Software Qual J (2014) 22:121–149
DOI 10.1007/s11219-013-9210-6

From requirements to UML models and back: how automatic processing of text can support requirements engineering

Mathias Landhäußer · Sven J. Körner · Walter F. Tichy

Published online: 5 July 2013

Abstract Software engineering is supposed to be a structured process, but manual tasks leave much leeway. Ideally, these tasks lie in the hands of skilled analysts and software engineers. This includes creating the textual specification of the envisioned system as well as creating models for the software engineers. Usually, there is quite a bit of erosion during the process due to requirement changes, implementation decisions, etc. To deliver the software as specified, textual requirements, models, and the actual software need to be synchronized. However, in practice, the cost of manually maintaining consistency is too high. Our requirements engineering feedback system automates the process of keeping textual specification and models consistent when the models change. To improve overall processing of natural language specifications, our approach finds flaws in natural language specifications. In addition to the already published workshop paper, we show how well our tools support even non-software-engineers in improving texts. The case studies show that we can speed up the process of creation texts with fewer flaws significantly.

Keywords Natural language specification · Ontology · Modeling · UML

1 Introduction

Most of the time, requirements documents and domain descriptions are provided in natural language (Mich et al. 2004). The first steps of a software project comprise the transfer of

M. Landhäußer (✉) · S. J. Körner · W. F. Tichy
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: mathias.landhaeusser@kit.edu

S. J. Körner
e-mail: sven.koerner@kit.edu

W. F. Tichy
e-mail: tichy@kit.edu

natural language specifications into semi-structured documents. Eventually, (semi-)formal models form the basis for the subsequent software development process.

The quality of models and the time needed to build the models depend on the quality of the natural language specifications and the skills of the modeler. Several approaches aim at enhancing the quality of specification documents to provide a good basis for the subsequent software engineering process. After the specification is written down and signed off by the customer, it is often transferred manually into more formal models (e.g., UML models). This transfer creates two separate representations—one in written text and one as models. Again, the manual transformation allows the quality to vary depending on the skills of the modeler. Tools should support the modeler in creating high-quality models more or less independent of his or her skill level. A good tool should ask the appropriate questions and point out possible problems.

The representations initially (and ideally) are equivalent; but this is not always the case: a modification in one representation should be accompanied by a change in the other. If synchronization is not maintained in the development process, the representations diverge from one another. Before incorporating new information from the customer in the software model, one has to assess how the demanded changes affect the model.

Many requirements engineering software systems (Volere 2009) only support very distinct parts of the software production process and few tackle the problems that arise when one works with natural language specifications.

We believe that a system that supports requirements engineers and customers alike must consider multiple issues: At first, requirements must be written down in high quality. Then, the problem domain must be modeled in a repeatable and consistent fashion. The impact of change requests should be estimated before integrating them (Arkley and Riddle 2005; Berry 2004). Last but not least, changes must be adopted as well as on the requirements as on the resulting models.

Our requirements engineering complete automation approach (RECAA) emphasizes on natural language specifications (Körner et al. 2012) and supports the software production process from quality assurance of requirements via automatic UML model generation to synchronizing models (Landhäußer et al. 2012) and specifications and impact analysis. The model generation covers class diagrams, state charts, and activity diagrams. As Fig. 1 shows, our process does not (re-)generate the customer's specification but integrates the changes into it. In general, text generation from models is fine. But if you start with a text given by the customer, an entirely new text might confuse the customer, even more so when the specification's structure and style is changed completely. Manually detecting the actual changes (derived from model changes) in an entirely new document is cumbersome. Integrating the changes into the customer's document is the goal REFS achieves. Also, (re-)generating the entire model after the specification is changed is fine in general. But changes in the customer's wishes occur quite often during the development phase. Regeneration would break the links between the model and the already developed artifacts.

This article is an extended version of a workshop paper published at RAISE 2012 (Landhäußer et al. 2012). It gives an overview of how our tools interact with one another to improve requirements engineering. It includes additional statistics about user studies conducted to test the usability of our tools. It seems that especially RESI is suitable for stakeholders and requirements analysts as well. If we manage to introduce more and more tool support for natural language processing (NLP), we will be closer to Parnas' (1985) vision of automatic programming. Since software becomes ubiquitous and a part of everyday life, we need more and more programmers. Since the number of people capable

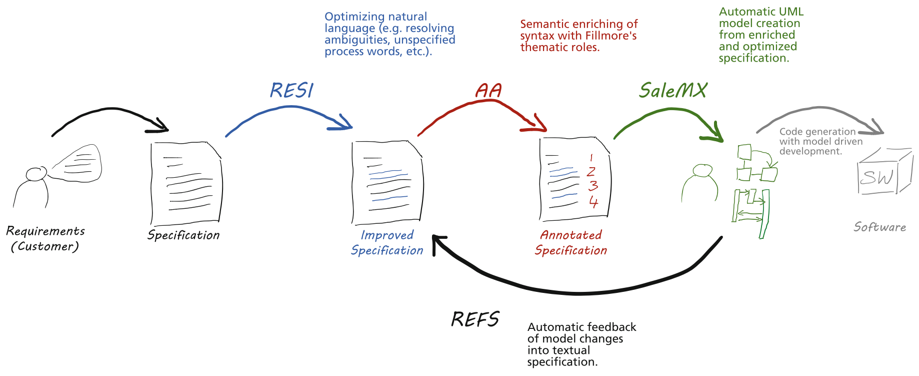

Optimizing natural language (e.g. resolving ambiguities, unspecified process words, etc.).
Semantic enriching of syntax with Fillmore's thematic roles.
Automatic UML model creation from enriched and optimized specification.
RESI
AA
SaleMX
Code generation with model driven development.
SW
Requirements (Customer)
Specification
Improved Specification
Annotated Specification
Software
REFS
Automatic feedback of model changes into textual specification.

Fig. 1 The RECAA process considers requirements elicitation, quality assurance, model generation, and change management

of programming is small and cannot grow as fast as devices enter our lives, we need to enable the average person to do some of the programming themselves.

2 Related work

In 2000, Nuseibeh and Easterbrook (2000) drafted a road map, especially emphasizing on bridging the gap between contextual inquiry and formal representations. Natt och Dag et al. (2004, 2005) showed that one can use NLP to link users' wishes with requirements in large-scale software projects. Their approach speeds up requirements management significantly when one deals with large-scale user bases and many requirements. In 2007, Cheng and Atlee (2007) showed that requirements engineering (RE) activities are more iterative compared with other software engineering activities. The involved tasks do not yet have pervasive tool support. Automating more parts of software engineering should speed up the processing times of requirements and decrease error rates. Figure 2 shows where RECAA is situated in the software development process and which parts of the development stages it supports and ideally combines. It bridges the gap between the manual tools used for requirements administration (*RE Tools*) and the semi-automatic tools (*CASE-Tools*) used in the design and implementation phase.

Today's RE relies mostly on the combination of certain tools and an experienced analyst (Volere 2009). Latter of which knows which tools to use to achieve the best results. Many tools focus on requirements management, their elicitation, and documentation. Other tools support the model creation process and code–stub generation. Our comparison of the tools is summarized in Tables 1 and 2: + indicates their strengths,—their shortcomings, and *o* indicates an average performance.

2.1 Improving requirement specifications

Researchers often demand for complete, correct, and unmistakable specifications (IEEE Computer Society 1998). But there are only a few papers that describe which concrete problems can occur and which aspects a human analyst should consider.

Berry et al. (2003) help writers to avoid linguistic ambiguities, explain the according problems by example, and show how to avoid them. Rupp and the SOPHIST group (2006)

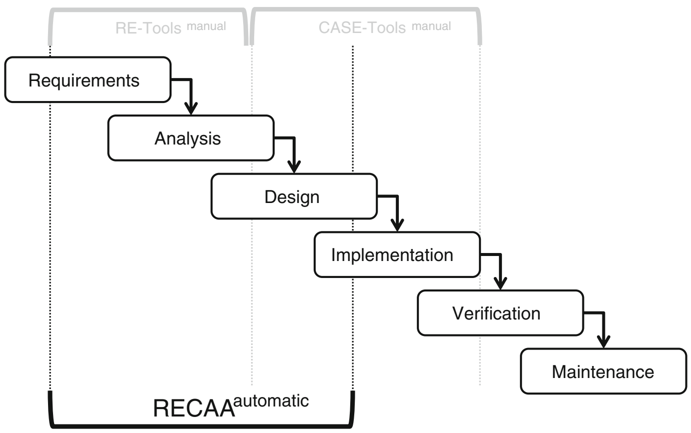


Fig. 2 RECAA and its support for software development

dedicate a complete chapter to finding and avoiding "linguistic defects" in natural language specifications. These defects are produced unintentionally when formulating sentences in natural language and lead to inferior specifications.

A manual technique that has been used for decades are inspections which base on the ideas of Fagan (1976) (Ackerman et al. 1989). The process itself depends largely on the skill-set of the inspectors and their experience. Also, the inspectors tend to classify flaws and errors into minor and major problems according to their experience. Then they tend to overlook or dismiss the *minor* problems which might be a major problem in the given context. We feel that inspections lack the deterministic behavior of a tool that assesses text following strict rules. This is one of the reasons why Kamsties et al. (2001) provided an improved inspection type, as inspections were sometimes seen as one-dimensional (Kiyavitskaya et al. 2008).

To leverage the power of computers, one can encode requirements in a machine-readable format. While this greatly eases the detection of specification flaws, one loses readability. As only few stakeholders are trained in formal representations, it is almost impossible for them to participate in the software engineering process (Heitmeyer et al. 1996). From a specialist's side, we point out that even though the necessary training is available (Pease and Murray 2003), practitioners perceive formal specification languages as difficult to use (Konrad and Cheng 2005).

A less formalized approach is controlling natural language. The allowed spectrum of language is being reduced and writers must adhere to certain rules that make the texts less ambiguous. One of the most popular approaches is Attempto Controlled English (ACE) (Fuchs et al. 1999). Denger et al. (2003) present an approach that is pattern-based but also report that existing specifications can be patternized only with difficulties. Propel (Smith et al. 2002) accompanies a controlled textual specification with a finite-state machine thereby using the advantages of both representations. Keeping them synchronized requires a huge effort though. We conclude that constrained natural languages lead to similar problems like formal languages, and the effort of applying them to existing documents is often a problem.

The research community has also come up with tools that detect defects in natural language specifications or rate their quality. Davis et al. (1993) present 24 criteria to measure the quality of a specification with the help of metrics. They point out that the

Table 1 Comparing existing RE tools and their features: part I

Manufacturer	Name	Manage textual requirements	Requirements modeling	Error tracking	Manual text preparation	Speech optimization	NLP	T2M	M2T	Roundtrip
Accept Software, Inc.	Accept 360°	+	o	+	n/a	n/a	n/a	n/a	n/a	n/a
IBM	IBM Rational Requirements Composer	+	+	o	n/a	n/a	n/a	n/a	n/a	n/a
IBM	IBM Rational RequisitePro	o	o	o	n/a	n/a	n/a	n/a	n/a	n/a
Borland	CaliberRM 2005	+	o	-	n/a	n/a	n/a	n/a	n/a	n/a
IBM	IBM Rational DOORS	+	+	o	n/a	n/a	n/a	n/a	n/a	n/a
Leap Systems	Leap SE	n/a	+	n/a	o	n/a	o	+	n/a	n/a
Hewlett Packard	HP Quality Center 9.2	+	−	+	n/a	n/a	n/a	n/a	n/a	n/a
Foresight Systems Inc.	Foresight	n/a	+	o	n/a	n/a	n/a	o	n/a	n/a
iRise	iRise	o	−	−	n/a	n/a	n/a	n/a	n/a	n/a
Polarion Software	Polarion Requirements	+	o	n/a	n/a	n/a	n/a	n/a	n/a	o
Gentleware	Poseidon for UML	n/a	+	n/a	n/a	n/a	n/a	n/a	n/a	n/a
RBC Product Development	RMTrak	+	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
OpenSource	GATE	n/a	n/a	n/a	o	−	+	+	o	-
OpenSource	OpenNLP	n/a	n/a	n/a	o	−	+	−	−	−
Universität Trento, IT	NL-OOPS	n/a	n/a	n/a	n/a	+	o	n/a	n/a	n/a
TU Chemnitz	TESSI	o	+	n/a	+	−	+	+	−	+
CoGenTex, Inc.	LIDA	o	+	n/a	o	n/a	o	+	o	o
Univ. of Limerick, IR	SUGAR / UMGAR	−	+	n/a	−	−	+	o	n/a	−
SSEC-FMT Lab	QuARS	o	n/a	n/a	n/a	+	+	n/a	n/a	n/a
Accenture	RAT	+	n/a	−	−	o	o	n/a	n/a	−

Table 2 Comparing existing RE tools and their features: part II

Manufacturer	Text creation	Impact analysis	Project planning/ collaboration	GUI	End-user suitability	Available	Metrics/ statistics	Office-integration
Accept Software, Inc.	n/a	n/a	o	+	+	Buy	n/a	o
IBM	n/a	+	–	+	+	Buy	o	o
IBM	n/a	o	–	+	o	Buy	o	o
Borland	n/a	+	o	+	+	Buy	–	o
IBM	n/a	–	+	+	+	Buy	o	o
Leap Systems	n/a	n/a	n/a	o	o	Buy	n/a	+
Hewlett Packard	n/a	+	o	+	o	Buy	o	+
Foresight Systems Inc.	n/a	o	n/a	o	o	Buy	o	o
iRise	n/a	n/a	n/a	+	o	Buy	n/a	o
Polarion Software	n/a	o	+	+	+	Buy	+	+
Gentleware	n/a	n/a	n/a	+	o	Buy	n/a	n/a
RBC Product Development	n/a	–	n/a	n/a	+	Buy	n/a	+
OpenSource	o	n/a	n/a	n/a	–	OpenSource	n/a	n/a
OpenSource	–	n/a	n/a	–	–	OpenSource	n/a	n/a
Universität Trento, IT	n/a	n/a	n/a	–	–	n/a	n/a	n/a
TU Chemnitz	+	o	n/a	o	–	On request	o	n/a
CoGenTex, Inc.	–	o	n/a	+	o	On request	n/a	n/a
Univ. of Limerick, IR	n/a	n/a	n/a	–	–	On request	n/a	n/a
SSEC-FMT Lab	n/a	n/a	n/a	o	n/a	On request	+	n/a
Accenture	n/a	n/a	o	+	+	n/a	–	+

perfect specification cannot exist since the individual quality criteria affect and sometimes even contradict each other (e.g., legibility vs. redundancy).

Wilson et al. (1997) categorize reoccurring expressions of natural language specifications and define metrics to evaluate a specification's completeness, consistency, and so on.

Chantree et al. (2006) scrutinized ambiguities in specifications. They automatically detect ambiguities, but they must be removed manually. They further differentiate between nocuous ambiguities (i.e., different readers are likely to interpret these passages differently) and non-harmful ones. To reduce the manual effort, they heuristically detect non-harmful ones and leave them unchanged.

In 2010, Yang et al. (2010) presented a machine learning-based analysis that identifies whether ambiguous anaphors are nocuous and therefore should be rephrased. Their approach focuses not on the detection of all ambiguous anaphors but on the ones that are likely to be misinterpreted. The classifier is trained with ambiguity information obtained in surveys; it identifies nocuous ambiguities with high recall and acceptable precision.

Fabbrini et al. (2001) developed a tool called quality analyzer of requirement specification (QuARS). It searches for words that indicate potential problems. Depending on the density of the indicators for a given passage, it is being flagged as problematic or non-problematic. Thereby QuARS pinpoints potential flaws to the user. Berry et al. offer an extension to QuARS in their *New Quality Model* work (Berry 2008). Fantechi et al. (2002) use QuARS as conceptual basis for their own metrics.

Pisan (2000) takes another route and begins with specifications that are evaluated and known to be of high quality. These already existing specifications or parts thereof are then being used as building blocks for new specifications.

2.2 Ontologies in requirements engineering

Today, several approaches use domain-specific ontologies to cope with the problems that occur in the RE process (Kaiya and Saeki 2005, 2006; Saeki 2004; Zhang et al. 2006; Meng et al. 2006). They use ontology-based systems to correctly classify textual information that has been delivered from stakeholders. Other projects, for example, make sure that the correct (domain specific) wording is used when the specification documents are elicited from different stakeholders. Some projects have a narrow field of application and are specified for certain conditions and use cases (Liu and Singh 2004; Havasi et al. 2007). Until today, real-world applications as listed in reference (Volere 2009) have not yet adopted many of the solutions resulting from current research.

2.3 Automatic model creation and text synthesis

After elicitation, requirements are transformed into models that give a more formal representation of the described software system. These models are usually not intended for use with the client but with the software architects and the programming team. The average client cannot understand these models. As a result, the analyst usually maintains two models: one (semi) formal model for the development team and one informal description in natural language for the client. These models have to be kept synchronized during requirements evolution. Dawson and Swatman (1999) argue that the mapping between informal and formal models is ad hoc and often results in divergent models. This strongly suggests to fill the gap between textual specifications and the models.

In 1997, Moreno and van de Riet (1997) set the foundation for model extraction. Juristo et al. (2000) explain that a systematic procedure to dissect and process natural language information is strongly needed. They hint to the disadvantages of manual tasks which dominate the RE process until today. They postulate that this procedure must be independent from the analyst and his individual skills.

In 2000, Harmain developed CM-Builder, a NLP tool which generates an object-oriented model from textual specifications (Harmain and Gaizauskas 2000). Montes et al. (2008) describe a method of generating an object-oriented conceptual model (like UML class diagrams) from natural language text. Hasegawa et al. (2009) describe a tool that extracts requirements models (abstract models of the system) from natural language texts. Instead of relying solely on NLP techniques, they perform text mining tasks on multiple documents to extract relevant words (nouns, verbs, adjectives, etc.), assuming that important and correct concepts of the domain are contained in multiple distributed documents.

Gelhausen presented an approach to generate UML domain models directly from textual specifications (Gelhausen and Tichy 2007; Gelhausen et al. 2008). The domain model includes class diagrams, activity diagrams, and state charts. He uses a graph as an intermediate representation of text. The graph contains typed nodes for sentences and words.

Edges stand for thematic roles and are the heart of his approach because they represent the semantic information given in the text. The roles are inspired by the work of Fillmore (1969) on which other researchers in the software and RE community based their work as well (e.g., Liaskos et al. 2006; Niu and Easterbrook 2008). Gelhausen adapted Fillmore's roles for the purpose of model extraction (Gelhausen et al. 2008). Graph transformation rules are then used to build an UML representation. Following his approach, the first step of model driven development can be automated.

Requirements engineering is an iterative process, as Fig. 3 shows (Glinz et al. 2009). Often, stakeholders change the textual software specification while the software is already in the making (Wiegers 2003). The direct approach would be to generate a new model after modifications. This approach leads to information loss if work on the existing models has already begun. It is the requirements analyst's job to assess the impact of the changes on the existing domain model. Ideally, one calculates the necessary changes in the models and keeps the unchanged parts of the models as well as the other software artifacts that have been created already. Our aim is to provide a fast and accurate evaluation of the situation when changes occur. We help the analyst to decide whether changes are worthwhile or whether they imply an overhaul of the architecture and the implementation.

Overmyer's Linguistic Assistant for Domain Analysis (LIDA) is a tool for requirements engineers who want support in the iterative RE process (Overmyer et al. 2001). LIDA analyzes the lexical content of natural language specifications written by domain experts. It identifies and marks lexical items corresponding to candidate model elements.

The analyst creates the UML class model according to model elements proposed by LIDA. It supports the analyst with a well-arranged document that he or she can use to extract the system domain model. When the models change, LIDA generates a new requirements specification instead of updating the initial text. This is unfortunate if you start with a customer-provided specification: Then you end up with a totally different text—LIDA provides no support for identifying the changes in the original specification. In contrast to LIDA, our approach keeps the connection between the specification and the various model types without re-generating the specification.

Kroha's (2000) TESSI is another automatic model generator that is capable of supporting iterative processes. TESSI helps the analyst to complete requirements. The analyst needs to specify the roles of words in the text. The problem is that the analyst needs to know every role of every word during the modeling process, because incomplete role arguments lead to incomplete UML models. The model is then used to synthesize a new specification from the model, i.e., to provide a model-derived requirements description. Again, the original specification is not updated but discarded. With SUGAR, Deeptimahanti et al. offer a tool to extract models from text (Deeptimahanti and Sanyal 2009). Changes in the specification text require a rerun of the process, and models from the

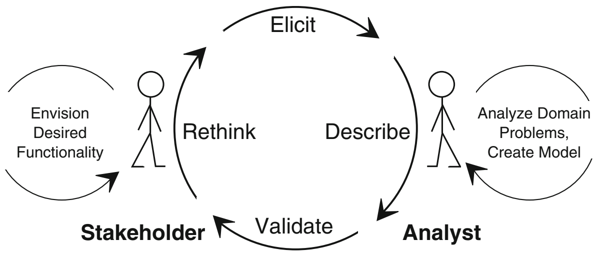

Elicit
Envision Desired Functionality
Rethink
Describe
Analyze Domain Problems, Create Model
Stakeholder
Validate
Analyst

Fig. 3 Requirements engineering is iterative

previous run are discarded. This can be a major drawback, if the development phase already has started. Model changes are not fed back to the text.

Mala's system uses a NLP pipeline to generate a model without the help of a domain expert (Mala and Uma 2006). Mala states that the yielded results are at least as good as or exceeding human-made class diagrams. Other tools that extract models from natural language with the phrase pattern approach come from Fliedl et al.(2004) and Li et al. (2005). Bajwa's UMLG extracts nouns and verb combinations from input texts and maps the nouns and verbs to UML class elements and relations, respectively (Bajwa and Choudhary 2006). Unfortunately, none of these tools support iteration or impact analysis.

Adding new information to the model needs to be expressed in the textual specification, too. An example would be a new class element that has been added to the UML domain model. In this case, natural language would have to be generated from the model. Research projects from Reiter, Meziane, and Kroha focus on this (Meziane et al. 2008; Kroha et al. 2006; Reiter and Dale 2000). But still, this cannot be considered as synchronization between model and textual specification rather than document generation from models. There is no direct connection to the initial specification.

2.4 Impact assessment

Being able to determine the impact of changes on a software specification is a well-known problem that has existed ever since software development became an industry. Bohner and Arnold (1996) define impact analysis as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change". To do that, one has to maintain traceability among various entities of the software development process. Also, one has to detect possible side and ripple effects of the changes.

Kung et al. (1994) use impact analysis to focus testing efforts on hot spots. They describe a formal model to identify changes and their impact on an object-oriented software library. Chaumun et al. (2002) use impact analysis methods to assess maintainability.

Most impact analysis approaches focus on changes of the program code, whereas Han (1997) used dependencies defined between software artifacts to identify the impact of a change. Briand et al. (2003) propose a tool that uses a set of OCL constraints to detect the differences between two versions of an UML model and their impact on unchanged model elements. Xing and Stroulia (2005) present an automatic tool called UMLDiff that detects structural model changes of two subsequent UML class models. Their approach is similar to ours regarding the detection method: When comparing two versions of an UML model, they identify pairs of classes that are identical in both models. Modified classes are matched based on their name, neighborhood (e.g., associations to already paired classes), and contents (e.g., when the name of a class changes but not its attributes and methods). UMLDiff then produces a concise list of changes based on the pairs (unmodified classes or modified classes) and unpaired classes (deleted from the old model or introduced in the new model).

3 RECAA components

The RECAA process (as outlined in Fig. 1) comprises several stages—and several tools that collectively support the requirements engineer. The following subsections cover the quality assurance with RESI, the automatic UML model generation with AutoAnnotator and Sal$_e$ mx, and the model and text synchronization with REFS. The technical basis of

REFS also allows for impact analysis when the specification text or the domain models change. Each tool has been evaluated with case studies (see Sect. 4).

3.1 Quality assurance: RESI

Before we start a software production process based upon natural language specifications, we make sure that the natural language text has as few defects as possible. During the work on SAL$_e$ mx, Gelhausen and Körner discovered that many problems resulting from linguistic defects could be detected automatically right at the beginning of the process. Körner's and Brumm's (2010) RESI uses NLP tools and ontological reasoning to detect linguistic defects such as distortion, incompletely specified process words, and so on. For every detectable defect type, there is a "rule" which can be applied to the specification under inspection. The user can decide which rules and in which order to apply to the text. RESI then iterates over the specification, applying one rule after the other. Many of the defect types can not only be detected: RESI often can make suggestions and sophisticated guesses on how to repair the defects. The suggestions and possible solutions are being delivered to the analyst who then can decide to fix the problem or to consult the customer first.

3.2 Model extraction support: AUTOANNOTATOR

To derive UML domain models (class, activity, and state diagrams) from textual specifications, SAL$_e$ mx relies on explicitly encoded semantic information (Gelhausen and Tichy 2007). While the extraction is relatively straightforward, the annotation process is not (Körner and Landhäußer 2010). To encode the semantics, one can choose from about 70 thematic roles and has to follow very strict annotation rules (Gelhausen 2010). If one confuses the provided structures, one ends up with subtle defects in the model, which have to be corrected by the analyst unnecessarily. Especially for beginners and large documents, this drawback might render manual annotation impracticable. AUTOANNOTATOR supports the analyst in properly encoding the semantics with thematic roles.

AUTOANNOTATOR uses a pipeline of proven natural language processing techniques (such as part-of-speech taggers and parsers) that can be used to analyze grammatical structures. On top of that, our tool uses ontologies (e.g., WordNet and Cyc) to query further semantic information. If AUTOANNOTATOR cannot determine what to encode, it relies on user interaction. The guided annotation speeds up the annotation process greatly and scales well with the size of the documents (see Sect. 4.3)

For now, AUTOANNOTATOR only processes English texts but can be adapted easily for other languages as long as the underlying natural language processing tools support them.

3.3 Transferring model changes to the specification: REFS

As RECAA uses Gelhausen's model extractor SAL$_e$ mx (Gelhausen and Tichy 2007), we extended its underlying graph representation as Fig. 4 illustrates. With the connection of textual nodes with their UML counterparts, we make it possible to synchronize changes in the UML models with the text and vice versa. It does not matter if the the UML model type is class diagram, activity diagram, or state chart. All types of models created by SAL$_e$ mx are considered. The left part of the figure shows the text subgraph for the phrase "The WHOIS client makes a text request to the WHOIS server, then, the WHOIS server replies text content." The right part shows an excerpt of the corresponding UML subgraph; typed edges connect the class and method nodes. We added tracking edges that connect text nodes with UML nodes; the tracking edges are added to the graph during the UML generation and are shown as dashed lines. This way, the UML representation is tightly linked with the underlying text. To account for possible repetitions in the text, we allow a given UML element to be linked with multiple text nodes. We store the original specification and the connected model together to reflect model changes back to text later on.

After the first UML models have been created from the textual specification, software architects make design decisions, rearrange UML model elements, group parts of the models, create superclasses, and so on. These changes are carried out with UML tools. Essentially, updates, creations, and deletions occur. Updates on relations are being treated as a combination of deletions and creations. Simple name updates are reflected directly into the text.

To reflect model changes to the specification, the initial model (M_i) and the changed model (M_c) need to be compared and matched. We use EMFCompare (Project 2010) of the Eclipse Modeling Framework to compute the difference between M_i and M_c. EMFCompare identifies model elements of M_i that also occur in M_c by hierarchically (type-aware) matching model elements: At first, the name of the model element is considered; then, all references to other elements are being examined. After that, the attributes are analyzed, and finally, the type of the element (i.e., the meta-model type) is considered. Every comparison gives a similarity value between 0 and 1. After comparing names, EMFCompare sums up the weighted combination of the values. If the sum is above a certain threshold, the elements are considered equal.

All elements that are being matched are either unchanged or can be identified for modification. All other elements of M_i have been removed in M_c; all unmatched elements of M_c are considered new. This way, we create a list of creations, updates, and deletions and successively integrate them into the original specification. Changes are processed in a create/update/read sequence to assure the correct order of changes without ripple effects. Changed and deleted elements can be identified in the text graph using the tracking edges for updating the text or removing the text elements. New elements are appended to the specification using simple templates; readability could be increased using more sophisticated approaches. At the end of the process, a modified specification text can be generated; parts not affected by the model changes remain untouched during the text modification.

The updated specification can be handed over to the stakeholders, who can also use text comparison to review the changes. A list of changes can be used for a quick overview. Figure 9 shows an updated specification after some modifications (see Sect. 4.5)

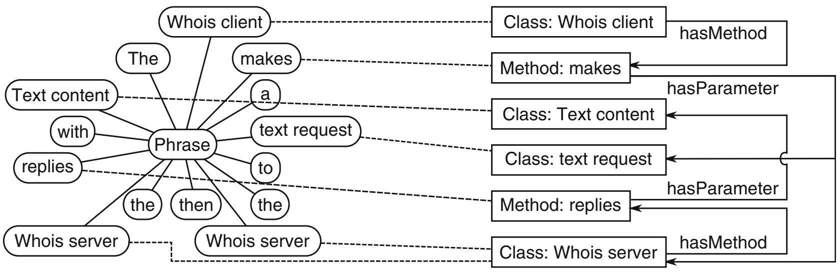


Fig. 4 Tracking edges connect the textual nodes with their UML counterparts

3.4 Using interconnection to assess impact of textual changes

If the stakeholders introduce new requirements after the modeling and development of software have already started, our interconnection from model to text can support impact assessment on a textual basis as well. This means that changes in the requirements text could be evaluated and assessed according to their possible side effects, i.e., cost and time. With the introduction of the bidirectional connection of text and model, we are able to detect the results of textual changes in the corresponding model.

To find out what would change in a model, we create an UML model of the initial specification and later of the altered specification. Then, we compute the differences between the initial and the altered model using EMFs model compare function (Project 2010). Inspired by the function point method, the users attach a weight factor to every UML change. As with function point, weight-factors need to be defined during the evaluation of finished projects, so that reliable numbers are being used. Subsuming these factors, we assess the impact of specification changes to their corresponding models. This impact measure can then be used as a possible value whether new requirements should be incorporated in the process (if the impact score is lower than a defined threshold) or not (if it is higher than a defined threshold).

This technique is straightforward for additional text, but also works for text changes and deletions. This can mean anything from changing a class name to altering the parameter list of a class' method. Deletions are simpler—we check whether the deleted text element appears anywhere else in the text. If not, the corresponding model element is deleted.

Reordering the sentences of a specification has no effect on the model an thus no impact at all.

4 Case studies

In this section, we show the effectiveness of our approach. At first, we show how different types of users work with RESI. Then, we report on the effectiveness of AutoAnnotator. Finally, we show how REFS transfers model changes back to a specification text, and we explain how model differencing can help determine the impact of specification changes on UML domain models.

The specifications used to run the following case studies are the modal window (Chen 2011), the musical store (Deeptimahanti and Sanyal 2008, 2009), Circe (Ambriola and Gervasi 2006), monitoring pressure (Berry et al. 2003; Courtois and Parnas 1993), the ATM (Rumbaugh et al. 1991), the steam boiler (Abrial et al. 1996), and the ABC video rental (Kiyavitskaya et al. 2008) examples. All specifications can be found in the Appendix.

4.1 RESI case study

Table 3 shows the comparison of all tested specifications and the errors found using RESI. The possible flaws are categorized into:

- Ambiguities: shows the number of ambiguous words and possible additional and more detailed meanings.
- Nominalizations.
- Process words: shows the number of incomplete process words and their missing arguments.

Table 3 Results of RESI specification improvement

Specification	Modal win.	Musical store	Circe	Monit. press.	ATM	Steam boiler	ABC video
# Words	33	133	138	99	170	188	222
# Phrases	1	17	12	6	10	7	17
Found flaws							
Ambiguities							
# Ambig. win.	8	46	28	26	57	55	44
# Add. mean.	13	91	45	43	141	98	198
# Det. mean.	3	41	22	19	46	42	12
# Nominaliz.	1	9	7	4	4	2	4
Process words							
# Incomplete	0	5	1	3	2	4	14
# Miss. arg.	0	4	1	5	2	4	18
# Synonyms	0	0	0	0	1	1	11
Sets and references							
# Quantors	0	1	1	0	5	4	8
# Def. art.	3	8	23	13	18	29	24
# Indef. art.	1	2	2	7	5	10	6

- Synonyms.
- Sets + references: shows the number of numerical values or sets/enumerations incorrectly used in the text. Definite and indefinite articles are a part of that.

4.2 RESI user case study

In 2010, Körner and Brumm (2010) showed the effectiveness of RESI. To further investigate the usability of RESI and user acceptance, we conducted a case study involving professional developers (P), Ph.D. students (D), and test persons without software engineering background (N). Each group provided four subjects. We used test specifications published by Kiyavitskaya et al. (2008) (ABC video rental, text 1) and a text taken from Berry's "Ambiguity Handbook" (Berry et al. 2003) (monitoring pressure, text 2). RESI detects more defects than the manual inspections in Berry's and Kiyavitskaya's publication (95 and 339, respectively in total). We removed the possibility of false positives to be able to compare the manual with the tool results. We used a factorial design (three user groups and two texts with and without RESI, respectively, see Table 4) and limited the test time to 15 min. Experience showed that well-trained analysts complete the tasks within 20–30 min with the support of RESI. Therefore, we did not expect the participants to complete the tasks whether manual or tool supported in the available time. And no one did.

Overall, the use of RESI results in a higher number of found defects: For text 1, the detection rate increases by 31 % (62 instead of 47 defects), see Fig. 5. For text 2, the detection rate increases even by 88 % (46 instead of 24 defects), see Fig. 6. The participants reported that they liked being guided through the specification by RESI.

Table 5 shows the total number of flaws that could be found in the case study's texts in the first row. The second row $\varnothing \sum \text{Flaws}_{\text{CS}}$ shows the number of flaws found be the test subjects using manual approaches or tool support. The recall of the case study Recall_{CS} shows a significant increase in found flaws using RESI.

The results of the Ph.D. students (D1–D4) show little difference with or without tool support (see Table 6). We conclude that the continuous training in RE during courses leads to these consistent results. One can see that the participants tend to search for specific problems and that they favor searching some defect types over others. Since RESI lets the

Table 4 Factorial design

Person	Run 1	Run 2
Subject 1	Text 1, manual	Text 2, w/RESI
Subject 2	Text 1, w/RESI	Text 2, manual
Subject 3	Text 2, manual	Text 1, w/RESI
Subject 4	Text 2, w/RESI	Text 1, manual

Subjects were assigned tests by lot

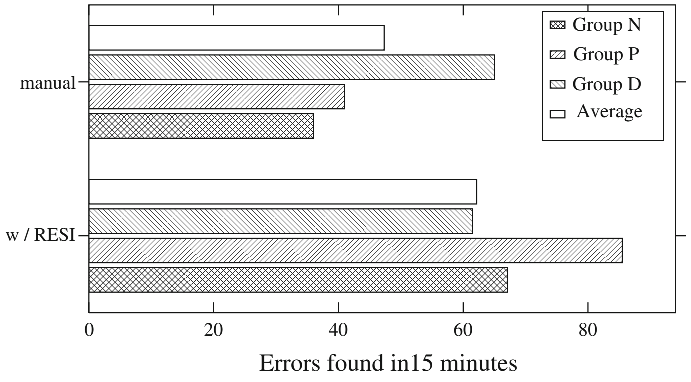


Fig. 5 Results of the inspection of the ABC video rental specification (Kiyavitskaya et al. 2008). Errors included: 339

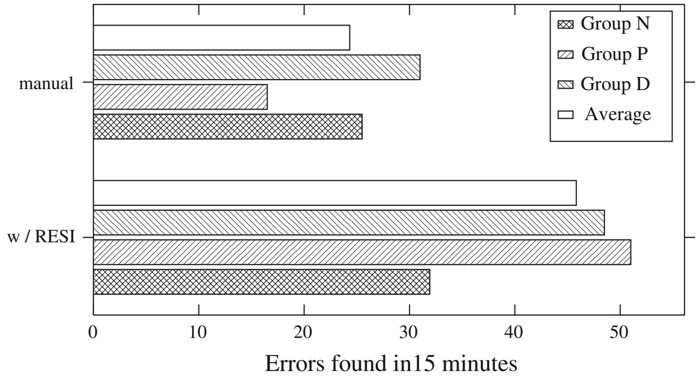


Fig. 6 Results of the inspection of the monitoring pressure specification (Berry et al. 2003). Errors included: 95

user decide which rules to apply and in which order, the results of the RESI session also exhibit that pattern.

The results for the professional developers (P1–P4) differ from the Ph.D.'s results (see Table 7). Similar to the Ph.D. students, the professionals iterated defect-class-wise over the text manually and the test persons exhibit preferences for different defect types. Comparing the results with and without RESI, there is a difference in this group: Using the tool, the professionals found more defects in the same amount of time. Since the manually found defects were true positives, we conclude that it is easier for them to read, interpret, and correct defects than to find (subtle) linguistic flaws themselves.

The final part of the case study involved persons without software engineering background (N1–N4) that represent stakeholders (see Table 8). We wanted to see, whether RESI can help stakeholders to improve their requirements or problem descriptions before handing them to an analyst. The results of this group are similar to these of the developer group, and tool support increases their performance as well.

Some users spent much time understanding and questioning the definitions and suggestions of RESI. Especially, the ontology's terminology seemed to distract them; we believe that this is a minor drawback that would diminish after some training. They also reported that taking notes for later discussions with the imaginary client took too much time. We also believe that training could improve this situation—especially because we got that remark from very few participants.

Table 5 Recall and precision for ABC video and monitoring pressure examples

	ABC video rental		Monitoring pressure	
	Manual	RESI	Manual	RESI
$\sum \text{Flaws}_{\text{Total}}$	399		95	
$ø \sum \text{Flaws}_{\text{CS}}$	47.33	62.17	24.33	45.83
$\text{Recall}_{\text{CS}} = \frac{\sum \text{Found Flaws}}{\sum \text{All Flaws}}$	0.1396	0.1834	0.2561	0.4825

Table 6 Flaws found by group 1 (Ph.D. students)

Specification	ABC video rental				Monitoring pressure			
Type of eval	Manual		RESI		RESI		Manual	
Test subject	D1	D4	D2	D3	D1	D4	D2	D3
# Ambiguous words	16	22	4	18	15	14	15	15
# Additional meanings	0	16	1	11	21	10	12	2
# More detailed meaning	0	3	2	10	6	5	0	0
# Nominalization	2	0	0	3	0	0	0	0
# Incomplete process words	14	11	0	1	0	1	3	3
# Missing arguments	18	13	0	0	0	1	5	5
# Synonyms	0	1	0	0	0	0	0	0
# Quantors	2	3	0	0	0	0	0	0
# Def. articles	8	1	0	0	12	4	1	0
# Indef. articles	0	0	0	0	4	1	1	0

4.3 AutoAnnotator case study

AutoAnnotator shows that implicit semantics can be automatically denoted (annotated) in textual specifications (Körner and Landhäußer 2010). The quality of the annotations depends largely on the quality of the provided texts. This is due to the (semantic) knowledge of the ontologies which is better or worse depending on subject and text quality. Using a domain ontology improves the results tremendously, if available. As of today, a manual inspection of the AutoAnnotator results is necessary, but results show that detection rates are acceptably high.

Table 9 shows the specifications that were annotated alongside with the correct, incorrect, and missing annotations. As can be seen, the correct annotation rate is between

Table 7 Flaws found by group 2 (professional software developers)

Specification	ABC video rental				Monitoring pressure			
Type of eval	Manual		RESI		RESI		Manual	
Test subject	P1	P4	P2	P3	P1	P4	P2	P3
# Ambiguous words	6	9	17	18	15	15	5	13
# Additional meanings	4	3	15	27	22	19	2	2
# More detailed meaning	1	1	6	5	6	5	0	2
# Nominalization	0	0	0	0	1	1	0	0
# Incomplete process words	10	10	2	1	2	1	2	3
# Missing arguments	14	13	2	0	2	1	3	1
# Synonyms	2	0	0	0	0	0	0	0
# Quantors	3	1	8	0	0	0	0	0
# Def. articles	2	0	24	1	12	12	0	0
# Indef. articles	1	2	6	1	1	6	0	0

Table 8 Flaws found by group 3 (non-professionals)

Specification	ABC video rental				Monitoring pressure			
Art d. eval	Manual		RESI		RESI		Manual	
Test subject	N1	N4	N2	N3	N1	N4	N2	N3
# Ambiguous words	26	11	40	23	6	0	15	15
# Additional meanings	1	7	30	11	10	0	2	2
# More detailed meaning	0	0	9	3	5	0	0	1
# Nominalization	0	0	0	0	0	0	0	0
# Incomplete process words	6	3	3	0	1	0	3	3
# Missing arguments	6	3	4	0	2	0	5	5
# Synonyms	0	0	0	0	0	0	0	0
# Quantors	4	3	4	8	0	0	0	0
# Def. articles	0	0	22	21	12	12	0	0
# Indef. articles	1	1	6	6	6	6	0	0

61 and 77 %. The specifications and the annotation results are comparable to corresponding papers that treated these specifications.

Now, if we compare the results of AutoAnnotator after having used RESI on a specification and having made several changes to the text due to suggestions, we find that this does not affect the annotation process in average (see Fig. 7). As can be seen, the Chen example would not compute at all in the original version due to errors that occur in the parser from Stanford. After making changes to the text following RESIs suggestions, the example can be processed but the amount of correct annotations is a meager 45 %.

Table 9 Qualitative and timely evaluation of automatic annotations with AutoAnnotator

Specification	Modal window	Musical store	Circe	Monitoring pressure	ATM	Steam boiler
# Words	33	133	138	99	170	188
# Phrases	1	17	12	6	10	7
# Annot.	–	124	120	84	156	158
# Correct	–	110	88	72	121	125
# Wrong	–	14	32	12	35	33
# Missing	–	19	26	33	33	36
# Total	–	143	146	117	189	194
% Correct	–	76.92	60.27	61.54	64.02	64.43
Runtime (in s)						
Init. time	–	2.99	2.91	2.99	3.34	3.17
Calc. time	–	9.82	12.32	12.78	30.71	23.67
Proc. time	–	109.82	72.3	9.71	62.83	23.75
Total time	–	122.64	87.54	25.49	96.88	50.59
Calc. time per word	–	0.074	0.089	0.129	0.181	0.126
Total time per word	–	0.92	0.63	0.26	0.57	0.27

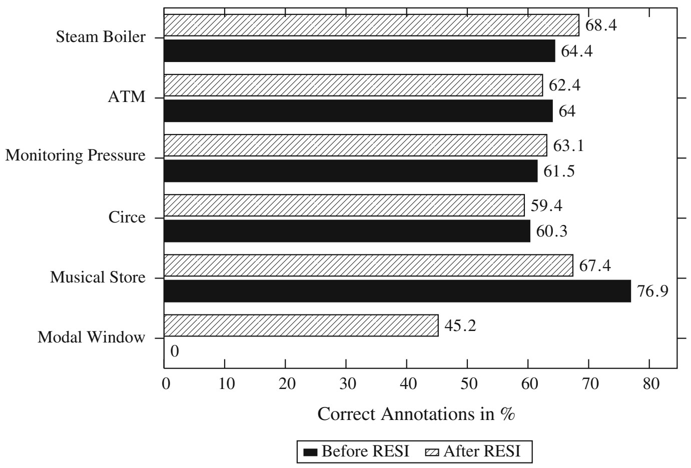


Fig. 7 Correct annotations with AutoAnnotator before and after using RESI

4.4 Feedback loop between the model and the text

Table 10 shows the detection rates of changes and deletions performed on the corresponding models. To avoid overfitting, the changes and deletions were made randomly to the text. This makes it impossible to evaluate the semantics and meaning of the sentences, and leaves the comparison of made deletions and changes and their detection in the model. We can then check whether the detection has been correctly fed back to the text.

Modal window, musical store, and monitoring pressure The changes and deletions of both texts have been fully detected and fed back to the text.

Circe (with errors) REFS detects all changes in the Circe example, but makes mistakes in the feedback to the text. One change is not detected as update, but as deletion and new creation of a model artifact. The biggest problem are sets and enumerations which tend to get lost when the feedback loop from model to text is run. This is due to the incomplete annotation of AutoAnnotator and leads to problems in the REFS process.

ATM (with errors) The feedback of updates and deletions is incomplete. One enumeration is lost which leaves and orphaned sentence ATM reads the cash card. Also, REFS deletes parts of the last sentence which is a mistake. The parts which are deleted in the last sentence are still in the model and must stay in the text as well. This is an error.

Steam boiler REFS detects all updated and deletions, but makes mistakes in the feedback loop again. In the third sentence, REFS stops after the first value of an enumeration and finishes the sentence with a full stop, though no other element of the enumeration had been deleted in the model.

4.5 Feedback loop between the text and the model

To illustrate REFS, we worked with the WHOIS protocol specification (IETF's RFC 3912) as printed in the WHOIS protocol (Daigle 2004) in the Appendix.

Model to text synchronization To exemplify the model to text synchronization, we use a generated class diagram for the WHOIS protocol specification. The diagram (see Fig. 8 for an excerpt) has been generated from the unmodified specification as shown in the WHOIS protocol (Daigle 2004) in the Appendix. Model elements can be updated, deleted, or created. For our example, we change the model as follows: we delete the class ASCII_LF

Table 10 Detection rates of random (U)pdates and (D)eletions and the feedback to text

Text	# Words	Random		Detected		Feedback to text	
		D	U	D (%)	U (%)	D	U
Modal window	30	2	3	100	100	2	3
Musical store	131	7	13	100	100	7	13
Circe	132	7	13	100	90	7 +other	3 +other
Monitoring pressure	89	4	9	100	100	4	9
ATM	110	6	1	100	100	6 +other	1 +other
Steam boiler	163	8	16	100	100	8 +other	16 +other

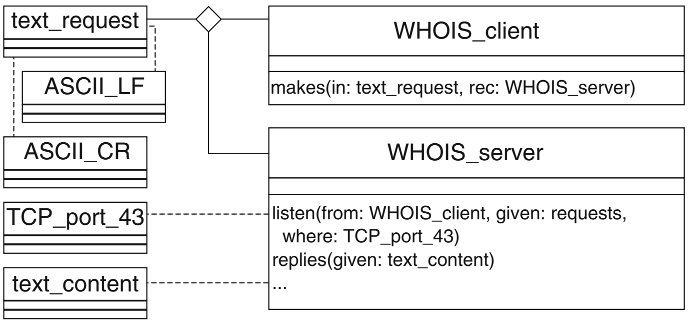
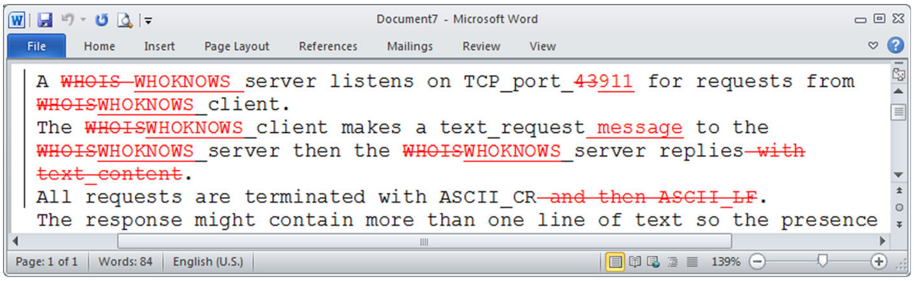

text_request
ASCII_LF
ASCII_CR
TCP_port_43
text_content
WHOIS_client
makes(in: text_request, rec: WHOIS_server)
WHOIS_server
listen(from: WHOIS_client, given: requests, where: TCP_port_43)
replies(given: text_content)
...

Fig. 8 An excerpt of the generated UML class diagram for the WHOIS protocol specification



```
A ~~WHOIS ~~WHOKNOWS_server listens on TCP_port_~~43~~911 for requests from
~~WHOIS~~WHOKNOWS_client.
The ~~WHOIS~~WHOKNOWS_client makes a text_request message to the
~~WHOIS~~WHOKNOWS_server then the ~~WHOIS~~WHOKNOWS_server replies~~ with
text_content~~.
All requests are terminated with ASCII_CR~~ and then ASCII_LF~~.
The response might contain more than one line of text so the presence
```

Fig. 9 Using Microsoft Word to present model changes in textual specifications

and its members as well as the class text_content and the corresponding parameter of the method replies. Also, we rename the WHOIS_server and WHOIS_client to WHOKNOWS_server and WHOKNOWS_client, respectively. Furthermore, we want the server to listen on TCP_Port_911 instead of 43. With this modified model, we run REFS to transfer the changes back to text.

The resulting text and the comparison to the original text are shown in Fig. 9. We show the first three sentences only, but a deletion in the model can lead to multiple deletions in the textual specification; also the renaming of server and client is propagated to the entire specification.

Text to model synchronization Assume the last two sentences of the WHOIS specification are missing in the initial specification. If a stakeholder now enters this additional information, the model has to be extended. Elements are not modeled repeatedly: If already existing model elements appear in new text, they are reused. Table 11 shows the detected natural language elements from the last two sentences. The right column shows the UML model elements that were detected and added if not already existing.

Treating updates and creations Until now, we have described the mutual synchronization process when parts of the UML model are deleted or parts of text are added to the specification. Of course, our approach also allows modifications, deletions of text, and the creation of new model elements. For newly created model elements, we need natural language generators to add the changes to the specification. So far, we use only simple templates to create sentences (Derre 2010). Updates of model elements and text passages are handled in a similar manner. Updates are usually treated as deletions followed by creations. A few exceptions update the text or model elements directly thereby preserving contextual information. If applicable, we prefer deleting and creating objects to avoid orphans.

Evaluating random modifications To assess, whether our approach provides a viable feedback loop between models and textual specifications, we conducted a small study. We used three specifications where we applied random modifications; one member of our team modified the texts and another independently modified the models using Altova UModel. Both randomly determined the elements to be modified or deleted. The complete specifications can be found on our website (Körner et al. 2012) with a detailed report.

Table 12 shows an excerpt of the results: every entry *a*/*b* states how many random modifications (*b*) have been made and how many modifications have been correctly transferred in the opposite direction (*a*). The 3rd and 4th columns show the results of the analysis of text modifications on the UML models; columns 5 and 6 show the number of model changes correctly transferred to the specification. For example, in the timbered house specification, we made three modifications to the text and deleted seven words; all three updates have been correctly mapped to the corresponding model elements and two of the deletions had an effect on the model. As can be seen, the random text deletions (and updates) sometimes modified elements that were detected but irrelevant for an UML class diagram. Elements that are not used for the automatic model creation are omitted in the REFS feedback. Furthermore, we made three updates to the model and deleted seven

Table 11 Text additions create new UML model elements unless they are already existing

Text addition	UML model element
WHOIS server	Class (already existing)
Closes	Method of class WHOIS_server
Connection	Class
Output	Class
Is finished	Method
Closed	Attribute of class connection
Indication	Method of an indetermined class
WHOIS client	Class (already existing)
Response	Class (already existing)
Received	Method of class response
Closes	Method of class WHOIS_server

Table 12 Results of the random modifications experiment

Case study		Text to model		Model to text	
Text	Size	Deletions	Updates	Deletions	Updates
Cinema	153 Words	6/7 1 irrel.	1/6 +5 irrel.	7/7	4/4
Timbered House	88 Words	2/7 +5 irrel.	3/3	7/7	3/3
WHOIS Protocol	100 Words	2/2	8/8	2/3 +1 incorr. crea/del	5/6

model elements; all changes were correctly transferred to the text. The model to text feedback is not yet perfect: One update to the model was incorrectly identified as a deletion and a creation; this information was transferred to the specification, but the resulting text's readability was reduced. Also, Altova UModel creates extra tool-specific packages. These packages are detected, but are irrelevant for the feedback loop. Therefore, they are removed before processing the model with REFS.

5 Conclusion

This article explains our vision of a (complete) requirements engineering tool chain that eases the work with natural language specifications—RECAA.

Requirements engineers spend a lot of time improving requirement specifications. Detecting defects before they emerge in later production stages is vital. With RESI, we provide an automated approach to improve textual specification which applies ontologies to provide common sense for machines. We showed that software utilizing semantics is indeed capable of solving some of the issues analysts deal with daily. The tool has many advantages over a human centered process. This way, we ensure that the quality of the requirements does not rely exclusively on the behavior and the skills of the analyst: every analyst can use the software to gain access to valuable information about possible flaws and errors in the specification.

With AutoAnnotator, we were using sentence grammar structures and ontologies to determine the correct semantics of a sentence. We use NLP tools for the pre-processing of natural language texts. The results of the evaluation suggest that the proposed approach is capable of deriving the semantic tags of $\textsc{Sal}_e$ mx. Together with its user interactive component to resolve mistakes, AutoAnnotator integrates a feedback loop in the annotation process. Combined with Gelhausen's (2010) UML diagram building process, the analyst could identify and correct the derived semantics on the fly. We are convinced that only if the analyst is faster and receives the same quality models as with the manual process, automatic model creation can support software development.

With our tool REFS, we presented a novel approach to synchronizing changes in UML model representations with textual specifications elicited from stakeholders. We explained how model changes can occur and how we transform these changes back into the textual specification and vice versa. We deliver the changed specification to the stakeholder for verification in an easy to read format, such as Microsoft Word. In future, we expect serious impact and true benefits if stakeholders can participate with their ideas.

All these tools combined deliver a good coverage of manual and error-prone tasks in software development. Focusing on improving the RE part of software engineering will be our target for years to come. If we are able to improve language processing and integrate regular people more into the software engineering process, we might eventually be able to enable the stakeholders themselves to program (Parnas 1985).

Acknowledgements We would like to thank Alexander M. Turek, Bugra Derre, and Fatih Ok. Research on the RECAA/AUTOMODEL project is funded by legodo.com in cooperation with the Karlsruhe Institute of Technology (KIT), Germany.

Appendix: Requirements specifications

The following texts were used during the evaluation of RESI, REFS, and AutoAnnotator.

Modal window (Chen 2011)

A modal window is a child window that requires the user to interact with it before they can return to operating the parent application, thus preventing any work on the application main window.

Musical store (Deeptimahanti and Sanyal 2008, 2009)

The musical store receives tape requests from customers. The musical store receives new tapes from the Main office. Musical store sends overdue notice to customers. Store assistant takes care of tape requests. Store assistant update the rental list. Store management submits the price changes. Store management submits new tapes. Store administration produces rental reports. Main office sends overdue notices for tapes. Customer request for a tape. Store assistant checks the availability of requested tape. Store assistant searches for the available tape. Store assistant searches for the rental price of available tape. Store assistant checks status of the tape to be returned by customer. Customer can borrow if there is no delay with return of other tapes. Store assistant records rental by updating the rental list. Store assistant asks the customer for his address.

Circe (Ambriola and Gervasi 2006)

The system is made of the Web interface, of Cico, of the view modules, and of the view selector. The Web interface receives from the user requirements and glossary. Requirements contain data on the team, on the author and on the revision. The Web interface transmits to Cico requirements and glossary. If the project is cooperative, the Web interface sends requirements and glossary to the repository, too. Cico computes abstract requirements using requirements, glossary, MAS-rules, predefined glossary and team data. If the project is cooperative, Cico requests team data to the repository. The view modules receive abstract requirements from Cico. The view modules can be dedicated to modeling, validation or metrication. From abstract requirements, view modules compute a view. The view module sends the view to the view selector. The user requests a view to the view selector.

Monitoring pressure (Berry et al. 2003)

The system monitors the pressure and sends the safety injection signal when the pressurizer's pres-sure falls below a "low" threshold. The human operator can override system actions by turning on a "Block" button and resets the manual block by pushing on a "Reset" button. A manual block is permitted if and only if the pressure is below a "permit" threshold. The manual block must be automatically reset by the system. A manual block is effective if and only if it is executed before the safety injection signal is sent. The "Reset" button has higher priority than the "Block" button.

ATM (Rumbaugh et al. 1991)

Design the software to support a computerized banking network including both human cashiers and automatic teller machines ATMs to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computer. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate record keeping and security provisions. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

Steam boiler (Abrial et al. 1996)

The general purpose of the steam boiler system, as shown in Fig. 1, is to ensure a safe operation of the steam boiler. The steam boiler operates safely if the contained amount of water never exceeds a certain tolerance, thus avoiding damage to the steam boiler and the turbine driven by the produced steam. Basically, the steam boiler system consists of the steam boiler itself, a measuring device for the water level, a pump to provide the steam boiler with water, a measuring device for the pump status, a measuring device for the amount of steam produced by the steam boiler, an operator desk, and a message transmission system for the signals produced. During operation, the water level is kept within the tolerance level as long as possible, using the measuring devices and the pump and producing status information for the operator desk. But even with some devices broken, the system can still successfully monitor the steam boiler. If no safe operation is possible any longer, control is handed over to the operator desk. Additionally, the operator can stop the system at any time via the operator desk.

ABC video rental (Kiyavitskaya et al. 2008)

Customers select at least one video for rental. The maximal number of tapes that a customer can have outstanding on rental is 20. The customer's account number is entered to retrieve customer data and create an order. Each customer gets an id card from ABC for identification purposes. This id card has a bar code that can be read with the bar code reader. Bar code Ids for each tape are entered and video information from inventory is displayed. The video inventory file is updated. When all tape Ids are entered, the system computes the total bill. Money is collected and the amount is entered into the system. Change is computed and displayed. The rental transaction is created, printed and stored. The customer signs the rental form, takes the tapes and leaves. To return a tape, the video bar code ID is entered into the system. The rental transaction is displayed and the tape is marked with the date of return. If past-due amounts are owed they can be paid at this time; or the clerk can select an option which updates the rental with the return date and calculates past-due fees. Any outstanding video rentals are displayed with the amount due on each tape and the total amount due. Any past-due amount must be paid before new tapes can be rented.

Cinema

This text was used in a software engineering exam at our chair.

At first, the user selects the movie show for which he would like to book tickets. This he does by clicking on the relevant film or the desired date. Depending on the selection, a list with movie shows for the selected film or a list of movie shows of the selected day is displayed. Then he clicks on the movie show for which he would like to order tickets. If there are any of the 30 orderable tickets left, he is prompted to enter his name, his e-mail address, and the desired number of cards. He can also specify whether he wants to be reminded of the movie show by e-mail. After a click on order, the system attempts to allocate the desired cards. If this is possible, a confirmation page is displayed on which all the provided information is summarized again. If the tickets for this movie show can not be reserved, there will be an error message and the user is prompted to reduce the number of cards or select a different idea.

Timbered house

This text was used in a software engineering exam at our chair.

A timbered house consists of 5–10 logs, 200–400 mud-bricks and 1,000 to 2,000 nails. Each building material, whether log, brick, or nail, is a component in exactly one timbered house. Each timbered house has a certain number of rooms and floors. For the construction of a timbered house is at least one carpenter in charge, which has a name and an individual hourly wage. For the construction of a timbered house, each carpenter uses his own tools, consisting of exactly one hammer and exactly one saw. Any carpenter can work on up to one timbered house at the same time.

WHOIS protocol (Daigle 2004)

A WHOIS server listens on TCP port 43 for requests from WHOIS clients. The WHOIS client makes a text request to the WHOIS server then the WHOIS server replies with text content. All requests are terminated with ASCII CR and then ASCII LF. The response might contain more than one line of text so the presence of ASCII CR or ASCII LF characters does not indicate the end of the response. The WHOIS server closes the connection as soon as the output is finished. The closed connection is the indication to the WHOIS client that the response has been received.

References

Abrial, J.-R., Börger, E., & Langmaack, H. (1996). The steam boiler case study: Competition of formal program specification and development methods. In *Formal methods for industrial applications. Specifying and programming the steam-boiler control* (pp. 1–12). Berlin: Springer. URL: http://citeseerx.ist.psu.edu/viewdoc/download.

Ackerman, A. F., Buchwald, L. S., & Lewski, F. H. (1989). Software inspections: An effective verification process. *Software, IEEE, 6*(3), 31–36, ISSN 0740-7459. doi:10.1109/52.28121.

Ambriola, V., & Gervasi, V. (2006). On the systematic analysis of natural language requirements with circe. *Automated Software Engginiering, 13*, 107–167, ISSN 0928-8910. doi:10.1007/s10515-006-5468-2. URL:http://portal.acm.org/citation.cfm?id=1107757.1107761.

Arkley, P., & Riddle, S. (2005). Overcoming the traceability benefit problem. In *Proceedings of 13th IEEE International Requirements Engineering Conference (RE)* (pp. 385–389). doi:10.1109/RE.2005.49.

Bajwa, I. S., & Choudhary, M. A. (2006). Natural language processing based automated system for UML diagrams generation. In *The 18th Saudi National Computer Conference on computer science (NCC18).* Riyadh: The Saudi Computer Society (SCS).

Berry, D. M. (2004). The inevitable pain of software development: Why there is no silver bullet. In M. Wirsing, A. Knapp, & S. Balsamo (Eds.), *Proceedings of Monterey Workshop 2002: Radical Innovations of Software and Systems Engineering in the Future, LNCS 2941*. Berlin: Springer.

Berry, D. M. (2008). In A. Bucchiarone, S. Gnesi, & G. Trentanni (Eds.), *A new quality model for natural language requirements specifications*. URL:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.5268.

Berry, D. M., Kamsties, E., & Krieger, M. M. (2003). From contract drafting to software specification: Linguistic sources of ambiguity—a handbook. URL:http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf.

Bohner, S. A., & Arnold, R. S. (Eds.). (1996). An introduction to software change impact analysis. In *Software change impact analysis* (pp. 1–26). Los Alamitos: IEEE Computer Society Press.

Briand, L. C., Labiche, Y., & O'Sullivan, L. (2003). Impact analysis and change management of uml models. Technical Report SCE-03-01, Carleton University.

Chantree, F., Nuseibeh, B., de Roeck, A., & Willis, A. (2006). Identifying nocuous ambiguities in natural language requirements. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)* (pp. 56–65). Washington, DC: IEEE Computer Society, ISBN 0-7695-2555-5. doi:10.1109/RE.2006.31.

Chaumun, M. A., Kabaili, H., Keller, R. K., & Lustman, F. (2002). A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming, 45*(2–3), 155–174, ISSN 0167-6423. doi:10.1016/S0167-6423(02)00058-8

Chen, R. (2011). The old new thing. MSDN blogs. URL:http://blogs.msdn.com/b/oldnewthing/archive/2011/12/12/10246541.aspx.

Cheng, B. H. C., & Atlee, J. M. (2007). Research directions in requirements engineering. In *Proceedingfs of Future of Software Engineering FOSE '07*. doi:10.1109/FOSE.2007.17, pp. 285–303.

Courtois, P.-J., & Parnas, D. L. (1993). Documentation for safety critical software. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering* (pp. 315–323). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-89791-588-7.

Daigle, L. (2004). WHOIS protocol specification. Internet. URL:http://www.ietf.org/rfc/rfc3912.txt. The Internet Engineering Task Force (IETF).

Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., & Theofanos, M. (1993). Identifying and measuring quality in a software requirements specification. In *Proceedings of the First International Software Metrics Symposium* (pp. 141–152). ISBN 0-8186-3740-4. doi:10.1109/METRIC.1993.263792.

Dawson, L., & Swatman, P. A. (1999). The use of object-oriented models in requirements engineering: A field study. In *ICIS* (pp. 260–273). doi:10.1145/352925.352949.

Denger, C., Berry, D. M., & Kamsties, E. (2003). Higher quality requirements specifications through natural language patterns. In *Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering (SWSTE '03)* (p. 80). Washington, DC: IEEE Computer Society. ISBN 0-7695-2047-2. doi:10.1109/SWSTE.2003.1245428.

Derre, B. (2010). Rückkopplung von Softwaremodelländerungen in textuelle Spezifikationen. Master's thesis. Karlsruhe: Karlsruhe Institute of Technology.

Deva Kumar, D., & Sanyal, R. (2008). Static UML model generator from analysis of requirements (SUGAR). In *Advanced software engineering and its applications, 2008 (ASEA 2008)* (pp. 77–84). doi:10.1109/ASEA.2008.25.

Deva Kumar, D., & Sanyal, R. (2009). An innovative approach for generating static UML models from natural language requirements. In *Advances in Software Engineering*, vol. 30 of *Communications in Computer and Information Science*, pp. 147–163. Berlin: Springer. ISBN 978-3-642-10241-7 (Print) 978-3-642-10242-4 (Online). doi:10.1007/978-3-642-10242-4_13.

Eclipse Modeling Framework Project. (2010). Eclipse modeling framework compare. URL:http://www.eclipse.org/emf/compare/. Last visited: 05/07/2012.

Fabbrini, F., Fusani, M., Gnesi, S., & Lami, G. (2001). The linguistic approach to the natural language requirements quality: Benefit of the use of an automatic tool. In *SEW '01: Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop* (p. 97). Washington, DC: IEEE Computer Society. ISBN 0-7695-1456-1.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 15*(3), 182–211. doi:10.1109/ASEA.2008.25

Fantechi, A., Gnesi, S., Lami, G., & Maccari, A. (2002). Application of linguistic techniques for use case analysis. In *Proceedings of the IEEE International Conference on Requirements Engineering, 2002* (pp. 157–164). ISSN 1090-705X. doi:10.1109/ICRE.2002.1048518.

Fillmore, C. J. (1969). Toward a modern theory of case. In D. A. Reibel, & S. A. Schane (Eds.), *Modern studies in English* (pp. 361–375). Englewood Cliffs: Prentice Hall.

Fliedl, G., Kop, C., & Mayr, H. C. (2004). Recent results of the NLRE (natural language based requirements engineering) project. *EMISA Forum, 24*(1), 24–25.

Fuchs, N. E., Schwertel, U., & Schwitter, R. (1999). Attempto controlled English—not just another logic specification language. *Lecture Notes in Computer Science, 1559*, 1–20, ISSN 0302-9743. doi:10.1007/3-540-48958-4_1.

Gelhausen, T. (2010). *Modellextraktion aus natürlichen Sprachen: Eine Methode zur systematischen Erstellung von Domänenmodellen*. Ph.D. thesis. Karlsruhe: Karlsruhe Institute of Technology.

Gelhausen, T., Derre, B., & Gei, R. (2008). Customizing grgen.net for model transformation. In *Proceedings of GRaMoT '08* (pp. 17–24). ACM. ISBN 978-1-60558-033-3. doi:10.1145/1402947.1402951.

Gelhausen, T., & Tichy, W. F. (2007). Thematic role based generation of UML models from real world requirements. In *Proceedings of the ICSC 2007,* (pp. 282–289). doi:10.1109/ICOSC.2007.4338360.

Glinz, M., Heymans, P., Persson, A., Sindre, G., Aurum, A., Madhavji, N. H., Paech, B., Regev, G., & Wieringa, R. (2009). Report on the working conference on requirements engineering: Foundation for software quality (REFSQ'09). *ACM SIGSOFT Software Engineering Notes, 34*(5), 40–45.

Han, J. (1997). Supporting impact analysis and change propagation in software engineering environments. In *Proceedings of the 8th IEEE International Workshop on Software Technology and Engineering Practice* (pp. 172–182). doi:10.1109/STEP.1997.615479.

Harmain, H. M., & Gaizauskas, R. J. (2000). CM-Builder: An automated NL-based CASE tool. In *ASE* (pp. 45–54). doi:10.1109/ASE.2000.873649.

Hasegawa, R., Kitamura, M., Kaiya, H., & Saeki, M. (2009). Extracting conceptual graphs from Japanese documents for software requirements modeling. In M. Kirchberg, & S. Link (Eds.), *APCCM*, vol. 96 of *CRPIT* (pp. 87–96). Australian Computer Society. ISBN 978-1-920682-77-4.

Havasi, C., Speer, R., & Alonso, J. B. (2007). ConceptNet 3: A flexible, multilingual semantic network for common sense knowledge. In *Recent advances in natural language processing*. Borovets, Bulgaria. URL:http://web.media.mit.edu/~jalonso/cnet3.pdf.

Heitmeyer, C. L., Jeffords, R. D., & Labaw, B. G. (1996). Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering Methodology, 5*(3), 231–261, ISSN 1049-331X. doi:10.1145/234426.234431.

IEEE Computer Society. (1998). IEEE recommended practice for software requirements specifications. *IEEE Standard 830-1998*, pp. 1–40. doi:10.1109/IEEESTD.1998.88286.

Juristo, N., Moreno, A. M., & López, M. (2000). How to use linguistic instruments for object-oriented analysis. *IEEE Software, 17*(3), 80–89. doi:10.1109/52.896254.

Kaiya, H., & Saeki, M. (2005). Ontology based requirements analysis: Lightweight semantic processing approach. In *Proceedings of Fifth International Conference on Quality Software (QSIC 2005)* (pp. 223–230, 19–20). doi:10.1109/QSIC.2005.46.

Kaiya, H., & Saeki, M. (2006). Using domain ontology as domain knowledge for requirements elicitation. In *Proceedings of the IEEE International Conference Requirements Engineering* (pp. 189–198, 11–15). doi:10.1109/RE.2006.72.

Kamsties, E., Knethen, A. V., Philipps, J., & Schätz, B. (2001). An empirical investigation of the defect detection capabilities of requirements specification languages. In *EMMSAD'01: Proceedings of the Sixth CAiSE/IFIP8.1 International Workshop on Evaluation of Modelling Methods in Systems Analysis and Design*.

Kiyavitskaya, N., Zeni, N., Mich, L., & Berry, D. M. (2008). Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requirements Engineering, 13*(3), 207–239, ISSN 0947-3602. doi:10.1007/s00766-008-0063-7.

Konrad, S., & Cheng, B. H. C. (2005). Facilitating the construction of specification pattern-based properties. *Requirements Engineering, IEEE International Conference on* (pp. 329–338). doi:10.1109/RE.2005.29.

Körner, S. J., & Brumm, T. (2010). Natural language specification improvement with ontologies. *International Journal of Semantic Computing (IJSC), 03*, 445–470.

Körner, S. J., & Landhäußer, M. (2010). Semantic enriching of natural language texts with automatic thematic role annotation. In *Proceedings of the Natural language processing and information systems, and 15th international conference on Applications of natural language to information systems, NLDB'10* (pp. 92–99). Berlin: Springer. ISBN 3-642-13880-2, 978-3-642-13880-5. http://dl.acm.org/citation.cfm?id=1894525.1894537.

Körner, S. J., Landhäußer, M., Gelhausen, T., & Derre, B. (2012). RECAA – the requirements engineering complete automation approach. URL:https://svn.ipd.uni-karlsruhe.de/trac/mx.

Kroha, P. (2000). Preprocessing of requirements specification. In M. T. Ibrahim, J. Küng, & N. Revell (Eds), *Database and expert systems applications*, vol. 1873 of *Lecture Notes in Computer Science* (pp. 675–684). Berlin: Springer. ISBN, 978-3-540-67978-3. doi:10.1007/3-540-44469-6_63.

Kroha, P., Gerber, P., & Rosenhainer, L. (2006). Towards generation of textual requirements descriptions from UML models. In *Proceedings of the 9th International Conference Information Systems Implementation and Modelling ISIM2006* (pp. 31 – 38). ISIM.

Kung, D. C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., & Chen, C. (1994). Change impact identification in object oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance* (pp. 202–211). doi:10.1109/ICSM.1994.336774.

Landhäußer, M., Körner, S. J., & Tichy, W. F. (2012). Synchronizing domain models with natural language specifications. In *Proceedings of the Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE'2012)*. doi:10.1109/RAISE.2012.6227965.

Li, K., Dewar, R. G., & Pooley, R. J. (2005). Towards semi-automation in requirements elicitation: Mapping natural language and object-oriented concepts. In *RE05* (pp. 5–7).

Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., & Mylopoulos, J. (2006). On goal-based variability acquisition and analysis. In *Proceedings of the 14th IEEE International Requirements Engineering Conference, RE '06* (pp. 76–85). Washington, DC: IEEE Computer Society. ISBN 0-7695-2555-5. doi:10.1109/RE.2006.45.

Liu, H., & Singh, P. (2004). ConceptNet—a practical commonsense reasoning tool-kit. *BT Technology Journal, 22*. URL:http://larifari.org/writing/BTTJ2004-ConceptNet.pdf.

Mala, G. S. A., & Uma, G. V. (2006). Automatic construction of object oriented design models [UML diagrams] from natural language requirements specification. In *PRICAI*(pp. 1155–1159). Guilin, China. doi:10.1007/11801603_152.

Meng, W. J., Rilling, J., Zhang, Y., Witte, R., & Charland, P. (2006). An ontological software comprehension process model. *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006). October 1st, Genoa, Italy*.

Meziane, F., Athanasakis, N., & Ananiadou, S. (2008). Generating natural language specifications from UML class diagrams. *Requirements Engineering, 13*(1), 1–18. doi:10.1007/s00766-007-0054-0.

Mich, L., Franch, M., & Inverardi, P. (2004). Market research for requirements analysis using linguistic tools. *Requirements Engineering, 9*, 40–56, ISSN 0947-3602. doi:10.1007/s00766-003-0179-8.

Montes, A., Pacheco, H., Estrada, H., & Pastor, O. (2008). Conceptual model generation from requirements model: A natural language processing approach. In E. Kapetanios, V. Sugumaran, & M. Spiliopoulou (Eds.), *NLDB*, vol. 5039 of *Lecture Notes in Computer Science* (pp. 325–326). Berlin: Springer. ISBN 978-3-540-69857-9.

Moreno, A. M., & van de Riet, R. P. (1997). Justification of the equivalence between linguistic and conceptual patterns for the object model.

Natt och Dag, J., Gervasi, V., Brinkkemper, S., & Regnell, B. (2004). Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International, RE '04* (pp. 283–294). Washington, DC: IEEE Computer Society. ISBN 0-7695-2174-6. doi:10.1109/RE.2004.47.

Natt och Dag, J., Regnell, B., Gervasi, V., & Brinkkemper, S. (2005). A linguistic-engineering approach to large-scale requirements management. *Software, IEEE, 22*(1), 32–39. doi:10.1109/MS.2005.1.

Niu, N., & Easterbrook, S. (2008). Extracting and modeling product line functional requirements. In *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference, RE '08* (pp. 155–164). Washington, DC: IEEE Computer Society. ISBN 978-0-7695-3309-4. doi:10.1109/RE.2008.49.

Nuseibeh, B., & Easterbrook, S. (2000). Requirements engineering: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering* (pp. 35–46). New York, NY: ACM Press. ISBN 1-58113-253-0. doi:10.1145/336512.336523.

Overmyer, S. P., Lavoie, B., & Rambow, O. (2001). Conceptual modeling through linguistic analysis using LIDA. In *Proceedings of the ICSE '01* (pp. 401–410). Washington, DC: IEEE Computer Society. ISBN 0-7695-1050-7.

Parnas, D. L. (1985). Software aspects of strategic defense systems. *Communications of the ACM, 28*(12), 1326–1335, ISSN 0001-0782. doi:10.1145/214956.214961.

Pease, A., & Murray, W. (2003). An english to logic translator for ontology-based knowledge representation languages. In *Natural Language Processing and Knowledge Engineering. Proceedings of the 2003 International Conference on* (pp. 777–783). doi:10.1109/NLPKE.2003.1276010.

Pisan, Y. (2000). Extending requirement specifications using analogy. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (pp. 70–76). New York, NY: ACM. ISBN 1-58113-206-9. doi:http://doi.acm.org/10.1145/337180.337190.

Reiter, E., & Dale, R. (2000). Building natural language generation systems. Natural language processing. Cambridge: Cambridge University Press. doi:10.2277/052102451X.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design*. Upper Saddle River, NJ: Prentice-Hall, Inc. ISBN 0-13-629841-9.

Rupp, C., & die SOPHISTen (2006). *Requirements-Engineering und Management* (4th ed.). Munich: Carl Hanser Verlag. ISBN 3-446-40509-7.

Saeki, M. (2004). Ontology-based software development techniques. *ERCIM News, 58*, 14–15. URL: http://www.ercim.org/publication/Ercim_News/enw58/EN58.pdf.

Smith, R. L., Avrunin, G. S., Clarke, L. A., & Osterweil, L. J. (2002). Propel: An approach supporting property elucidation. In *ICSE 2002: Proceedings of the 24rd International Conference on Software Engineering* (pp. 11–21). ISBN 1-58113-472-X.

Volere. (2009). List of requirement engineering tools. URL:http://www.volere.co.uk/tools.htm.

Wiegers, K. E. (2003). *Software requirements : Practical techniques for gathering and managing requirements throughout the product development cycle* (2nd. ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8; 978-0-7356-1879-4.

Wilson, W. M., Rosenberg, L. H., & Hyatt, L. E. (1997). Automated analysis of requirement specifications. In *ICSE '97: Proceedings of the 19th International Conference on Software Engineering* (pp. 161–171). ISBN 0-89791-914-9.

Xing, Z., & Stroulia, E. (2005). Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05* (pp. 54–65). New York, NY: ACM. ISBN 1-58113-993-4. doi:10.1145/1101908.1101919.

Yang, H., de Roeck, A., Gervasi, V., Willis, A., & Nuseibeh, B. (2010). Extending nocuous ambiguity analysis for anaphora in natural language requirements. In *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, RE '10* (pp. 25–34). Washington, DC: IEEE Computer Society. ISBN 978-0-7695-4162-4. doi:10.1109/RE.2010.14.

Zhang, Y., Witte, R., Rilling, J., & Haarslev, V. (2006). An ontology-based approach for traceability recovery. In *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*. Jean-Marie Favre Dragan Gasevic Ralf Lämmel Andreas Winter.

Author Biographies

Mathias Landhäußer is a Ph.D. student at Prof. Tichy's group at the Karlsruhe Institute of Technology (formerly University Karlsruhe), Germany, since 2010. His research interests are software engineering and software testing, especially applying natural language processing techniques in these areas. He is also interested in empirical software engineering. Mathias earned an M.Sc. in Information Engineering and Management from Karlsruhe Institute of Technology in 2010. He is a member of ACM and GI.

Sven J. Körner has been Ph.D. student at Prof. Tichy's group at the Karlsruhe Institute of Technology (formerly University Karlsruhe), Germany, since 2005. His research interests are software engineering, natural language processing, automatic requirements processing, and software model creation. Sven earned a M.Sc. in Computer Science from University of Karlsruhe in 2006. He is a member of IEEE.

Walter F. Tichy has been professor of Software Engineering at the Karlsruhe Institute of Technology (formerly University Karlsruhe), Germany, since 1986, and was dean of the faculty of computer science from 2002 to 2004. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and served 6 years on the faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is currently directing research on a variety of topics, including empirical software engineering, software architecture, and programming languages and tools for multi/manycore computers. He has consulted widely for industry. He earned an M.S. and a Ph.D. in Computer Science from Carnegie Mellon University in 1976 and 1980, resp. He is director at the Forschungszentrum Informatik, a technology transfer institute in Karlsruhe. He is co-founder of ParTec, a company specializing in cluster computing. He has helped organize numerous conferences and workshops; among others, he was program co-chair for the 25th International Conference on Software Engineering (2003). He received the Intel Award for the Advancement of Parallel Computing in 2009 and was named ACM Fellow in 2013. Dr. Tichy is a member of ACM, GI, and the IEEE Computer Society.