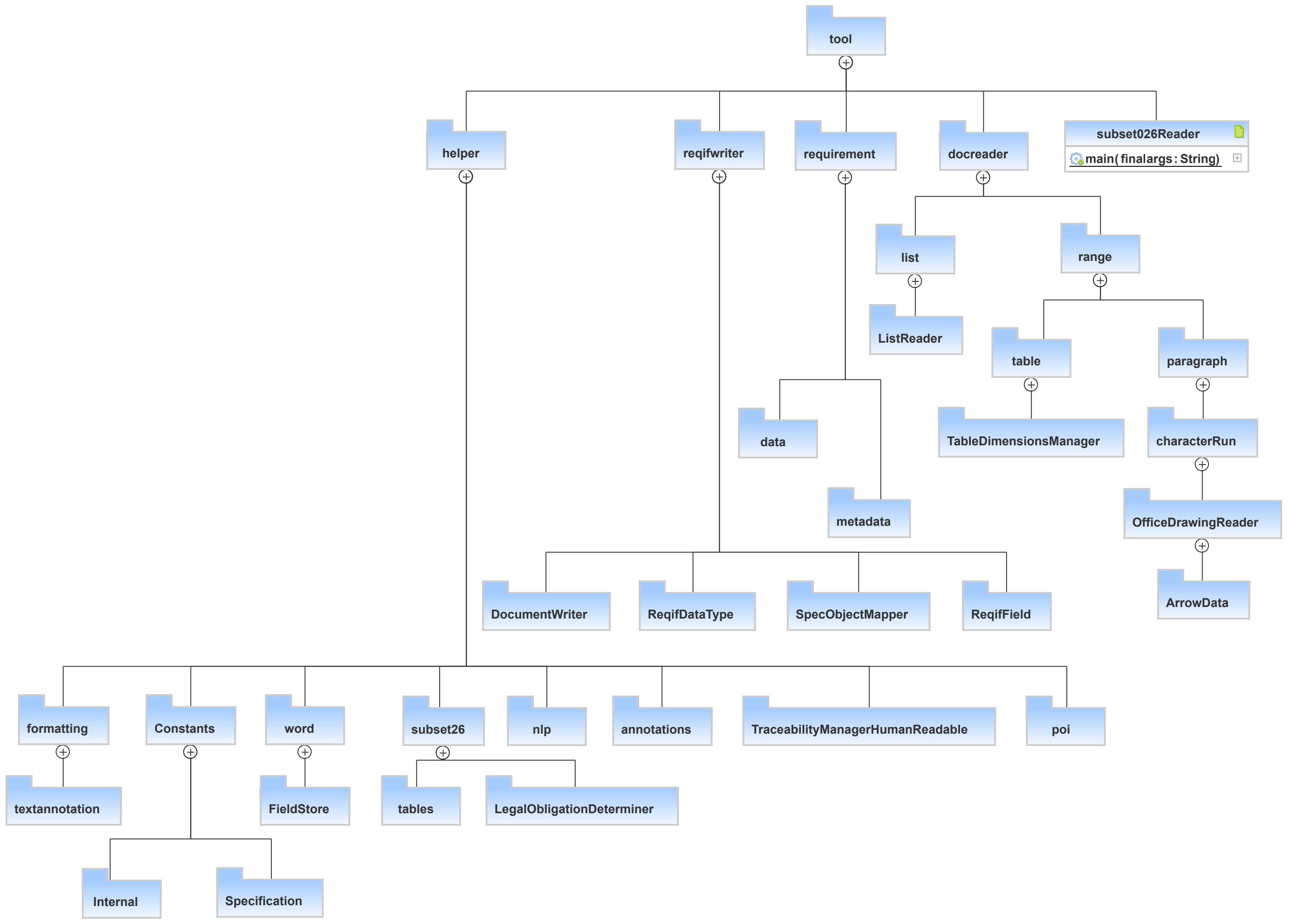


Figure 19: Overview of the tool's package structure

`textannotation` (not to be confused with the `annotation` package whose explanation will be postponed to Section 3.6). Although such encapsulation is often regarded an anti-pattern (so-called "package by layer" rather than "package by feature" [Hir15]), it was crafted with adaptability in mind. Thus, by having access to the source code, the user is not only enabled to easily amend the way annotations are displayed but also to tweak a large amount of other settings, most of which are pooled inside `Constants` and its subpackages. All of this is possible without any deeper knowledge of the actual algorithms responsible for the treatment of the specification files.
Besides these static data there are also various kinds of wrappers to adapt external functionality to the tool's needs: The package `word` contains methods to access common internal structures of the DOC files and to convert data formats specific to DOC into their generic equivalents (e.g. to handle so-called *twips*, a measure for lengths, and to correctly map special character entities). A similar aim is followed by the two packages `poi` and `nlp`. However, instead of abstracting the functionality of a file format, they focus on external libraries. The former inherits its name from Apache POI providing the low level reading facilities for DOC files, and employs a fair bit of reflection to circumvent some bugs in that external code[40]. The latter links to an NLP library (Section 3.4.2) and mostly provides enhancements for carrying out the respective work on the multiple processing units of modern computers simultaneously. The correct construction (Listing 4 on page 39) of tracestrings is taken care of by `TraceabilityManagerHumanReadable`. Ultimately, `subset26` encapsulates a good portion of functionality specific to the Subset-026, such as the abstract table definitions (Section 2.3.1).

As a final note it should be clear that Figure 19 does not depict a UML package diagram in its purest form. This is mostly because of how Java handles classes and how those are mapped onto a meaningful visual representation[41]. For this Figure regular packages (starting with a lower case letter), as well as the parent class of a set of nested classes and enum-classes[42] (the latter two both start with upper case letters) have all been drawn as "packages".

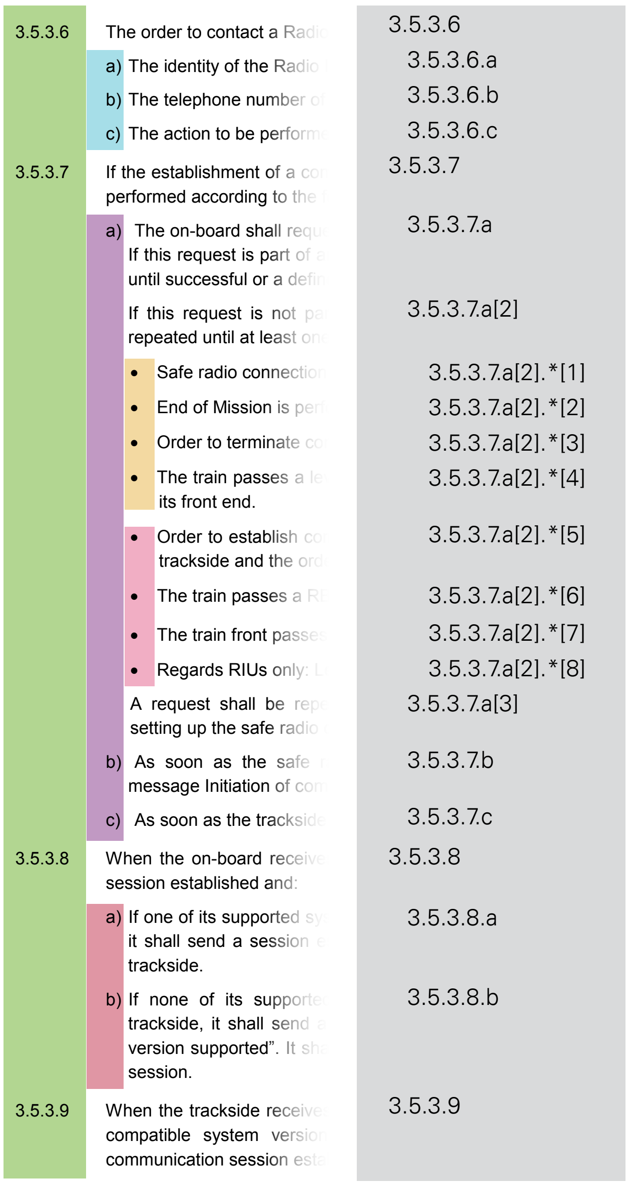
3.4.1 List hierarchy algorithm

Section 2.3 outlined a very straightforward implementation of an algorithm to compute the tracestring for an artifact within the Subset-026. In spite of illustrating the core idea behind the tracestring-computation very well, this algorithm turns out to be much too simple for real world use.
Consider Figures 20 and 21 on on pages 65–66 to get an idea of the problem scope from a more technical point of view. Each colored vertical bar in those Figures marks an independent list. Hence, in Figure 20 the specification authors used six different lists instead of sticking with only the outermost green one and utilizing additional levels to cover for the various sublists. Under these circumstances some computation has to take place which correctly determines the hierarchical relationships of the individual lists in order to process their items into the tracestrings shown in the gray region on the right of the Figure. On top of that, this algorithm must be

[40] In addition, this library has also seen extensive patching. However, this will not be discussed here.
[41] As with Figure 22 on page 67, the entire diagram construction was automatically performed by Yatta Solutions' UML Lab.
[42] Technically their enum-values (read: enumeration values) constitute a set of singleton-classes, i.e. a predetermined number of objects.

capable of both merging technically independent but visually continuous lists (the two bulleted lists below 3.5.3.7.a) and doing the opposite if items of a hierarchically more significant list interrupt a continuous list (blue list of Figure 21).

As simple and intuitive as this may sound, such behavior is hard to grasp for a computer. This difficulty is mostly owed to the iterative nature of the list processing implementation in the DOC file format. While scanning such a file from beginning to end, an algorithm has no idea what item will come next. Therefore, any decisions regarding the hierarchical position of this item can only be based on historical information (i.e. everything that was seen before). Such behavior is genuinely different from the holistic approach towards a list commonly taken by a human.

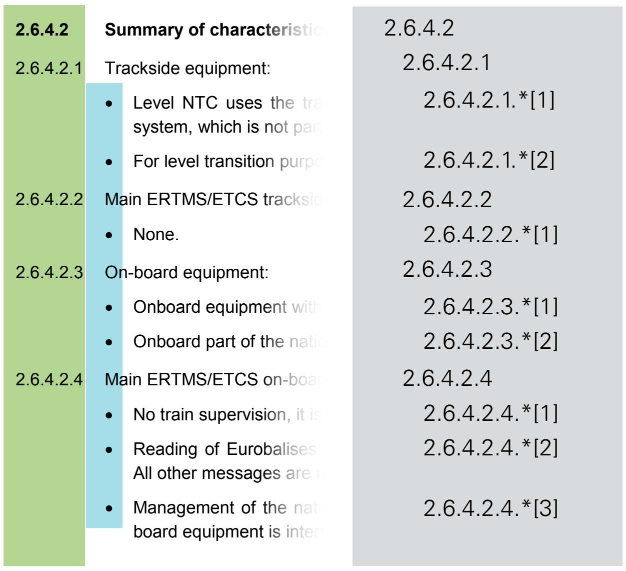


3.5.3.6	The order to contact a Radi	3.5.3.6
	a) The identity of the Radio	3.5.3.6.a
	b) The telephone number o	3.5.3.6.b
	c) The action to be perform	3.5.3.6.c
3.5.3.7	If the establishment of a co performed according to the	3.5.3.7
	a) The on-board shall requ If this request is part of a until successful or a defi	3.5.3.7.a
	If this request is not pa repeated until at least on	3.5.3.7.a[2]
	• Safe radio connectio	3.5.3.7.a[2].*[1]
	• End of Mission is per	3.5.3.7.a[2].*[2]
	• Order to terminate co	3.5.3.7.a[2].*[3]
	• The train passes a le its front end.	3.5.3.7.a[2].*[4]
	• Order to establish co trackside and the ord	3.5.3.7.a[2].*[5]
	• The train passes a R	3.5.3.7.a[2].*[6]
	• The train front passe	3.5.3.7.a[2].*[7]
	• Regards RIUs only: L	3.5.3.7.a[2].*[8]
	A request shall be rep setting up the safe radio	3.5.3.7.a[3]
	b) As soon as the safe message Initiation of co	3.5.3.7.b
	c) As soon as the tracksid	3.5.3.7.c
3.5.3.8	When the on-board receive session established and:	3.5.3.8
	a) If one of its supported sy it shall send a session e trackside.	3.5.3.8.a
	b) If none of its supporte trackside, it shall send a version supported". It sh session.	3.5.3.8.b
3.5.3.9	When the trackside receive compatible system versio communication session est	3.5.3.9

Figure 20: List numbering example from chapter 3 of Subset-026

For an overview of the tool's classes involved with list processing consider Figure 22 on page 67. The lowest level is populated by `ListReaderPlain`. This class encapsulates the logic necessary to correctly calculate the displayed numberText of any list paragraph. It was developed from scratch, using the respective DOC file format documentation provided by Microsoft (specifically [Mic14a, clauses 2.4.6.3 f]; other sources are stated directly in the code) because no comprehensive open-source implementation of the DOC list processing was otherwise available (LibreOffice/Openoffice, for instance, fail to correctly compute all numberTexts present in the Subset-026). Meanwhile, the respective code has been extended by an external person to cover for the `*.docx` file format as well and is due to be integrated into Apache TIKA, a project on top of Apache POI [ADB+15].
The `ParagraphListAware`-object expected as a parameter to the function `getFormattedNumber()` of `ListReaderPlain` is an example of a wrapper provided by the package `helper.poi` (see Figure 19) which takes care of emending certain internal properties of list items. Together with its sibling `getLevelTuple()`, this method is called by `ListReader` which eventually processes this information into a proper hierarchy. To account for multiple nesting levels, specifically lists

inside table cells (confer with Figure 3 on page 19), this class is accompanied by a dedicated stack (`SublistStack`) providing independent list data stores for each nesting level. There is also a `FakeLsidManager` computing valid *Lsids* (unique identifiers attributed by DOC to each individual list) for paragraphs that visually resemble a list item but are technically not part of any list. These are prevalent in the appendix to chapter 3 of Subset-026.



2.6.4.2	**Summary of characteristi**	2.6.4.2
2.6.4.2.1	Trackside equipment:	2.6.4.2.1
	• Level NTC uses the tr system, which is not par	2.6.4.2.1.*[1]
	• For level transition purp	2.6.4.2.1.*[2]
2.6.4.2.2	Main ERTMS/ETCS tracksi	2.6.4.2.2
	• None.	2.6.4.2.2.*[1]
2.6.4.2.3	On-board equipment:	2.6.4.2.3
	• Onboard equipment wit	2.6.4.2.3.*[1]
	• Onboard part of the nati	2.6.4.2.3.*[2]
2.6.4.2.4	Main ERTMS/ETCS on-boa	2.6.4.2.4
	• No train supervision, it is	2.6.4.2.4.*[1]
	• Reading of Eurobalises All other messages are	2.6.4.2.4.*[2]
	• Management of the nat board equipment is inte	2.6.4.2.4.*[3]

Figure 21: List numbering example from chapter 2 of Subset-026

The two main methods exposed by `ListReader` are `getAsPrinted()` which returns the plain numberText destined for the `ListNumberText`-field of the ReqIF file (Section 3.2.1), and `getFullyQualified()` which returns the tracestring to appear in the `requirementID`-field.
This `ListReader` is wrapped again by a `ListToRequirementProcessor` whose `processParagraph()` method is what will actually be called repeatedly for all paragraphs during phase 2 of Section 3.1. The purpose of this last-mentioned class is to properly handle skipped levels for the in-memory tree store (Section 2.2) and to generate the resulting artifacts accordingly.
The diagram in Figure 22 also contains a few collapsed classes which are not of any deeper interest here. As a side note, `TableContentOverrideManager` makes the bridge to the arrow handling presented in Section 3.3.3.

The iterative insertion of a new list item into the tracestring hierarchy is the actual magic within this entire process. The responsible code is deeply buried inside `SublistManager`. Listing 6 on page 68 depicts its core method stripped of any comments to squeeze it onto a single page. The method is part of the `LevelTuple` class which represents a single level in a list[43]. Such `LevelTuples` are tied together by a `LevelStore` representing an entire list. Since nesting of lists is arbitrary, each `LevelTuple` can itself link to multiple `LevelStores` to account for sublists (i.e. in such a case this `LevelTuple` would constitute the parental element as discussed in Section 3.3.2).
Upon insertion of a new list item belonging to a previously unknown list, the logic starts by calling the method shown in Listing 6 of the most significant `LevelTuple` contained in the most significant `LevelStore`. This is usually the number of the currently processed chapter of the Subset-026. It then loops over all levels of the current list and compares (line 7) their various properties (mainly left indentation and outline level) against the item which is to be inserted. If a stored level is found to be less significant (i.e. the condition in line 8 triggers), the loop exits. In a second step (lines 18 ff.) sublists are accounted for. Line 20 checks if the `LevelTuple` which

[43]For the sake of the following explanation `LevelTupleA` mentioned in lines 1 and 3 of Listing 6 as well as `LevelTupleWForce` of line 27 may be regarded as a simple `LevelTuple`.

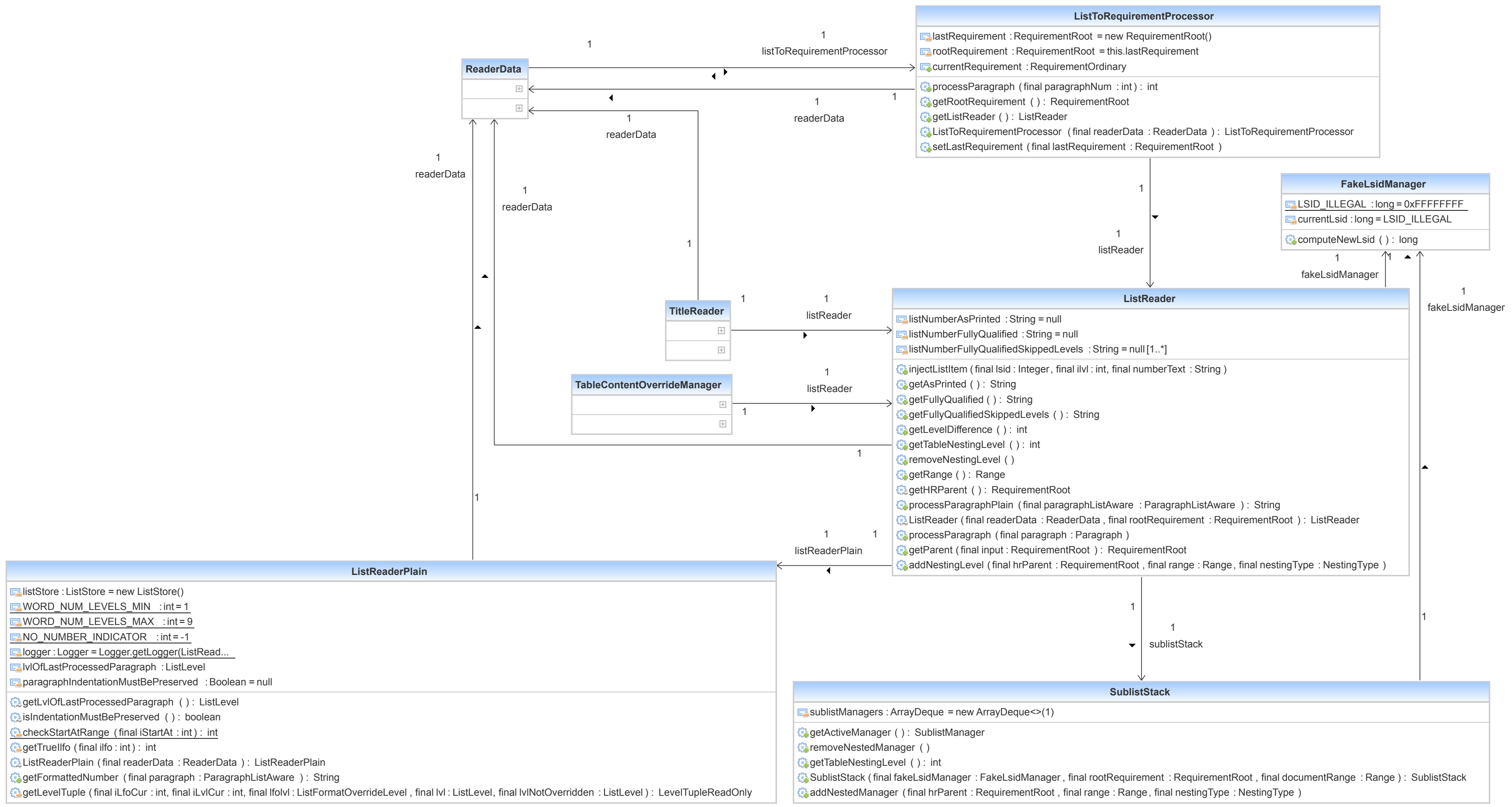

ReaderData
ListToRequirementProcessor
lastRequirement : RequirementRoot = new RequirementRoot()
rootRequirement : RequirementRoot = this.lastRequirement
currentRequirement : RequirementOrdinary
processParagraph (final paragraphNum : int) : int
getRootRequirement ( ) : RequirementRoot
getListReader ( ) : ListReader
ListToRequirementProcessor (final readerData : ReaderData) : ListToRequirementProcessor
setLastRequirement (final lastRequirement : RequirementRoot)
FakeLsidManager
LSID_ILLEGAL : long = 0xFFFFFFFF
currentLsid : long = LSID_ILLEGAL
computeNewLsid ( ) : long
TitleReader
TableContentOverrideManager
ListReader
listNumberAsPrinted : String = null
listNumberFullyQualified : String = null
listNumberFullyQualifiedSkippedLevels : String = null [1..*]
injectListItem (final lsid : Integer, final ilvl : int, final numberText : String)
getAsPrinted ( ) : String
getFullyQualified ( ) : String
getFullyQualifiedSkippedLevels ( ) : String
getLevelDifference ( ) : int
getTableNestingLevel ( ) : int
removeNestingLevel ( )
getRange ( ) : Range
getHRParent ( ) : RequirementRoot
processParagraphPlain (final paragraphListAware : ParagraphListAware) : String
ListReader (final readerData : ReaderData , final rootRequirement : RequirementRoot) : ListReader
processParagraph (final paragraph : Paragraph)
getParent (final input : RequirementRoot) : RequirementRoot
addNestingLevel (final hrParent : RequirementRoot , final range : Range , final nestingType : NestingType)
ListReaderPlain
listStore : ListStore = new ListStore()
WORD_NUM_LEVELS_MIN : int = 1
WORD_NUM_LEVELS_MAX : int = 9
NO_NUMBER_INDICATOR : int = -1
logger : Logger = Logger.getLogger(ListRead...
lvlOfLastProcessedParagraph : ListLevel
paragraphIndentationMustBePreserved : Boolean = null
getLvlOfLastProcessedParagraph ( ) : ListLevel
isIndentationMustBePreserved ( ) : boolean
checkStartAtRange (final iStartAt : int) : int
getTrueIlfo (final ilfo : int) : int
ListReaderPlain (final readerData : ReaderData) : ListReaderPlain
getFormattedNumber (final paragraph : ParagraphListAware) : String
getLevelTuple (final iLfoCur : int, final iLvlCur : int, final lfolvl : ListFormatOverrideLevel , final lvl : ListLevel, final lvlNotOverridden : ListLevel) : LevelTupleReadOnly
SublistStack
sublistManagers : ArrayDeque = new ArrayDeque<>(1)
getActiveManager ( ) : SublistManager
removeNestedManager ( )
getTableNestingLevel ( ) : int
SublistStack (final fakeLsidManager : FakeLsidManager , final rootRequirement : RequirementRoot , final documentRange : Range) : SublistStack
addNestedManager (final hrParent : RequirementRoot , final range : Range , final nestingType : NestingType)
1
listToRequirementProcessor
1
readerData
1
readerData
1
readerData
1
readerData
1
listReader
1
fakeLsidManager
1
fakeLsidManager
1
listReader
1
listReader
1
listReaderPlain
1
sublistStack
1
1
1
1
1
1
1
1
1

Figure 22: UML class diagram showing the list processing subsystem of the tool

```java
private LevelTupleA findCorrectInsertionPoint(final ParagraphPropertiesDeterminer pProperties, final String numberText) {
    assert pProperties != null && numberText != null;
    final LevelTupleA outputTuple;
    detection: {
        LevelTupleA previousTuple = null;
        for (final LevelTupleA currentTuple : this.levels.values()) {
            final int hierarchyComparison = currentTuple.getLevelTuple().compareTo(pProperties);
            if (hierarchyComparison > 0) {
                final LevelTupleA candidate = forceOnThisLevel(numberText, currentTuple);
                if (candidate.force) {
                    outputTuple = candidate;
                    break detection;
                }
                break;
            }
            previousTuple = currentTuple;
        }

        if (previousTuple != null) {
            assert previousTuple.getLevelTuple().compareTo(pProperties) <= 0;
            if (previousTuple.getLevelTuple().getChild() != null) {
                final LevelTupleA childTuple = previousTuple.getLevelTuple().getChild().findCorrectInsertionPoint(pProperties, numberText);
                if (childTuple != null) {
                    final int hierarchyComparison = childTuple.getLevelTuple().compareTo(pProperties);
                    if (hierarchyComparison == 0) {
                        final LevelTupleA currentLevelCandidate = isOnSameListLevel(numberText, childTuple.getLevelTuple());
                        if (currentLevelCandidate.getLevelTuple().isPredecessor()) {
                            if (childTuple.force) outputTuple = new LevelTupleWForce(currentLevelCandidate.getLevelTuple());
                            else outputTuple = currentLevelCandidate;
                            break detection;
                        }
                        else if (!childTuple.force) {
                            assert !currentLevelCandidate.getLevelTuple().isPredecessor();
                            previousTuple = currentLevelCandidate;
                        }
                    }
                    else if (hierarchyComparison < 0) {
                        childTuple.getLevelTuple().setPredecessor(false);
                        previousTuple = childTuple;
                    }
                    if (childTuple.force) {
                        outputTuple = childTuple;
                        break detection;
                    }
                }
                else previousTuple.getLevelTuple().removeAllChildren();
            }
            else {
                assert previousTuple.getLevelTuple().compareTo(pProperties) <= 0;
                final LevelTupleA currentLevelCandidate = isOnSameListLevel(numberText, previousTuple.getLevelTuple());
                if (currentLevelCandidate.getLevelTuple().isPredecessor()) previousTuple = currentLevelCandidate;
            }
            outputTuple = previousTuple;
        }
        else outputTuple = null;
    }
    return outputTuple;
}
```

Listing 6: Java algorithm to insert a new list item into the tracestring hierarchy

was last encountered by the loop (i.e. the last one not causing it to terminate) has a sublist[44] attributed. If so, the entire method shown in this Listing is invoked again starting with the most significant level of that sublist. A successful recursive call (i.e. the condition in line 22 yields `true`) indicates the presence of a sublist level which is more significant than the item to be inserted. Thus, this item now constitutes a member of a sublist's sublist (confer with the bulleted list below artifact 3.5.3.7.a[2] shown in Figure 20 on page 65). The following lines deal with edge cases related to items of unknown lists which are on the same hierarchical level as items of previously encountered lists (with the two bulleted lists of Figure 20 again making a good example here). The helper function `isOnSameListLevel()` is not only capable of determining this property for the case of adjacent bullets but can also account for consecutive numbering which is a strong indicator of hierarchically matching lists. Line 45 takes care of the opposite case exemplified by the two intertwined lists of Figure 21 on page 66: It resets all knowledge about sublists every time a more significant level is encountered.
As a result, the last `LevelTuple` which is more significant than or equally significant as the new item is returned. In a subsequent step, the new item will be added to the governing `LevelStore` accordingly. As explained earlier, the combination of `LevelStores` and `LevelTuples` constitutes a hierarchy from which the tracestring can be easily computed.

The explanation above is intended to give at least a rough idea of how the list processing works. However, some parts of Listing 6, such as the checks against the `force` flag and the case if the algorithm returns `null` for a non-recursive call, have been purposely ignored to keep this Section comprehensible. The interested reader may find the discussed code including the omitted explanatory comments in `docreader.list.SublistManager.LevelStore`.
It should also be noted that the current implementation bears quite some potential for quality improvements. The method `isOnSameListLevel()`, for instance, exhibits non-obvious side effects and there are generally too many violations of the Law of Demeter[45].

3.4.2 Techniques for natural language content

Currently, the tool tackles natural language contents by two means: Regular expressions (regexes) and Natural Language Processing (NLP). The former can be understood merely as an extensive search syntax. To utilize it, first a pattern is defined, then compiled into an efficient binary representation and subsequently matched against the texts of Subset-026. This process has the advantage of being very fast, but comes at the cost of limitation by the expressiveness of the search pattern. So a good pattern should only match exactly those contents it was intended for, nothing more and nothing less. The more complex those patterns become and the less is known about the structure of the content they are matched against, the more likely it is for this contract to break. However, there is hardly any better way, especially when dealing with uniform information extraction from large piles of input that share some common properties.
To limit the number of mistakes referring to regexes, they are rarely ever used alone in the tool but rather combined with a secondary measure. The table matcher of Listing 3 on page 37, for

[44]Although `LevelTuples` can have several sublists, only the most recently added one returned by `getChild()` is of interest here.
[45]This is: The code does not stick to communication with only its nearest neighbors. Simply put, all lines containing more than one dot (`.`) in Listing 6 constitute such a violation.

instance, uses regexes only after all other structural expectations of the table have been fulfilled. In addition, there are numerous jUnit test-cases for the various classes employing regex technology. These use data from the actual Subset-026 to further restrict erroneous results.
One major downside of regexes is their limited ability to account for context. Modern PCRE implementations, including the one of the Java language, do feature support for so-called *lookarounds*, which somewhat mitigate this problem. However, they still fall short when it comes to any deeper concepts of contextual relations other than "search term is preceded by phrase *x* and followed by phrase *y*". An example[46] of this functionality is given by the following regex which is part of the unknown legal obligation keywords of Appendix A.2.1:

```
(?<!MAY␣)OPTIONALLY
```

This simply states that the keyword `OPTIONALLY` must be regarded a match unless it is preceded by the word `MAY` followed by a space. The embraced part is therefore called a *negative lookbehind*.
Patterns, especially user-amendable ones, are often post-processed by the tool into a much larger combined pattern. For the above example this means all the keywords indicating an unknown legal obligation will be concatenated in one single pattern. In a subsequent step, this will be preceded by a qualifier to indicate the entire search shall be performed case-insensitively.

One way of overcoming this drawback and to actually derive meaning from textual contents is the employment of algorithms from the domain of NLP. Specifically, the tool uses a Probabilistic Context-Free Grammar (PCFG) parser developed by the Natural Language Processing Group at Stanford University [Sta15] to identify the predicate and headphrase (that is: the subject combined with a bit of context) of any sentence within an artifact. This information is later visualized in the `implementerEnhanced`-field of the ReqIF output (Table 4 on page 50).
A parser usually stands at the very beginning of any NLP pipeline. It takes a piece of text and splits it into sentences. Each sentence then goes through a so-called *part-of-speech tagging* attributing each word its respective function in that sentence. This step also yields a dependency tree which can be queried to find the said predicate and headphrase. For an example of the entire process try out the interactive parser demo accessible through Stanford's website (see above).
The probabilistic approach of this parser suggests that its algorithms perform some kind of guessing. This is true in the sense that they were previously trained by their authors on a certain set of texts totally unrelated to the Subset-026. During this training a number of so-called *production rules* was extracted describing the construction of natural language sentences. It comes at no big surprise that applying those generic rules[47] onto the Subset-026 does not always yield perfect results. For instance, consider the phrase "Handing Over RBC" prevalent in numerous artifacts of chapter 3. Although the parser does take the capitalization of these three words into account, it still fails to recognize this unusual phrase as an atomic entity rather than a succession of verb, preposition and noun. This mistake could be ruled out by either teaching the parser this phrase as a *named entity* or by actually training the underlying lexical model of the extraor-

[46]As the mindful reader may have noticed, this Section will refrain from explaining the foundations of the actual regex syntax. Numerous thick books on this subject are available.
[47]As they are usually ambiguous, the use of statistical methods is needed to choose between them. This is the actual reason for the term "probabilistic" in the parser's name.

dinary English language used within the Subset-026. The latter is a pretty time-consuming endeavour and requires a fair amount of training data written in the same manner as the specification texts, which simply was not available. Thus, it was decided to use this technology only as an enhancement of suggestive quality within the `implementerEnhanced` field.

NLP generally bears a lot of potential for the RM-domain. However, this can only be properly unleashed if the number of false positives is cut down to a manageable size. This means either the specifications must see some "linguistic smoothing" (as suggested here [Kof05a, Sec. 3.1.2]) or the parser needs dedicated training (see above). A reduction of the number of erroneously parsed sentences to zero is unlikely to happen without the use of a clearly defined (i.e. constrained) input language. Accordingly, NLP's apparent lack of precision is often seen as a major deal breaker for the employment of such technology in the requirements domain [Rya92, Sec. 4].
For an example why this situation cannot be easily improved, consider the use of the keyword "note" as an identifier to artifacts which are only informative (as defined in Subset-026, clause 1.7.1.4). Depending on the context "note" may be a noun ("musical note", "A note to the reader") or a verb ("note this wonderful example") [Ben13, p. 19]. Unfortunately, within the Subset-026 this term hardly ever comes with context. A typical use (taken from artifact 8.4.1.3.2) therefore looks like this:

> Note: orientations are in any case always referred to the directionality of balise group (balise transmission), directionality of loop (Euroloop transmission) or directionality of LRBG (radio transmission).

Since the appearance of "note" in this example essentially constitutes a single word sentence, it remains unclear if this is an imperative verb or a simple noun (and the lack of an active predicate in the following sentence does not make it any easier). It would take only minimal effort to avoid such ambiguity, if the word "note" was simply swapped for the term "annotation" which comes with a distinct noun-making suffix and does not double as a verb in the English language. Although this example may seem a little contrived, it should nonetheless illustrate the general difficulty which also emerges in other, far more complex scenarios.

Given that the previously stated obstacles can be overcome, there are five main areas were NLP could be of valuable use:

Atomicity attribution Specifically, this refers to the improvement of the algorithms responsible for the state of the `atomic`-flag (Section 3.2.1). By utilizing the more-sophisticated sentence detection of an NLP toolkit combined with a count of nouns rather than conjunctions of a sentence, such an enhancement could be accomplished with relative ease.

Context-aware annotation of keywords For the case of weak word detection (Appendix A.2.2) in the automotive domain this has been shown to significantly reduce false-positives [Kri13]. A similar technique could be employed for use within the `implementerEnhanced`-field (Table 4 on page 50).

Detection of ambiguities within the texts of a requirement Consider artifact 3.5.3.9 shown below for an example:

> When the trackside receives the session established report or the information *that no compatible system version is supported by the on-board*, it shall consider the communication session established.

Here, it remains unclear if the emphasized part is a relative clause dependant exclusively on "information", or if the "session established report" (which again makes an example of an atomic phrase a parser would have to be taught in advance) also governs this clause (i.e. there are two entities "report" and "information" both of which are capable of conveying what is said in that clause).
Such an ambiguity detection may be implemented by comparing the different internal trees a probabilistic parser naturally generates. Usually only that of the highest likelihood is made available for further processing. However, the Stanford parser is open-sourced which allows this to be easily changed.
As a side note: Resolving the ambiguity could be as easy as placing a comma behind "report".

Resolution of relative references A typical example of this is a sentence structure like the following (taken from artifact 6.6.3.4.5.b):

> data is deleted (i.e. it is not sent to the receiver)

The intention here is to link the pronoun "it" with the noun "data" to make clear what "it" stands for. This process is commonly referred to as *coreference* or *anaphora resolution* and respective algorithms are readily available.

Improvement of the linked phrase detection The main problem with those phrases (which are discussed in Section 3.2.2) is that they can come in various flections. As this frequently happens with natural language in general, part of the work of a parser is therefore to find the lemma of each processed word (so-called *stemming*). This functionality could be used to find even more subsequent mentions of a previously stored phrase.

NLP has not yet seen widespread use in the RM-domain. This is not only due to its inherent inaccuracy but can also be attributed to the poor running time and excessive memory usage of most algorithms in this field (confer with the respective comment in Section 3.1). Existing tools for information retrieval from specification documents therefore either focus on the automatic processing of some constrained grammar [OM96, Sec. 2] or essentially require a human to perform the work of a parser [Kof05b, Sec. 4], [CCM$^+$10, Sec. 5.3.3], [PYG12, Sec. 3.1]. A notable exception of a commercial software that claims to employ NLP is the RQA tool described here [The15a]. It belongs to the group of quality analyzers to be introduced in the next Section.

3.5 Comparison to other tools

Until now the *raison d'être* of the tool has never been really questioned. This is chiefly because it fits a niche purpose, which simply does not exist in the perfect world other tools are often designed for.

In the mindset of the traditional V-Model the result of each step is assumed to be set in stone. As a matter of course, specifications in this model are flawless. Although practitioners[48] more and more come to the conclusion that this V-Model does not exactly represent reality even for the most rigorous development processes, most software safety standards, including EN 50128 for the railway domain [CEN11, Sec. 5.3.2], still endorse its use.
This attitude essentially yields two classes of tools for the general market: Quality analyzers and specification importers. The former are used during the writing of requirements and assist the author by marking textual properties such as inadequate wording, obvious inconsistencies or too long sentences. Examples of this class include [LGF+05, Sec. 4], [Hei10] and [Hoo15]. The latter usually forms a part of a much bigger RM-application where this component is intended to read in the completed requirements after authoring (and perhaps quality checking). Both tool classes therefore represent some kind of a waterfall-like toolchain whose result is ultimately passed on to the next step of the V-Model where the implementer takes over.

As stated in the introduction (Section 1.1), the tool of this thesis distinctly differs from those approaches. Specifically, it addresses the implementer rather than the primary author. This is because the Subset-026, and ETCS specifications in general, are, indeed, more or less set in stone[49]. Given this situation, an implementer therefore does not (or rather: is not eligible to) care about, say, too long sentences. He/She simply has to live with them. This requires the tool to limit its quality assessment to aspects which are also useful to an implementer. Section 3.2.1 details what this precisely means for the case of Subset-026. The same attitude applies to the interpretation of the DOC input file. Other tools usually necessitate to craft those documents in a specific way in order to facilitate their proper interpretation. DOORS, for instance, relies on a certain structure and wording of the requirements. Reqtify employs regexes as its primary strategy for information extraction and therefore is even more in need of consistent wording. And IBM's RequisitePro, which was used for the *EuRailCheck*-project introduced in Section 1.2, requires the user to place special markers in such files[50] denoting their individual elements [IBM13]. Thus, providing documents without any such features is likely to break the respective import routines (justifying the dashed arrow in Figure 1 on page 16). For the case of DOORS this can be exemplified with intra-cell arrows (Section 3.3.3) and backgrounds of cells (as seen in artifact 2.6.8.3[2].[t]1), both of which would simply be omitted. Another non domain-specific example is depicted in Figure 23 on page 75 where DOORS (top) manages to garble equations. In comparison, the tool of this thesis whose output is rendered by ProR (middle) performs a lot better because its VBA-based conversion approach (Section 3.1.1) employs the very same algorithms as Microsoft Word. Besides, the lack of indentation of these formulas poses challenges for a proper hierarchy extraction[51].

The further enhancements that the tool provides for implementation and subsequent V & V activities are unparalleled by commercial off-the-shelf (COTS) software due to their high degree of domain-specificity. This is particularly true for the linked phrases detection (Section 3.2.2) and the various supplemental metadata of an artifact (Section 3.2.1). Some by-products such as the

[48]E.g. the openETCS staff at DB Netz who rely on agile methodologies, instead.
[49]To be fair, requirements can be altered for a future version. However, the process of integrating changes is very cumbersome and only allows certain named bodies to actually propose any amendments [Gra09, Sec. 2.6.1], [Eur15].
[50]Technically: Vanished Text. This feature of DOC has already been discussed in a different context in Table 3 on page 49.
[51]This effectively constitutes a case when the algorithm of Listing 6 on page 68 will return `null`. Confer with Section 3.4.1.

tagging of elements within artifacts (Section 2.2.2) can even be considered novel altogether.

Alas, academic papers on this, admittedly very practical, problem are scarce. It can only be speculated whether some of the reasons for this lack can be found in the difficulties involved with generalizing the respective ideas. Thus, the few papers that do exist desperately try to abstract from the specific challenges of a certain domain [ASB$^+$08, Sec. 2] or start their analysis from scratch [RAC11]. Low-level discussions of information retrieval techniques applicable to a specific file format are mostly confined to the various XML-derived formats of the World Wide Web [MMK01] and PDF [FFT$^+$08]. A prototype for an Excel to ReqIF converter targeted at model-driven development is discussed here [ADT11].
Hands-on descriptions from the industrial reality are even harder to find. [Cur08] makes one notable exception, using DOORS' DXL automation facilities to read in military standards. For the ETCS specifications some, allegedly error-prone, VBA scripts are available [AHLM14, Sec. 2.1 f].

3.6 Applying this tool to other documents

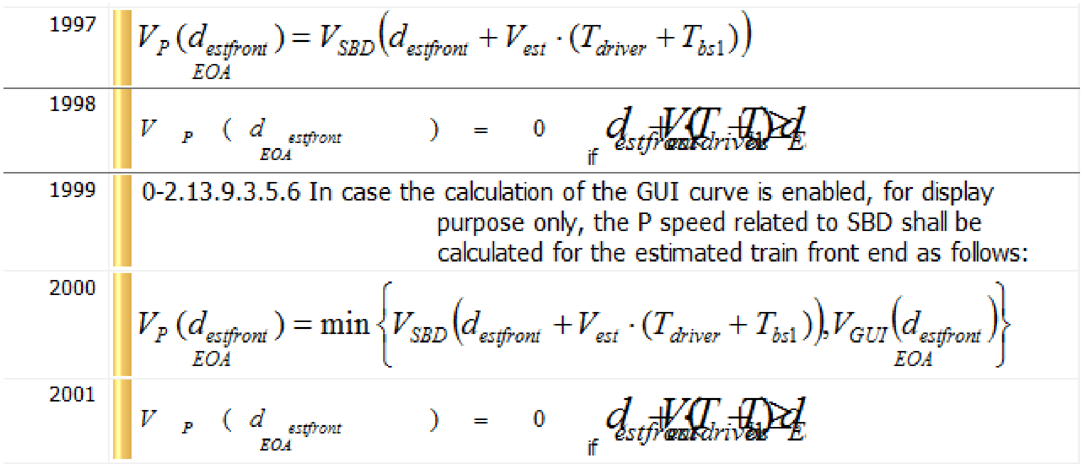
Although the preceding Sections may have created a different impression, applying the tool to documents other than Subset-026 is indeed feasible. However, there is not yet any sorcery[52] employed in its algorithms. Specific adaption to the new documents is therefore a necessary preliminary step.
To easily spot the code sections which are likely to need modifications, the tool makes use of a dedicated `@DomainSpecific` annotation (to be found in the package `helper.annotations`; see Figure 19 on page 63) which labels methods, constructors and fields exhibiting a high degree of customization to the Subset-026. For a rough idea how much code is affected by this annotation consider the following statistics: Currently the tool consists of 1065 methods, 212 constructors and 776 fields of which only 148 (approx. 7%) are tagged as `@DomainSpecific` (Counts by Eclipse's "Java search" respectively its "References" feature).

Provided that the new document is also available in the DOC format and its main structuring element is a hierarchical list, there is a good chance adaption can be performed with manageable effort. The necessary steps can be summarized as follows:

1. Go through the various, comprehensively documented parameters of `helper.Constants` and amend them as necessary. This specifically applies to elementary properties such as the delimiter character of captions and the style names for the proper detection of headings (Section 3.3.1).
   Keywords of the `implementerEnhanced`-field as listed in Appendix A.2 can also be set here.

2. Check the class `docreader.range.TitleReader`. It is used in the very beginning of the tool's processing (specifically: it constitutes the first operation in phase 2 of Section 3.1) and can therefore cause it to fail right away. Alas, due to the layout of Subset-026, it consists almost entirely of `@DomainSpecific` code.

[52]In less twaddling terms: Machine Learning. Confer with Section 4.



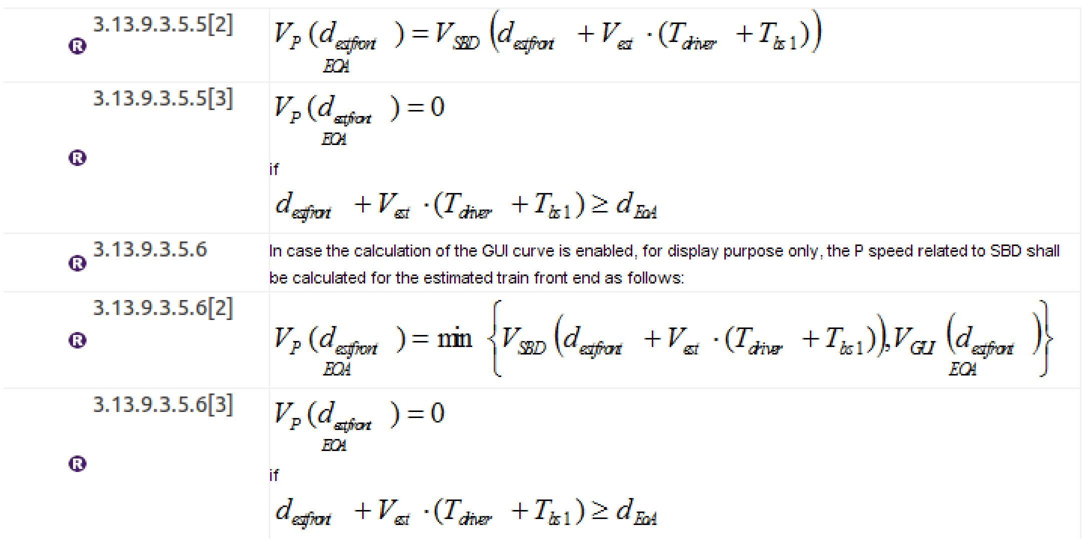
1997 $V_P(\underset{EOA}{d_{estfront}}) = V_{SBD}\left(d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1})\right)$

1998 $V \quad P \quad (\quad \underset{EOA}{d \quad estfront} \quad) \quad = \quad 0$ if $d_{estfront} + V(T_{driver} + T_{bs1}) \geq d_E$

1999 0-2.13.9.3.5.6 In case the calculation of the GUI curve is enabled, for display purpose only, the P speed related to SBD shall be calculated for the estimated train front end as follows:

2000 $V_P(\underset{EOA}{d_{estfront}}) = \min\left\{V_{SBD}\left(d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1})\right), V_{GUI}\left(\underset{EOA}{d_{estfront}}\right)\right\}$

2001 $V \quad P \quad (\quad \underset{EOA}{d \quad estfront} \quad) \quad = \quad 0$ if $d_{estfront} + V(T_{driver} + T_{bs1}) \geq d_E$



Ⓡ 3.13.9.3.5.5[2] $V_P(\underset{EOA}{d_{estfront}}) = V_{SBD}\left(d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1})\right)$

Ⓡ 3.13.9.3.5.5[3] $V_P(\underset{EOA}{d_{estfront}}) = 0$

if

$d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1}) \geq d_{EoA}$

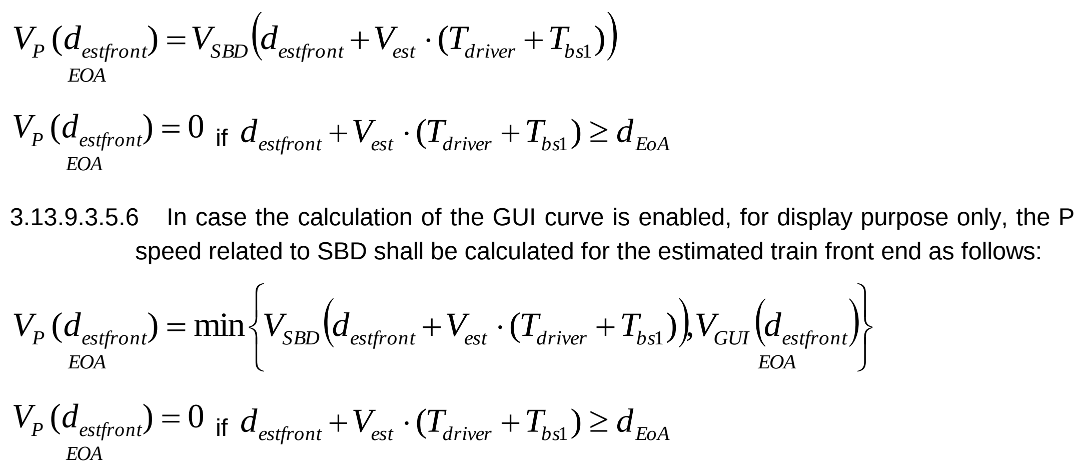
Ⓡ 3.13.9.3.5.6 In case the calculation of the GUI curve is enabled, for display purpose only, the P speed related to SBD shall be calculated for the estimated train front end as follows:

Ⓡ 3.13.9.3.5.6[2] $V_P(\underset{EOA}{d_{estfront}}) = \min\left\{V_{SBD}\left(d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1})\right), V_{GUI}\left(\underset{EOA}{d_{estfront}}\right)\right\}$

Ⓡ 3.13.9.3.5.6[3] $V_P(\underset{EOA}{d_{estfront}}) = 0$

if

$d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1}) \geq d_{EoA}$



$$V_P(\underset{EOA}{d_{estfront}}) = V_{SBD}\left(d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1})\right)$$

$$V_P(\underset{EOA}{d_{estfront}}) = 0 \text{ if } d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1}) \geq d_{EoA}$$

3.13.9.3.5.6 In case the calculation of the GUI curve is enabled, for display purpose only, the P speed related to SBD shall be calculated for the estimated train front end as follows:

$$V_P(\underset{EOA}{d_{estfront}}) = \min\left\{V_{SBD}\left(d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1})\right), V_{GUI}\left(\underset{EOA}{d_{estfront}}\right)\right\}$$

$$V_P(\underset{EOA}{d_{estfront}}) = 0 \text{ if } d_{estfront} + V_{est}\cdot(T_{driver}+T_{bs1}) \geq d_{EoA}$$

Figure 23: Result of DOORS' DOC importer (top). ProR's (middle) and Microsoft Word's (bottom) rendering is shown for comparison.

3. Replace the existing table definitions (Section 2.3.1) with substitutes matching the tables of the new document. This can be achieved by simply creating a new class which extends from `helper.subset26.tables.GenericTable` and implements its abstract methods. This new definition must then be made known through a respective mention in the list of `helper.subset26.tables.TableServiceLocator`.
4. *(optionally)* Extend the available annotations for the `implementerEnhanced`-field (Table 4 on page 50). [SGC$^+$09], for instance, contains numerous fitting proposals for the domain of legal documents.
   The respective code lives in `helper.formatting.textannotation.AnnotationBuilder`.
5. *(optionally)* Amend the fields shown in the ReqIF output as desired. This can be done by simply changing the calls in step four of the `read()` method in `docreader.DocumentReader`.

Ultimately try and run the tool on the new document. If it fails, it will most likely exit with an explanatory message and a stack trace. Both should help to easily pin down the problem.

3.7 EN 50128 tool qualification

The European Standard on railway software EN 50128 primarily targets the development principles of the actual (most likely safety-relevant) production software. However, the current 2011 version of this Standard was extended by a section on auxiliary tools and the requirements to qualify for use within an SDLC.
Such tools are divided into three classes: *T3* for programs which directly contribute to the final output (e.g. a compiler), *T1* for anything on the periphery (e.g. a text editor) and *T2* for everything in-between (e.g. tools dealing with tests of the final software) [CEN11, clauses 3.1.42 ff.]. The tool developed within the scope of this thesis is clearly a *T1*-tool since it only deals with the requirements which describe the final software. By the definition of EN 50128, this is a very abstract, not-so-much-safety-relevant area. In fact, "requirement support tools" are even explicitly stated as an example of *T1*-tools in [CEN11, clause 3.1.42].

In order to qualify a tool for any tool class, the first step is to provide a tool manual [CEN11, clause 6.7.2]. In the present case, the preceding Sections of this thesis shall be considered as such, in combination with the Javadoc documentation which comes with the sources of the tool. Subsequently, the qualification requirements mentioned in the table below [CEN11, clause 6.7.4.12], need to be accounted for, as they differ for each tool class. *T1* mandates the implementation of only a single such clause, namely [CEN11, clause 6.7.4.1]. This very generic requirement basically requests the selection of a truly suitable tool for a certain job, as opposed to a piece of software that requires a lot of error-prone manual labour accompanying its use.
In the words of EN 50128, a tool "shall be selected and demonstrated to be compatible with the needs of the application" [CEN11, line 1154]. Since the tool discussed in this thesis was specifically crafted for its job and thus by no means is a COTS software, no further explanation is needed to justify its **compatibility**. **Cooperation** of that tool with other tools, as mandated in [CEN11, lines 1151 ff.], is accounted for by the use of ReqIF as a standardized and openly documented exchange format (see Section 2.1.3). The final requirement towards **availability** "[. . .]

over the whole lifetime of the [resulting] software" [CEN11, line 1155] can only be striven for by using technologies with little vendor lock-in (e.g. there are numerous implementations of Java runtimes on the market) and opening the source code, so chances are that somebody can take over maintenance even if the original author is no longer available (see also Section 2.1).

This Section, despite its brevity, is all EN 50128 requires for a *T1* tool qualification and it may therefore be regarded as a "tool validation report" according to [CEN11, clause 6.7.3].

4 OUTLOOK

deleted epigraph – Thomas Karl, [Kar13, p. 3]

Processing unconstrained specification documents is a task with copious degrees of freedom[53]. Despite the elaborate diligence of the preceding Sections, this thesis can therefore only scratch the surface of the overall problem.
As it stands, the solution presented is suitable for the given task (see Section 5). However, there are numerous starting points for improvement. While the further evolution of specific technologies used within the tool have, for the most part, already been discussed in the respective Sections, the big picture still deserves scrutiny.

The basic approach of this thesis was clearly motivated by the solution-oriented mentality of a practical engineer. Therefore, only little effort has been spent on the theoretical foundations underlying the limits and possibilities of the various utilized techniques. This emphasis on pragmatism is notably manifested in the fundamental rule-based nature of the tool: Although an intuitive if-then-else sequence may guarantee quick and reproducible results, it is likely to prove too inflexible in the long run – especially when the number of possible inputs becomes more diverse.
One possible escape from this dilemma may be found in the broader employment of Machine learning (ML). While this is already a part of the utilized NLP-library, algorithms from this field have not yet seen any wider adoption at other places within the tool. This limitation can be mainly attributed to the lack of suitable training data and the strict orientation towards the Subset-026 which can be tackled with simple rules alone. Nevertheless, [DD95] and [Fre00] show promising approaches to the domain of layout and structure recognition in documents using ML. Possible future enhancements to the tool dedicated towards its applicability in a more generic context are therefore likely to benefit from this technology.
Yet, a first and more simply implemented step in this direction could be made by employing more generic metrics for the various detection algorithms, rather than by focusing exclusively on domain-specific facts. Consider the example of the recurring phrases elicitation (item 2 of

[53]This is, in fact, a respectful quote by a coworker at DB Netz who always found the self-chosen topic of this paper to be far too unrewarding for a Diploma Thesis.

Section 3.2.2). Currently, the associated logic expects the first mention of such a phrase to always come enclosed in quotation marks (“*first mention*”). By utilizing a statistical measure such as tf-idf (term frequency – inverse document frequency) combined with stemming as outlined in Section 3.4.2, interesting candidates for recurring phrases can be spotted irrespective of their individual formatting.
The tf-idf calculation is pretty straightforward and works as follows [MRS08, Sec. 6.2 ff]:

$$\begin{aligned}
\mathrm{tf}_{t,d} &= \text{number of occurrences of term } t \text{ in document } d && (1)\\
\mathrm{df}_{t} &= \text{number of documents containing the term } t && (2)\\
\mathrm{idf}_{t} &= \log \frac{N}{\mathrm{df}_{t}} && (3)\\
\mathrm{tfidf}_{t,d} &= \mathrm{tf}_{t,d} \cdot \mathrm{idf}_{t} && (4)
\end{aligned}$$

So essentially a *term frequency* $\mathrm{tf}_{t,d}$ (Equation 1) and a *document frequency* df_t (Equation 2) are computed. Then the latter is normalized over the number of documents N, inversed and scaled logarithmically. The result (Equation 3) is called *inverse document frequency* idf_t and constitutes a measure for the degree of importance of a term t in respect to all documents. This idf_t is eventually multiplied by the *term frequency* to obtain the final tf-idf measure (Equation 4).
As idf_t tends to be high for a rare term and much lower for a frequent one, $\mathrm{tfidf}_{t,d}$ will be a (relatively) big number if this term t occurs often within a small number of documents. By replacing the term “document” with “artifact”, it becomes clear that this is a decent measure to determine candidates for recurring phrases.
As a side note, the logarithm of Formula 3 has no deeper meaning other than to provide a nice scaling for the result (its base may therefore be altered at will). The same applies to the inversion, whose primary duty is to ensure a positive result (confer $\log \frac{x}{y} = -\log \frac{y}{x}$). Furthermore, Equation 1 gives the so-called *raw frequency* of a term. Since artifacts are meant to convey only a single idea (i.e. their `atomic` flag ought to be set for most of all cases, confer with Section 3.2.1), an even simpler binary weighing scheme may be more appropriate here.

A second area which bears great potential for the development of the tool is the domain of automated ontology / taxonomy generation. This refers to the discipline of mining conceptual relations from natural language contents (think: *has-a*, *is-a* relationships) and storing them in such a way (usually by means of a tree) that a computer can derive meaning from them. [MS00, Sec. 4] exemplifies this nicely with the varying configurations of hotel rooms in Mecklenburg-West Pomerania.
However, this technology is mostly aimed at dynamic queries such as “List all necessary preconditions for a Movement Authority” (i.e. “Movement Authority” *has-a* number of preconditions), whereas ReqIF is designed as a static data store. Thus, these two concepts do not particularly lend themselves for combination.

Besides all these high-level considerations, a number of smaller, more technical improvements are also conceivable. On the input side, above all, the lack of support for `*.docx` and the limited implementation of DOC are concerned. The latter is heavily tailored towards the specific needs of Subset-026 and therefore does not encompass structures perhaps encountered in dif-

ferent specification documents, such as support for textboxes or visual elements other than arrows (see Section 3.3.3). In addition, there is currently no extensive support for the handling of *revision marks*. Since the ERA provides specifications not only in a "sanitized" form, but also with plentiful comments (technically: *revision marks*), it may be worth extracting them. Subsequently, they may become attached to the respective artifacts and serve as some kind of a "history of origins".
On the content processing side, room for improvement lies primarily in traceability into images. Alas, this task poses tremendous difficulties which have already been explained in Section 3.3.4. Other than that, there are numerous small items worth of consideration, such as the (semi-) automated generation of standardized visualizations (e.g. Systems Modeling Language (SysML) requirement diagrams) from statistical output (Section 2.2.1), or more detailed categories for the `implementerEnhanced`-field (e.g. the various groups listed in [LGF+05, Table 4]).
This field may also be used as a foundation for the application of formal methods, such as Event-B (confer with Figure 1 on page 16). Eclipse features sophisticated tooling support for this method through its Rodin platform, which can be tightly integrated into ProR (see also [CSN+15, Sec. 6]). Since entities (single words, short phrases) referenced within Event-B commonly constitute only a part of a requirement, they may be easily mapped onto the various annotations of the `implementerEnhanced`-field (Table 4 on page 50). See [HJL14, Sec. 5] for details (referenced entities are there called *phenomena*).
Another option for making use of the provided ReqIF-files within the Eclipse framework is the employment of reqcycle [Ecl15]. This is a novel tool still under heavy development which can provide traceability links to UML- and SysML-models for use in model-driven development.

Of course, all of the above imply that future versions of the Subset-026 will not differ *fundamentally* from the currently published documents. However, despite significant potential for improvement, this is unlikely to happen anytime soon.

5 CONCLUSION

deleted epigraph – margin note in [GKP93, p. 142]

This thesis proposed a novel software tool for the processing of specification documents published in the Microsoft Word file format. In its current version, the tool was specifically crafted to meet the demands of Subset-026, a core specification in the ETCS domain. Numerous examples demonstrated its fitness to process all eight independent chapters of this document in a fully automated manner.

In essence, there were two groups of stakeholders who biased the final outcome of this paper: On the one hand, the practitioners within openETCS who desperately needed a workable traceability foundation for the implementation of the EVC. And on the other hand, the academic researchers at TU Dresden who always found traceability rather insipid and favored a deeper understanding of the specification contents, instead. This thesis aims to satisfy both.

	Problem	Tool mitigation
1.	There are too many of them.	defined structure (Section 3.2.1) and supportive links (Section 3.2.2)
2.	They are unstable.	possibility of computing deltas between different requirement versions (confer with Section 2.3.3) using XML-aware diff-tools such as the comparison functionality of ProR or Altova's DiffDog software.
3.	They are ambiguous.	detection of inappropriate wording (Table 4 on page 50) and various context-aware warnings (Table 3 on page 49)
4.	They are incomplete.	decomposition into a tree to simplify the identification of missing elements (Section 2.2.1)

Table 5: Common problems with requirements according to [Gla98, p. 21]

Despite its rather generic approach, Section 2 primarily targeted the practitioners. Besides extensively covering the computation of unique, context-sensitive requirement identifiers, so-called *tracestrings* that make the foundation to traceability, this part also showed various metrics that can be extracted from specifications independent of their domain. In Section 3 the actual implementation of the tool was presented, placing special emphasis on the specific challenges imposed by the natural language contents of the processed documents. This approach generated a much stronger formalization of those contents, which is reflected in the clear structure of the resulting ReqIF files.
Accordingly, the presented tool completely fulfills the requirements of the problem definition ("Aufgabenstellung" on page 7) except two minor points: There is currently no direct integration into the Eclipse platform, respectively its modelling framework, due to a lack of such demand within openETCS. This point is mainly a usability issue and has no influence on the quality of the output. Secondly, the tool does not yet feature any contradiction detection worth its name. Section 3.4.2 details the reasons why this was not feasible within the scope of this thesis.

Nonetheless, the generated ReqIF-output constitutes a major improvement over the original Microsoft Word files. Not only can ReqIF be read and interpreted in a standardized way by a multitude of different requirements management applications. But those files also finally allow to tackle the most common and tedious causes of trouble with specification documents using decent tool support.
Table 5 identifies four main groups of such requirement-related problems and lists the respective countermeasures offered by the tool. However, due to the postprocessing nature of the software, these can at best be mitigative actions, as their root cause still remains deeply buried in the swamp of European bureaucracy.
Indeed, this circumstance can only be tackled in the long run. Above all, it would take the wider adoption of ReqIF instead of Microsoft Word. Thus, not only implementers would benefit from its superior extent of formalization but also its authors and eventually all the other parties involved.
An application of the tool to other domains is also conceivable and Section 3.6 detailed the necessary preparatory steps. The successful processing of Subset-023 (Baseline 3.0.0) of the ETCS specification can serve as an encouraging example here. In this context, a future integration of the tool into a bigger framework dedicated to document interpretation may be a worthwhile prospect [CMD$^+$04].

DECLARATION OF AUTHORSHIP

I hereby certify that the work presented here is, to the best of my knowledge, original and the result of my own investigations, except where otherwise indicated.

Hierdurch erkläre ich, dass ich die von mir am heutigen Tage eingereichte Diplom-Arbeit selbstständig verfasst und andere als die angegebenen Hilfsmittel nicht benutzt habe.

Dresden, June 19th, 2015

A APPENDICES

deleted epigraph – Abstract of 2013's April Fools' Day RFC, [BKR13]

A.1 Postprocessing statistics data

A tool run (Section 3.1) will produce a `statistics`-subdirectory with two CSV-files:

nodes.csv which contains a line for each processed artifact (*SpecObject*) and thus represents the *nodes* of the tree in Figure 5 on page 26. The meaning of the individual fields is as follows:

Index	Name	Type	Description
1	ID	`String`	a unique ID (equal to `Spec Object | Identifier` shown in Figure 12 on page 48)
2	Label	`String`	*plainText* of the artifact (as in ReqIF)
3	Level	`Integer`	Hierarchical level of this artifact
4	Implement	`Boolean`	Value of the *implement*-field of this artifact (as in ReqIF)

Fields *Level* and *Implement* are not strictly required by any subsequent tool but can be used for filtering purposes.

edges.csv where each line corresponds to an *edge* that connects two *nodes*. The meaning of the individual fields is as follows:

Index	Name	Type	Description
1	Source	`String`	unique ID of the source of this edge
2	Target	`String`	unique ID of the target of this edge
3	Weight	`Float`	a number which distinguishes different types of edges. `3.0` denotes a hierarchical relation, `2.0` a cross-reference and `1.0` a link between recurring phrases (see Section 3.2.2 for a definition of the latter two).

The IDs of the fields *Source* and *Target* match with an ID from `nodes.csv`. Again, the field *Weight* is not strictly required for subsequent processing but can be used for filtering.

A.1.1 Clean up spurious external links

The tool's algorithms do produce a few false-positive links (read: *SpecRelations* in ReqIF terminology, Section 3.2.2), which may falsify some means of statistical postprocessing. Specifically,

the presence of those links may severely confuse automatic graph layout algorithms as introduced in Section 2.2.1.
Here is how to get rid of them:

1. Open `nodes.csv` and jump to the end.
2. While scanning backwards through the file, look for spurious nodes. Note down their identifier (first column) and delete the entire line which contains them.
3. Stop scanning when you see the first node belonging to the current chapter (external nodes are always written after the internal nodes).
4. Save and close the file.
5. Open `edges.csv`
6. Search for the identifiers of the second step in the third column.
7. Delete the entire line if a match is found.
8. Save and close the file.

A.1.2 Merge data of several tool runs

This Section briefly describes how statistics data gained from several runs of the tool (e.g. after processing different chapters of the Subset-026) can be combined into one output.
Example code is meant to be run in a POSIX-compliant Unix environment using a shell.

1. For each `nodes.csv`: Remove the first line and concatenate the remainder to a big file called `nodesCombined.csv`.
   E.g.: `sed '1d' nodes.csv >> nodesCombined.csv`
2. For each `edges.csv`: Remove the first line and concatenate the remainder to a big file called `edgesCombined.csv`.
   E.g.: `sed '1d' edges.csv >> edgesCombined.csv`
3. Remove duplicate nodes from `nodesCombined.csv`. These are external links encountered several times in different chapters. However, we only want to keep the node which represents the actual target.
   E.g.: `sort --field-separator="," -k 1,1 nodesCombined.csv | tac | sort -u --field-separator="," -k 1,1 > nodesCombinedCleaned.csv`

   (this is: sort by the first column, then reverse the result and ultimately sort again, retaining only the first entry – which thanks to the reversing will always be the longest text and thus the real target – of any duplicates).
4. Prefix `nodesCombinedCleaned.csv` with the original CSV-header.
   E.g.: `head -n1 nodes.csv | cat - nodesCombinedCleaned.csv > nodesFinal.csv`
5. Prefix `edgesCombinded.csv` with the original CSV-header.
   E.g.: `head -n1 edges.csv | cat - edgesCombined.csv > edgesFinal.csv`
6. `nodesFinal.csv` and `edgesFinal.csv` can now be used as an input to Gephi, Excel, R, . . .

A.2 Subset-026 keywords

The keywords must be interpreted as case-insensitive regexes with Java-specific escapes. Matching is performed from left to right (so the first matching keyword will cause all the remaining ones to be skipped in that particular run). Spaces are given as ␣.

A.2.1 Legal obligation

The *mandatory* and *optional* keywords are taken from clause 1.7.1.1 in Subset-026. The rest is a manual compilation.

mandatory	`SHALL`
optional	`MAY`
unknown	`CAN(?:NOT)?` , `MUST` , `WILL` , `MIGHT` , `(?<!MAY␣)OPTIONALLY`

A.2.2 Weak words

These are words unwelcome in a requirement text because of their inherent lack of precision. Other literature may also call these "vague words" or "weasel words".

Weak words from literature These were compiled from [Kni12], [Dup98], [FFGL01, Table 3], [WRH97, p. 164], [ISO11, clause 5.2.7]:

`above` , `adequate` , `anything` , `approximately` , `as␣soon␣as` , `bad` , `believe` , `below` , `best` , `better` , `but␣not␣limited␣to` , `clear` , `cyclically` , `easy` , `eventually` , `extremely` , `feel` , `generally` , `good` , `hope` , `if␣appropriate` , `if␣needed` , `if␣possible` , `immediately` , `in␣round␣numbers` , `more␣or␣less` , `overall` , `possibly` , `recent` , `repeatedly` , `rough` , `seem` , `significant` , `something` , `strong` , `think` , `useful` , `very(?!␣(?:first|last))` , `worst`

Weak words from Subset-026 These were compiled by hand:

`all␣necessary` , `at␣(?:minimum|least)` , `defined␣time` , `e\\.g\\.` , `for␣example` , `etc\\.` , `even␣(?:if|when)` , `if␣necessary` , `no[nt]␣exhaustive(?:ly)?` , `some␣(?:information|situation(?:s|\\(s\\))?)` , `temporarily` , `once␣(?:\\w+\\s)+?is␣terminated` , `other␣(?:\\w+\\s)+?sources` , `certain␣moment(?:s|\\(s\\))?` , `obviously` , `hereafter` , `tends␣(?:to)?` , `mostly` , `suddenly` , `accidental` , `(?:(?:an)?other|different)␣reason(?:s|\\(s\\))?` , `continuously` , `\\.\\.\\.` , `when␣needed`

A.2.3 Other keywords for the *implementerEnhanced*-field

These were all compiled by hand.

Condition

`if` , `when(?:␣applicable)?` , `in␣case(?:␣of)?` , `whether` , `where␣available`

Loop

`For␣(?:all|each|every)` , `again` , `repeat(?:ed(?:ly)?)?` , `repetition(?:s|\\(s\\))?`

Time

`while` , `during` , `until` , `after` , `not␣(?:\\w+\\s)?yet` , `waiting␣time` , `time␣delay` , `timer?` , `delay` , `wait(?:ing)?` , `as␣long␣as`

Again The authors of the Subset-026 often tend to write *re-evaluate* instead of *reevaluate*. These regexes make use of that:

`re-\\w+` , `revalidat(?:ed?|ion)` , `reenter(?:ed)?`

External Entities These were mostly compiled from ETCS Subset-023, which is a glossary to other Subsets:

`driver(?:[’']s)?(?!␣ID)` , `signalman(?:[’']s)?` , `external(?:␣(?:interface|device))?` , `(?-i:TRK)` , `trackside` , `(?-i:RBC)(?!␣ID)` , `Radio␣Block␣Cent(?:er|re)` , `(?-i:LEU)` , `Line␣?side␣electronic␣unit` , `National␣system` , `(?-i:RIU)` , `Radio␣In-?fill␣Unit` , `(?-i:LRBG(?:s|\\(s\\))?)` , `(?:Last␣Relevant␣)?balise␣group(?:s|\\(s\\))?` , `(?:Euro)?(?:balise|loop)(?:s|\\(s\\))?(?!␣(?:antenna|telegram))` , `(?-i:LTM)` , `Loop␣Transmission␣Module`

Self references

`ERTMS/ETCS␣on-?board(?:␣equipment|unit)?` , `on-?board␣(?:equipment|unit)`

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

GLOSSARY

Numbers following the description text indicate the pages where the respective term is used.

Apache POI An open-source Java library to process Microsoft Office file formats. 38, 64, 65

BLOB Binary Large Object. 23, 37, 44

COM Component Object Model. A methodology invented by Microsoft to ease software reusability. 90

COTS commercial off-the-shelf. 73, 76

CSS Cascading Style Sheets. 27

CSV comma-separated values. 22, 25, 43, 44, 83, 84

digraph When talking about graph theory (Section 2.2.1) this is just short for a generic directed graph. However, regarding legacy computer systems, the term describes two adjacent characters which are to be considered as one. For example the Pascal programming language supports writing `(*` and `*)` instead of `{` and `}`. The latter is the intended meaning in Section 3.3.4. 25, 61

DOC binary Microsoft Word file format used since Word 97. 3, 9, 17–20, 22, 23, 36–38, 40, 41, 44–46, 49, 51, 52, 54, 56, 57, 61, 62, 64–66, 73–75, 78–80

DOORS Dynamic Object Oriented Requirements System. A software by IBM for requirements management. For this thesis version 9.6.1 was used. 16, 25, 33, 45, 47, 55–57, 73–75, 89

DXL DOORS eXtension Language. A scripting language to automate DOORS. 74

Eclipse Mainly a development environment for Java and other computer programming languages. Through its versatile plugin-system it may also be used as a platform to integrate different small tools under a common interface. 79, 80, 90

EMF Windows Enhanced Metafile. A refined version of WMF introduced with Windows NT 3.1. See [Mic15a] for details. 44

EN 50128 A European Standard on software safety in the railway domain. Official title: *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*. [CEN11] contains the respective bibliography entry. 9, 14, 73, 76, 77

ERA European Railway Agency. 14, 17, 79

ERTMS European Rail Traffic Management System. Umbrella term for ETCS and GSM-R. 13, 89

ETCS European Train Control System. 3, 13–17, 23, 40, 46, 48, 52, 73, 74, 79, 80, 86, 89, 91

EU European Union. 14

EUG ERTMS Users Group. 13

EVC European Vital Computer. 14, 79, 90

Event-B A formal method for system modelling based on the B method. 79

FAT File Allocation Table. 37

GDI Graphics Device Interface. A component of the Microsoft Windows operating systems to perform graphical output. See [Mic15c] for details. 44

GSM-R Global System for Mobile Communications - Rail. 89

GUID Globally Unique Identifier. 40

IBM International Business Machines Corporation. 55, 56, 89

Javadoc A documentation generator for Java source code from formatted text snippets. 36, 52, 58, 76

jUnit A unit testing framework for the Java programming language. 41, 54, 70

LZB Linienzugbeeinflussung. 14

MathML Mathematical Markup Language. An XML-based format for storing equations. There are two dialects: *Content MathML* which is more focused on semantics, and *Presentation MathML* which is more oriented towards layout. 45

ML Machine learning. Umbrella term for algorithms that can learn from a set of data *A* and then make predictions on a new set of data *B* on this basis. 9, 74, 77

MTEF Math Type Equation Format. Proprietary file format used by Design Science's MathType software, the foundation of the Equation editor in Microsoft Word [Des14]. 37, 45, 90

NLP Natural Language Processing. 9, 41, 43, 50, 64, 69–72, 77

OCR Optical Character Recognition. 61

OLE Object Linking and Embedding. A proprietary Microsoft technology which allows to embed documents of different types into each other based on COM. 37, 44, 45, 54, 61

OMML An XML-based successor to MTEF. Specified in [ISO12, clauses 22.1.2.777 ff]. 45

openETCS Research and Development project led by Deutsche Bahn Netz AG and aimed at creating a manufacturer-independent, open-source EVC software implementation. 14, 15, 73, 79, 80

OS Operating System. 41

PCFG Probabilistic Context-Free Grammar. 70

PCRE Perl Compatible Regular Expressions. A regular expression with the same syntax and semantics as in Perl 5. 39, 70

PDF Portable Document Format. 17, 18, 20, 23, 40, 74

PNG Portable Network Graphics. 37, 43, 44

POSIX Portable Operating System Interface. 84

ProR An Eclipse plugin which can read and process ReqIF files.
For this thesis version 0.13.0.201505160302 with the "formalmind Studio"-extension in version 1.0.0.201505161006 was used. 45–48, 51, 52, 54, 55, 59, 73, 75, 79, 80

regex Regular Expression. 9, 35, 36, 38, 39, 59, 69, 70, 73, 85, 86

ReqIF Requirements Interchange Format. A standardized, XML-based file format. 3, 9, 21–24, 27, 35, 40, 41, 43–46, 51, 53–57, 62, 66, 70, 74, 76, 78–80, 83, 90

Reqtify A requirements management tool by Dassault Systèmes. 15, 16, 73

RFC Request for Comments. 83

RM Requirements Management. 3, 9, 15, 16, 22, 27, 28, 33, 41, 43–46, 55, 59, 71–73, 80, 90

RTF Rich Text Format. 45

SCADE Safety-Critical Application Development Environment. A proprietary suite for the development of safety-critical software by Esterel/Ansys. 15

SDLC systems development life cycle. 52, 76

SIL Safety Integrity Level. 14

singleton A standard term in object-oriented programming for a class which can only be instantiated once. 46, 64

SLOC Source lines of code. A metric for the size of a computer program. `sloccount` available at [Whe12] was used for its computation. 41

Subset-026 A part of the ETCS specifications which deals with its core functionality. In its own words: "The purpose of this document is to specify the unified European Train Control System (ETCS) from a technical point of view" (clause 1.5.1.1). The version discussed in this thesis can be obtained here [Eur12]. 3, 9, 13–15, 17, 23, 25–28, 33, 35, 37, 38, 41, 42, 44, 45, 52, 54–59, 61, 62, 64–66, 69–71, 73, 74, 77–79, 84–86

SysML Systems Modeling Language. A graphical modelling language on the basis of UML. 79

UML Unified Modeling Language. 18, 22, 64, 67, 79, 91

UNISIG Union Industry of Signalling. 13, 14

URI Uniform Ressource Identifier. 23

V & V Verification & Validation. The process of checking that a piece of software meets its specifications (Verification) and fulfills its intended purpose (Validation). 15, 23, 73

VBA Visual Basic for Applications. 24, 73, 74, 88

V-Model An extension to the waterfall-model (i.e. one step after another) whose individual steps are usually drawn in the shape of a V. Each step on the right stroke of that V depends on input from a step on the left stroke. Widely used for the different phases of traditional software development. See [CEN11, Fig. 4] for a graphical representation. 15, 22, 23, 73

WMF Windows Metafile. An image format for both vector- and bitmap-components in use since Windows 3.0. See [Mic15b] for details. 44, 89

XHTML Extensible Hypertext Markup Language. 27, 28, 43, 55, 57

XML Extensible Markup language. A language for hierarchically structuring a text file. 3, 18, 22, 23, 27, 40, 45, 51, 74, 80, 90, 91

XPath XML Path Language. 27, 51

XSD XML Schema Definition. 27

Terms specific to this thesis

(traceable) artifact Every entity of the input file which can be processed individually and thus has a tracestring attached. The number of these artifacts depends on the granularity. In their collectivity they form a superset to all the requirements captured in the input file. See the beginning of Section 2.3 for a more in-depth explanation. 23–25, 27, 28, 31–38, 40, 41, 43, 45, 46, 48–55, 57–59, 61, 62, 64, 66, 69–74, 78, 79, 83, 92

granularity A measure for the size of a single artifact and thus the precision of the overall traceability. See also [EGHB07, Sec. 2.1]. 18, 28, 31, 38, 40, 53, 54, 60, 92

numberText Microsoft terminology for the displayed number of a list item. I.e for "1.2.3 Some text" *1.2.3* would be the `numberText` of this list item. 28, 40, 51, 53, 54, 58, 65, 66

traceability The ability of knowing why an entity exists (backward traceability) and where it is used (forward traceability). See Section 2.2 and the definitions in [IEE98, Sec. 4.3.8] and [CEN11, clause D.58]. 9, 15, 17, 23, 28, 38, 41, 46, 51, 79, 80, 92

tracestring A unique identifier attributed to each traceable artifact. See also the more thorough definition in Section 2.3. 20, 27, 31–40, 43, 46, 49, 52, 55, 57, 60, 64, 66, 68, 69, 80, 92

BIBLIOGRAPHY

[ADB+15] ALLISON, Tim; DORKA, Moritz; BEDNÁRIK, Filip; BURCH, Nick ; MATTMANN, Chris A.: *[TIKA-1315] Basic list support in WordExtractor*. `https://issues.apache.org/jira/browse/TIKA-1315`. Version: 2015, last checked: 2015-06-05

[ADT11] ADEDJOUMA, Morayo; DUBOIS, Hubert ; TERRIER, François: Requirements exchange: From specification documents to models. In: *Proceedings - 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*. Las Vegas, NV : IEEE Comput. Soc. Press, 2011. – ISBN 978–0–7695–4381–9, 350–354

[AHLM14] ASBACH, Lennart; HUNGAR, Hardi; LEMMER, Karsten ; MEYER ZU HÖRSTE, Michael: Formalisation of test procedures for a high level of automation. In: *Signal+Draht* 106 (2014), No. 9, pp. 63–69. – ISSN 0037–4997

[And15] ANDERSPITMAN: *Former Boeing Engineer on an Integer Overflow Bug in the 787 Dreamliner and how that relates to Traceability*. `https://news.ycombinator.com/item?id=9477941`. Version: 2015, last checked: 2015-05-03

[ASB+08] ALVES, V.; SCHWANNINGER, C.; BARBOSA, L.; RASHID, a.; SAWYER, P.; RAYSON, P.; POHL, C. ; RUMMLER, a.: An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In: *2008 12th International Software Product Line Conference* (2008), pp. 67–76. `http://dx.doi.org/10.1109/SPLC.2008.18`. – DOI 10.1109/SPLC.2008.18. ISBN 978–0–7695–3303–2

[Ben13] BENDER, Emily M.: *Linguistic Fundamentals for Natural Language Processing : 100 Essentials from Morphology and Syntax*. Morgan & Claypool Publishers, 2013. – ISBN 978–1–6270–5012–8

[BHJ09] BASTIAN, M; HEYMANN, S ; JACOMY, M: Gephi: An Open Source Software for Exploring and Manipulating Networks. In: *Proceedings of the Third International ICWSM Conference*. San Jose : Association for the Advancement of Artificial Intelligence, 2009, pp. 361–362

[BKR13] BARNES, R.; KENT, S. ; RESCORLA, E.: *RFC 6919 - Further Key Words for Use in RFCs to Indicate Requirement Levels*. `https://tools.ietf.org/html/rfc6919`. Version: 2013, last checked: 2015-05-22

[Bur14] BURETTE, Thomas: *So You Want To Write Your Own CSV code?* `http://tburette.github.io/blog/2014/05/25/so-you-want-to-write-your-own-CSV-code/`. Version: 2014, last checked: 2015-06-15

[CCM+09] CAVADA, Roberto; CIMATTI, Alessandro; MARIOTTI, Alessandro; MATTAREI, Cristian; MICHELI, Andrea; MOVER, Sergio; PENSALLORTO, Marco; ROVERI, Marco; SUSI, Angelo ; TONETTA, Stefano: Supporting Requirements Validation: The EuRailCheck Tool. In: *2009 IEEE/ACM International Conference on Automated Software Engineering* (2009), November, 665–667. `http://dx.doi.org/10.1109/ASE.2009.49`. – DOI 10.1109/ASE.2009.49. ISBN 978–1–4244–5259–0

[CCM+10] CHIAPPINI, A; CIMATTI, A; MACCHI, L; REBOLLO, O; ROVERI, M; SUSI, A; TONETTA, S ; VITTORINI, B: Formalization and validation of a subset of the European Train Control System. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10* vol. 2. New York, New York, USA : ACM Press, 2010. – ISBN 978–1–6055–8719–6, 109–118

[CEN11] CENELEC: *EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*. 2011

[CMD+04] CLAVIER, Eric; MASINI, Gérald; DELALANDRE, Mathieu; RIGAMONTI, Maurizio; TOMBRE, Karl ; GARDES, Joël: DocMining: A Cooperative Platform for Heterogeneous Document Interpretation According to User-Defined Scenarios. In: LLADÓS, Josep (publ.); KWON, Young-Bin (publ.): *GREC 2003*, Springer, 2004. – ISBN 978–3–540–25977–0, pp. 13–24

[CRST10] CIMATTI, Alessandro; ROVERI, Marco; SUSI, Angelo ; TONETTA, Stefano: Formalization and Validation of Safety-Critical Requirements. In: BUJORIANU, Manuela (publ.); FISHER, Michael (publ.): *Workshop on Formal Methods for Aerospace* vol. 20. Eindhoven : Electronic Proceedings in Theoretical Computer Science, March 2010. – ISSN 2075–2180, 68–75

[CSN+15] CAVALLI, Ana; SANTOS, Joao; NGUYEN, Huu-Nghia; BEHRENS, Marc ; RIEGER, Stefan: *D.4.2.1 1st interim V & V report on the applicability of the V&V approach to the formal abstract model*. `https://github.com/openETCS/validation/blob/45adc586dea78c5a824af8a0aa3bd1b8a668230d/Reports/D4.2/D4.2.1-VV-Model/D4.2.1.pdf`. Version: 2015, last checked: 2015-06-15

[Cur08] CURTIN, Stephen: *MICROSOFT WORD EXPORT TO DOORS FOR MILITARY STANDARD (MIL-STD) SPECIFICATIONS*. `http://download-na.telelogic.com/download/ugcagenda/Stephen_Curtin_Microsoft_Word_Export_to_DOORS.pdf`. Version: 2008, last checked: 2015-05-24

[DD95] DENGEL, Andreas; DUBIEL, Frank: Clustering and classification of document structure – a machine learning approach. In: *Proceedings of 3rd International Conference on Document Analysis and Recognition* vol. 2, 1995. – ISBN 0–8186–7128–9, pp. 587–591

[Des99] DESIGN SCIENCE: *How MTEF is Stored in Files and Objects*. `http://web.archive.org/web/20010304111449/http://mathtype.com/support/tech/MTEF_storage.htm`. Version: 1999, last checked: 2015-06-14

[Des14] DESIGN SCIENCE: *MathType's Equation Format (MTEF)*. `https://www.dessci.com/en/reference/sdk/#MTEF`. Version: 2014, last checked: 2015-05-14

[Dor15] DORKA, Moritz: *subset026reader: first public release*. Version: 2015. `http://dx.doi.org/10.5281/zenodo.18706`. – DOI 10.5281/zenodo.18706

[Dup98] DUPRÉ, Lyn: *BUGS in Writing : A Guide to Debugging Your Prose*. 9th Edition. Addison-Wesley Longman, Inc., 1998. – ISBN 0–201–37921–X

[Ecl15] ECLIPSE FOUNDATION: *ReqCycle | PolarSys - Open Source tools for the development of embedded systems*. `https://www.polarsys.org/projects/polarsys.reqcycle`. Version: 2015, last checked: 2015-06-10

[ECM06] ECMA: *Standard ECMA-376 - Part 4: Markup Language Reference*. `http://www.ecma-international.org/publications/standards/Ecma-376.htm`. Version: 1st Edition, 2006, last checked: 2015-06-15

[EGHB07] EGYED, Alexander; GRUNBACHER, Paul; HEINDL, Matthias ; BIFFL, Stefan: Value-Based Requirements Traceability: Lessons Learned. In: *15th IEEE International Requirements Engineering Conference (RE 2007)* (2007), 115–118. `http://dx.doi.org/10.1109/RE.2007.16`. – DOI 10.1109/RE.2007.16. ISBN 0–7695–2935–6

[EJ12] EBERT, Christof; JASTRAM, Michael: ReqIF: Seamless requirements interchange format between business partners. In: *IEEE Software* vol. 29, 2012. – ISSN 07407459, pp. 82–87

[ERT15] ERTMS SOLUTIONS: *ERTMSFormalSpecs – Open Source*. `https://www.ertmssolutions.com/products/ertmsformalspecs-open-source/`. Version: 2015, last checked: 2015-05-04

[Est15] ESTEREL TECHNOLOGIES SA: *SCADE LifeCycle Requirements Management Gateway*. http://www.esterel-technologies.com/products/scade-lifecycle/requirements-management-traceability/scade-lifecycle-requirements-management-gateway/. Version: 2015, last checked: 2015-05-14

[Eur12] EUROPEAN RAILWAY AGENCY: *Subset-026, Baseline 3.3.0*. http://www.era.europa.eu/Document-Register/Documents/Index004-SUBSET-026.zip. Version: 2012, last checked: 2015-05-29

[Eur15] EUROPEAN RAILWAY AGENCY: *Change Control Management*. http://www.era.europa.eu/Core-Activities/ERTMS/Pages/Change-Control-Management.aspx. Version: 2015, last checked: 2015-06-07

[Feu12] FEUSER, Johannes: *Open Source Software for Train Control Applications and its Architectural Implications*, Universität Bremen, PhD-thesis, 2012. http://nbn-resolving.de/urn:nbn:de:gbv:46-00103095-16

[FFGL01] FABBRINI, F.; FUSANI, M.; GNESI, S. ; LAMI, G.: The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. In: *Proceedings 26th Annual NASA Goddard Software Engineering Workshop*, IEEE Comput. Soc, 2001. – ISBN 0–7695–1456–1, 97–105

[FFT+08] FAZZINGA, B; FLESCA, S; TAGARELLI, A; GARRUZZO, S ; MASCIARI, E: A wrapper generation system for PDF documents. In: *Proceedings of the ACM Symposium on Applied Computing*, ACM Press, 2008. – ISBN 978–1–5959–3753–7, 442–446

[Fir05] FIRESMITH, Donald: Are Your Requirements Complete? In: *Journal of Object Technology* 4 (2005), No. 1, 27–43. http://www.jot.fm/issues/issue_2005_01/column3/

[Fis96] FISHMAN, Charles: *They Write the Right Stuff*. http://www.fastcompany.com/28121/they-write-right-stuff. Version: 1996, last checked: 2015-06-04

[Fre00] FREITAG, D: Machine learning for information extraction in informal domains. In: *Machine learning* 39 (2000), No. 2-3, 169–202. http://dx.doi.org/10.1023/A:1007601113994. – DOI 10.1023/A:1007601113994. – ISSN 1573–0565

[GF94] GOTEL, O.C.Z.; FINKELSTEIN, C.W.: An analysis of the requirements traceability problem. In: *Proceedings of IEEE International Conference on Requirements Engineering*, IEEE Comput. Soc. Press, 1994. – ISBN 0–8186–5480–5, 94–101

[GKNV93] GANSNER, Emden; KOUTSOFIOS, Eleftherios; NORTH, Stephen ; VO, Kiem P.: Technique for drawing directed graphs. In: *IEEE Transactions on Software Engineering* 19 (1993), No. 3, pp. 214–230. http://dx.doi.org/10.1109/32.221135. – DOI 10.1109/32.221135. – ISSN 0098–5589

[GKP93] GRAHAM, Ronald L.; KNUTH, Donald E. ; PATASHNIK, Oren: *Concrete mathematics*. 2nd Edition. Reading, MA : Addison-Wesley, 1993. – ISBN 978–0–2011–4236–5

[Gla98] GLASS, Robert L.: *Software Runaways*. Upper Saddle River : Prentice Hall, 1998. – ISBN 0–13–673443–X

[GM89] GORDON, Gary; MCMAHON, Elizabeth: A greedoid polynomial which distinguishes rooted arborescences. In: *Proceedings of the American Mathematical Society* 107 (1989), No. 2, pp. 287–298. http://dx.doi.org/10.1090/S0002-9939-1989-0967486-0. – DOI 10.1090/S0002–9939–1989–0967486–0. – ISSN 0002–9939

[Gra09] GRALLA, Christoph: *Zur Gestaltung einer ETCS-Migration eines Eisenbahnverkehrsunternehmens*, Technische Universität Braunschweig, Dissertation, 2009. http://www.digibib.tu-bs.de/?docid=00030082

[HB07] HASSAN, Tamir; BAUMGARTNER, Robert: Table recognition and understanding from PDF files. In: *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR* vol. 2, 2007. – ISBN 0–769–52822–8, pp. 1143–1147

[Hei10] HEIDENREICH, Martin: Metriken und Werkzeugunterstützung zur Überprüfung von Anforderungen. In: *ObjektSpektrum* (2010), No. RE/2010, 1–4. `http://www.sigs-datacom.de/fachzeitschriften/objektspektrum/archiv/artikelansicht.html?tx_mwjournals_pi1[pointer]=0&tx_mwjournals_pi1[mode]=1&tx_mwjournals_pi1[showUid]=6651`. – ISSN 0945–0491

[Hir15] HIRONDELLE SYSTEMS: *Java Practices -> Package by feature, not layer*. `http://www.javapractices.com/topic/TopicAction.do?Id=205`. Version: 2015, last checked: 2015-07-04

[HJL14] HALLERSTEDE, Stefan; JASTRAM, Michael ; LADENBERGER, Lukas: A method and tool for tracing requirements into specifications. In: *Science of Computer Programming* 82 (2014), 2–21. `http://dx.doi.org/10.1016/j.scico.2013.03.008`. – DOI 10.1016/j.scico.2013.03.008. – ISSN 0167–6423

[HMBMn11] HERRANZ, Ángel; MARPONS, Guillem; BENAC, Clara ; MARIÑO, Julio: *Mechanising the Validation of ERTMS Requirements and New Procedures*. `https://www.fi.upm.es/catedra-ibmrational/sites/www.fi.upm.es.catedra-ibmrational/files/MechanisingtheValidationofERTMS.pdf`. Version: 2011, last checked: 2015-06-15

[Hoo15] HOOD GROUP: *DESIRe®*. `http://www.hood-group.com/en/products/tools/requirements-engineering/desirer/`. Version: 2015, last checked: 2015-06-07

[IBM13] IBM: *IBM Knowledge Center - Working with Microsoft Word documents*. `http://www-01.ibm.com/support/knowledgecenter/SSSHCT_7.1.0/com.ibm.reqpro.help/w_documents/working_ms_word/t_work_word_docs.html?lang=de`. Version: 2013, last checked: 2015-06-07

[IBM15] IBM: *Display modes*. `http://www-01.ibm.com/support/knowledgecenter/SSYQBZ_9.6.0/com.ibm.doors.requirements.doc/topics/c_displaymodes.html?lang=en`. Version: 2015, last checked: 2015-05-08

[IEE98] IEEE: *IEEE Recommended Practice for Software Requirements Specifications*. Version: revised Edition, October 1998. `http://dx.doi.org/10.1109/IEEESTD.1998.88286`. – DOI 10.1109/IEEESTD.1998.88286. ISBN 0–738–10332–2

[ISO11] ISO/IEC/IEEE: *ISO/IEC/IEEE 29148:2011 Systems and software engineering - Life cycle processes - Requirements engineering*. Version: 1st Edition, 2011. `http://dx.doi.org/10.1109/IEEESTD.2011.6146379`. – DOI 10.1109/IEEESTD.2011.6146379

[ISO12] ISO: *ISO/IEC 29500-1 Information technology — Document description and processing languages — Office Open XML File Formats — Part 1: Fundamentals and Markup Language Reference*. 3rd Edition. Geneva, September 2012. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=61750`

[Jac95] JACKSON, Michael: *Software Requirements & Specifications*. Wokingham : Addison-Wesley, 1995. – ISBN 0–201–87712–0

[JPD14] JASTRAM, Michael; PETIT-DOCHE, Marielle: *Report on the Final Choice of the Primary Toolchain*. `https://itea3.org/project/workpackage/document/download/1469/11025-openETCS-WP-7-D71.pdf`. Version: November 2014

[Kar13] KARL, Thomas: Standards through operational requirement specifications. In: *Signal+Draht* 105 (2013), No. 7+8. – ISSN 0037–4997

[Kle02] KLENNER, Markus: *Eisenbahn und Politik : Vom Verhältnis der europäischen Staaten zu ihren Eisenbahnen*, Universität Wien, Dissertation, 2002

[Kni12] KNIGHT, Robert M.: Red Flags and No-Nos. In: *Writing Public Prose : How to Write Clearly, Crisply, and Concisely*. Portland : Marion Street Press, LLC, 2012. – ISBN 978–1–936–86327–3, Chapter 8, pp. 99–108

[Kof05a] KOF, Leonid: An Application of Natural Language Processing to Domain Modelling – Two Case Studies. In: *International Journal of Computer Systems Science & Engineering* 20 (2005), No. 1, pp. 37–52. – ISSN 0267–6192

[Kof05b] KOF, Leonid: Natural Language Processing: Mature Enough for Requirements Documents Analysis? Version: 2005. `http://dx.doi.org/10.1007/11428817_9`. In: MONTOYO, Andrés (publ.); MUŃOZ, Rafael (publ.) ; MÉTAIS, Elisabeth (publ.): *Natural Language Processing and Information Systems*. Springer Berlin Heidelberg, 2005. – DOI 10.1007/11428817_9. – ISBN 978–3–540–26031–8, pp. 91–102

[Kri13] KRISCH, Jennifer: *Identifikation kritischer Weak-Words aufgrund ihres Satzkontextes in Anforderungsdokumenten*, Universität Stuttgart, Diplomarbeit, 2013. `http://fg-re.gi.de/fileadmin/gliederungen/fg-re/Treffen_2013/Krisch.pdf`

[LGF+05] LAMI, Giuseppe; GNESI, Stefania; FABBRINI, Fabrizio; FUSANI, Mario ; TRENTANNI, Gianluca: An Automatic Tool for the Analysis of Natural Language Requirements. In: *International Journal of Computer Systems Science & Engineering* 20 (2005), No. 1. `http://shining.isti.cnr.it/WEBPAPER/2004-TR-40.pdf`. – ISSN 0267–6192

[Lio96] LIONS, Jacques-Louis: *Ariane 5 Flight 501 Failure*. Version: July 1996. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`, last checked: 2015-06-15

[Mic08a] MICROSOFT CORPORATION: *MICROSOFT OFFICE WORD 97-2007 BINARY FILE FORMAT SPECIFICATION [*.doc]*. `http://download.microsoft.com/download/0/B/E/0BE8BDD7-E5E8-422A-ABFD-4342ED7AD886/Word97-2007BinaryFileFormat(doc)Specification.pdf`. Version: 2008, last checked: 2015-06-15

[Mic08b] MICROSOFT CORPORATION: *Office Automation Using Visual C++*. `https://support.microsoft.com/en-us/kb/196776/en-us`. Version: 2008, last checked: 2015-04-30

[Mic11] MICROSOFT CORPORATION: *Understanding Graphics in Office Binary File Formats*. `http://msdn.microsoft.com/en-us/library/office/gg985447(v=office.14).aspx`. Version: 2011, last checked: 2015-05-14

[Mic14a] MICROSOFT CORPORATION: *[MS-DOC]: Word (.doc) Binary File Format*. `http://download.microsoft.com/download/2/4/8/24862317-78F0-4C4B-B355-C7B2C1D997DB/%5BMS-DOC%5D.pdf`. Version: 2014, last checked: 2015-06-15

[Mic14b] MICROSOFT CORPORATION: *[MS-ODRAW]: Office Drawing Binary File Format*. `http://download.microsoft.com/download/2/4/8/24862317-78F0-4C4B-B355-C7B2C1D997DB/%5BMS-ODRAW%5D.pdf`. Version: 2014, last checked: 2015-06-15

[Mic15a] MICROSOFT CORPORATION: *Enhanced-Format Metafiles*. `https://msdn.microsoft.com/en-us/library/dd162600%28v=vs.85%29.aspx`. Version: 2015, last checked: 2015-05-12

[Mic15b] MICROSOFT CORPORATION: *Windows-Format Metafiles*. `https://msdn.microsoft.com/en-us/library/dd145202%28v=VS.85%29.aspx`. Version: 2015, last checked: 2015-05-12

[Mic15c] MICROSOFT CORPORATION: *Windows GDI*. `https://msdn.microsoft.com/en-US/library/dd145203%28v=vs.85%29.aspx`. Version: 2015, last checked: 2015-05-12

[MMK01] MUSLEA, Ion; MINTON, Steven ; KNOBLOCK, Craig A.: Hierarchical Wrapper Induction for Semistructured Information Sources. Version: 2001. `http://dx.doi.org/10.1023/A:1010022931168`. In: *Autonomous Agents and Multi-Agent Systems* vol. 4. Kluwer Academic Publishers, 2001. – DOI 10.1023/A:1010022931168. – ISSN 1387–2532, pp. 93–114

[MRS08] MANNING, Christopher D.; RAGHAVAN, Prabhakar ; SCHÜTZE, Hinrich: *Introduction to Information Retrieval*. Cambridge University Press, 2008 `http://www-nlp.stanford.edu/IR-book/`. – ISBN 0–521–86571–9

[MS00] MAEDCHE, Alexander; STAAB, Steffen: Discovering Conceptual Relations from Text. In: HORN, W (publ.): *Proceedings of the 14th European Conference on Artificial Intelligence*. Amsterdam : IOS Press, 2000, 321–325

[Mur06] MURRAYS3: *MathML and Ecma Math (OMML)*. `http://blogs.msdn.com/b/murrays/archive/2006/10/07/mathml-and-ecma-math-_2800_omml_2900_-.aspx`. Version: 2006, last checked: 2015-05-28

[Mö14] MÖLLE, Daniel: Papierkrieg. In: *iX - Magazin für professionelle Informationstechnik* (2014), September, No. 9, pp. 74–78. – ISSN 0935–9680

[Obj13] OBJECT MANAGEMENT GROUP: *Requirements Interchange Format (ReqIF)*. Version 1.1. Needham, MA, October 2013. `http://www.omg.org/spec/ReqIF/1.1/PDF`

[OM96] OSBORNE, M.; MACNISH, C.K.: Processing natural language software requirement specifications. In: *Proceedings of the Second International Conference on Requirements Engineering*, IEEE Comput. Soc. Press, 1996. – ISBN 0–8186–7252–8, 229–236

[PMS14] PÖSCHL, Martin; MUTH, Bertil ; SEIBERTZ, Achim: *ReqIF Implementation Guide*. Version 1.4.3. Darmstadt, November 2014

[PYG12] PERES, Florent; YANG, Jing ; GHAZEL, Mohamed: A Formal Framework for the Formalization of Informal Requirements. In: *International Journal of Soft Computing and Software Engineering* 2 (2012), August, No. 8, 14–27. `http://dx.doi.org/10.7321/jscse.v2.n8.2`. – DOI 10.7321/jscse.v2.n8.2

[RAC11] RAUF, Rehan; ANTKIEWICZ, Michał ; CZARNECKI, Krzysztof: Logical structure extraction from software requirements documents. In: *Proceedings of the 2011 IEEE 19th International Requirements Engineering Conference, RE 2011* (2011), pp. 101–110. `http://dx.doi.org/10.1109/RE.2011.6051638`. – DOI 10.1109/RE.2011.6051638. – ISBN 978–1–457–70923–4

[Rya92] RYAN, Kevin: The role of natural language in requirements engineering. In: *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE Comput. Soc. Press, 1992. – ISBN 0–8186–3120–1, 240–242

[Sch12] SCHROEDER, M.: *Cost Benefit Assessment of ETCS Baseline 3*. `http://www.era.europa.eu/Document-Register/Documents/120320_ERA_EE_005319_B3.doc`. Version: 2012, last checked: 2015-05-17

[Sch14] SCHÜRMANN, Tim: Dem Vergessen entreißen. In: *Linux Magazin* (2014), No. 10, pp. 62–68. – ISSN 1432–640X

[Sei09] SEIBEL, Peter: *Coders at work*. New York, NY : Apress, 2009. – ISBN 1–430–21948–3

[SGC+09] Spinosa, PierLuigi; Giardiello, Gerardo; Cherubini, Manola; Marchi, Simone; Venturi, Giulia ; Montemagni, Simonetta: NLP-based metadata extraction for legal text consolidation. In: *Proceedings of the 12th International Conference on Artificial Intelligence and Law - ICAIL '09* (2009). http://dx.doi.org/10.1145/1568234.1568240. – DOI 10.1145/1568234.1568240. ISBN 978–1–60–558597–0

[Sip06] Sipser, Michael: *Introduction to the Theory of Computation*. 2nd Edition. Boston, MA : Course Technology, 2006. – ISBN 0–619–21764–2

[Sma12] SmartMatix: *Numbering Traps*. http://www.smartmatix.com/Resources/RQMTipsTraps/NumberingTraps.aspx. Version: 2012, last checked: 2015-01-14

[SP12] Sathyam, Ujwal S.; Prahl, Scott A.: *MathType MTEF v.5*. http://rtf2latex2e.sourceforge.net/MTEF5.html. Version: 2012, last checked: 2015-06-15

[Sta15] Stanford Natural Language Processing Group: *The Stanford Parser: A statistical parser*. http://nlp.stanford.edu/software/lex-parser.shtml. Version: 2015, last checked: 2015-06-06

[The15a] The Reuse Company: *Improve your software quality with RQS*. http://www.reusecompany.com/requirements-quality-suite. Version: 2015, last checked: 2015-06-07

[The15b] The SQLite team: *SQLite Home Page*. https://www.sqlite.org/. Version: 2015, last checked: 2015-05-30

[UIC90] UIC: *UIC leaflet 563 - Fittings provided in coaches in the interests of hygiene and cleanliness*. Paris, 1990

[UNI13] UNIFE: *ERTMS history*. http://www.ertms.net/?page_id=49. Version: 2013, last checked: 2015-04-25

[W3C04] W3C: *XML Schema Part 2: Datatypes Second Edition*. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#ID. Version: 2004, last checked: 2015-05-14

[War09] Warden, Shane: *Masterminds of programming*. 1st Edition. O'Reilly, 2009. – ISBN 0–596–51517–1

[Whe12] Wheeler, David A.: *SLOCCount*. http://www.dwheeler.com/sloccount/. Version: 2012, last checked: 2015-06-09

[WP06] Walter, Stephan; Pinkal, Manfred: Automatic Extraction of Definitions from German Court Decisions. In: Califf, Mary E. (publ.); Greenwood, Mark A. (publ.); Stevenson, Mark (publ.) ; Yangarber, Roman (publ.): *Proceedings of the Workshop on Information Extraction Beyond The Document*. Sydney : Association for Computational Linguistics, 2006. – ISBN 1–932432–74–4, 20–28

[WRH97] Wilson, William M.; Rosenberg, Linda H. ; Hyatt, Lawrence E.: Automated analysis of requirement specifications. In: *Proceedings - International Conference on Software Engineering*, 1997. – ISBN 0–897–91914–9, pp. 161–171

[XML14] XMLStarlet developers: *XMLStarlet Command Line XML Toolkit*. http://xmlstar.sourceforge.net/. Version: 2014, last checked: 2015-05-02