Text to Software

Developing Tools to Close the Gaps in Software Engineering

Walter F. Tichy
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
walter.tichy@kit.edu

Sven J. Koerner
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
sven.koerner@kit.edu

ABSTRACT

Software development relies heavily on manual processes for transforming requirements into software artifacts such as models, source code, or test cases. Requirements are the starting point for these transformations, and they are typically written in natural language. However, hardly any automated tools exist that translate natural language texts into software artifacts.

We propose to adapt recent advances in natural language processing and semantic technologies to generate UML models, test cases, and perhaps even source code, from natural language input. Though earlier efforts in automatic programming had limited success, we are now in a situation where extracting meaning from text has made substantial progress. For example, encouraging results for generating UML models from textual requirements have been achieved. It might even be possible to generate executable test cases. An intermediate step would be to generate tests from API documentation (which would also be a useful capability in itself). An even greater advance would be to automate rote coding.

In all of these cases, entirely new classes of software tools would be needed to extract and process the semantics inherent in natural language texts, augment them with tacit knowledge from ontologies and domain models, and, where necessary, ask humans to clarify ambiguities.

Though speculative, a determined, long-term effort in translating text to software could automate and accelerate software development to an unprecedented degree.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.10 [**Software Engineering**]: Design—*Methodologies, Representation*

General Terms

Design, Documentation, Experimentation, Theory

FoSER-18, November 7–8, 2010, Santa Fe, New Mexico, USA.

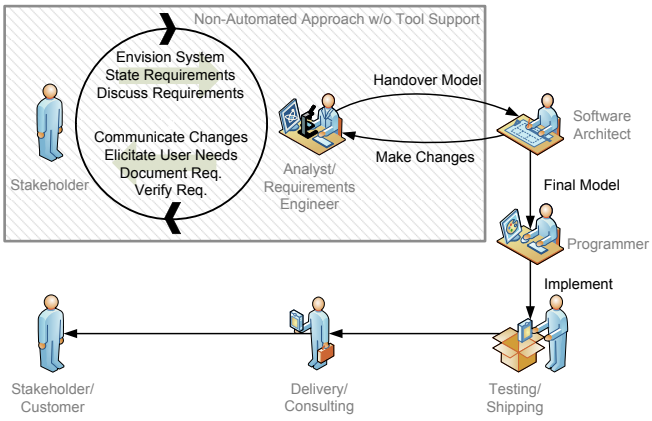

Figure 1: The Software Development Process Lacks Tool Support in Requirements Engineering

Keywords

Requirements Engineering, Automation, Model Extraction, Test Case Generation

1. INTRODUCTION

At its core, software development involves the translation of human intent into software (compare Fig. 1). Usually, a skilled analyst is needed to elicit this intent; the result is a requirements document. This document is written in natural language to make it easy for stakeholders to understand. Class diagrams, state diagrams, and other models are extracted from the requirements, also by hand. The models are refined into software designs, which are then translated into source code. Test cases must be written. All of these tasks are manual. Only the final step, translating source code into executable code, is fully automated. Software engineering research should seek to automate more of these manual tasks.

Since all software development processes hinge on requirements, we think it paramount that tools are able to extract information from this source. This means that natural language processing capabilities are needed, because 95% of requirements are recorded in natural language [20]. Later on, one might add speech input, but for now, textual input is challenging enough.

A number of researchers have demonstrated independently that extracting UML models (class diagrams, state diagrams, and sequence diagrams) from natural language text is feasi-

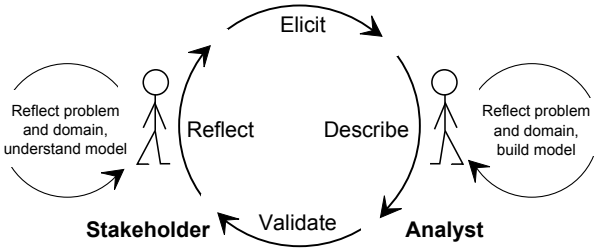

Figure 2: Iterative Process in RE

ble [9, 23, 5]. Additional research has shown that some of the ambiguities inherent in natural language requirements can be identified [6, 11, 14] and corrected automatically by using ontologies [16, 24]. Much more needs to be done, but from these results we conclude that more of the translation processes could be automated. In particular, it might be possible to extract test cases from requirements. By knowing how requirements were (machine-) translated into classes, methods, state machines, and sequence diagrams, it might be possible to produce source code for simple tests. However, generating test cases is a big leap; we therefore propose to start with something simpler, i.e., the generation of test cases from API descriptions, such as Javadoc. The advantage of API documentation is that it is more concrete than requirements. Furthermore, generating test cases form API documentation would be a valuable capability in itself.

An even greater leap would be to generate source code statements. We think that generators will not be able to invent complex algorithms in the near future, but some amount of rote coding might be possible, in particular if state machines, sequence diagrams, and temporal relations can be extracted from detailed requirements. Sequence diagrams could be used to produce code skeletons, for instance.

A brief note about "automatic programming" is appropriate. The term describes a type of programming where software is produced by machine. David Parnas [22] traced the history of the term and concluded that "automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer." In fact, one of the early compilers was called Autocode. Perhaps a generator that produces software from text or spoken input will be the ultimate autocoder.

Today's IDEs, generative techniques, "wizards", and other approaches automate aspects of software development, but they perform relatively mundane bookkeeping or generation tasks or are extremely narrow in application domain. Cheng and Atlee call for better tools and more automation in requirements engineering [3]. Nuseibeh and Easterbrook also demand new technologies that bridge the gap between requirements elicitation approaches based on contextual inquiry and more formal specification and analyses techniques [21]. What is missing is extracting meaning from requirements and augmenting that with tacit knowledge.

Natural language processing (NLP) has advanced to a point where extracting meaning is now practicable. Jurafsky's and Martin's book [10] provides an extensive overview of the state of the art. We use several of their NLP tools, such as parser, parts-of-speech taggers, and named entity recognizer. For tacit knowledge, ontologies are a promising source. Cyc [4] is a huge ontology that encodes a large

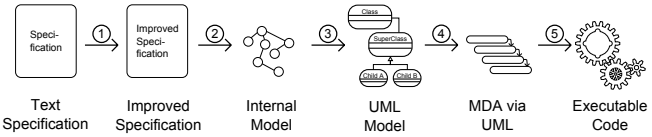

Figure 3: Automating the Software Development Process

body of world knowledge as well as some domain knowledge. The latest release of Cyc includes $500,000+$ concepts and $5,000,000$ assertions (facts and rules). It uses some $26,000+$ relations that interrelate, constrain, and (at least partially) define the concepts. Furthermore, note that requirements typically introduce new concepts. Interestingly, Kof describes the automated building of domain specific ontologies directly from requirements text [15] . Thus, powerful tools are in place. We agree with Kof that NLP is ready for requirements engineering.

Automatically generating software artifacts from natural language descriptions would enable workers with less expertise than today to take part in the software development process. It would permit faster, more reliable, and more deterministic software development. "Text to Software" is a long-term vision that will take decades to realize. Given some early successes, it is a goal that seems attainable and might yield deep insights into how software is actually constructed.

In the following, we expand on the idea of generating models and test cases from text and discuss an important methodological issue (benchmarking).

2. MODELS FROM TEXT

Extracting models from text is all about semantics; the syntax of text is not important. We extract semantic information from text using an extended set of Fillmore's thematic roles [8]. In a first approach, requirements texts had to be annotated with these roles by hand. This approach allowed us to determine the semantic content that was needed for generating software models, without having to solve the general natural language processing problem. As a next step, we are now investigating how the appropriate thematic roles can be discovered automatically [19].

Our approach to automating natural language requirements processing (NLRP) is depicted in Fig. 3. First we improve the textual specification itself (1), since many flaws in software development result from faulty requirements. Changes are made with customers' approval; working with text makes this easy for the customers. The resulting specification is then translated into an internal graph structure that is machine processable (2). The internal graph is based on the semantics of the text, not its syntax. Our extraction tool then generates UML models (3). These models could be passed to model driven architecture (MDA) tools (4). The idea of MDA is to create executable code from UML models (5).

2.1 Automatic Specification Improvement

It is well-known that requirements documents are often incomplete, inaccurate, faulty, or contradictory. Today's requirements engineering tools (for an extensive list see Volere [25]) offer modeling and management functions, but can

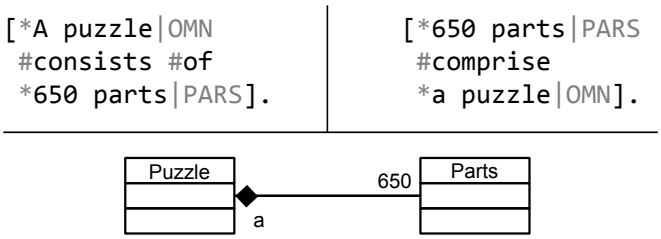

Figure 4: SaleMX Syntax: Semantics over Syntax

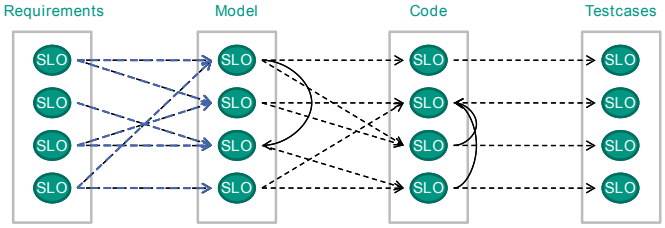

Figure 5: Software Lifecycle Objects

hardly cope with the content and meaning of natural language specifications. Therefore, analysts must inspect specifications for problems and resolve them in dialog with customers. Step-by-step rules [11, 2, 7] have been proposed for improving specifications manually.

Tools such as RESI [16] and RAT [24] help the analyst identify some of these problems. RESI relies on Cyc for supplying world knowledge, i.e. common sense to make decisions. As an example, consider the requirements statement `Username and password are entered`. Here it is unclear who enters the *password* and where. With Cyc, RESI identifies the word "enter" as an incompletely specified process word and asks the user for clarification. Thus, RESI helps by identifying problem spots, but it does not replace the human decision maker except in clear cases.

2.2 Creating UML Models

Figure 4 shows two example sentences describing the same situation. Both sentences are syntactically quite different, but describe exactly the same facts. The two sentences result in the same UML model, shown underneath. We use semantic annotations with thematic roles which, in this example, describe a composition: `OMN` (Omnium: the whole) and `PARS` (Pars: parts of it). A collection of 60 thematic roles and associated transformation rules generate UML models directly from text.

During the requirements engineering part of software engineering, there will always be parts that require user interaction. These parts cannot be automated, but a tool should limit user interaction to a minimum and automate all simple or tedious work. Our research prototypes indicate that such tools are feasible [9, 18, 17].

2.3 Enabling Round Trip Engineering

The process of requirements engineering is iterative. As depicted in Fig. 2[1], the stakeholder and the analyst need to synchronize their understanding of the project many times before the actual software can be developed . Also, the effort of synchronization sometimes excels the effort of the actual implementation. This is one of the main reasons why software requirements, implementation, and documentation underly an erosion process. Considering the fact that requirements specifications often found the base for legally binding contracts, we postulate the necessity to maintain the connection between Software Lifecycle Objects (SLOs) [1] by offering a complete round trip engineering approach.

Bohner and Arnold describe a SLO as any entity used in the software development cycle. It has dependencies to and from other SLOs. A SLO can be anything: a requirement statement, a model element, a segment of code, a test case, etc. SLOs exist during the lifetime of a product and have $n-ary$ relations with each other. An example can be seen in Fig. 5.

Vertical dependencies describe the mutual connection of SLOs within the same type of SLO, such as requirements, models, code, and test cases. Horizontal dependencies show that each SLO can be coupled with one or many other SLOs of other vertical types. Creating and maintaining these dependencies is costly. Helping the software analyst, architect, and stakeholder to maintain the connections among requirements, models, code, and tests would improve software development dramatically. For example, changes in the models should be reflected back in the requirements. Changes in the code might cause changes to test cases, models, and specifications.

One of our current research projects aims at feeding UML model changes back into the existing textual requirements. The changes are highlighted in the text, so the analyst can spot them easily. For example, Fig. 6 shows two UML models. The model on the left is the original model created from the text underneath it. The model on the right depicts the altered model. The text shown in grey shows the automatic changes that were performed on the specification in response to the model changes on the right. This tool connects the software model to the actual software requirements specification and keeps the two SLOs consistent. The tool provides valuable help in the iterative change process with stakeholders.

3. TEST CASES FROM TEXT

Generating test cases from text is much more speculative than generating models from text, as we have no preliminary results to support our arguments. However, it appears plausible that as soon as we bring (automated) semantic knowledge into the software development process, we may be able to automate other steps.

3.1 Test Cases from APIs

Test case generation is an active research area. Existing techniques use code or formal specifications, but none work off natural language specifications, even though that is by far the most common form of API specification. For example, consider the API of a stack in *java.util.stack*. A useful test would push elements on the stack and then pop them off and check for LIFO order. At present, this would be a test that is exceedingly difficult to generate. Using the textual API specification and the world knowledge of ontologies, it might be possible to derive such a test in the future.

For example, the ontology Cyc contains knowledge about data structures and describes a stack as *an organized pile*

[1]Figure according to Dr. Martin Glinz, University of Zürich

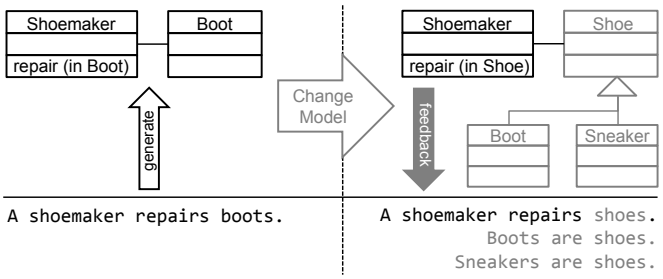

Figure 6: Changes in UML Alter Specifications

with its contents lying *on top of another*. It also defines LIFO and FIFO order. By exploiting this information, it might be possible to generate useful tests, not only about stacks but other container data structures as well. In particular, one would have to use the concept of storage to generate tests that insert data, extract it again, and compare the result. To make progress in this difficult area, one should start with simple cases, such as tests that exercise setters and getters, and then progress to more difficult semantics.

3.2 Test Cases from Models

Some research tools already create test cases from models, such as UML models [12]. Since we generate models from text, we should be able to use such test generation tools directly. Understanding the concept of data structures and knowing the relationships of the different entities (as given by UML classes), one could even create test oracles. Again, this might work by utilizing the model and additional information from ontologies. We envision transformation rules that create test cases directly from enriched models and additional constraints expressed in the requirements text.

This process could be improved using domain-specific ontologies. Domain-specific ontologies allow the discovery of semantics with higher precision and detail than general world knowledge.

3.3 Extracting Timelines

Discovering temporal constraints (before/after relations) in textual specifications is already partly possible. UML state or sequence charts are generated from these constraints. This information could be used for generating assertions that check that the ordering constraints are actually obeyed during execution.

For example, a *WrongOrderException()* would be thrown whenever the execution sequence does not follow the order given in the text or API. This part would improve the automatic exception coding and handling and support developers to not miss (timely) constraints. Again, generating test cases from text is an extremely difficult problem. Whether workable methods will be discovered cannot be predicted with any certainty at this time.

4. BENCHMARKS

Benchmarks will be important for progress in this area. A benchmark for natural language requirements processing (NLRP) would contain requirements documents of graded difficulty. Each requirement document would be accompanied by a "solution", i.e., a list of which model elements should be extracted, what inaccuracies or ambiguities are present, or what aspects should be tested. Independent teams can then apply their NLRP tools to the benchmark and measure recall and precision objectively. Initially, researchers would start with short and simple requirements, and over time the documents would become longer and more difficult to handle. As with all benchmarks, they need to evolve in order to prevent overfitting.

Benchmarks help generate quantifiable, objective results and foster a competitive climate that accelerates progress. Benchmarks have been extremely successful in driving research in computer architecture, speech recognition and translation, robotics, databases, SAT solving, and other areas. A particular successful example is the DARPA challenge for robotic vehicles. In all of these cases, benchmarks resulted in swift and substantial progress. Successful techniques were quickly adopted by other teams and improved upon. By providing benchmarks for NLRP, the research community might achieve similar progress. We have already begun to collect examples of requirements documents and will make them available on a website. A first version of the website is available [13].

It turns out that real requirements are surprisingly difficult to find. Textbooks contain few examples, and they seem to be made up by the authors or copied from other authors, who also made them up. Real requirements are quite different from textbook examples, but we also found that companies are reluctant to make their requirements documents publicly available. Furthermore, research in NLRP often use artificial, strongly restricted languages. There seems to be a need for realistic requirements with which to improve tools.

5. CONCLUSIONS

Any user of the web has noticed that search engines can translate web pages. These translations are not perfect, but surprisingly useable. On mobile phones, speech-to-speech translators are now available (see for instance Gibbigo in Apple's Appstore). A user of the translator speaks into the phone in English and the phone produces a spoken Spanish translation; the reverse direction works as well. This technology seems to be straight out of a science-fiction book, yet it is real and improving rapidly. Why should automatic translation of requirements to software artifacts not be possible? At present, machine translation relies on statistical methods, and perhaps these methods are not applicable to software because the statistical basis is too small. Thus, researcher will have to use other techniques, such as ontologies, but we also expect the statistical basis to improve with time. Early successes with generating models from text are quite encouraging. Software engineering researchers should take note of the advances in machine translation and semantic technologies and capitalize on them. Generating software from textual description would without a doubt be a significant step forward.

6. REFERENCES

[1] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[2] M. Ceccato, N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry. Ambiguity identification and measurement in natural language texts.

[3] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *Proc. Future of Software Engineering FOSE '07*, pages 285–303, 23–25 May 2007.

[4] Cycorp Inc. ResearchCyc. http://research.cyc.com/.

[5] D. K. Deeptimahanti and R. Sanyal. An innovative approach for generating static UML models from natural language requirements. In *Advances in Software Engineering*, volume 30 of *Communications in Computer and Information Science*, pages 147–163. Springer Berlin Heidelberg, 2009.

[6] C. Denger, D. M. Berry, and E. Kamsties. Higher quality requirements specifications through natural language patterns. volume 0, page 80, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[7] C. R. . die SOPHISTen. *Requirements-Engineering und -Management*. Carl Hanser Verlag, 4 edition, 2006.

[8] C. J. Fillmore. Toward a modern theory of case. In D. A. Reibel and S. A. Schane, editors, *Modern Studies in English*, pages 361–375. Prentice Hall, 1969.

[9] T. Gelhausen and W. F. Tichy. Thematic Role based Generation of UML Models from Real World Requirements. In *First IEEE International Conference on Semantic Computing (ICSC 2007)*, volume 0, pages 282–289, Irvine, CA, USA, Sept. 2007. IEEE Computer Society.

[10] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, Englewood Cliffs, NJ, 2. ed., [pearson international edition] edition, 2009.

[11] E. Kamsties, D. M. Berry, and B. Paech. Detecting Ambiguities in Requirements Documents Using Inspections. In *Proceedings of the First Workshop on Inspection in Software Engineering (WISE'01)*, pages 68–80, 2001.

[12] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz. Less is more: A minimalistic approach to uml model-based conformance test generation. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:82–91, 2008.

[13] KIT. The NLRP Benchmark Homepage, 2010. http://nlre.wikkii.com.

[14] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry. Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requir. Eng.*, 13(3):207–239, 2008.

[15] L. Kof. Natural Language Processing for Requirements Engineering: Applicability to Large Requirements Documents. In *Automated Software Engineering, Proceedings of the Workshops*, Linz, Austria, Sept. 2004.

[16] S. J. Körner and T. Brumm. RESI - A Natural Language Specification Improver. *International Conference on Semantic Computing*, 0:1–8, 2009.

[17] S. J. Körner and T. Brumm. Natural Language Specification Improvement with Ontologies. *International Journal of Semantic Computing (IJSC)*, 03(04):445–470, 2010.

[18] S. J. Körner and T. Gelhausen. Improving Automatic Model Creation using Ontologies. In Knowledge Systems Institute, editor, *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering*, pages 691–696, July 2008.

[19] S. J. Körner and M. Landhäußer. Semantic Enriching of Natural Language Texts with Automatic Thematic Role Annotation. NLDB 2010, June 2010.

[20] L. Mich, M. Franch, and P. N. Inverardi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56, Feb. 2004.

[21] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[22] D. L. Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335, 1985.

[23] V. S. Sharma, S. Sarkar, K. Verma, A. Panayappan, and A. Kass. Extracting high-level functional design from software requirements. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 35–42, Washington, DC, USA, 2009. IEEE Computer Society.

[24] K. Verma and A. Kass. Requirements analysis tool: A tool for automatically analyzing software requirements documents. In *7th International Semantic Web Conference (ISWC2008)*, October 2008.

[25] Volere. List of requirement engineering tools, 2009. http://www.volere.co.uk/tools.htm.