2007-09

Innovations for Requirements Analysis, From Stakeholders' Needs to Formal Designs

Monterey, California: Naval Postgraduate School.

Lecture Notes in Computer Science 5320

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Barbara Paech Craig Martell (Eds.)

Innovations for Requirements Analysis

From Stakeholders' Needs to Formal Designs

14th Monterey Workshop 2007
Monterey, CA, USA, September 10-13, 2007
Revised Selected Papers

Springer

Volume Editors

Barbara Paech
University of Heidelberg
Im Neuenheimer Feld 326
69120, Heidelberg, Germany
E-mail: paech@informatik.uni-heidelberg.de

Craig Martell
Naval Postgraduate School
Department of Computer Science
1411 Cunningham Road
Monterey, CA 93943, USA
E-mail: cmartell@nps.edu

Library of Congress Control Number: 2008940145

CR Subject Classification (1998): D.2.1, I.2.7, H.5.2, I.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-89777-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-89777-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12573790 06/3180 5 4 3 2 1 0

Preface

We are pleased to present the proceedings of the 14th Monterey Workshop, which took place September 10–13, 2007 in Monterey, CA, USA. In this preface, we give the reader an overview of what took place at the workshop and introduce the contributions in this Lecture Notes in Computer Science volume. A complete introduction to the theme of the workshop, as well as to the history of the Monterey Workshop series, can be found in Luqi and Kordon's "Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs" in this volume. This paper also contains the case study that many participants used as a problem to frame their analyses, and a summary of the workshop's results.

The workshop consisted of three keynote talks, three panels, presentations of peer-reviewed papers, as well as presentations of various position papers by the participants.

The keynote speakers at this year's workshop were Daniel Berry, Aravind Joshi, and Lori Clarke. Each of their talks was used to set the tone for the presentations and discussions for that particular day. Daniel Berry presented an overview of the needs and challenges of natural language processing in requirements engineering, with a special focus on ambiguity in his talk "Ambiguity in Natural Language Requirements." Aravind Joshi provided an overview of current natural language processing research in discourse analysis in the talk "Some Recent Developments in Natural Language Processing." Finally, Lori Clarke showed how to combine formal requirements specification with natural language processing to cope with the complex domain of medical information processes in "Getting the Details Right." We are grateful to each of them for their time and energy. For extended abstracts of the talks, please see "Part I: Abstracts" in this volume.

The panels examined a wide range of topics related to natural language processing and requirements engineering. The active discussions that took place at these panels stimulated many ideas for both the workshop and for the papers presented here. The titles and participants of the panels were:

1. *Advances in Requirements Engineering*

   Chairs: Christine Choppy (University of Paris 13), Sol Shatz (University of Illinois at Chicago)
   Panelists: Jeff Besser (SPAWAR), John Gibson (Naval Postgraduate School), Douglas Lange (SPAWAR), Julio Leite (PUC-Rio), and Steve Yau (Arizona State University).
   Date: September 11, 2007

2. *State of the Art in Natural Language Processing and Requirements Engineering*

   Chairs: Michel Lemoine (ONERA) and Kane Kim (University of California, Irvine)
   Panelists: Swappan Bhattacharya (National Institute of Technology, Durgapur), Nabendu Chaiki (University of Calcutta), Alan Rieffer (DISA), Chen-Yu (Phillip) Sheu (University of California, Irvine), and Oleg Sokolsky (University of Pennsylvania).
   Date: September 12, 2007

3. *Pro's and Con's of Proposed Approaches for Requirements Engineering*

   Chairs: Doris Carver (Louisiana State University) and Daniel Cooke (Texas Tech University)
   Panelists: Mikhail Auguston (Naval Postgraduate School), Valdis Berzins (Naval Postgraduate School), David Hislop (U.S. Army Research Office, Retired), Mohammad Ketabchi (Savvion), Peter Musial (VeroModo, Inc.), William Roof (IntelliDOT Corporation), Nelson Rushton (Texas Tech University), John Salasin (National Institute of Standards).
   Date: September 13, 2007

Finally, of the papers presented at the workshop, the authors of 11 were invited to revise and expand their papers. These make up "Part II: Papers" in this volume. The papers fell into two broad categories:

1. Innovative requirements engineering techniques
2. Innovative applications of natural language processing techniques

1 Innovative Requirements Engineering Techniques

The six papers in this group present several challenges for requirements engineering and discuss innovative solution ideas.

In "Could an Agile Requirements Analysis Be Automated?—Lessons Learned from the Successful Overhauling of an Industrial Automation System," Thomas Aschauer, Gerd Dauenhauer, Patricia Derler, Wolfgang Pree, and Christoph Steindl describe a recent successful requirements analysis of a complex industrial automation system that combined a talented expert, who was willing to dig into the domain details, with a committed customer and a motivated team. Martin Feather, in "Defect Detection and Prevention," presents the DDP process and tool which supports the exploration of and decision-making for complex requirements documents. His abstract (to be found in "Part I") characterizes and summarizes the most important literature on this approach. In "Model-Driven Prototyping-Based Requirements Elicitation," Jicheng Fu, Farokh Bastani, and I-Ling Yen present a requirements elicitation approach that is based on model-driven prototyping. They apply a "rapid program synthesis" approach to speed up prototype development. Michael Goedicke and Thomas Herrmann, in "A Case for ViewPoints and Documents," consider how various stakeholders provide their requirements from different points of view, and how to deal with the

fact that these various points of view can often lead to vague and inconsistent requirements specifications. Allyson Hoss and Doris Carver, in "Towards Combining Ontologies and Model Weaving for the Evolution of Requirements Models," address the challenges of software change that result from adding new requirements. They do this by combining ontologies and model weaving to assist in software evolution. Finally, in "Reducing Ambiguities in Requirements Specifications via Automatically Created Object-Oriented Models," Daniel Popescu, Spencer Rugaber, Nenad Medvidovic, and Daniel Berry describe a three-step, semi-automatic method for identifying inconsistencies and ambiguities in requirement specifications. Their method automatically generates a diagram of the objects, classes and methods of the specified system for a human to review.

2 Innovative Applications of Natural-Language Processing Techniques

The five papers in this section all deal, in some way, with using natural language processing to help with the requirements engineering process.

Valdis Berzins, Craig Martell, Luqi, and Paige Adams, in "Innovations in Natural Language Document Processing for Requirements Engineering," evaluate the potential contributions of natural language processing to requirements engineering and suggest some improvements to natural language processing systems that may be useful in this context. Nikhil Dinesh, Aravind Joshi, Insup Lee, and Oleg Sokolsky, in "Logic-Based Regulatory Conformance Checking," describe an approach to formally assess whether an organization conforms to a body of regulation. This is done via a logic in which statements can formally refer to and reason about other statements. They present preliminary work on using natural language processing to assist in the translation of regulatory sentences into this logic. In "On the Identification of Goals in Stakeholders Dialogs," Leonid Kof shows that the often unstated, and sometimes unknown, goals of stakeholders can lead to contradictory requirements, and that making these goals explicit as early in the process as possible facilitates the resolution of these contradictions. He describes how these goals can be derived by systematic analysis of stakeholders' dialogs. Douglas Lange, in "Text Classification and Machine Learning Support for Requirements Analysis Using Blogs," describes how text classification and machine learning technologies are being use to support management requirements in military command centers. He then explores how these technologies might be used in a requirements analysis environment. Finally, in "Profiling and Tracing Stakeholder Needs," Pete Sawyer, Ricardo Gacitua, and Andrew Stone show how shallow natural language techniques can be used to assist in the analysis of stakeholder-elicited information and help with the synthesis of the user requirements. These same techniques can be used for subsequent management of requirements and in identifying unprovenanced requirements.

It has been a pleasure and an honor to serve as Program Committee Chairs for the 2007 Monterey Workshop. First of all, we would like to thank the Workshop

Chairs, Luqi and Fabrice Kordon, for their continuous support and advice during the workshop and the preparation of these proceedings. Secondly, we would like to thank the members of the Program Committee, who acted as anonymous reviewers and provided valuable feedback to the authors. We are also grateful to the authors for their active participation in the workshop and their timely responses during the preparation of the proceedings. Doris Keidel-Müller was a great help in reviewing the layout of the papers.

Finally, none of this would have worked as smoothly as it did without the continuous support of Willi Springer. Many thanks!

September 2008

Barbara Paech
Craig Martell

Organization

The Monterey 2007 Workshop was run by an Organizing Committee of two Workshop Chairs and a Technical Program Committee.

Workshop Chairs

Luqi	Naval Postgraduate School, Monterey, USA
Fabrice Kordon	Pierre & Marie Curie University, Paris, France

Technical Program Committee

Barbara Paech	University of Heidelberg, Germany
Craig Martell	Naval Postgraduate School, Monterey, USA

Program Committee

Daniel M. Berry	University Waterloo, Canada
Christine Choppy	University Paris XIII, France
Steven Clark	Oxford University, UK
Lori A. Clarke	University of Massachusetts, USA
Rance Cleveland	University of Maryland, USA
Vincenzo Gervasi	University of Pisa, Italy
Aravind Joshi	University of Pennsylvania, USA
Kane Kim	University of California, Irvine, USA
Leonid Kof	Technical University of Munich, Germany
Fabrice Kordon	Pierre & Marie Curie University, Paris, France
Bernd Krämer	FernUniversität Hagen, Germany
Mitch Marcus	University of Pennsylvania, USA
Bashar Nuseibeh	The Open University, UK
Manuel Rodriguez	National Research Council, USA
Sol Shatz	University of Illinois at Chicago, USA
Phillip Sheu	University of California, Irvine, USA

Table of Contents

Part I: Abstracts

Part II: Papers

Innovative Requirements Engineering Techniques

Innovative Applications of Natural-Language Processing Techniques

Ambiguity in Natural Language Requirements Documents
(Extended Abstract)

Daniel M. Berry

Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
dberry@uwaterloo.ca

1 Introduction

This paper is an extended abstract of an invited talk at the workshop that I put together using material from other talks and from papers that I and colleagues have written. The purposes of this extended abstract are to summarize the talk and to allow the reader to find the source materials for the talk directly.

2 Natural Language Is Key in Requirements Engineering

An overwhelming majority of requirements specifications (RSs) are written in natural language (NL). Virtually every initial conception for a system is written in NL. Virtually every request for proposal (RFP) is written in NL [1]. However, we all know that NL is *so* ambiguous, and so inherently so. No wonder RSs are such messes.

There is an old tradeoff: A RS can be written (1) in a NL or (2) in a mathematics-based (MB) formal language (FL)[1]. A NL has the disadvantage that it is inherently ambiguous, but the advantages (1) that there is always someone who can write with it and (2) that a RS written in it is always more or less understood by all stakeholders, albeit somewhat differently by each. A MB FL has the advantage that it is inherently unambiguous, but the disadvantages (1) that there is not always someone who can write a RS with it and (2) that a RS written with it is not understood by most stakeholders, although all that do understand it understand it the same.

A lot of research in requirements engineering (RE) is directed at solving the problem of ambiguous RSs by (1) convincing people to use MB FLs and (2) addressing the negatives of MB FLs, by making them more accessible [3], sometimes with the help of tools [4,5].

However, the reality is that there is no escaping NL RSs. Michael Jackson [6] reminds us that "Requirements engineering is where the informal meets the formal." In order to write software, ideas, which are inherently informal, have to be converted

[1] The term "mathematics-based" is used to distinguish the kinds of FLs I am referring to from other semi-formal notations, e.g., UML [2], that are often called "formal".

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 1–7, 2008.

somehow to code, which is inherently formal. There needs to be a transition from informal to formal somewhere along the way from ideas to code. That transition generally happens the first time ideas are written in an even informal notation, during RE. Therefore, NLs are inevitable, even if it is only for the initial conception.

Even if one moves immediately to FLs, the inherent ambiguity of the NL initial conception can strike as the transition is made. What the formalizer understands of the conception may be different from what the conceiver meant. The phenomenon of *subconscious disambiguation* strikes [7].

In subconscious disambiguation, the reader of an ambiguous phrase is not even aware that there is an interpretation other than the one that came first to his or her mind. The reader understands an interpretation and thinks that it is the only one. In fact, *here* is where it is most important to catch ambiguity: right up front, when the requirements analyst (RA) is getting raw information, be it goals, business rules, or requirements, from the clients and users. The RA must find *each* ambiguity and ask the clients and users what they mean with it. The flip side of subconscious disambiguation is *subconscious ambiguation*, the inadvertent introduction of ambiguity during writing by an author who believes that all readers will understand what he or she was thinking during the writing.

In a semi-formal language such as any of the UML notations, there are two sources of ambiguity. Ambiguity can still strike when going from the conception to a model, and the model itself is not uniguous[2].

Therefore, there is a group of researchers focusing on solving the problem of ambiguous RSs by trying to improve our writing, understanding, and processing of NLs.

3 Avoiding or Detecting Ambiguities

There are several approaches to avoiding the ambiguity of NLs:

1. Learn to write less ambiguously, avoiding those constructions that tend to create ambiguities [8,9,10,11,12].
2. Learn to detect ambiguity either manually [13,14,15], or with the help of tools [16,17,18,19,20,21,22]. Manual detection is helped by being able to recognize constructions that tend to create ambiguities. Someone who is aware of writing pitfalls can detect ambiguities manually more easily than someone who is not aware of the pitfalls.
3. Use a restricted NL which is inherently unambiguous [23,24,25] but may not be so natural.

4 Taxonomy and Defintions of Ambiguity

Figure 1 shows a taxonomy of the kinds of ambiguity that can be encountered in a NL RS [26]. Most of the tree is based on the traditional ambiguity-and-related-phenomena literature [e.g., 27]. The new portions of the tree are based on more recent literature about software-engineering ambiguity [28,29,30,31] and language-error ambiguity

[2] "Uniguous" means "not ambiguous".

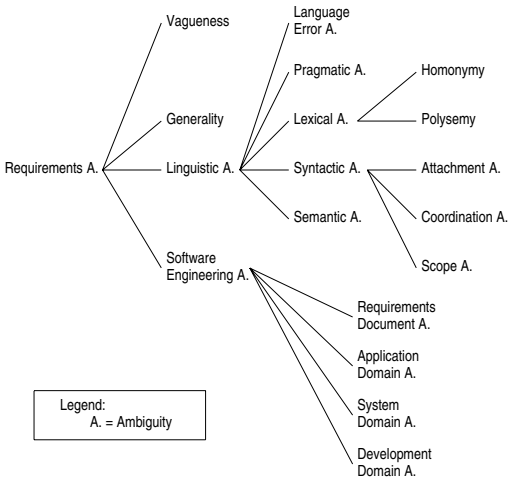


Fig. 1. Taxonomy of Ambiguity Types

[9,10]. Berry and Kamsties describe all of these types of ambiguities, including the new ones, with the help of examples [26].

Probably the most useful of these definitions from RE viewpoint is that of Alan Davis [31], who has has suggested a test for ambiguity to serve as a definition: "Imagine a sentence that is extracted from an SRS, given to ten people who are asked for an interpretation. If there is more than one interpretation, then that sentence is probably ambiguous." The problem with this test is that, as in software testing, there is no guarantee that the eleventh person will not find another interpretation. However, this test does capture the essence of a *useful* SRS, which is unambiguous for most practical purposes.

Our experience has identified another category of ambiguity, *language error*. As with all other categories, language error may not be mutually exclusive of other categories. A language-error ambiguity occurs when a grammatical, punctuation, word choice, or other mistake in using the language of discourse leads to text that is interpreted by a receiver as having a meaning other than that intended by the sender. The most common language-error ambiguities are:

– misplacement of only and also, e.g., does the author of

  The spam filter only marks the e-mail it considers to be spam.

  mean

  (1) what is actually written,

  (2) The spam filter marks only the e-mail it considers to be spam.,

  or

  (3) The spam filter only marks only the e-mail it considers to be spam.;

– use of all or a plural as a sentence subject, leaving unclear whether the phenomenon described in the sentence applies to each element of the subject set or to the whole set, e.g., as would happen with

    Students enroll in six courses per term.

  and

    Students enroll in hundreds of courses per term.

  when the reader has no domain knowledge[3];
– a pronoun with an unclear referrent, e.g., to what part of the preceding text does the This in the sentence

    This prevents security breaches.

  refer?

They are certainly very difficult to detect by someone who is not aware of the problems [10].

5 Tools

Given any process in SE or RE, we want to build a tool that does the process or at least helps carry it out, and the ambiguity-in-RE area is no different. However, such tools a good idea? Back in 1993, Kevin Ryan concluded that for the foreseeable future, AI approaches to language understanding do not work well enough for RE work [32].

I would add that such a tool would not even be desirable, because it would take the thinking requirements analyst out of the loop, making it less likely that he or she would notice serious omissions and questionable, albeit logically okay, requirements. While a fully understanding NLP tool is out of the realm of possibility or desires, there are ways that less powerfull NLP tools can help the practicing, thinking requirements analyst find instances of potentially ambiguous sentences [17,18,19,22,33].

The total amount of information to deal with for any real problem is *huge* and repetitive. We desire assistance in extracting useful information that is less than what is in the original document, with 100% recall of the information in the original document and with 100% precision. That is, from a 500 page RS, we want 5 pages containing *all* and *only* the meaningful information in the 500 page RS.

A tool that looks for instances of a particular ambiguity should have 100% recall, even at the expense of some imprecision, i.e., false positives; otherwise, the user will learn not to trust the tool to find every instance, and the user will end up manually searching the whole document anyway. Thus, a tool should not be based on a NLP process, e.g., parsing, that inherently has less than 100% recall. Instead, the tool must be based on a process that has 100% recall *by design*, e.g. a lexical analyzer looking for all instances of particular keywords [22]. For example, a tool that finds every instance of the word only is useful, because it is guaranteed to find every instance even though the user will suffer some imprecision as he or she examines each instance to decide if the instance is ambiguous. Actually, a little nonburdensome imprecision may help keep the human RA engaged, especially if the instances of imprecision are funny examples of the stupidity of computers and algorithms!

[3] I now avoid writing sentences with plural subjects except when talking about properties of a whole set.

6 Conclusion

NL is unavoidable in RSs, even if only at the very beginning when you are talking with the client. Subconscious ambiguation strikes in writing. Subconscious disambiguation strikes in reading.

Ambiguity abounds in places you never even thought of, e.g., in only, in all, and in plural.

Any tool must have 100% recall and good summarization at the expense of some imprecision.

The most important lesson of this talk is that in reality, we are never going to prevent ambiguity. So we must learn to spot it, not only in polished RSs, but also, and especially, *in goals, business rules, and initial RSs*, in whose reading subconscious disambiguation first strikes. Then we must ask the client what he or she means.

Acknowledgments

Berry's work was supported in part by NSERC grant NSERC-RGPIN227055-00.

References

1. Mich, L., Franch, M., Inverardi, P.N.: Market research for requirements analysis using linguistic tools. Requirements Engineering Journal 9, 40–56 (2004)
2. Rumbaugh, J., Jacobson, I., Booch, G. (eds.): The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
3. Henninger, K., Kallander, J., Shore, J., Parnas, D.: Software requirements for the A-7E aircraft. NRL Memorandum Report 3876, Naval Research Laboratory, Washington, DC, USA (1978)
4. Bharadwaj, R., Heitmeyer, C.: Model checking complete requirements specifications using abstraction. Automated Software Engineering 6, 37–68 (1999)
5. Heitmeyer, C.L., Kirby, J., Labaw, B.G.: The scr method for formally specifying, verifying, and validating requirements: Tool support. In: Proceedings of the 19th International Conference on Software Engineering ICSE 1997, pp. 610–611 (1997)
6. Jackson, M.A.: The role of architecture in requirements engineering. In: Proceedings of the IEEE International Conference on Requirements Engineering, vol. 241, IEEE Computer Society, Los Alamitos (1994)
7. Gause, D.C.: User DRIVEN Design—The Luxury that has Become a Necessity, A Workshop in Full Life-Cycle Requirements Management. ICRE 2000 Tutorial T7, Schaumberg, IL, USA (2000)
8. Götz, R., Rupp, C.: Regelwerk natürlichsprachliche methode. Technical report, Sophist (1999), `http://www.sophist.de`
9. Berry, D., Kamsties, E.: The syntactically dangerous *all* and plural in specifications. IEEE Software 22, 55–57 (2005)
10. Berry, D., Kamsties, E., Krieger, M.: From contract drafting to software specification: Linguistic sources of ambiguity. Technical report, University of Waterloo, Waterloo, ON, Canada (2003), `http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf`

11. Kovitz, B.L.: Practical Software Requirements: A Manual of Content and Style. Manning, Greenwich, CT, USA (1998)
12. Dupré, L.: Bugs in Writing: A Guide to Debugging Your Prose, 2nd edn. Addison-Wesley, Reading (1998)
13. Kamsties, E., Berry, D., Paech, B.: Detecting ambiguities in requirements documents using inspections. In: Lawford, M., Parnas, D.L. (eds.) Proceedings of the First Workshop on Inspection in Software Engineering (WISE 2001), pp. 68–80 (2001)
14. Kamsties, E.: Surfacing Ambiguity in Natural Language Requirements. PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany(2001); also Volume 5 of Ph.D. Theses in Experimental Software Engineering, Fraunhofer IRB Verlag, Stuttgart, Germany (2001)
15. Denger, C.: High quality requirements specifications for embedded systems through authoring rules and language patterns. Master's thesis, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany (2002)
16. Osborne, M., MacNish, C.: Processing natural language software requirement specifications. In: Proceedings of the International Conference on Requirements Engineering (ICRE 1996), pp. 229–236 (1996)
17. Wilson, W.M., Rosenberg, L.H., Hyatt, L.E.: Automated analysis of requirement specifications. In: Proceedings of the Nineteenth International Conference on Software Engineering ICSE 1997, pp. 161–171. ACM Press, New York (1997)
18. Mich, L., Garigliano, R.: Ambiguity measures in requirement engineering. In: Feng, Y., Notkin, D., Gaudel, M. (eds.) Proceedings of International Conference on Software—Theory and Practice ICS 2000. Sixteenth IFIP World Computer Congress, pp. 39–48. Publishing House of Electronics Industry, Beijing (2000)
19. Kiyavitskaya, N., Zeni, N., Mich, L., Berry, D.M.: Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. Requirements Engineering Journal 13, 207–240 (2008)
20. Berry, D.M., Bucchiarone, A., Gnesi, S., Lami, G., Trentanni, G.: A new quality model for natural language requirements specifications. In: Proceedings of the International Workshop on Requirements Engineering: Foundation of Software Quality, REFSQ 2006 (2006)
21. Tjong, S., Hartley, M., Berry, D.: Extended disambiguation rules for requirements specifications. In: Proceedings of Workshop in Requirements Engineering, WER (2007), `http://wer.inf.puc-rio.br/index.html`
22. Tjong, S.F.: Avoiding Ambiguity in Requirements Specifications. PhD thesis, Faculty of Engineering & Computer Science, University of Nottingham, Malaysia Campus, Semenyih, Selangor Darul Ehsan, Malaysia (2008)
23. Comer, J.: An experimental natural-language processor for generating data type specifications. SIGPLAN Notices 18, 25–33 (1983)
24. Enomoto, H., Yonezaki, N., Saeki, M., Chiba, K., Takizuka, T., Yokoi, T.: Natural language based software development system tell. In: O'Shea, T. (ed.) Advances in Artificial Intelligence, ECAI 1984, pp. 721–731. Elsevier, Amsterdam (1984)
25. Fuchs, N., Schwertel, U., Schwitter, R.: Attempto controlled english (ace) language manual version 3.0. Technical Report No. 99.03, Institut für Informatik der Universität Zürich, Zürich, Switzerland (1999)
26. Berry, D.M., Kamsties, E.: Ambiguity in requirements specification. In: Leite, J., Doorn, J. (eds.) Perspectives on Requirements Engineering, pp. 7–44. Kluwer, Boston (2004)
27. Lyons, J.: Semantics I and II. Cambridge University Press, Cambridge (1977)
28. Schneider, G.M., Martin, J., Tsai, W.T.: An experimental study of fault detection in user requirements documents. ACM Transactions on Software Engineering and Methodology 1, 188–204 (1992)

29. Gause, D.C., Weinberg, G.M.: Exploring Requirements: Quality Before Design. Dorset House, New York (1989)
30. IEEE: IEEE Recommended Practice for Software Requirements Specifications, ANSI/IEEE Standard 830-1993. Institute of Electrical and Electronics Engineering, New York, NY, USA (1993)
31. Davis, A.M.: Software Requirements: Objects, Functions, and States. Prentice-Hall, Upper Saddle River (1993)
32. Ryan, K.: The role of natural language in requirements engineering. In: Proceedings of the IEEE International Symposium on Requirements Engineering (ISRE 1993), CA, USA, pp. 240–242. IEEE Computer Society Press, Los Alamitos (1993)
33. Bucchiarone, A., Gnesi, S., Pierini, P.: Quality analysis of NL requirements: An industrial case study. In: Proceedings of the Thirteenth IEEE International Requirements Engineering Conference (RE 2005), pp. 390–394 (2005)

Towards Discourse Meaning

Aravind K. Joshi

Department of Computer and Information Science and Institute for Research in Cognitive Science
University of Pennsylvania Philadelphia PA USA
joshi@seas.upenn.edu

Abstract. The overall goal is to discuss some issues concerning the dependencies at the discourse level and at the sentence level. However, first I will briefly describe the Penn Discourse Treebank (PDTB)*, a corpus in which we annotate the discourse connectives (explicit and implicit) and their arguments together with "attributions" of the arguments and the relations denoted by the connectives, and also the senses of the connectives. I will then focus on the complexity of dependencies in terms of (a) the elements that bear the dependency relations, (b) graph theoretic properties of these dependencies such as nested and crossed dependencies, dependencies with shared arguments, and (c) attributions and their relationship to the dependencies, among others. I will compare these dependencies with those at the sentence level and discuss some issues that relate to the transition from the sentence level to the level of "immediate discourse" and propose some conjectures.

An increasing interest in moving human language technology beyond the level of the sentence in text summarization, question answering, and natural language generation , among others, has recently led to the development of several resources that are richly annotated at the discourse level. Among these is the Penn Discourse TreeBank. (PDTB), a large-scale resource of annotated discourse relations and their arguments over the one million word Wall Street Journal (WSJ) Corpus. Since the sentence-level syntactic annotations of the Penn Treebank [2] and the predicate-argument annotations of the Propbank [4] have been done over the same target corpus, the PDTB thus provides a richer substrate for the development and evaluation of practical algorithms while supporting the extraction of useful features pertaining to syntax, semantics and discourse all at once. The PDTB is the first to follow a lexically - grounded approach to the annotation of discourse relations. Discourse relations, when realized explicitly in the text, are annotated by marking the necessary lexical items – called discourse connectives - expressing them, thus supporting their automatic identification.

PDTB adopts a theory-neutral approach to the annotation, making no commitments to what kinds of high-level structures may be created from the low level annotations of relations and their arguments. This approach has the appeal of allowing the corpus to be useful for researchers working within different frameworks. This theory neutrality also permits investigation of the general question of how structure at the sentence level relates to structure at the discourse level, at least that part of the discourse structure that is “parallel” to the sentence structure [6]. In addition to the argument structure of discourse relations, the PDTB provides sense labels for each relation following a hierarchical classification scheme. Annotation of senses highlights the polysemy of

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 8–9, 2008.

connectives, making the PDTB useful for sense disambiguation tasks [3]. Finally, the PDTB separately annotates the attribution of each discourse relation and of each of its two arguments. While attribution is a relation between agents and abstract objects and thus not a discourse relation, it has been annotated in the PDTB because (a) it is useful for applications such as subjectivity analysis and multi-perspective QA [5], and (b) it exhibits an interesting and complex interaction between sentence-level structure and discourse structure [1]. The first preliminary release of the PDTB was in April 2006. A significantly extended version was released as PDTB-2.0 in February 2008, through the Linguistic Data Consortium (LDC), see http://www.seas.upenn.edu/~pdtb, for the annotation manual, published papers, tutorial slides and a link to LDC.

References

1. Dinesh, N., Lee, A., Miltsakaki, E., Prasad, R., Joshi, A., Webber, B.: Attribution and the (non)-alignment of syntactic and discourse arguments of connectives. In: Proceedings of the ACL Workshop on Frontiers in Corpus Annotation II: Pie in the Sky, Ann Arbor, Michigan (2005)
2. Marcus, M.P., Santaroni, B., Marcinkiewicz, M.A.: Building a large annotated corpus of English: The Penn Treebank. Computational Linguistics 19(2), 313–330 (1993)
3. Miltsakaki, E., Dinesh, N., Prasad, R., Joshi, A., Webber, B.: Experiments on sense annotation and sense disambiguation of discourse connectives. In: Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories (TLT 2005), Barcelona, Spain (2005)
4. Palmer, M., Guildea, D., Kingsbury, P.: The proposition Bank: an annotated corpus of semantic roles. Computational Linguistics 31(1), 71–106 (2005)
5. Prasad, R., Dinesh, N., Lee, A., Joshi, A., Webber, B.: Annotating attribution in the Penn Discourse Treebank. In: Proceedings of the COLING/ACL Workshop on Sentiment and Subjectivity in Text, pp. 31–38 (2006)
6. Lee, A., Prasad, R., Joshi, A., Dinesh, N., Webber, B.: Complexity of Dependencies in Discourse: Are Dependencies in Discourse More Complex than in Syntax? In: Proceedings of the 5th International Workshop on Treebanks and Linguistic Theories, Prague, Czech Republic (December 2006)

Getting the Details Right

Lori A. Clarke

Department of Computer Science
University of Massachusetts,
Amherst, Massachuseets USA
clarke@cs.umass.edu

Keywords: Requirements engineering, Property specifications, Finite-state verification, Medical Safety.

1 Overview

Requirement engineering usually involves repeated refinements of the requirements specifications, starting with high-level systems goals and constraints, to more precise and measurable specifications of intended behavior, to detailed, focused statements that provide the basis for formal reasoning. We refer to these more detailed, mathematically rigorous specifications as *property specifications*. Although care must be taken when defining requirements at all these levels of abstraction, it is particularly difficult to accurately capture all the subtle details associated with property specifications. To help understand these decisions, the PROPEL (*PROP*erty *EL*icitation) system [1, 2] provides templates for commonly occurring property patterns [3] in which the options that need to be considered for each pattern are explicitly represented. PROPEL currently provides three views of each template and its associated options: natural language phrases to be selected, a set of hierarchical questions to be answered, or a finite-state automaton with optional labels, transitions, and accepting states to be selected. After all the options have been selected for a template, the finite-state automaton view provides a mathematically precise property specification.

We evaluated PROPEL and this approach to requirement refinement as part of the UMASS Medical Safety Project. In this project, medical professionals are working with computer scientists to define and improve life-critical medical processes. In this work, processes are first modeled in the Little-JIL process definition language[4]. Little-JIL provides a high-level, graphical representation of the process, but is designed to also facilitate modeling important process considerations such as concurrency and exception handling. Such complex models need to be carefully validated before being used for important decision-making.

Using medical guidelines and protocols as the high-level requirements and working with domain experts, in this case medical professionals, we then refined these requirements, first to more detailed natural-language statements and then, using PROPEL, to precise property specifications. We found that often just doing this mapping helped uncover errors in the process models or in the higher-level requirements statements. Frequently important details about the requirements were not captured by the natural language descriptions, and domain experts had to provide the missing information.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 10–12, 2008.

Not surprisingly, it took several iterations of improvements before the process model and the property specifications were deemed to be reasonable representations. At this point, FLAVERS [5], a finite-state verification system, was used to determine if the model of the process was indeed always consistent with each stated property specification. If an inconsistency was found, counter example traces through the model were provided that helped reveal the reason for the inconsistencies. By examining the counter examples, errors were isolated and corrected. Typically, it took many iterations before a satisfactory process model that adhered to the set of property specifications was actually obtained.

Using this approach for creating detailed process models, for mapping high-level requirements to collections of property specifications, and then using formal verification to determine if the model is consist with these specifications, we were able to find important and interesting errors. Not surprising, often errors were found in the process model or in the property specification. Having these two, relatively independent representations, allowed us to validate both. We then used these validated artifacts to help discover errors in the actual processes. When process errors were found, modified processes were proposed by the medical professionals and then verified be the computer scientists before being implemented in the medical setting. Process modifications occurred relatively frequently, for example, in response to errors being found, new technologies being introduced, or new staffing constraints. In contrast, the properties remained quite stable, with new properties sometime being added to address new concerns that arose. Thus, the collection of properties was invaluable to validating process modifications or finding process errors. More information about the medical safety project, the errors that were detected, and the technologies have been reported elsewhere [6, 7].

Acknowledgements. Leon J. Osterweil and George Avrunin are major contributors and co-principal investigators on the UMASS medical safety project. Many other individuals have also contributed to this project including Dave Brown, Lucinda Cassels, Bin Chen, Stefan Christov, Rachel Cobleigh, Heather Conboy, Elizabeth Henneman, Philip Henneman, Wilson Mertens, and Sandy Wise.

This research was partially supported by the National Science Foundation under awards CCF-0427071, CCR-0205575, and CCF-0541035, and by the U.S. Department of Defense/Army Research Office under awards DAAD19-01-1-0564 and DAAD19-03-1-0133.

References

1. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: PROPEL: An Approach Supporting Property Elucidation. In: 24th International Conference on Software Engineering, Orlando, FL, pp. 11–21 (2002)
2. Cobleigh, R.L., Avrunin, G.S., Clarke, L.A.: User Guidance for Creating Precise and Accessible Property Specifications. In: 14th International Symposium on Foundations of Software Engineering, Portland, OR, pp. 208–218 (2006)
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: 21st International Conference on Software Engineering, Los Angeles, CA, pp. 411–420 (1999)

4. Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton Jr., S.M., Wise, A.: Little-JIL/Juliette: A Process Definition Language and Interpreter. In: Proceedings of 22nd International Conference on Software Engineering, Limerick, Ireland, pp. 754–758 (2000)
5. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M., Naumovich, G.: Flow Analysis for Verifying Properties of Concurrent Software Systems. ACM Transactions on Software Engineering and Methodology 13, 359–430 (2004)
6. Chen, B., Avrunin, G.S., Henneman, E.A., Clarke, L.A., Osterweil, L.J., Henneman, P.L.: Analyzing Medical Processes. In: 30th International Conference on Software Engineering, Leipzig, Germany (to appear, 2008)
7. Clarke, L.A., Avruinin, G.S., Osterweil, L.J.: Using Software Engineering Technology to Improve the Quality of Medical Processes. In: 30th International Conference on Software Engineering, Leipzig, Germany (to appear, 2008)

Defect Detection and Prevention (DDP)

Martin S. Feather

Jet Propulsion Laboratory, California Institute of Technology
Martin.S.Feather@jpl.nasa.gov

The Defect Detection and Prevention (DDP) decision support process, developed at JPL, has over the last 8 years been applied to assist in making a variety of spacecraft decisions. It was originally conceived of as a means to help select and plan hardware assurance activities (inspections, tests, etc) [1], generally late in the development life-cycle. However, since then it has been used predominantly in early phase of system design, when information is scarce, yet many critical decisions are made. Its range of application has extended to encompass a wide variety of kinds of systems and technologies. Its predominant role has been to assist in planning the maturation of promising new technologies to help guide the next steps in their development as they emerge from the laboratory and seek to mature sufficiently to become acceptable to spacecraft missions [2]. Although this may at first glance seem far removed from terrestrial considerations, the factors that come into play in this kind of decision-making are universal - unclear and inconsistent perceptions about requirements and capabilities, uncertainty of what are the driving concerns that should be addressed and how best to address them, challenges of gathering and combining information from experts of multiple difference disciplines, and inevitably the lack of sufficient resources (money, time, CPU, power, ...) to do everything one would wish. Other significant applications of DDP have been as the risk management tool for entire spacecraft projects in their early phases of development, as an aid to planning portfolios of mission activities (e.g., [3]), and as a means to help guide R&D decisions (e.g., [4], [5]).

DDP achieves this versatility by providing an information model appropriate to these kinds of decision making challenges, supported by custom-developed software, and conducted in facilitated group sessions. Its information model [6] is somewhat akin to that of Quality Function Deployment (QFD) [7], but with a probabilistic risk basis. In the world of risk management tools, DDP fills a niche between the qualitative methods (such as SEI's Continuous Risk Management approach), and sophisticated quantitative modeling of Probabilistic Risk Assessment. DDP's custom software is used to help gather, combine, analyze and present (via several cogent visualizations) [8] the model information.

Requests for DDP software should be made through https://download.jpl.nasa.gov/

Acknowledgements

The research to develop DDP was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 13–14, 2008.

References

1. Cornford, S.L.: Managing Risk as a Resource Using the Defect Detection and Prevention Process. In: 4th International Conference on Probabilistic Safety and Management, International Association for Probabilistic Safety Assessment and Management, pp. 1609–1614 (1998)
2. Feather, M.S., Cornford, S.L., Hicks, K.A., Johnson, K.R.: Applications of too support for risk-informed requirements reasoning. Computer Systems Science and Engineering 20(1), 5–17 (2005)
3. Tralli, D.M.: Programmatic Risk Balancing. In: IEEE Aerospace Conference, pp. 2_775–2_784, Big Sky, MT (2003)
4. Shapiro, A.A., Cornford, S.L., Feather, M.S., Price, G., Gawdiak, Y.O., Ricks, W.R.: Planning a Large-Scale progression of R&D - a Pilot study in the Aerospace Domain. In: IEEE Aerospace Conference, Big Sky, MT (2006)
5. Feather, M.S., Uckun, S., Hicks, K.A.: Technology Maturation of Integrated System Health Management. In: Space Technology and Applications International Forum, pp. 827–828, Albuquerque, NM,(2008)
6. Feather, M.S., Cornford, S.L.: Quantitative Risk-based Requirements Reasoning. Requirements Engineering Journal 8(4), 248–265 (2003)
7. Akao, Y.: Quality Function Deployment. Productivity Press, Cambridge (1990)
8. Feather, M.S., Cornford, S.L., Kiper, J.D., Menzies, T.: Experiences using Visualization Techniques to Present Requirements, Risks to Them, and Options for Risk Mitigation. In: International Workshop on Requirements Engineering Visualization, Minneapolis / St. Paul, MN (2006)

Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs*

Luqi[1] and Fabrice Kordon[2]

[1] Naval Postgraduate School, Monterey, California, USA
luqi@nps.edu
[2] LIP6, Université Pierre et Marie Curie, Paris, France
Fabrice.Kordon@lip6.fr

Abstract. The lions's share of the software faults can be traced to requirements and specification errors, so improvements in requirements engineering can have a large impact on the effectiveness of the overall system development process. A weak link in the chain is the transition from the vague and informal needs of system stakeholders to the formal models that support theoretical analysis and software tools.

This paper explains the context for the 2007 Monterey workshop that was dedicated to this problem. It provides the case study that participants were asked to use to illustrate their new methods, and summarizes the discussion and conclusions of the workshop.

1 Introduction

The Monterey Workshop Series. The objective of the entire series of 15 Monterey workshops since 1992 has been to "increase the practical impact of formal methods in computer-aided software development". The workshop seeks to improve software practice via application of engineering theory and to encourage development of engineering theory that is well suited for this purpose.

Previous workshops have reduced the gap between theoretical and practical aspects of software/system engineering and have produced a consensus that the pain of system development could be reduced via computer aid for or automation of software engineering subtasks based on particular theories and various kinds of formal models. A common theme has been to hide theoretical results and complex mathematical ideas inside tools with simple interfaces so that practitioners could use them without the need to fully understand the theory behind them.

However, there has also been general agreement that the pain of development cannot be eliminated completely. No matter what you do, somewhere in the process some people have to think clearly and in detail to reach agreement on what problems should be solved by the software to be developed. Consequently,

* This work was supported in part by ARO grant 45614CI.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 15–24, 2008.

requirements, response to changes, and human aspects of programming have been identified as potentially fruitful areas for improvement.

Goal of the 2007 Monterey Workshop. The 2007 workshop is focused on requirements, particularly the process of transforming vague and uncoordinated needs of individual stakeholders into consistent and well defined requirements that are suitable for supporting automated and computer aided methods for engineering subtasks in the development process to follow.

Errors or failures of software-based systems are due to a variety of causes, e.g. misunderstanding of the real world, erroneous conceptualization, or problems in representing concepts via the specification or modeling notations. Precise specification is a key success factor as are communication and the deliberation about whether the specification is right and whether it has been properly implemented. Not all stakeholders are familiar with the formal models and notations employed. Some important requirements might be difficult to quantify and/or express using formal languages, such as the desire that a system should be user-friendly or easily maintainable. Better technologies for requirements analysis should thus be considered.

The majority of requirements are given in natural language, either written or orally expressed. Other requirements might also be visually expressed in terms of figures, diagrams, images or even gestures. Artificial-intelligence approaches might be used to develop prototypes, which can then be re-engineered using more conventional requirements technologies and safety assurance techniques. For example, we might employ large amounts of semantic and statistical data, knowledge bases and theorem provers to infer as much contextual information as possible from the (vague) textual or visual requirements. Then, some extra questions could be raised to system/software stakeholders to point out some fuzzy (or missing) requirements to be refined or some conflicting requirements to be reconciled.

Accurate automatic analysis of natural language expressions has not yet been fully achieved, and interdisciplinary methodologies and tools are needed to successfully go from natural language to accurate formal specifications. Conformance of a system implementation to its requirements requires dynamic and efficient communication and iteration among system stakeholders. It is in supporting this process, and not in supplanting it, that innovative approaches to requirements analysis need to find their proper role.

We want to gain a better understanding of how to deal with natural language as the vehicle from which we derive system/software requirements, how to use intelligent agents as entities to facilitate semi-automatic requirements-documentation analysis, and how to build automatic systems to aid in requirements/specifications elicitation. The overall aim is to exchange ideas for continued research in the intersection of these two areas and to reduce the gap between theory and practice.

A good case study for these issues is to consider how to extract a conceptual model of the goals and requirements of the software needs discussed in a *blog*. As blogs are unstructured natural language, they represent one of the most difficult

challenges for natural language processing. All workshop participants have been requested to use the case study given in Section 4 of this paper to illustrate their work.

2 Focus Areas

The three days of the workshop were organized around the following focus areas:

- *Recent Advances in Requirements Engineering.* Feather compares various approached to specification and requirements analysis. Aschauer et al explore success factors for agile requirements analysis. Dinesh et al present an approach for regulatory conformance checking.
- *Human and Linguistic aspects of Requirements Engineering.* Kof addresses identification of goals in stakeholder dialogs. Sawyer examines profiling and tracing of stakeholder needs. Goedecke explores the relation between viewpoints and documents.
- *Computer Aid for Requirements Engineering.* Fe describes how model-driven prototyping can help elicit requirements. Popescu et al explain how automatically created OO models can be used to improve the quality of requirements specifications.

The panels and discussion sections interleaved with the presentations were focused on integrating, balancing, and assessing the various viewpoints presented at the workshop to reach a consensus on where we are, how emerging capabilities for natural language processing and computer aided requirements elicitation methods can contribute, and to identify the best paths forward.

3 Workshop Case Study

All workshop participants were asked to use the case study given below to illustrate their work. The participants in the case study discussion are:

- a representative from the Transportation Security Administration (TSA);
- a representative from the Federal Aviation Administration (FAA);
- a representative from Airport screening and security (ASS).

Their discussion on the blog is reproduced hereafter.

The objective of the case study exercise is to answer the following questions based on the discussion above:

1. What was the topic(s) of the discussion? Have you noticed any contradictions?
2. What are the realistic requirements that FAA suggests for increasing airport security?
3. What long-terms goals should be set by TSA?
4. What concrete changes should be enforced by Airport screening and security?

Who	Post
FAA	We have to ban on airplane passengers taking liquids on board in order to increase security following the recent foiled United Kingdom terrorist plot. We are also working on technologies to screen for chemicals in liquids, backscatter, you know...
ASS	Technologies that could help might work well in a lab, but when you use it dozens of times daily screening everything from squeeze cheese to Chanel No. 5 you get False Alarms... so it is not quite ready for deployment!
FAA	Come on! Generating false positives helped us stay alive; maybe that wasn't a lion that your ancestor saw, but it was better to be safe than sorry. Anyway, I want you to be more alert - airport screeners routinely miss guns and knives packed in carry-on luggage.
ASS	Well... It's not easy to move 2 million passengers through U.S. airports daily. And people can't remain alert to rare events, so they slip by
TSA	We can deal with it. What if you guys take frequent breaks? And also we are going to artificially impose the image of a weapon onto a normal bag in the screening system as a test. Then screeners learn it can happen and must expect it. Eventual solution will be a combination of machine and human intelligence.
AAS	Sounds good though we do take breaks and are getting inspected. We do not get annual 'surprise' tests - sometimes we get them everyday; and if a screener misses too may of these consistently, they are sent to training.
TSA	We have yet to take a significant pro-active step in preventing another attack - everything to this point has been reactive. Somebody hijacks a plane with box cutters? - Ban box cutters. Somebody hides explosives in their shoes? - X-ray shoes, and then ban matches. We are well behind!
FAA	What do you suggest? Yes, there is an uncertainty. On each dollar that a potential attacker spends on his plot we had to spend $ 1000 to protect. There are no easy solutions. We are trying to federalize checkpoints and to bring in more manpower and technology.
TSA	We need to think ahead. For instance, nobody needs a metal object to bring down an airliner, not even explosives. Practically everything inside the aircraft is easily flammable, except for the people, so all anyone needs is oxidizer. Do any of the automated screening devices detect oxidizers? Are the human screeners trained to recognize them?
FAA	Good point. Airlines need to take the lead on aviation security. The corporate response was to market cheap tickets and pass security off on the federal government. Have a trained group of security officers on every flight. Retrain flight attendants as security officers. Forget about passing around the soda and peanuts - that should be secondary.
AAS	Sir, a lot of airlines are not doing well and are on the Government assistance. Prices go up, baggage get mishandled. There are constant changes in screening rules – liquids/no liquids/3-1-1 rule. Anything radical will not only cost a lot of money but also deter people. I mean an economic threat is also a threat.
TSA	I think that enforcing consistency in our regulations and especially in their application will be a good thing to do. Another thing is that even if an airline goes bankrupt there are still advantages: bankruptcy makes it easier to rearrange company assets and to renegotiate vendor and supplier contracts.
FAA	Ok, we had very productive discussion. Now back to work. I want you to come up with some concrete measures based on what we have been talking about. You should finally generate some ROI for that money we have been spending. And do not forget, the examples listed above are not all-inclusive.

4 Synthesis of Workshop Discussions

The main points that emerged from workshop discussions are the following:

Getting Unambiguous Specifications. A major focus of the workshop was the transition from informal ideas to formal models, and the associated problems of resolving ambiguities. This transition is an inescapable part of software development because stakeholder needs, which are inherently informal, must be transformed into software, which is inherently formal. Synthesizing an unambiguous model of stakeholder needs is a major part of requirements engineering; another major part is ensuring that this model is accurate.

It was recognized that natural language is an inescapable part of the process, because most of the communication with stakeholders is carried out in natural language. There has been a great deal of past work on requirements engineering that has advocated the use of formal models to represent requirements by creating notations and tools to make such models accessible to a wider audience.

However, this does not avoid the need for resolving ambiguities. Despite all the past advances, it is still the case that most stake holders are unable to write formal models. These models are constructed by specially trained experts, who construct models on behalf of the stakeholders based on their natural language statements. These experts are at risk of subconscious disambiguation; they construct formal models based on their understanding of stakeholders' statements even though their interpretations could be different than what the stakeholders meant. The model builder may not even be aware that there is an interpretation of the natural language other than the first one that comes to mind and was understood [2].

A similar problem applies to approaches that use unambiguous subsets of natural language. These subsets are made unambiguous by rules and restrictions that admit only one interpretation. Subconscious disambiguation in this case can have the reader relying on an understanding and interpretation of the constrained natural language that differs from the one chosen by the rules and used by all of the software tools based on those rules.

The workshop recognized that it is not possible to write unambiguous natural language, and that it is useful to reduce the amount of ambiguity where possible. Some details can be found in [2,12]. Other suggested approaches included using fault tolerance strategies to engineer systems that can tolerate ambiguity and to use examples to clarify which interpretation is intended. Examples could be supplied by stakeholders or generated from formal models and checked by stakeholders.

Ambiguity Management. Sometimes ambiguity may be used to deliberately express disagreements among different stakeholders. In such cases, questions must be raised to get the correct interpretation. The clients may not know the answer, so negotiations or additional information may be needed to get a reliable resolution.

The workshop concluded that natural language processing and aid for resolving ambiguities would be useful, but should be used to support current processes rather than replacing them. Reasons for this include (1) that there is a lot more

to requirements engineering than just translating stakeholder statements into formal models and (2) that current accuracies of automated natural language processing are less than 100%.

Requirement Engineering. Requirements engineering tasks include finding implied but unstated requirements, detecting conflicts between needs of different stakeholders, and resolving such conflicts. Communication gets increasingly difficult as systems scale up. Stakeholder are typically comprised of diverse groups, each of which has its own specialized domain knowledge, jargon, and unique tacit understanding of the problem. Bridging the gaps becomes key to success as complexity increases because each group typically has only a partial understanding of the issues, constraints, possible solutions and cost implications [14,9].

Accuracy of the requirements engineering process is crucial. Requirements engineering is a critical part of the system development process because requirement errors cost roughly 100 times less to correct during requirement engineering than after system delivery [4]. This imposes extreme constraints on the accuracy of natural language processing and that we might use to derive system requirements. However, natural language processing accuracies are currently in the 90%-92% range, at best [3]. Therefore natural language processing must be augmented with other methods for removing residual errors, and accuracy must be greatly improved if it is to be seriously used for Requirements engineering.

Towards Computed Aided Requirement Engineering. To be useful, tools must find all possible instances of a problem, and it is acceptable to have some false positives in the warnings and error reports, otherwise engineers will not be able to afford to rely on the results of the tool. Since the delivered system is unlikely to be any better than the requirements, accuracy of the requirements has great importance. Existing manual processes for deriving requirements form informal stakeholder statements therefore incorporate a variety of checking procedures that include reviews, storyboarding, simulation and prototype demonstration, dependency tracing, consistency checking, and many others. Natural language processing of requirements engineering must be integrated with such checking procedures to achieve needed accuracy.

Accuracy of natural language processing can be improved by specializing the problem to the context of requirements engineering and using the extra information provided by that context. For details, see [3]. Developing tools and methods for augmenting and supporting current requirements engineering processes with tools that incorporate natural language processing appears to be a promising realistic goal, if it is coupled with integration into error checking and correction processes already used in requirements engineering. Total automation of requirements engineering does not appear to be feasible in the foreseeable future, in view of the gap between promises and actual results of AI research of the past several decades. Natural language processing is highly context dependent, both on the subject domain and the questions being asked. In the context of requirements engineering there are an effectively unlimited number of domains.

Given the huge size of requirements documents for real projects, even imperfect heuristic methods that can improve confidence that something important

was not overlooked. Some problems that have been explored in detail include identifying goals in stakeholder dialogs [10], support for dealing with requirements changes [1,8], using shallow natural language processing techniques to aid in synthesizing requirements [13,11], and requirements validation [6].

To automate the process it is useful to rely on representations and methods for detecting conflicts or unfounded constraints in the requirements (this can be seen as a second focus that emerged from the workshop). Methods based on logic were proposed for checking conformance of requirements to regulations [5]. A notation and method for analyzing conflicts, ambiguities, and imprecision in requirements based on viewpoints of different stakeholders were explored [7]. These directions are promising because they attempt automation of the processes that cannot be effectively done manually when requirements are very complex. The reason for this is that the analyses are non-local in nature and can depend on interactions between wildly separated parts of the requirements. People are effective at analyzing small bits of text in depth, but not at finding widely separated connections in very long documents. Progress in these directions should be possible in the not too distant future.

Synthesis of Discussions during the Workshop. Traditionally, Monterey Workshops leave a large space to discussion between participants. In 2007, the workshop discussions resulted in the following conclusions:

- End-to-end integration is necessary for all of the component technologies to realize their possible contributions to real software development processes. To achieve this, a necessary step is to clarify the interface between natural language processing and requirements engineering. [3] contains a step toward this goal.
- Domain specific approaches can help natural language processing perform better. Context information such as the goals of the speaker, the speaker's area of expertise, and expected output of the process can narrow the search space for disambiguation and condition the probabilities governing the most likely interpretations.
- Natural language processing for requirements engineering needs to handle domain specific jargon and acronyms.
- Generating accurate natural language from formal models is easier and more accurate than the reverse process, and can be very helpful for finding errors. However problems with subconscious disambiguation [2] are still present.
- Generating summary descriptions is useful for finding defects, especially errors of omission.
- To have practical impact, automatic methods contributing to the transformation from natural language to formal requirement models have to be faster and more accurate than current manual methods.

5 Conclusion

Overarching goals of the rest of the series of Monterey Workshops are to create a shared community-wide articulation of the system/software engineering

enablement challenge, reach consensus on the set of intellectual problems to be solved, and create a common vision of how the solutions to these problems will fit together in a comprehensive engineering environment.

The Monterey Workshop has been able to bring the brightest minds in Software Engineering together with the purpose of increasing the practical impact of formal methods for software development so that these potential benefits can be realized in actual practice. In the workshop, attendees and organizers work to clarify what good formal methods are, what are their feasible capabilities, and what are their limits. Overall, the workshop strives to reduce the gap between theory and practice. This has been a slow and difficult process because theoreticians and practitioners do not normally talk to each other, and did not at the beginning of the workshops. This gap has been gradually reduced. In particular, researchers have focused on problems that are relevant to the practitioners, and have helped demonstrate how recent theory can be applied to solve current problems in software development practice.

Here are the workshops:

N	Year	Theme	Location	Chairs
0	1992	Concurrent and Real-Time Systems	Monterey	Luqi, Gunter
1	1993	Software Slicing, Merging and Integration	Monterey	Berzins
2	1994	Software Evolution	Monterey	Luqi, Brockett
3	1995	Specification-Based Software Architecture	Monterey	Luqi
4	1996	Computer-Aided Prototyping	Monterey	Luqi
5	1997	Requirements Targeting Software and Systems Engineering	Bernried	Broy, Luqi
6	1998	Engineering Automation for Computer Based-Systems	Carmel	Luqi, Broy
7	2000	Modeling Software System Structures in a Fastly Moving Scenario	Santa Margherita Ligure	Astesiano, Broy, Luqi
8	2001	Engineering Automation for Software Intensive System Integration	Monterey	Luqi, Broy
9	2002	Radical Innovations of Software and Systems Engineering in the Future	Venice	Wirsing
10	2003	Embedded Systems	Chicago	Shatz
11	2004	Compatibility and Integration of Software Engineering Tools	Vienna	Manna, Henzinger
12	2005	Networked Systems	Irvine	Sztipanovits, Kordon
13	2006	Composition of Embedded Systems	Paris	Kordon, Sokolsky
14	2007	Innovations for Requirements Analysis	Monterey	Luqi, Kordon
15	2008	Foundations in Computer Software	Budapest	Dobrowiecki, Sztipanovits

The 2007 workshop highlighted some differences between generic natural language processing and natural language processing in the context of requirements engineering. Researchers from both communities learned about relevant recent advances from each of the communities and became more aware of the open problems in the gaps between the two fields. It is becoming clear that many software problems originate in the gap between the fuzzy needs of the human stakeholders and the formal models used in software design. This are is gaining increasing attention from the scientific community.

The Monterey workshops have helped focus the attention of the community on many productive directions. For example, since the 1995 workshop identified specification-based architectures as a key means to achieve system flexibility and reuse, there has been a great deal of activity in these areas. A great deal of research has produced architecture description languages and associated analysis methods, there have been commercial advances on "plug and play" hardware and software, adoption of service-based architectures in electronic commerce, and a move toward open architectures in government and defense systems. Currently the practical impact of software architecture is no longer in doubt

We look forward to comparable advances in computer aided requirements analysis in the decade to come.

Acknowledgments

The Monterey Workshops were initiated under the support of Dr. Hislop at ARO and many others at NSF, ONR, AFOSR, and DARPA. We would like to thank DARPA and NSF for their financial support of the 2007 workshop, NRC for support of two talented postdoctoral fellows Dr. Rodriguez and Dr. Ivanchenko who contributed to the proposal, workshop case study and material for the web page, the program committee chairs Barbara Paech and Craig Martell and committee members for their efforts on reviewing papers and putting together the workshop program, and the local chair Craig Martell for handling endless practical details. All of the workshop participants contributed to the ideas summarized in this paper.

References

1. Aschauer, T., Dauenhauer, G., Derler, P., Pree, W., Steindl, C.: Could an Agile Requirements Analysis be Automated? In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 25–42. Springer, Heidelberg (2008)
2. Berry, D.: Ambiguity in Natural Language Requirements Documents: Extended Abstract. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 1–7. Springer, Heidelberg (2008)
3. Berzins, V., Martell, C., Luqi, Adams, P.: Innovations in Natural Language Document Processing for Requirements Engineering. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 125–146. Springer, Heidelberg (2008)

4. Boehm, B.: Software Engineering Economics, Upper Saddle River, NJ, USA. Prentice Hall PTR, Englewood Cliffs (1981)
5. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: Logic-based Regulatory Conformance Checking. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 147–160. Springer, Heidelberg (2008)
6. Fu, J., Bastani, F., Yen, I.: Model-Driven Prototyping Based Requirements Elicitation. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 43–61. Springer, Heidelberg (2008)
7. Goedicke, M., Herrmann, T.: A Case for ViewPoints and Documents. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 62–84. Springer, Heidelberg (2008)
8. Hoss, A., Carver, D.: Towards Combining Ontologies and Model Weaving for the Evolution of Requirements Models. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 85–102. Springer, Heidelberg (2008)
9. Kelly, D.: A software chasm: Software engineering and scientific computing. IEEE Software 24(6), 119–120 (November-December 2007)
10. Kof, L.: On the Identification of Goals in Stakeholders Dialogs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 161–181. Springer, Heidelberg (2008)
11. Lange, D.: Text Classification and Machine Learning Support for Requirements Analysis Using Blogs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 182–195. Springer, Heidelberg (2008)
12. Popescu, D., Rugaber, S., Medvidovic, N., Berry, D.: Reducing Ambiguities in Requirements Specifications via Automatically Created Object-Oriented Models. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 103–124. Springer, Heidelberg (2008)
13. Sawyer, P., Gacitua, R., Stone, A.: Profiling and Tracing Stakeholder Needs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 196–213. Springer, Heidelberg (2008)
14. Stone, A., Sawyer, P.: Identifying tacit knowledge-based requirements. Software, IEE Proceedings 153(6), 211–218 (2006)

Could an Agile Requirements Analysis Be Automated?—Lessons Learned from the Successful Overhauling of an Industrial Automation System

Thomas Aschauer[1], Gerd Dauenhauer[1], Patricia Derler[1], Wolfgang Pree[1], and Christoph Steindl[2]

[1] C. Doppler Laboratory Embedded Software Systems, Univ. Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at
www.cs.uni-salzburg.at
[2] Catalysts GmbH,
Prager Str. 6, 4040 Linz, Austria
steindl@catalysts.cc
www.catalysts.cc

Abstract. This paper sketches a recent successful requirements analysis of a complex industrial automation system that mainly required a talented expert, with a beginner's mind, who has been willing to dig into the domain details together with a committed customer and a motivated team. With these key factors and the application of an appropriate combination of well-established and some newer methods and tools, we were able to efficiently elicit, refine, and validate requirements. From this specific context, we try to derive implications for innovative requirements analysis. We argue that in projects that go beyond simple, well defined, and well understood applications, automated requirements analysis is unlikely to lead to a successful specification of a system.

Keywords: requirements analysis, agile development, use cases, automation systems.

1 Introduction

Our research group cooperates with an industry partner that is a dominant player in the area of a specific kind of test automation systems that are used, for example, in the automotive industry. These automation systems need to be tailored to customer demands. For the software solution our research partner currently offers, this tailoring process is not supported well. Thus we were asked to develop a system that radically improves the customization and operation process of such systems.

The inherent complexity of the domain and the vagueness of the original requirements document we were provided with were major challenges for the requirements engineering process. We chose an agile, prototype driven approach with short feedback cycles. In conjunction with an unbiased team, which consisted of a top software scientist and four motivated software engineers, we were able to successfully elicit

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 25–42, 2008.

and analyze the requirements and to come up with an innovative solution. We are confident that it is able to solve the current system's shortcomings and to sustainably improve our partner's competitive advantage.

The main contribution of this paper is twofold. First it presents a successful requirements analysis process for an industrial innovation project. Second it argues that in this particular case automatic requirements analysis methods were not applicable. As such it serves as a reality check for natural language processing methods in requirements analysis.

The remainder of this section briefly introduces the target domain and gives a short overview of the customization process in the current system. Section 2 describes the project context, the initial requirements and the team structure. The actual requirements analysis process and the development of the prototypes are described in section 3. Section 4 presents a case study on how the team's understanding of one particular requirement grew over time. Section 5 concludes that automated methods for analyzing requirements are not likely to have succeeded for this particular project setting.

1.1 The Domain of Test Automation Systems

This section briefly introduces the application domain of test automation systems. Typically, a test system is used to acquire measurement data from operations of a device under test. The resulting data is required, for example, for research and development or for quality assurance. Various variants of test systems are used in industry.
An automated test system typically comprises the following parts: a device under test (or device for short), automatic test equipment (or equipment for short) that simulates force, a mechanical link between the device under test and the automatic test equipment for force transmission, measurement equipment ranging from simple temperature sensors to sophisticated measurement devices, actuators such a throttles, I/O systems as interface to an automation system that controls the test procedure, and conditioning devices controlling supply for air, oil, water, etc.

This system structure is depicted in Figure 1: Boxes represent hardware components, the block arrow represents the mechanical link, solid lines represent electrical connections between components, and the dotted line represents media supply for air, oil, water, etc.
A typical test procedure for an automated test system has a duration ranging from minutes to hours or even days. During that time, up to millions of measurement values are recorded, which can amount to several gigabytes of measurement data. The automation system's software is responsible for controlling the device and the equipment in real-time, for executing test procedures, and for collecting and recording the measurement data. Evaluation of the measurement data is performed by separate post-processing tools.

An automated test system can be operated as stand alone system or in a larger context, the so-called *test-factory*. A test-factory is a set of separate automated test systems, with possibly different capabilities, that share common infrastructure such as measurement data archiving. The overall goal of a test-factory is to optimize the throughput by scheduling test orders accordingly.

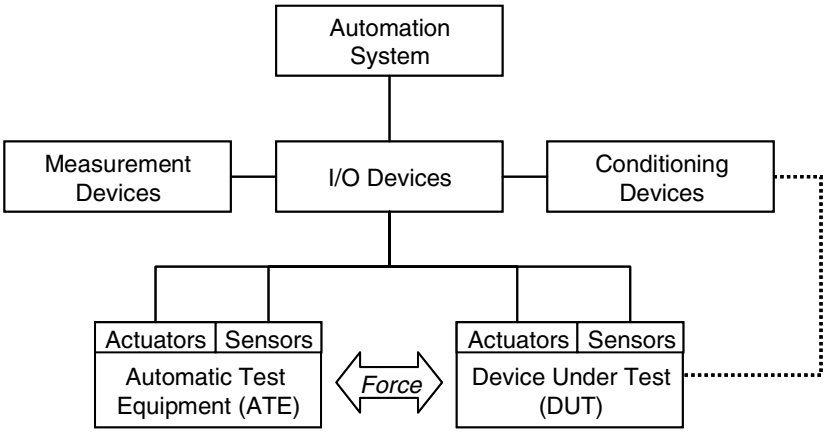


Fig. 1. Typical automated test system structure

1.2 Problems of the Current System

Our work is based on the cooperation with a dominant player in the field of test automation systems. Our research partner offers a software solution that can be applied to all kinds of test systems through customization to specific requirements. The software evolved over the last two decades during which its code base, mainly written in C++ and C, grew to about 1.5 million lines of code.

The automation system software consists of a number of specialized subsystems such as a hard real-time kernel executing the device under test and the automatic test equipment, and a subsystem for measurement data acquisition. The interactions between these subsystems are established through variables in globally shared memory. Thus the subsystems have to be configured consistently. Due to the evolution of these subsystems, they all use their own configuration file formats for customization, ranging from plain text files to binary files.

Configuration parameters describe properties of hardware and software. Properties of hardware are, for example, the device's weight. Software properties described in configuration files include the characteristic values for the equipment's controller, the safety limits for the force affecting the device, device driver settings for certain measurement devices, or user defined formulas and scripts to be executed by the automation system.

A major hurdle for users that have to customize the system is that the current tool directly reflects the automation system's software structure and low-level design decisions in the user interface. Moreover, all parameters are presented in a tabular form. The main view basically is a plain table, where each table row represents the associated subsystem of the automation system and is used to navigate to a detailed view for the subsystem. In other words, the configuration interface is presented as a set of interrelated spread-sheet pages.

As an example, the automation system has a separate subsystem for proportional-integral-derivative controllers (PID-controllers). When a user needs to modify the parameters for a specific controller of the equipment, the user would need to navigate

to that subsystem, browse through all PID-controllers in the system to find the right one, and then modify the corresponding parameters. This customization system forces engineers that are used working with the parts of the test system such as the device, the equipment, etc. to understand the internals of the software to be able to set parameters correctly.

In the customization process, engineers have to modify parameters in configuration files that are loaded by the automation system at system startup. In a typical setup there are about 10,000 configuration parameters with about 120,000 values to be set correctly. The creation of a consistent set of configuration files is a time consuming and error prone procedure which may take weeks or even months for a complex test system setup.

The current systems offers very limited support for the initial creation of these configuration files. Only the skeletons representing the structure of the test system can be created by a tool, the details have to be filled in manually. The time it takes to get the system running depends on the experience of the engineers in charge. They often use a form of ad-hoc reuse by using configuration files from previous similar projects as templates.

Once a set of configuration files is created, it has to be kept synchronized with the automation system software. When software is updated during the lifetime of a test system, its configuration files have to be modified accordingly. Again, tool support for the update process only barely exists. Updating configuration files has to be performed manually. As a consequence, customers update their automation system software to major revisions only when absolutely necessary, because the process of modifying configurations files is time consuming and error prone. Our research partner therefore has to invest a lot of development effort in the maintenance of many different software revisions in parallel.

2 Project Setup

Our research partner identified the necessity for improving the current customization process. Due to the fact that multiple tries to overcome the current system's shortcomings by the company itself have failed for different reasons, a research project in cooperation with our research institute was initiated. This section briefly describes the initial requirements and the project team structure.

2.1 Initial Requirements

The main mission goal is to develop a system that radically improves the usability of customization and operation of test automation systems. In the beginning, we were provided with a rather haphazard requirements document consisting of about 20 items. The list includes specific functional requirements as well as some general non-functional requirements such as maintainability and security issues. The most important requirements are summarized as follows:

a) Introduce components, i.e. named sets of parameters, that naturally map to domain entities such as device under test, automatic test equipment, PID-controller, etc. and that describe both their visualization and their parameters.

b) On-site extensibility, meaning that new functionality can be added to the system without the need to recompile any source code.
c) Provide different parameter views including guidance through customization tasks. The basic idea is that of separation of concerns [1], meaning the splitting of various aspects of a system into independent parts that can be dealt with independently. As an example, there should be a separate view for hardware-related parameters, such as the weight of the device, and another separate view for software-related parameters, such as the characteristic values of the PID-controller for the equipment.
d) Support users in mastering the complexity of test system setups, e.g. by hiding those parameters that are not needed for a specific task. For example, a service task concerned with finding the defect part between the automation system and a certain device does not require knowledge about the simulation model for the device.
e) Provide a context-aware work environment that supports the user in specifying only valid parameter values for a component by evaluating the component's context.
f) Do as many checks as possible as early as possible. Inconsistent measurement and consumption frequencies, for example, can be detected by comparing parameters of connected components when the connection is established, whereas the existence of a piece of hardware in a test system can only be checked when the system is connected to the actual test system. Furthermore, ensure that these checks can be integrated in different products to avoid duplicated implementations.
g) Replace configuration files by parameter sets, i.e. by components.
h) Provide an *operations view* describing a component's visualization and the parameters that are modifiable during the operation of a test system.
i) Compatibility to existing systems, which means supporting a wide range of tools and technologies.
j) Maintainability of components, which means support for versioning, change tracking, comparison, and interoperability between different systems and also between different software versions.

In addition, we also received a huge amount of user documentation, system requirements specifications for the existing system, and UML diagrams. The latter consisted of use case diagrams and use cases describing functionality at the level of specific technical details. These documents evolved along with the existing system during the last two decades. They were, however, hardly up to date.

2.2 Project Team and Location

Due to the importance of the project for our customer, the company is fully committed to it and we report to one of its executives. The project is set up around one of the company's most respected experts, who is also fully committed to the project goals. The project leader has more than one decade of experience in the domain and long time experience in successfully managing projects of comparable complexity, including innovative software development projects. Later on we realized that this particular project leader is like an *advocate* for the project, in the sense as Wile described knowledgeable advocates as crucial for the success of their domain specific language experiments [2].

Company representatives with in-depth domain knowledge as well as product managers were available in the requirements analysis phase. Additionally, we had access to employees that formerly were associated with competitors and also to developers of the current system.

The initial software development team consisted of one top software scientist as team leader, and four young software engineers with little or no project experience. The team leader has extensive software development experience, social skills training, and an additional solid background in automation systems, but had no prior knowledge of the particular automated test system.

During the course of the project, the team grew in size by two software developers and two domain engineers with background in automation systems and the target domain.

The project team intentionally resides at a different geographical location than our partner, which emphasizes the company's intention to strike a new path in the development of their software solutions.

3 Prototyping-Based, Agile Requirement Analysis

Considering the ambiguity in the provided requirements document (cf. section 2.1), the fact that the team had no prior knowledge of the domain and the overall vision of the project seemed somewhat unsettled, the right methods for the requirements engineering task had to be chosen.

We decided to stick to an agile approach for the following reasons: First, the short feedback cycles would allow us to quickly respond to changes in the requirements and to misunderstandings of the original requirements document. As stated by Hirsch, "the desired properties of the end product can not be known until at least part of the solution is built" [3]. Second, the project leader's intuition gave him the feeling that for an innovation project, a front-up design method would not lead to success. Third, the team leader had previous, successful experience in applying agile methods.

This section chronologically describes the project phases, beginning from the initial phases of paper prototyping to the current phase. In addition, the planned project phases are sketched to depict the different approaches necessary in the different phases.

3.1 Phase I: Paper Prototyping (September 2006 – February 2007)

Since the project team was completely new to the domain, we started the project with a 5 day workshop. We approached the problem from the user's point of view, first developing a global context with the user roles and their targets, then detailing the tasks of the users – completely unrestricted by the existing system. During the first workshops we looked also at systems from three competitors.

We wrote down the discussions in detailed workshop protocols, and we visualized the scenarios on slides, some with animations so that they resembled how a system could actually work. Some of these presentations were prepared from one workshop day to the other, so that we could start with a recapitulation of the previous day, and extend on it.

Figure 2 shows a conceptual drawing for how a *perfect parameterization system* would show the physical parts of a sample test system. Basically, boxes represent components which are pieces of hardware or software that are connected to other components. Concepts such as grouping, abstraction by hierarchically structuring components, and different ways of connecting components were applied and refined using these drawings.

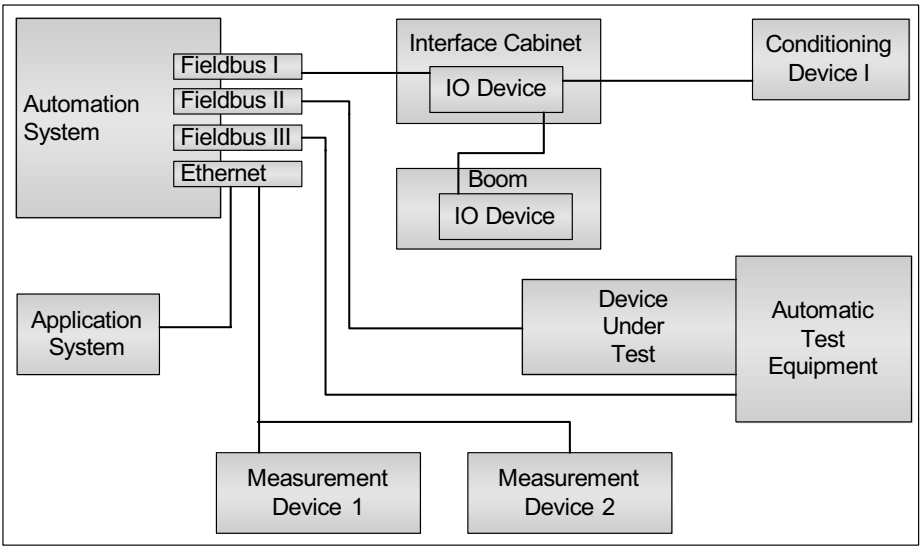


Fig. 2. Conceptual drawing showing the physical parts of a test system

We held several workshops in a row, with approximately a month in between, which gave us time to understand the existing system. Thus we were able to conceptualize the requirements step by step. This process was documented by writing a glossary comprising about 90 terms as well as by analyzing and writing some 130 use cases. At that time the project focus to develop a system that would eventually replace the parameterization tool of the current software solution was clearly communicated to the team.

Due to the radical departure from the original system, we knew that we had to present the ideas in an easy-to-grasp way; hence we decided to develop a mock-up prototype that would allow showing how various users, in their various roles, would use the system. For that we specified scenarios such that we could exactly define the click paths through the prototype for every user. Numerous concepts and ideas were proposed and discussed in simple drawings on paper, in slide presentations and figures drawn with common drawing tools. These drawings exemplified how the software could appear for each scenario.

Similar to the drawing in Figure 2, the mock-up prototype provided a view representing the physical components of a test system. Figure 3 shows the corresponding screen.

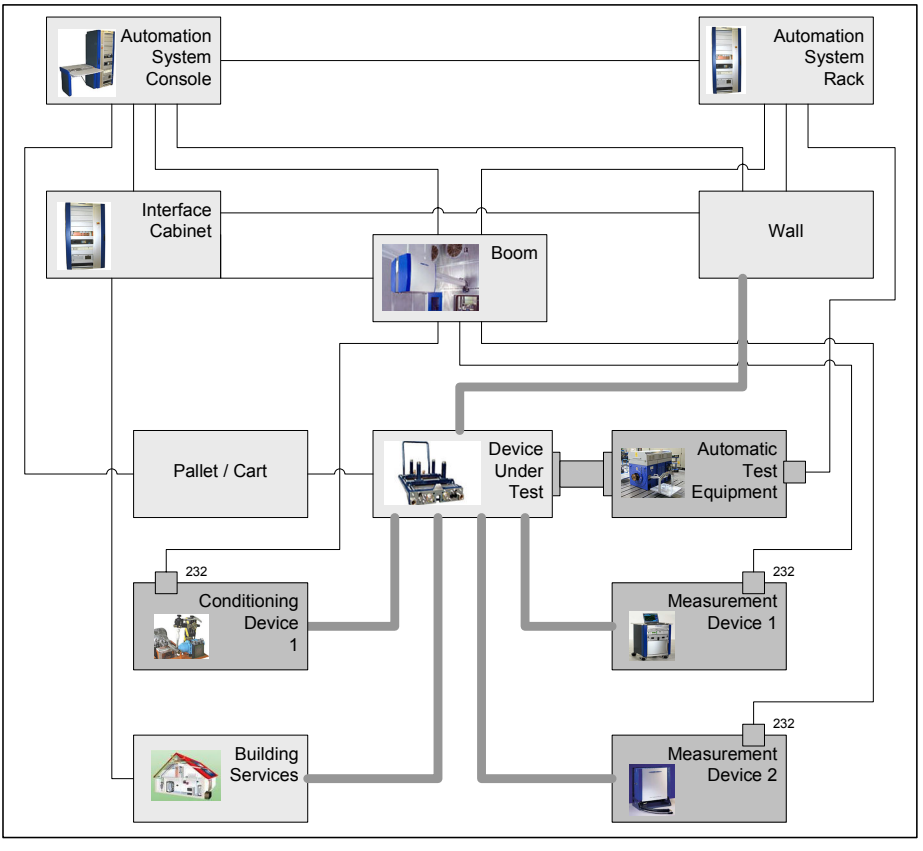


Fig. 3. Physical view of test system in mock-up prototype

The development of the mock-up prototype served as a vehicle to document and elicit customer requirements and to gain domain understanding, such that we were able to derive *16 core features* which represent the essence of the system's functionality. Each of those core features was meant to be orthogonal to the others; together they yield a powerful system to solve the underlying problem. The most essential core features are summarized as follows:

- Definition of the concept of domain components. Domain components are defined as sets of parameters that are grouped into self-contained units.
- Support for configuring domain components, that is, setting their parameters or connecting them to other domain components.
- Support for combining domain components in groups or hierarchies. By allowing domain components to be built hierarchically, that is, by layering component systems as described by Szyperski [4], complexity can be managed. Collapse and expand mechanisms help hiding unnecessary details.
- Support for management of domain components in libraries, which might be predefined by our customer or be user-specific.

- Support for comparing domain components and helping users to discover similarities and differences.
- Support for versioning of domain components.
- A mechanism for undo and redo management and for creating and recording macros at the user interface layer.
- Provide a mechanism for guiding users performing predefined tasks or for resolving problems. Furthermore, allow users to provide their own experience in form of guidance for other users.
- Management of different views of domain components and corresponding access rights.

For the actual demonstrations we decided to have one person clicking through the prototype, while another person would do the talking, watching the audience, being able to answer questions and to improvise, and to lead the questions back to the click paths that we had prepared beforehand. Eventually the team presented the prototype, which was enthusiastically received by the customer's top management in January 2007.

We captured the demonstrations as video sequences with a couple of introductory slides and an animated demo part. Those videos had several advantages:

- They allowed everyone to get some insights about what the project was about. With the prototype, getting these insights would not have been possible, since the prototype required in-depth knowledge of the prototype's implementation and the predefined click-paths. Only a small percentage of click-paths a typical user would do were implemented.
- They allowed us to preserve the presentations, so that we ourselves could have a look at them later on, e.g. when new people were to join the team.
- They allowed us to explain the system without having to conserve the executable or without installing the executables; hence it was easier to share.
- They forced us to get the user stories right and consistent. We had to remove all vagueness from the ideas.

However, the videos also had some drawbacks, e.g.:

- Even though we asked for feedback on the last slide shown in the videos, we did not get valuable feedback from the customer.
- We were told not to distribute information about the project in this format any more, since some of the customer's employees spent much time on them. Moreover, the videos caused disturbance in the current system's development team, since this was one of the first sources of information about our project that was made available to them. As Ramos et al. point out [5], the introduction of radically new software and the vision of a future work reality associated with it is never free of emotions.

The system to be built was split in two layers, a generic framework layer and an application layer built on top of the framework. The framework, developed by our team, provides a platform for building custom applications that can be developed by application engineers with profound domain knowledge, but without programming skills. While the framework incorporates generic domain knowledge, such as sensors,

measurement values, etc., the application incorporates specific domain knowledge such as the types of equipment applicable for a specific device and test system.

During the first project phase, the software development team gradually gained understanding for the customer's demand of a component-oriented framework for test automation systems. The different interpretations of the term *component framework* were one of the major causes of confusion between the customer representatives and the development team: The development team had components in mind as defined in software science, that is, components describing a unit of composition with contractually specified interfaces and explicit context dependencies only. Such a software component can be deployed independently and is subject to composition by third parties [6]. Furthermore, the development team had a *technical* component framework in mind, in the sense of e.g. the OSGi platform [7], while the customer representatives thought of a framework for modeling and assembling components specific to the *domain*, such as DUT and controllers. Domain components are somehow related to software science components, that is, they are units of composition, they explicitly describe dependencies, and they are units of deployment. For the further success of the project it was crucial to overcome this misunderstanding.

3.2 Phase II: Working Prototype Based on a Domain-Specific Language (March 2007 – September 2007)

Early in the project, we considered a domain-specific language (DSL) as crucial basis for describing test system components. Motivated by our project leader, the DSL was designed as a generic one for describing automation systems. This means that the DSL offers, for example, a means for describing data types of values, the construct of a generic component for grouping values and also for grouping associations between these components. The language for describing test systems is an extension of the generic one. We refer to the generic language as CDL (Component Description Language) and to the test-system-specific language as tsCDL.

For the refinement of CDL and tsCDL we applied an informal approach. Starting in April, we evaluated tools and methodologies that would best fit this task. We first used the Unifid Modeling Language (UML) syntax [8] to sketch and iteratively refine the key test automation system concepts such as electrical plugs, wires, mechanical connections, sensors, and actuators. It turned out that a simple UML diagram drawing tool with limited UML capabilities was better suited for this purpose compared to a full-featured UML editor.

In addition to the UML-based CDL and tsCDL refinement, we came up with a textual representation of CDL and tsCDL. We used both, the UML-based and text-based versions of tsCDL to describe test system components such as devices, data acquisition units and also complete test systems. The definition and refinement of the textual syntax of the language and the UML-based version were intertwined.

The following code fragment in Figure 4 sketches the description of a test automation system. The sample test automation system consists of three components, the automatic test equipment *Ate*, an I/O hardware called *IO*, and the automation system called *AuSys*. The component *Ate* is of type *AteType127*, the component *IO* is of type *IODevice*, and the *AuSys* component is of type *AutomationSystemPC*. The test automation system description also states which locations are relevant in this case: a location called *Floor3* and a *ControlRoom*. The second line in the *RELATIONS* section

harnesses the location description by stating that the *Ate* component is located on *Floor3*. The *Ate* component is hierarchically composed of other components, such as the *BendingBeam*. The *BendingBeam*'s plug *Plug2* is connected to plug *X17* of the *IO* component, which is specified in the first line of the *RELATIONS* section.

```
COMPONENT TestSystem
  COMPONENTS
    Ate   : AteType127
    IO    : IODevice
    AuSys : AutomationSystemPC
  END
  LOCATIONS
    Floor3
    ControlRoom
  END
  RELATIONS
    Ate.BendingBeam.Plug2 CONNECTS IO.X17
    Ate AT Floor3
  ...
  END
END
```

Fig. 4. Sample test automation system described textually in the tsCDL

Splitting the domain-description language in a generic one (CDL) and a test-system-specific one (tsCDL) is an example of an architectural aspect that could not be derived as requirement from the information we received from the customer. Nevertheless, this extra effort of coming up with both description languages turned out to be crucial for other system parts that rely on them. For example, we only needed to implement an interactive visual editor for CDL, not the much richer tsCDL which is constantly changed and extended. This is also true for the persistence layer for storing and retrieving domain components. A key objective for the project was the compatibility between components stored in different software versions. Another key objective was that software upgrades must not result in costly database schema migrations. As such, the persistence format has to be stable and should not change often during the further evolution of the software. Our experiments corroborated that we only need to define a database schema for the CDL, not for the the tsCDL.

The design of CDL and tsCDL as well as the development of the interactive visual editor, the persistence mechanism for CDL components and their versioning were inherently difficult to plan. Initial attempts to establish a development process, such as Scrum [9], were abandoned since the estimated efforts turned out to be unrealistic and the process became an overhead without any benefit.

The major milestone of Phase II was the development of a prototypical first version of the software system incorporating the most essential features of the 16 core features identified in the previous phase.

3.3 Decompression Phase III (September 2007 – February 2008)

After presenting the results of the second phase to the executives of our customer, the team entered a short *decompression phase* [10], which means the team performed a

retrospective to improve subsequent phases. The retrospective revealed the following key success factor: Defining concrete scenarios helped to focus the development of the prototype. Furthermore, the scenarios had to be defined in detail, so that all vagueness had to be eliminated and the concepts had to be sound and understandable from the user's point of view.

The following factor has been identified as restraining to the project success: Due to the inherent complexity of the application domain and its peculiarities, the team depends on a variety of information sources. We did not consistently question the quality and the completeness of the information we got.

3.4 From Research Prototype to Product (Starting February 2008)

The software system has reached a level of maturity so that domain engineers can use it to model real-world test system components. The foundation, based on CDL and tsCDL, is stable and additional features are continuously integrated enabling domain engineers to model the various aspects of test system components as they are found in test system products. The goal of a milestone in August 2008 is to demonstrate that the software system is capable of modeling, configuring and operating a real test system. The long-term plan is that the newly developed software system will be shipped as product to customers in 2010.

The additionally required features are derived from the feedback of the customer's domain engineers. We have established a bi-weekly release cycle now. The release planning incorporates the requests of domain engineers in the form of user stories, describing the expected behavior in terms of the user interface. These stories are usually a few lines of text and the effort to implement them ranges from one person-day to about one person-week.

These requests of domain engineers represents one source for our release planning. The other sources are the initial requirements as presented in section 2.1 and the refactorings suggested by the development team itself. We treat the identified refactorings of the existing code as user stories.

4 Case Study: Understanding the Versioning Requirement

We exemplify how our understanding of the requirements evolved over time by picking one of the 20 initial requirements which we consider as a representative example. The initial list of requirements contained the following text, which was summarized as the last bullet-hole item in section 2.1:

«12. Maintainability: it must be possible to version parameters and parameter sets; Change logging, i.e. who changed what and when; Export/import among test fields, also language independent; Search/find; Difference of parameters and parameter sets; Undo; Interoperability of previous software versions with data in newer version and vice versa»

We were quite aware that this key requirement was intentionally phrased quite vaguely, for example, the "and vice versa" phrase. Therefore, we tried to de-scope some of the requirements for the initial project Phase I:

«12. Maintainability:…»

« ➔ we will propose a concept for the operation until the end of 2006 ➔ the domain model and meta-model will allow for versions ➔ since the implementation would require major changes to the existing system, we won't perform them until the end of 2006»

So for a while we turned back to the more challenging requirements and developed concepts, prototypes etc. as explained in section 0 above. One of the 16 core features identified in Phase I was the following:

«13. Updating of components with a transport mechanism for changes: It is possible to deliver application components in a new version and deploy them. Macros can be used as transport mechanism for changes. There will be a language for describing:

- *How old data shall be migrated*
- *Whether the new version must be deployed or can (optionally) be deployed*
- *Whether user interaction / acknowledgement is necessary or whether the update shall be performed silently*

Updating application components is not about updating software, but about updating descriptions of components (together with the underlying data).
Updating of system functions would require a software update which we do not address in the first release. »

Even then we thought that updating would "simply" mean that we need some flexible mechanism to get data of older versions migrated to the schema of the new version. During a workshop with another project team of the customer in February 2007 they presented the following requirements or conclusions:

«'Import mechanism is enabled to do needed data migration'
'Migration Framework is a MUST!'
'Be migration aware'
'Versioning - Implemented within our storage services'»

We took those statements again as hints that we will only need to import old data in new versions of the software. We acknowledged the need for a migration framework and versioning but deferred the topic nevertheless, believing that we will also be able to implement it in the persistence layer with some import / export filters.

In March 2007, we augmented the requirements with use cases. We identified the following use cases:

- UC Versioning 1 – Select a version
- UC Versioning 2 – Browse version log
- UC Updating 1 – Define data migration
- UC Updating 2 – Perform data migration

However, we sketched only the main scenario for the versioning use cases, and left the updating use cases undefined. Back then, leaving everything open was the best we could do, since any detail would have been speculation.

At the end of March 2007, we had an architecture workshop with the customer where the development manager of the existing system mentioned that the new system will have to sustain the concurrent operation of automated test systems in

multiple versions. We considered it sufficient if our software were able to cope with new and old versions of the data.

In Phase III at the beginning of December 2007, we discussed the topic in detail with our advocate. The discussion was summarized with the following versioning requirements:

- In a test field, multiple test systems will be in use with various versions of the new software system.
- The new software system must be able to process old and new components.
- It shall be possible to migrate components in old versions to newer versions, such that test systems with new versions of the software system can use the old components.
- If possible, it shall be possible to use the new components even on old test systems, possibly just in a read-only mode.

We discussed the implications of those requirements on the various layers of the system and how changes in each layer would affect upper layers. Analogies from books on database refactoring were drawn, e.g. the idea of *scaffolding code* in the database, which transparently enables one version of the software to work with several versions of the data model. As described by Ambler [11], this can be achieved by introducing views and triggers in the database layer. Furthermore, we drew analogies from related scientific papers, dealing for example with the problem how to co-evolve a model when the corresponding meta-model evolves, as described by Wachsmuth [12].

We identified two principal approaches to deal with version changes: to track all transformations, i.e. a priori, versus to derive modifications from delta detection, i.e. a posteriori. The latter approach was ruled out by construction of examples that showed its deficiencies.

However, we still did not really accept the need for bidirectional compatibility, i.e. that new versions of the software can work with old and new data, *and* that old versions of the software can work with old and new data.

At the end of December 2007, our advocate kept pushing towards bidirectional compatibility. In January 2008, we finally accepted the challenge of bidirectional compatibility and gave it a try, i.e. we did a so-called *spike* in eXtreme Programming terminology [13]:

- We refined the implementation from the user's point of view.
- We implemented the solution, which required several extensions of the persistence layer and upper software layers.
- We demonstrated to the customer how data model transformations can be defined and how a new version of the system can then automatically transform data from the old format into the new format. Furthermore we demonstrated how an old version of the system can automatically transform data from the new format into the old format, given that a bidirectional mapping between old and new meta-model exists.

Summarizing the case study,

- we considered versioning and updating as a black box for a long time
- we ignored repeated hints by the customer, or we did not understand them
- we placated our advocate for a long time

Finally, we worked through the problem within three calendar weeks, and we came up with an appropriate solution for a problem that the customer has had for decades but that resisted several previous attempts to be solved. In the end, all the extensions did not have a negative impact on the existing architecture. We think that it would have been impossible to derive the requirements of this aspect from documents we received from the customer.

5 Limits of Automated Requirements Analysis

This real-world project corroborates, in our point of view, that requirements analysis can barely be automated if the stakeholders do not have a clear understanding about a software system. In this case it was the feeling of the customer that the current system could be improved significantly. The customer and its team were somehow trapped in the existing system. Knowing too many details and worrying about significant changes made it virtually impossible to come up with appropriate requirements for an overhauled system. The required creativity cannot be expected from tools. To quote Deming [14]: "As a good rule, profound knowledge comes from the outside, and by invitation. A system cannot understand itself."

The beginner's mind [15] allowed the team to profoundly analyze the features of the current system as well as its strengths and weaknesses. This is a quality already pointed out by Berry [16]. He describes a computer-system-savvy person without any knowledge of the domain as the person asking ignorant, not stupid, questions to expose tacit assumptions made by domain-expert stakeholders assuming incorrectly that all other domain-expert stakeholders understand. By making those assumptions explicit, conflicts in the understanding are discovered at an early stage in the software development.

5.1 Could Automated Support for Requirements Analysis Have Been Beneficial?

Reflecting on potential use of automated approaches to requirements analysis, we identified two areas where application of such approaches could have been beneficial in our case: term extraction and preventing ambiguity. For a recent overview of state-of-the-art approaches to requirements engineering in general see Cheng and Atlee [17].

Since the project team was completely new to the problem domain, automated support for extracting the domain specific terms could have been applied. As Kof points out [18], a thorough understanding of domain concepts is essential and a precise definition for each concept is required. An approach to semi-automatically extract ontology from requirements documents is proposed. Such an approach or similar ones are, however, likely to have failed in our case for the following reasons:

- As pointed out in section 0, the initial requirements document we received consisted of only 20 items that just briefly described the system to be built. Domain specific terms occurred in the document, but due to the document's limited size the usage of a semi-automated or an automated tool for term extraction is not likely to

have produced substantially better results than performing this task manually. In the paper prototyping Phase I, as described in section 0, we created a glossary for the essential domain concepts.

- Along with the initial requirements document, we also received a huge amount of documents related to the current system, such as requirements specifications and user documentation. When the team sifted through these documents, it soon became obvious that most of the information was not relevant in the early project phases. It still is in question whether the majority of the material will be of any use at all since it deals with specific technical details and peculiarities. Applying a system for term or ontology extraction on these documents would have been a challenge on its own due to the size of the documents. It is not clear how such a system could have helped in the decision which concepts to ignore, and which not to ignore, in particular if one keeps in mind that the number of essential concepts is very small compared to the overall number of concepts. For example, an automatic analysis of the documentation would likely have identified the *normname* as one of the most relevant concepts in the domain just by the number of references. Normnames are, however, just a necessity of the current system's implementation: A normname is the unique name of a variable in the global shared-memory which is used to connect the different functions and subsystems, as mentioned in section 1.2. These global variable names are one major shortcoming of the current system that we could get rid of in the new system.
- Important concepts of the new system were completely missing in the current system; they were only described by general terms in the initial list of requirements. The versioning and compatibility requirement as described in section 0 is an example. Term extraction techniques would not have been helpful for understanding these requirements either.

Another area where application of natural language processing tools would have been conceivable is in preventing ambiguity in the documents we generated. For example, Fantechi et al. [19] present an approach that analyzes use cases written in natural language and provide certain metrics for measuring aspects related to ambiguity. These might have improved the consistency of the use case documents we created in Phase I as described in section 0. Because these use cases were not the final specification of the system to be built, but just a vehicle to further understand the requirements and to structure the problem domain, fewer ambiguities in these documents would have just been a minor benefit.

For a project of this type, i.e. searching for a creative and revolutionary solution, the successful application of automated techniques is unlikely. Typically, the customer would not present a fully specified requirements document and expect a development team to return a working program after a certain amount of time, within a predefined budget. Instead, in an iterative process with frequent workshops, demonstrations and presentations, the customer can see how the project is evolving and how the team performs. Moreover, the team can gradually gain better understanding of the customer's *real* demands.

6 Conclusion

We assume that none of the tools that automate requirements analysis could lead to a successful completion of the requirements analysis for our project, because the available inputs from the customer are too haphazard and the terminology is not precise enough—a situation that is typical for many real-world software projects. In such a context the sketched agile requirements analysis with short feedback cycles together with the communication vehicle of a throw-away prototype has turned out to be an appropriate requirements analysis method. We are convinced that no automated system would have been able to support, let alone accomplish something close to such a successful requirements analysis and specification based on the available natural language descriptions of the requirements, the current system and its envisioned features.

References

1. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
2. Wile, D.: Lessons Learned from Real DSL Experiments. In: Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS 2003). IEEE Computer Society Press, Los Alamitos (2003)
3. Hirsch, M.: Moving from a Plan Driven Culture to Agile Development. In: ICSE 2005 the 27th International Conference on Software Engineering, St. Louis (invited talk) (2005)
4. Szyperski, C.: Component software and the way ahead. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 1–20. Cambridge University Press, New York (2000)
5. Ramos, I., Berry, D.M., Carvalho, J.Á.: The Role of Emotion, Values, and Beliefs in the Construction of Innovative Work Realities. In: Bustard, D.W., Liu, W., Sterritt, R. (eds.) Soft-Ware 2002. LNCS, vol. 2311, pp. 300–314. Springer, Heidelberg (2002)
6. Szyperski, C., Pfister, C.: Workshop on Component-Oriented Programming, Summary. In: Muehlhaeuser, M. (ed.) Object-Oriented Programming – ECOOP 1996 Workshop Reader. Dpunkt Verlag, Heidelberg (1997)
7. OSGi Alliance, Open Services Gateway initiative, `http://www.osgi.org/`
8. UML, Unified Modelling Language, `http://www.uml.org/`
9. Rising, L., Janoff, N.S.: The Scrum Software Development Process for Small Teams. IEEE Software 17(4), 26–32 (2000)
10. Gamma, E.: Agile, open source, distributed, and on-time: inside the eclipse development process. In: ICSE 2005 the 27th International Conference on Software Engineering, St. Louis (keynote talk) (2005)
11. Ambler, S.W., Sadalage, P.J.: Refactoring Databases: Evolutionary Database Design. Addison Wesley Signature Series. Addison-Wesley, Reading (2006)
12. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)
13. Beck, K.: Test-driven development: By example. Addison-Wesley, Reading (2002)
14. Deming, W.E.: The New Economics for Industry, Government, Education, 2nd edn. MIT Press, Cambridge (2000)
15. Suzuki, S.: Zen Mind, Beginner's Mind, Weatherhill (1973)
16. Berry, D.M.: The Importance of Ignorance in Requirements Engineering. Journal of Systems and Software (1995)

17. Cheng, B.H., Atlee, J.M.: Research Directions in Requirements Engineering. In: 2007 Future of Software Engineering, International Conference on Software Engineering. IEEE Computer Society, Washington (2007)
18. Kof, L.: Natural Language Processing: Mature Enough for Requirements Documents Analysis? In: Natural Language Processing and Information Systems, 10th International Conference on Applications of Natural Language to Information Systems, Alicante, Spain (2005)
19. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of Linguistic Techniques for Use Case Analysis. In: Proceedings of the 10th Anniversary IEEE Joint international Conference on Requirements Engineering. IEEE Computer Society, Washington (2002)

Model-Driven Prototyping Based Requirements Elicitation

Jicheng Fu, Farokh B. Bastani, and I-Ling Yen

Department of Computer Science
The University of Texas at Dallas
P.O. Box 830688, EC 31
Richardson, TX 75083-0688 USA
{jxf024000,bastani,ilyen}@utdallas.edu

Abstract. This paper presents a requirements elicitation approach that is based on model-driven prototyping. Model-driven development fits naturally in evolutionary prototyping because modeling and design are not treated merely as documents but as key parts of the development process. A novel rapid program synthesis approach is applied to speed up the prototype development. MDA, AI planning, and component-based software development techniques are seamlessly integrated together in the approach to achieve rapid prototyping. More importantly, the rapid program synthesis approach can ensure the correctness of the generated code, which is another favorable factor in enabling the development of a production quality prototype in a timely manner.

Keywords: Requirements Elicitation, Prototyping, Component-Based Software Development, Code Patterns, Model-Driven Development.

1 Introduction

The primary measure of success in a software system is the degree to which it meets the purpose for which it is intended [23]. Therefore, requirements engineering (RE) activities are vital in ensuring successful projects. In [11], RE is defined as a branch of software engineering concerned with real world goals, functions, and constraints on software systems. RE facilitates the transformation from informal requirements to formal specifications, which serve as the basis for subsequent development. However, the secret behind the scene for the transformations is difficult to formulate because of the problems of uncertainty, ambiguity, inconsistency, etc., inherent in the process.

Prototyping is a popular requirements elicitation technique because it enables users to develop a concrete sense about software systems that have not yet been implemented. By visualizing the software systems to be built, users can identify the true requirements that may otherwise be impossible. Prototyping was once regarded as the solution to RE. It has many advantages [4][15], including:

- Reduced time and cost. Problems can be detected in the early stages. Therefore, the overall cost is greatly reduced.
- Concretely present the system operations and facilitate design decisions.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 43–61, 2008.

- Stakeholders from all parties can actively get involved in the development process.

Prototyping is especially useful when there is a great deal of uncertainty or when early feedback from stakeholders is needed [4]. There are two types of prototyping, i.e., rapid prototyping (throwaway) [15][17] and evolutionary prototyping [16]. Rapid prototyping focuses on the demonstration of functionality and obtaining early feedback on requirements that are poorly understood. The essential idea is to develop a prototype system containing any unclear requirements as quickly as possible. There may be bugs in the prototypes and the overall quality of the implementation may not be good. But these are tolerable in rapid prototyping. Hence, this type of prototype is referred to as quick-and-dirty and will be discarded after any unclear requirements have been clarified [4].

Evolutionary prototyping, on the other hand, is developed as a portion of the actual system. It focuses on the requirements that have already been well understood. New requirements and features are incrementally added as the development proceeds in an iterative manner. The prototype is developed to be of production quality and will not be thrown away [4].

However, prototyping is not thriving as expected due to the following reasons [21]:

- Management can get confused by the prototype and the production quality version to be built. They may expect that the final deliverable will come quickly based on enhancement and refinement of the prototype;
- Poor quality codes from the prototype may remain in the final system due to the tendency of reusing previously written code fragments.
- Lack of mechanisms for requirements traceability.
- Prototypes may not be developed quickly due to the system complexity and technical limitations.

The last two reasons are the most important factors that hinder the use of prototyping. Technical people may tend to make the prototype overcomplicated, resulting in some artifacts that are not linked back to the original requirements. Another tendency is the omission of some functionality because stakeholders may be absorbed in some aspects, e.g., user interfaces, etc., while neglecting other aspects. The most valuable property of prototyping is the fact that it can be done quickly. The lack of systematic rapid development approaches makes it hard to fulfill this property. Without this property, prototyping cannot have significant impact on industry.

It is, therefore, desirable to have a prototyping approach that can leverage the advantages of rapid and evolutionary prototyping. Specifically, prototypes should be developed quickly and still maintain satisfactory quality. To achieve this goal, we need to meet the following objectives:

(1) Make software design a part of the development process.
(2) Achieve a certain level of automation to speed up the development.
(3) Make requirements traceable.

Based on these objectives, we propose a model-driven development (MDD) based prototyping approach. The use of model-driven approaches is especially amenable to requirements engineering because it meets the aforementioned objectives. First, in MDD, system design has become a part of the development process. UML 2.0,

developed to support MDD, has changed the view that UML diagrams only serve as temporary documents and will be put aside at later points during the development process. Combined with OCL (Object Constraint Language), UML is able to specify models in a formal way. OCL is a declarative and precise specification language, which has no side-effects and does not change the state of the system [30]. It enables errors to be found early in the life-cycle, when fixing a fault is relatively cheap.

Second, MDD can automate the generation of infrastructural code (i.e., code frames) through transformations between platform independent models (PIMs) and platform specific models (PSMs) and between PSM and code. Specifically, PIM and PSM are designed to raise the level of abstraction. PIMs are models with high level abstractions that are independent of the implementation technology [9]. PSMs are bound to specific platforms and implementation technologies. PSMs are generated from PIMs through transformation and the code is in turn generated from PSMs. These processes can be automated to increase productivity. Thus, developers can concentrate on the development of PIMs, which are at a higher level of abstraction than the actual codes. This is another favorable factor for speeding up the development process.

Third, traceability is a desired feature for the design of model-driven development tools. The existing MDD tools support a certain level of traceability. Hence, it makes the development process amenable to requirements changes. It is always easier to indicate what part of a PIM is affected by the changed requirements than to determine code segments that must be modified. When parts of the code are traced back to elements in the PIM, it would be much easier to make an impact analysis of the requested changes [9].

Although transformations that map models to the next level are typically used in MDD [24][28], there are some doubts about the practicality of generating complete systems solely via transformations. Transformations are good at generating infrastructure codes instead of business codes. In order to further speed up the development of prototypes, a novel program synthesis technique is applied to the proposed approach. The program synthesis technique combines AI planning and component-based synthesis techniques to achieve automated generation of business/logic code. Specifically, we design and implement a fast planning graph based iterative planner, called FIP [6]. It can deal with nondeterministic actions (actions that can generate multiple possible effects) and generate parameterized procedure-like generic reusable plans, which are called procedural plans. FIP can help automate the selection and organization of underlying components to achieve the given goal. The underlying component-based synthesis technique serves as the basis for the final code generation. It is based on a component-based software development (CBSD) technique, code pattern [13][14], which is concerned with reusing existing software components to build larger applications at a lower cost and risk and in less time. The AI planning and component-based program synthesis technique can be seamlessly integrated with MDA to achieve even more rapid program synthesis. In this hybrid system, the development of PIM still relies on human intervention. However, PIM is independent of any implementation details and has a higher abstraction level than code. Hence, the designers can put more efforts on the business-logic related aspects of the system. Then, the static aspect of the system will be generated through MDD's transformation technique and the dynamic aspect will be generated through the AI planning and component-based program synthesis technique.

The rest of this paper is organized as follows: Section 2 overviews the techniques involved in the proposed model-driven development based prototyping approach. Section 3 presents a novel rapid program synthesis approach, in which MDA, AI planning, and component-based program synthesis techniques are seamlessly integrated together. Section 4 discusses requirements elicitation through the proposed prototyping approach based on the advanced rapid program synthesis approach. Section 5 concludes the paper and identifies some future research directions.

2 Overview

As rapid prototyping focuses on unclear requirements and evolutionary prototyping focuses on well understood requirements, neither of them alone is sufficient to represent a complete system. The proposed approach intends to combine the advantages of both methods and develop a prototype in a timely manner and of production quality. In this sense, the proposed approach is a rapid evolutionary prototyping approach.

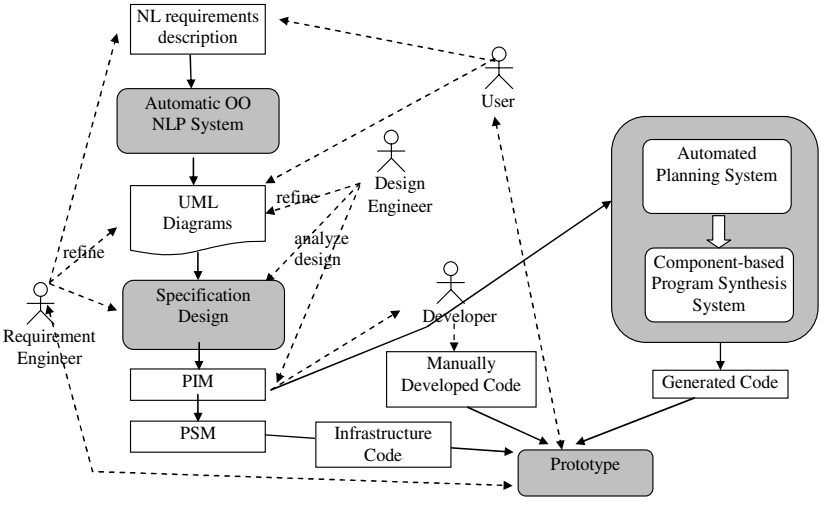


Fig. 1. People and techniques involved in the proposed approach

Fig. 1 gives an overview of the people and techniques involved in the proposed rapid evolutionary prototyping approach. During the requirements specification process, use cases are the first tangible things that stakeholders interact with. They document initial requirements and provide scenarios illustrating interactions with end users or other systems to achieve specific business goals. Use cases and other UML elements can be automatically generated by tools [7][25] that employ natural language processing (NLP) techniques to capture essential and relevant software requirements from natural language descriptions. This can help automate Object-Oriented Analysis (OOA) though the tools are not mature and only aid the requirements acquisition and analysis process. Human involvement is mandatory, especially when contradictions exist in the requirement specification.

Although MDA can generate the infrastructure code through transformations, it is not good at generating code for the dynamic aspects of the system. In order to speed up the development process as well as improve the quality of the implemented prototype, a novel rapid program synthesis approach is used. The techniques involved in the approach are MDA, AI planning, and component-based code synthesis, which are organized in a hierarchy and seamlessly integrated together. The top level is the PIM of MDA. PIM is specified using UML with OCL. It presents planning problems to the underlying automated planning system (APS), which is located in the middle of the hierarchy. Based on the planning problem, the AI planner in the APS generates a procedural plan, in which its underlying components are chosen and organized to achieve the given goal. The generated plan is then fed to the component-based synthesis system that is located at the lowest level. The final code is then generated by the code synthesis system. The developers only need to focus on the incomplete parts where the planner cannot find a suitable solution. This can alleviate the developers' burden and increase the development speed and reliability of the system. Section 3 discusses the details of the novel rapid program synthesis approach.

After the system is complete, users can visually operate it and formulate new requirements to cope with any problems. These will be fed back to the requirements engineers and the development cycle is repeated.

3 Rapid Program Synthesis

In our proposed evolutionary prototyping approach, rapid program synthesis technique plays a critical role. It ensures that the prototype is built in a timely manner and with production quality.

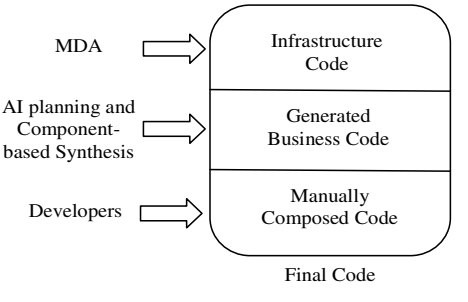


Fig. 2. Ways to obtain the final code

Fig. 2 shows how the final codes are obtained. The infrastructure codes (static aspects of the system) are generated by MDA through transformation. The AI planning and component-based synthesis method can generate business codes, which constitute the behavioral aspects of the system. The parts that cannot be generated automatically have to be implemented manually.

In Section 3.1, we introduce how MDA helps generate the infrastructural code through transformation. In Section 3.2, we introduce how the dynamic aspects of the

system are generated by the AI planning and Component-based synthesis approach. Then, in Section 3.3, we discuss how to integrate the AI planning and component-based synthesis approach with MDA so that both the static and some parts of the dynamic aspects of the system can be automatically generated.

3.1 MDA

Model-driven architecture (MDA) has attracted considerable research interests and is predicted to be the next generation software development method. MDA transforms models written in one language into models in another language. The direction of transformation is usually from high level models to low level models. Fig. 3 illustrates the relationships between PIM, PSM, and codes. PIM is designed independently of any implementation details. It comes at a higher level of abstraction. PIM is then transformed into PSM, which is a domain specific model that relies on specific domains and technology. In the final step, PSM is transformed into code. The mainstream MDA tools, e.g., [8], support these transformations.

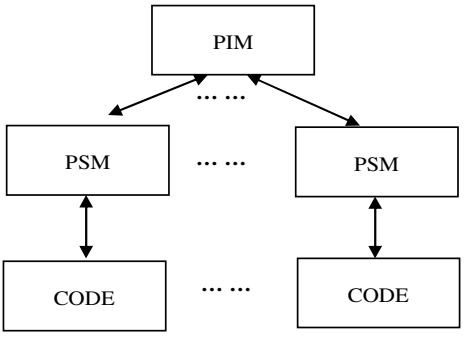


Fig. 3. Relationships between PIM, PSM, and code [9]

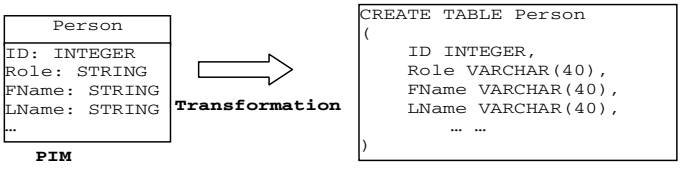


Fig. 4. Example of PIM to relational transformation

Transformation introduces automations in generating models and codes and, thus, the productivity is increased greatly. For example, assume that a "surprise" test management system is proposed to be developed for the case study in [19]. The system keeps track of the surprise test records of screeners. For a surprise test, the inspector purposely introduces fake weapons into a normal bag as a test. If the screener misses too many tests, he/she will be sent for training. This surprise test management system can be used to illustrate the transformation-based prototyping approach. The PIM to

relational transformation can be fully automated by generating the corresponding SQL clauses. Suppose that there is a database used to store users (screeners, inspectors, etc.), test to be conducted, test history, etc. For example, the PIM "Person" is defined as shown in Fig. 4. The attribute "Role" is used to distinguish inspectors and screeners.

It is very natural to do the transformation from PIM to its relational counterpart automatically through transformation. However, transformation is only good at generating code related to the static aspects of the system, i.e., infrastructure code. For example, Fig. 5 shows the transformations between PIM and PSM and between PSM and code. J2EE technology is used in this example to illustrate the idea. The PIM model "Person" is transformed into three PSM models tailored to fit within J2EE specifications. The PSM models (EJBs) are in turn transformed into code. We call the code as the infrastructure code because it only contains static code frames and/or getter/setter methods. The business code is absent from the transformation.

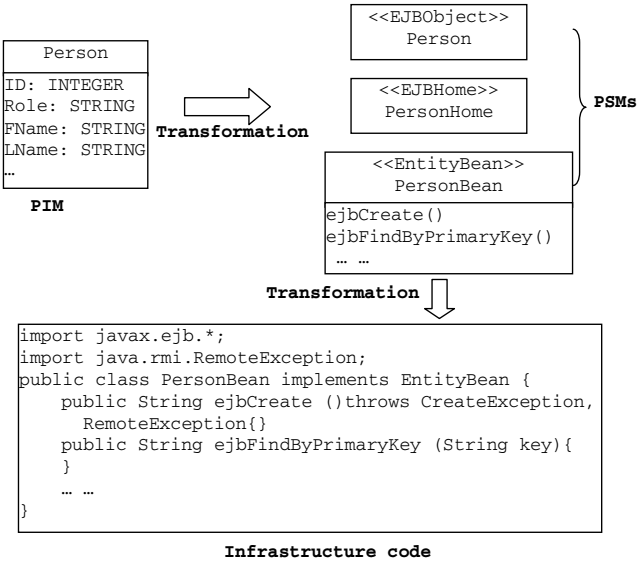


Fig. 5. Example of PIM to PSM and PSM to code transformations

To overcome the limitation of the transformation method, the concept of "heterogeneous models" [27] is introduced to empower MDA to generate business code. In this type of model, PIM and PSM are still specified with the original modeling language. Code segments written in low level languages are embedded in the appropriate parts of the high level components. The major advantages of this model are that existing codes can be reused and business code can be generated. However, this model has many disadvantages,

- The heterogeneous models mix high level models and low level code segments together and make the design difficult to understand.
- The heterogeneous models exacerbate maintenance difficulty because the changes in the high level models may lead to changes in the embedded code

segments. If the high level models are designed to be transformed to different platforms, code segments achieving the same functionality but supporting different platforms need to be added.

- The heterogeneous models neutralize MDA's benefits of portability and documentation.

In this sense, it is desirable to have a program synthesis technique that is not tightly coupled with the high level models and would not affect the benefits of MDA. Our AI planning and component-based synthesis method discussed in Section 3.2 can achieve this goal.

3.2 AI Planning and Component-Based Synthesis

Raising the level of abstraction and increasing the level of reuse have been proven to be the right way to develop software systems [22]. Our AI planning and component-based code synthesis approach closely follows this principle by integrating AI planning techniques with Component-Based Software Development (CBSD) methods. Specifically, AI planning is a problem solving technique that works on high level abstractions of actions. The problem solving process is declarative, i.e., users only need to focus on specifying the initial and goal conditions and the AI planner helps generate a plan leading the system from the initial state to the goal state.

Another reason that makes AI planning appealing is that it can overcome some limitations of the deductive code synthesis method [20], which was once regarded as the answer to code synthesis. Similar to AI planning, deductive code synthesis also enables users to work on high level specifications. Code can be generated as a by-product of the proof by the theorem prover. The first limitation is that the deductive code synthesis process may not terminate. Actually, this is a problem inherent in any deductive methods [29]. When the theorem prover runs longer than expected, it is not possible to infer whether no solution exists or whether the prover needs more time to finish the proof. The second limitation is that it is difficult for the deductive code synthesis methods to generate loop constructs. Even a short iterative program has been proven to be difficult to reason about [12]. The FIP planner is not subject to these limitations. FIP enhances classical Graphplan [1], which is guaranteed to terminate regardless of whether a plan exists or not. Also, FIP is designed to support the generation of loop and conditional constructs in its procedural plans. All of these make FIP a full-fledged technique for automated code synthesis.

To increase the level of reuse, CBSD techniques can be used to achieve the goal. CBSD is designed to use existing software components as building blocks to construct larger applications. This approach can help lower the overall development cost and reduce the development time. However, software developers face a steep learning curve to grasp under what conditions the components can be used, the ways the components can be composed together, and all the constraints on the usages of the components. The code pattern technique [13][14] is designed to overcome the problems facing CBSD and is applied to our automated code synthesis approach.

In Section 3.2.1, we briefly introduce the background knowledge of AI planning and the FIP planner. In Section 3.2.2, we introduce the CBSD approach using code patterns. In Section 3.2.3, we discuss how to integrate FIP and code patterns together to achieve automated program synthesis.

3.2.1 FIP

We first briefly define the terminologies that are used in this paper.

Definition 1 (Action). Traditionally, an action in AI planning is defined as a triple, $a = \langle pre(a), add(a), del(a) \rangle$, where $pre(a)$ is the precondition of the action a; $add(a)$ is the post-condition achieved by the action a; and $del(a)$ is the delete effect that is no longer valid after the execution of the action a.

Definition 2 (Planning Problem). A planning problem is defined as a triple $P = \langle s_0, g, O \rangle$, where s_0 is the initial condition of the planning problem; g is the goal to be achieved; and O is a set of actions.

Definition 3 (Plan). Given a planning problem $P = \langle s_0, g, O \rangle$, a plan is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ that leads the system from the initial condition s_0 to the goal state g.

The reason that we have developed our own AI planner instead of using existing techniques is two-folds. First, existing planning techniques are not sufficiently expressive. The majority of AI planners are limited to deterministic domains and deal with only sequential planning, i.e., the actions in the generated plan are organized in a sequential manner. These planners are called classical planners. Each action is deterministic, i.e., the application of the action brings the system from the current state to a single other state. For example, a robot uses its arm to move a block from position *A* to position *B*. For classical planners, the effect of this action can always be predictable. As long as the robot can hold the block, it will definitely move the block to the expected destination. However, in reality, due to mechanical constraints, the robot's arm may drop the block during the moving process. Therefore, classical planning techniques make some impractical assumptions about the real world. For requirement elicitation, [11] points out that requirements engineers tend to set up goals and make some assumptions that are too idealistic. These assumptions are either not achievable or are very likely to be violated. Hence, to model the real world with more precision, it is desirable to require AI planners to be able to deal with nondeterministic actions.

Second, some efficiency and scalability problems have been reported for existing AI planners. There have been research works on nondeterministic planning domains, in which actions can have multiple nondeterministic effects. MBP [3] and Kplanner [12] are two such examples. However, as reported in [10], the CPU time of MBP may grow exponentially as the size of the planning problem grows. Kplanner suffers from the scalability problem due to its inherent mechanism of trying different loop bounds to generate the final plan.

Based on these reasons, we have developed FIP that can deal with nondeterministic actions and achieve highly efficient planning. FIP is based on planning graph [1]. It decomposes a nondeterministic action into a set of classical actions and conducts the planning process in two phases. In the first phase, a weak plan [3] is generated. The plan is weak because it only indicates one possible path leading to the goal. In this plan, only the ideal situations generated by actions are included. It is actually the optimistic shortest path leading to the goal. Based on this weak plan, FIP deals with the effects that are omitted in the first phase and generates a complete plan in the second phase. The search for a complete plan is conducted based on the shortest path along the weak plan. Hence, the overall search distance is optimal. In addition, the

planning graph is not a complete state space. It only contains states that are derivable from the initial conditions. Thus, the search space is much smaller than those of MBP and Kplanner. All these factors have made FIP a powerful and efficient planner qualified for dealing with practical problems.

3.2.2 Code Pattern

Definition 4 (Code Pattern). A code pattern *cp* is a named functional unit that captures the typical structure and composition of a set of components. *cp* is represented by a triple $cp = (i, b, c)$, where *cp* is the pattern name, *i* is the interface, *b* is the body, and *c* is a pair of pre- and post-conditions $\{P, R\}$. The functionality of a pattern *p* can be represented as $\{P\}cp\{R\}$.

NAME	GetJDBCDBConnection
INTENT	Establish the connection with the database
CONTEXT	JAVA/JDBC
SOLUTION	1. Load JDBC driver; 2. Establish the DB connection

CODE TEMPLATE	Code_template
	Interface
	IN: String driver;
	String dbURL;
	String userName;
	String pwd;
	OUT: Connection con;
	End_interface
	Body
	try {
	Class.forName(driver);
	} catch(java.lang.ClassNotFoundException e) {
	System.err.print("ClassNotFoundException: ");
	System.err.println(e.getMessage());
	}
	
	try {
	con = DriverManager.getConnection(dbURL, userName,
	pwd);
	} catch(SQLException ex) {
	System.err.println("SQLException: " + ex.getMessage());
	}
	
	End_body
	Constraint
	Pre: Known(driver) and Known(dbURL) and
	Known(userName) and Known(pwd)
	Post: Known(con)
	End_constraint
	End_code_template

Fig. 6. Code pattern example for JDBC

For example, in the surprise test management system, database operations for storing, retrieving, and managing screeners' records are necessary. Code patterns can be used to capture the typical usages of the JDBC components as well as their interactions. In Fig. 6, a simple code pattern about how to obtain a JDBC DB connection is defined. A code pattern consists of an interface, a pattern body, and a constraint section. The pattern interface contains pattern parameters which are used to customize the pattern. Pattern parameters are also called ports. Three types of ports are possible, namely, input ports, output ports, and input/output ports. An input port is a data

source, an output port is a sink, and an input/output port can be either a source or a sink depending on the pattern context. For this particular example, there are four input ports and one output port. The precondition specifies the condition under which the code pattern can be applied while the post-condition indicates the effect achieved after the execution of the code template body.

Four code pattern composition operations, including one instantiation operation (Map) and three functional operations (Concatenate, Invert, and Splice) have been formally defined for glue code synthesis. The instantiation operation, *map*, is used to instantiate a pattern to obtain a concrete code segment. For example, "driver = sun.jdbc.odbc.JdbcOdbcDriver; dbURL = jdbc:odbc:AirTravel; …" can be used to instantiate the code template in the code pattern body in Fig. 6 to obtain a segment of concrete code. The concatenate operation is used to connect two or more code patterns together sequentially to form a flow of data or actions. The invert operation obtains a code pattern that performs the inverse operation of the original pattern. The splice operation joins two code patterns together according to their internal loop constructs. It interleaves the internal code of the two code patterns and merges code frames inside the loop constructs.

The code pattern approach is especially attractive for large enterprises because they may already have a substantial repository of existing software systems and, hence, seldom need to construct a new system from scratch. Code patterns can be used to record typical usages of code segments that have been proven to be correct and are repetitively used in the system construction. This kind of reuse is an effective way to save cost and time. When the number of code patterns grows large, they can be organized in a code pattern repository for future use.

3.2.3 The Integration of AI Planning and Code Pattern

As discussed above, the integration of AI planning and code pattern can raise the level of abstraction and increase the level of reuse in system construction. As shown in Fig. 7, the AI planning and code pattern based automated synthesis system consists of two major parts, namely, the prototype Code Pattern Integration System (CPIS) [14] and the Automated Planning System (APS).

The CPIS at the top portion of Fig. 7 consists of a code pattern repository, a graphical user interface, a code pattern parser, and a code pattern composer. The code pattern repository stores all the code patterns. The GUI interface is presented to enable the system users to add, retrieve, and edit code patterns from the repository. The code patterns in the repository come from two major sources, i.e., patterns that are input from the GUI interface and the composite patterns generated by the code pattern composer.

The code pattern parser is responsible for checking the validity of the code template. It is implemented based on JavaCC and supports multiple programming language grammars, e.g., C++, JAVA, etc. The errors in the code template of the code pattern can be detected when it is loaded into the CPIS.

The code pattern composer supports the pattern operations, namely, map, concatenate, invert, and splice, to compose code patterns. The system users work on the code patterns in the repository and use the pattern operations to compose composite patterns to achieve semi-automated synthesis of the glue code. The productivity would be increased greatly if the code pattern operations can be automated.

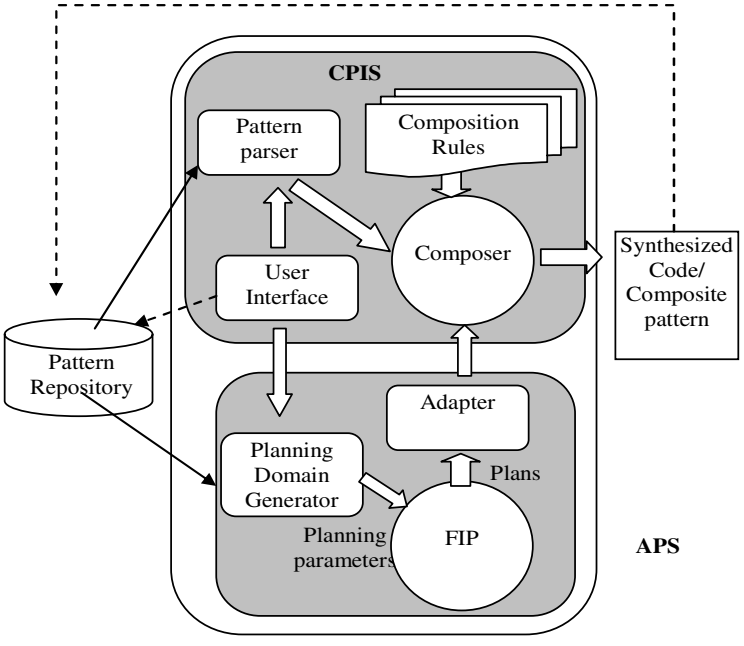


Fig. 7. Architecture of the code synthesis system

The automated planning system (APS) at the bottom portion of Fig. 7 is the core of the whole code synthesis system and can achieve the goal of automating the code pattern operations. It consists of three major parts, namely, planning domain generator, the FIP planner, and the plan adapter. As shown in Definition 4, a code pattern has a constraint portion consisting of pre-/post-conditions. The formalism of code pattern makes it naturally fit in the definition of AI planning actions. Hence, code patterns are modeled as planning actions. The planning domain generator serves as the bridge between code pattern repository and the APS and models code patterns as a planning domain. [5] also presents details about how code pattern operations are modeled in the planning system to facilitate plan generation as well as code synthesis. Given a planning problem, the AI planner, FIP, works on the actions that are derived from the code patterns and generates parameterized procedure-like generic reusable plans, i.e., *procedural plans*. The procedural plans are translated into preprocessed patterns by the plan adapter and fed to the code pattern composer to synthesize composite code patterns.

It should be emphasized that this program synthesis system is not limited to code patterns. The underlying components could be any components that can be abstracted with pre/post-condition constraints. For example, web services are well suited to the proposed architecture as shown in Fig. 7. The synthesis system is also an open system in which different component-based synthesis techniques may exist together. In this case, there will be multiple adapters for the AI planner to translate the generated plans to different underlying systems.

The input to the program synthesis system as shown in Fig. 7 is the planning problem, which is defined as a triple $P = (s_0, g, O)$, where s_0 is the initial condition, g is the

goal to be achieved, and O is the set of planning actions. As O is generated from the code pattern repository by the planning domain generator, the users just need to specify s_0 and g without having to know the details about how to achieve g.

3.3 The Integration of MDA and AI Planning and Component-Based Synthesis

As shown in Fig. 5, the code generated through transformation contains only code structures, which include the definitions of classes and operations, and/or the implementation of static getter/setter operations that are derived from the private attributes. The dynamic aspects of the system still remain to be completed. Therefore, we need a technique that is able to generate business code and complete some of the dynamic aspects of the system.

On the other hand, our AI planning and component-based code synthesis approach can generate business code based on the underlying code patterns. If it is integrated with MDA, it could greatly increase the productivity by generating the business code for the system dynamics.

To conduct the integration, we analyze MDA and its modeling process. UML is the de facto modeling language for MDA. However, UML is not good at modeling dynamic (or behavioral) parts [9]. The introduction of OCL 2.0 mitigates this problem and provides more choices for constructing high quality models. OCL is a formal modeling language that can be used to express conditions (pre-/post-conditions and invariants) and build software models. It is defined as an assistant language for UML. Hence, the combination of UML 2.0 and OCL 2.0 is the key to make the integration successful.

Specifically, pre-/post-conditions on operations can be used to express the system dynamics [9]. Formally, they can be expressed with a pair (P, R) representing the pre-condition and post-condition, respectively. As discussed in Section 3.2.3, the input to the AI planning and component-based code synthesis system is also a pair, (s_0, g), where s_0 is the initial state and g is the goal. The similarity of the two pairs (P, R) and (s_0, g) strongly suggests that the constraints on the operations can be formulated as a planning problem. Specifically, P is treated as the initial condition s_0 and R represents the goal g. The code synthesis system takes the input and generates the final code to fill in the body of the operation if the planning problem is solvable. The generated code is correct and is guaranteed to achieve the goal due to the following reasons:

(1) Code pattern is formally designed. Its functionality is expressed by the constraints, i.e., pre-/post-condition as shown in Definition 4.
(2) The code template in the code pattern is a proven solution to a recurring problem.

The AI planning and component-based code synthesis system tries to generate code for each operation based on its pre-/post-conditions. As shown in Fig. 2, the final code comes from three sources, namely, MDA, automated code synthesis system, and the developers. The multiple ways to automate the code synthesis could greatly speed up the development process and make the proposed prototyping method more practical. Moreover, the generated code (from MDA and code synthesis system) is correct and has good quality. This is another favorable factor for the proposed rapid evolutionary prototyping approach.

3.4 Analysis

Our rapid program synthesis approach (MDA + AI Planning + Component-based synthesis) has the same advantages as the heterogeneous models [27] discussed in Section 3.1, i.e., reuse the existing code to achieve the business code generation. However, our method is not subject to the disadvantages of the heterogeneous models.

First, the rapid program synthesis approach is not coupled with any high level models and does not hurt the MDA hierarchy. MDA, AI Planning, and component-based synthesis techniques can be seamlessly integrated together. Second, our approach does not complicate the maintenance process. The change of high level models will not result in maintenance burdens. Code can be regenerated along with the infrastructure code when the transformations are executed between different levels. Third, the rapid program synthesis approach does not hurt the MDA's benefits of portability and documentation. The generation of system dynamics is parallel in the transformation between PIM to PSM and from PSM to infrastructure code. In addition, the rapid program synthesis approach does not make any changes in the PIM. Thus, the PIM can still fulfill the function of high-level documentation that is needed for any software system [9].

4 Requirements Elicitation Via Prototyping

Based on the aforementioned advanced rapid program synthesis technique, we propose a prototyping approach that is intended to combine the advantages of the rapid and evolutionary prototyping. The rapid program synthesis technique ensures that the prototype can be developed rapidly and with good quality. In addition, the proposed prototyping approach implements the requirements regardless of whether they are poorly understood or well understood. In other words, the proposed approach will not be subject to the limitations of classical rapid prototyping and evolutionary prototyping.

Although the rapid program synthesis approach discussed in Section 3 can rapidly generate the correct code, it cannot generate a complete system fully. Manually composed code accounts for a certain portion in the prototype as shown in Fig. 2. In order to ensure that this portion of code does not compromise the prototype's quality, we apply a technique, baseline, that is similar to the operational prototyping approach [4]. A baseline corresponds to a well built prototype, in which the software is developed with production quality and only well understood requirements are included.

For the well-understood requirements, the standard MDA development cycle [9] (as shown in the left portion of Fig. 8) is followed to implement these requirements in a high quality manner. This is equivalent to the evolutionary prototyping. Then, a baseline is set up to record that the implemented prototype is of production quality. There is no poor quality code in the prototype within the baseline.

In the next step, the end users are trained to operate the prototype. This process may inspire them to clarify some of the unclear requirements or to come up with new requirements. The users may also experience some problems. All of these observations will be collected and sent to the requirements and design engineers.

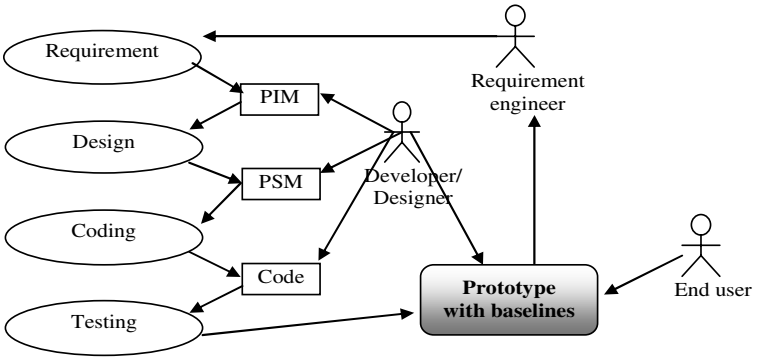


Fig. 8. Rapid evolutionary prototyping

For requirements that are critical but poorly understood, the throwaway (rapid) prototyping method is applied to implement them over the baseline. The implementation should be done as quickly as possible to illustrate the functionality to the users. After the users have identified the true requirements from the quick-and-dirty prototype, the portion that is not included in the baseline will be thrown away. The cost is not too high as the traditional throwaway prototyping is because the rapid program synthesis approach can help generate code to speed up the development. The automatically generated code is much cheaper than code that is manually composed.

Then, the MDA development cycle is repeated and the newly identified and well understood requirements are implemented with good quality to set up the next baseline. Afterwards, the implemented prototype is sent to the users again and the same processing steps are repeated if needed.

4.1 Discussion

With the rapid program synthesis technique presented in Section 3, the throwaway prototype over the baseline can be implemented quickly. Although the generated code is correct, it may not produce the desired effect because the requirements themselves may not be correct. The code generated based on the incorrect requirements are not valuable and must be discarded.

The proposed rapid evolutionary prototyping approach has some similarities with operational prototyping, e.g., the use of baselines, the way of handling poorly understood requirements, etc. There are also some major differences. First, operational prototyping uses conventional development strategies to implement requirements that are well understood. In contrast, the proposed approach applies the MDA development strategy, in which the design becomes part of the development and more stakeholders (software, design, and requirements engineers, etc.) are actively involved in the development process. The development focus has shifted from code to PIM, which is a higher level of abstraction. The artifacts that are created during the development process are models.

Second, our rapid program synthesis technique makes the proposed prototyping approach a practical method for requirements elicitation. It greatly speeds up the development process and ensures the quality of the generated code. The development cost

can be reduced as well. This is especially true for implementing poorly understood requirements. The code of the quick-and-dirty prototype would be discarded after the requirement elicitation. But the development cost of the automatically generated code is relatively cheaper than that of the manually composed code. If the generated code accounts for a large portion of the prototype, it implies that the cost can be greatly reduced.

4.2 Example

We still use the "surprise" test management system as an example to illustrate how the proposed prototyping approach works. As shown in Fig. 9, the surprise test management system consists of three subsystems, namely, user management, test management, and analysis subsystems.

The user management subsystem is a conventional information management system, which includes the major use cases of "add", "edit", "retrieve", and "delete" users. The requirements regarding the user management subsystem are well understood.

The test management subsystem is the key part of the system. It includes the major use cases of "test generation", "record test results", "retrieve tests", and "decision making". The requirements regarding this subsystem are not completely clear. Specifically, the users may have conflicting requirements about test generation. They cannot make an agreement about how screeners are chosen for the test and how inspectors are identified to conduct the test.

The analysis subsystem includes the major use cases of "report generation", "records evaluation", and "trend analysis". This subsystem is supposed to use the data mining technologies to implement the requirements. But the specific requirements regarding this subsystem are poorly understood as well. The users still do not have a clear idea about exactly what kind of reports needs to be generated and how the series of results could help in trend analysis.

The proposed prototyping approach implements the system in the following steps. First, the well understood requirements are implemented. Hence, the user management subsystem and part of the test management subsystem (e.g., record test results, retrieve tests, and decision making) are implemented in a quality manner. As the implementation relates to conventional database application development, abundant code patterns that capture the typical usages of JDBC and other database related operations exist for facilitating the code generation. A baseline is set up for the prototype indicating the completion of the well understood requirements.

Then, the users can operate the prototype and the problems found are collected. As the test generation function is absent from the prototype, the users cannot have a complete experience about the overall system. Hence, the test generation is critical but poorly understood. It is implemented by using the rapid (throwaway) prototyping strategy to illustrate its functionality over the baseline. In the prototype, the screeners are chosen at random for the test and the inspectors are identified in a round robin manner from the set of available candidates whose schedules are clear at the time when the test is supposed to be conducted.

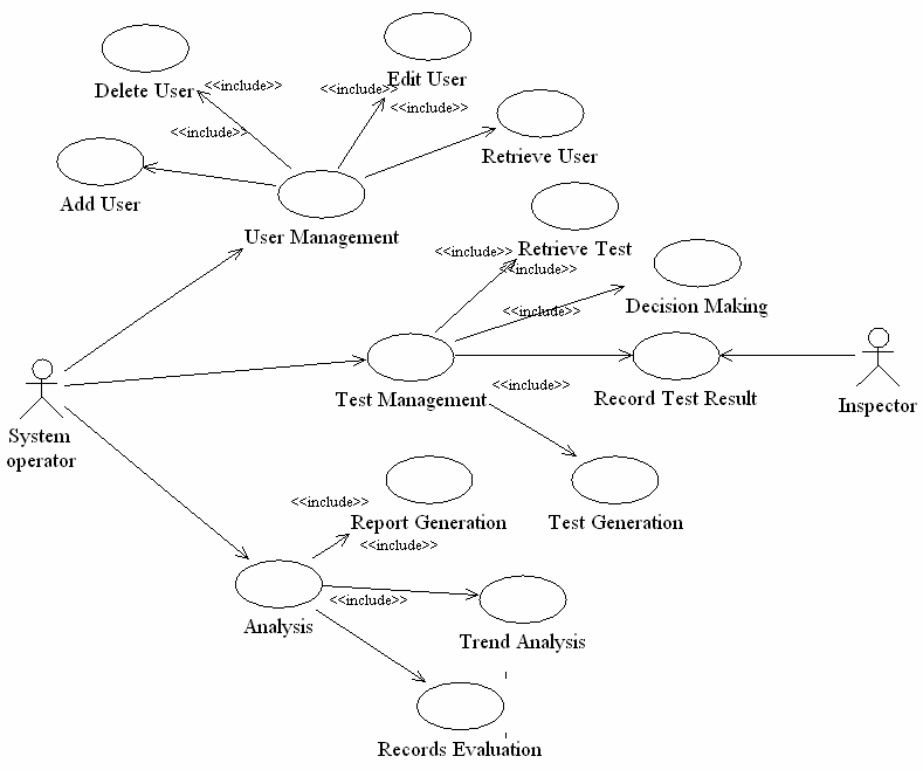


Fig. 9. Surprise test management system use case

Suppose the users reach an agreement after operating the prototype. They agree that the screeners who are most likely to fail the test should be chosen for the test and the way of identifying inspectors in the prototype is considered acceptable. After the requirements become clear, the part that is not included within the baseline is discarded, including the code for the inspector identification. Afterwards, the MDA development cycle is followed to implement the newly clarified requirements with good quality. Machine learning technique, e.g., Markov Decision Processes (MDPs) [26], can be used to predict which screeners are more likely to fail the test. Code patterns are available for solving the typical recurring problems in the MDPs area. The rapid program synthesis approach can help accelerate the prototype development. Then, another baseline is created and the prototype is sent to the users again.

For the analysis subsystem, the users can hardly come up with concrete requirements before they really touch the system and have the historical data available. This is especially amenable to the prototyping approach. After the prototype is in good shape to achieve the design goals for test management, the users will have better understandings about what they really need. The clarified requirements are incrementally fed back and implemented with the help of the proposed prototyping approach.

5 Conclusions

It is estimated that to fix a defect found during requirements engineering costs two orders of magnitude less than to fix the same defect after the product has been delivered [2][18]. This asserts the essential role of requirements engineering in the software development process. Software prototyping is an important requirements elicitation technique that can help find defects at an early stage and, thus, make the project more likely to succeed.

We have proposed a model-driven development based prototyping approach for requirements engineering. It inherits the advantages of prototyping elicitations without the disadvantages, such as untraceable requirements and tendency of reusing previously written code fragments, etc., by applying model-driven development principles and advanced program synthesis techniques, in which MDA, AI planning, and component-based software development techniques are seamlessly integrated together. The proposed approach is a rapid evolutionary process that iteratively refines the requirements, design, and implementation and yields high quality systems with the help of the novel rapid program synthesis technique.

References

1. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence 90, 281–300 (1997)
2. Boehm, B.: Industrial software metrics top 10 list. IEEE Software 4(5), 84–85 (1987)
3. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence 147(1–2), 35–84 (2003)
4. Davis, A.: Operational prototyping: A new development approach. Software 9(5), 70–78 (1992)
5. Fu, J., Bastani, F.B., Yen, I.: Automated AI planning and code pattern based code synthesis. In: ICTAI 2006, pp. 540–546 (2006)
6. Fu, J., Bastani, F.B., Ng, V., Yen, I., Zhang, Y.: FIP: A fast planning-graph-based iterative planner, Technical Report. UTDCS-03-08, UT-DALLAS (2008)
7. Harmain, H.M., Gaizauskas, R.: CM-Builder: A natural language-based CASE tool. Journal of Automated Software Engineering, 157–181 (2003)
8. http://www.andromda.org/
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
10. Kuter, U., Nau, D.: Forward-chaining planning in nondeterministic domains. In: Proceedings of the National Conference on Artificial Intelligence (AAAI-2004), pp. 513–518 (2004)
11. Lamsweerde, A., Letier, E.: Handling obstacles in goal-oriented requirements engineering. TSE 26(10), 978–1005 (2000)
12. Levesque, H.: Planning with loops. In: Proc. of the IJCAI 2005 Conference, Edinburgh, Scotland (2005)
13. Liu, J., Bastani, F.B., Yen, I.: Code Pattern: An approach for component-based code synthesis. In: Proceeding of the 7th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, FL, pp. 330–336 (2003)

14. Liu, J., Bastani, F.B., Yen, I.: A formal foundation of the operations on code Patterns. In: The International Conference on Software Engineering and Knowledge Engineering, Taipei, Taiwan, Republic of China ((2005)
15. Luqi: Knowledge-based support for rapid software prototyping. IEEE Expert 3(4), 9–18 (1988)
16. Luqi: Software evolution through rapid prototyping. Computer 22(5), 13–25 (1989)
17. Luqi, Berzins, V., Yeh, R.: A prototyping language for real time software. IEEE Transactions on Software Engineering 14(10), 1409–1423 (1988)
18. Luqi, Guan, Z., Berzins, V., Zhang, L., Dloodeen, D., Coskun, C., Pueett, J., Brown, M.: Requirements document based prototyping of CARA software. International Journal on Software Tools for Technology Transfer 5(4), 370–390 (2004)
19. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
20. Manna, Z., Waldinger, R.: Fundamentals of deductive program synthesis. IEEE Transactions on Software Engineering 8(18), 674–704 (1992)
21. Mcclendon, C.M., Regot, L., Akers, G.: The Analysis and Prototyping of Effective Graphical User Interfaces (October 1996)
22. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Reading (2004)
23. Nuseibeh, B., Easterbrook, S.: Requirements engineering: A roadmap. The Future of Software Engineering. In: 22nd International Conference on Software Engineering, pp. 35–46. ACM-IEEE (2000) (special issue)
24. Object Management Group: MDA Guide: Version 1.0.1, OMG document omg/03-06-01 (2005)
25. Overmyer, S.L.V., Rambow, O.: Conceptual modeling through linguistics analysis Using LID. In: 23rd international conference on Software engineering (2001)
26. Puterman, M.L.: Markov Decision Processes. Wiley, Chichester (1994)
27. Selic, B.: Model-driven development: Its essence and opportunities. In: 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC), pp. 313–319 (2006)
28. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley, Chichester (2006)
29. Stickel, M.E., Waldinger, R.J., Chaudhri, V.K.: A Guide to SNARK (2005), `http://www.ai.sri.com/snark/tutorial/tutorial.html`
30. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Reading (2003)

A Case for ViewPoints and Documents

Michael Goedicke[1] and Thomas Herrmann[2]

[1] Specification of Software Systems, ICB, University of Duisburg-Essen, 45117 Essen, Germany
[2] Institute of Applied Work Science, Ruhr-University of Bochum, 44780 Bochum, Germany
goedicke@s3.uni-due.de, thomas.herrmann@rub.de

Abstract. In this contribution we consider various sorts of vague and imprecise pieces of (requirements) specification information as different view points provided by different stakeholders. Usually there is an obvious "non convergence" in the stakeholders' views and it is important to address the various sources of ambiguity and inconsistency between such view points. We advocate addressing not only "traditional" inconsistency to drive development forward but include other forms of imprecision like ambiguity and vagueness. The aim is to provide a path from a decentralized viewpoint-oriented style to a document-oriented style of software requirements specification. We use parts of the airport security case study to show aspects of our approach.

1 Introduction

Usually many stakeholders are involved in software development processes to define the requirements from various points of views. Various research approaches address the consistency problem arising in such a context since the various partial specifications contained in a set of viewpoints are not consistent from the start. However, an area, which also needs to be addressed in such a context, is an additional notion of imprecision: ambiguity and lack of complete knowledge.

If it becomes apparent that a stakeholder is not able or willing to specify a certain situation or constellation, it is very helpful to record this fact in the specification. In the airport security scenario such situations can be identified in various parts of the recorded conversation. For example, the sub-activities of the screening process are not completely specified and it is not exactly clear what will happen if "the airlines are not doing well", or the list of measures to "increase security" is not complete (cf. fig. 2, 15, 16). The involvement of a number of different stakeholders in such a setting implies that each stakeholder provides such imprecise specification statements by uttering beliefs or his or her viewpoint. We have identified inconsistency in the view point-oriented research (see e.g. [3]) as a major driving force which can reveal hints on how to proceed in the software development process. We also argue here that dealing with the discernable cases of imprecision, ambiguity and other related forms of incomplete and potentially contradictory specification information provides support for the formulation of important questions in the process of eliciting new aspects and pieces of specification information from stakeholders. In this respect, ambiguity and imprecision can be seen as a special sort of inconsistency.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 62–84, 2008.

This states the challenge we would like to address here: based on a representation scheme for recording explicitly ambiguous and incomplete requirements specification (SeeMe [8], [10]) we identify how SeeMe can represent different view points, and we categorize how such combined pieces of information originating in different viewpoints can be combined and reconciled with each other. Usually, SeeMe-diagrams are used for the design of socio-technical systems (e.g. in the area of knowledge management) where formal structures (usually presented by technical components) and informal aspects (e.g. conventions of the social system) are combined. Therefore, SeeMe diagrams are developed in workshops where a facilitator's questions are answered by various stakeholders and continually documented by modifying the diagrams. Consequently, a single SeeMe-diagram can represent varying viewpoints. However, this workshop-approach does not work in those cases where a huge number of SeeMe-diagrams have been developed at several different locations and / or during various periods of time. Under these conditions, it is reasonable to provide techniques which automatically prepare a comparison of the varying viewpoints which are represented in a set of SeeMe-diagrams. For this purpose, SeeMe-diagrams have not yet been used due to a lack of technical support. The process and techniques we propose here lead to additional useful questions and development steps which serve as driving forces, especially for the upstream activities of the software development process.

In the following we give a brief outline of the overall approach and a brief account of the underlying ViewPoint concept for our work here. In addition the representation scheme SeeMe is briefly described in particular its ways of defining various forms of ambiguous and incomplete information. We then give a sketch of the necessary definitions for the various transformation steps, intermediate graph-based representation schemes and graph- based transformations. Finally, we briefly compare our work with existing approaches to ambiguity and incompleteness in specification. We conclude our presentation with a brief look at research issues currently being pursued.

2 Expressing Ambiguity and Imprecision Using ViewPoints

We briefly sketch the overall process since we propose here a combination of various concepts and techniques. As was pointed out above, we aim at increasing the usage of explicit statements regarding imprecision and vagueness in early stages in (software system) development processes. We assume here that the goal of the airline security scenario is actually to create and deploy a system, which is not explicitly expressed in the blog [13]. Thus, we assume that we see in the scenario of [13] the early stages of a development process.

The overall approach addresses the problem that from the start a common understanding of the problem(s) at hand is by no means perfect. This was the starting point of the research which led to the ViewPoint concept [2]. In the following we refer to the specific notion of view point in this research as *ViewPoint*. The basic idea is that relevant stakeholders express their view on the problem or given task in his/her representation scheme following a local process model. While this is still a promising approach to capture diverging opinions and views, such a decentralized approach to record relevant properties of a system under development induces the need to come up with a consistent set of requirement specifications upon which all involved parties can agree.

We believe that a decentralized approach is more realistic especially in large development teams. However, the need to come up with a single consistent document is also induced by management purposes and the nature of detailed technical software development as well. Later we will sketch the process we employ here.

The starting point is to create a set of ViewPoints using the SeeMe-representation scheme which specify the personal view of the stakeholders. We also assume here that either the stakeholder or a trained consultant will provide the respective personal ViewPoint. The overall process (see fig. 1) is divided into three stages. In stage one the various stakeholders create (or had created) their respective ViewPoints using SeeMe. These ViewPoints form the basis to discuss commonalities and differences. This is the focus of stage 2 where some automated support helps to find common parts and parts which differ in the ViewPoints. This is accomplished using appropriate internal graph-based representations and transformations (see section 4). The presentation of the common and different parts is then fed back to the stakeholders for further discussion and development e.g. back to stage 1. This may lead to a converging process which yields a consensus in the form of a common ViewPoint representing the consensus between stakeholders. Some differences may still exist but are considered unimportant for or irrelevant to further development. We argue to keep them, however, since these manifestations of diverging opinions are possibly important for future developments. They could also present the starting point for the discussion of variants in a product line development. However, this is beyond the scope of this paper and will be pursued as part of future work.

Once the common part represents the consensus among the stakeholders, a further development stage can be entered. Thus in stage 3 in fig. 1 a more detailed presentation of the common ViewPoint has been created from the SeeMe representation. This is again done by appropriate transformations. Depending on the degree of detail contained in the SeeMe representation, additional input from a moderator, consultant or stakeholder might be necessary to accomplish progress to stage 3.

In contrast to the original ViewPoint concept, we consider here – at least for the upstream development activities of stages 1 and 2 – only SeeMe as single representation scheme. This removes complexity for the presentation here and allows us to focus on the problem of representing vagueness and imprecision at the level of ViewPoints.

The ViewPoint approach [2] is helpful in analyzing the situation sketched in the case study. The stakeholders' view is presented using ViewPoints and given an appropriate representation scheme – first order logic e.g. xlinkit [15] or temporal logic will certainly fulfill the task – inconsistencies can be made explicit. However, this is only the starting point.

As was put forward in [17], emphasis has been put on living with inconsistencies and providing some repair actions if possible. The overall processes involving ViewPoints and related global properties (like convergence i.e. removal of inconsistencies) have to be elaborated and refined. If one regards requirements engineering in particular as a set of continuous activities parallel to creating other development artifacts, it is an important goal to integrate a document-oriented approach into the otherwise distributed views (c.f. stage 3 in fig. 1.). Thus the challenge is to integrate document centric views – in many cases expressed today in some UML dialects or sublanguages – with more interaction centric and decentralized views as represented by the blog of the three stakeholders of the case study.

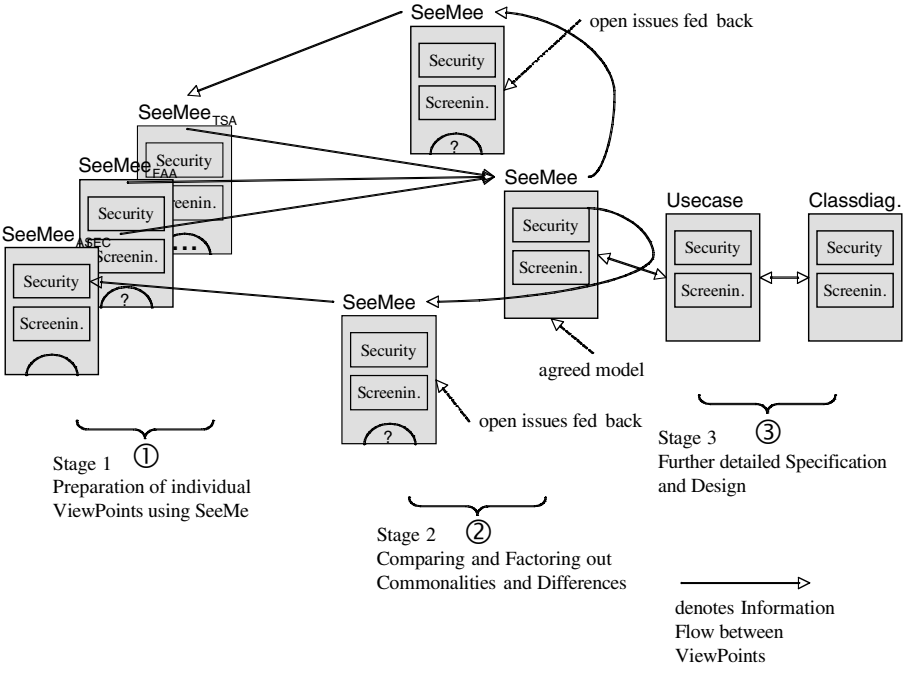


Fig. 1. Sketch of the overall Process

While much of our work was concerned with the first type of integration and inconsistencies, the latter is quite important. Earlier work [19] suggests that this is possible by modeling the involved layers (organization, informal relationships etc.) but additional results can be achieved if other forms of imprecise information are represented as well.

A contribution to this goal is to present vagueness explicitly as in the SeeMe approach [8], [10]. The characteristic of the diagrammatic SeeMe approach is to introduce explicit graphical elements in order to indicate ambiguities, vagueness or other type of imprecision.

Below we present the SeeMe representation scheme in more detail and then discuss the process of finding common and different parts of a SeeMe based ViewPoint.

3 Stating Ambiguous and Imprecise Specification Information: The SeeMe – Notation

SeeMe has been developed to mainly support the communication during the early phases of requirements engineering. It is based on communication theory which suggests that communicators only make explicit what is not already obvious by their context [11]. Therefore a modeling notation which represents a rich variety of aspects must allow the modeler to focus on the essential aspects. A design-oriented notation must not enforce the depiction of all details, as they are needed for context-free tasks

of programming, configuration or formulation of regulations. It must be possible to represent incomplete or uncertain information and to indicate those aspects of a model which are only incompletely specified. If misunderstandings are observed with respect to this incompleteness, it can be gradually reduced by making the diagrams more explicit and formal.

Therefore, for the early phases of designing socio-technical systems or processes it is reasonable to use a modeling notation to create diagrams which

- visualize the complex interdependencies between people, between humans and computers and between technical components
- can integrate overview sketches of the planned solution with the representation of rich details if a contributor wants to introduce them. Subsequently, it is not necessary to switch between different diagrams to see varying degrees of details
- integrate formal and informal structures as well as technical and social aspects
- handle incompleteness and vagueness (e.g. if it is not clear which sub-activities are part of a task or under which conditions these sub-activities are carried out.)
- and represent conventions, interests and multiple perspectives.

The development of SeeMe was triggered by the experience that available methods were not suitable to represent a combination of imprecisely as well as formally specified structures. We analyzed a set of common modeling methods for their appropriateness for socio-technical systems ([5], [7], [14], [18], [20], [25]). SeeMe is inspired by the extended-event-process-chain (eEPC) developed by Scheer [21], by use-case diagrams [20] and by State-Charts [6]. We have combined aspects of these methods and extended them with possibilities to express vagueness which includes incompleteness and uncertainty. Vagueness in SeeMe is related to a qualitative lack of information and not to a quantitative measurement of the probability of the occurrence or correctness of a certain modeling element.

SeeMe is designed to support communication. This being its main purpose, it is closely related to natural language. It can mirror the imprecision and context dependency which characterize the usage of natural language. SeeMe-diagrams can easily be derived from natural language documents, by indicating the keywords which are to be transformed into graphical elements. Figure 2 shows a tool, which supports this step. The modeler has to indicate the keywords, arrange the elements geometrically and connect them with relations.

Such a tool helps to incorporate the overall approach in a setting that starts with natural language documents and derives and maintains the other requirements and design artifacts incrementally. However, we will not pursue this here since we want to concentrate on explicit representation of imprecision and vagueness.

SeeMe helps to describe the interaction between people and physical or technical objects of the world, and therefore differentiates between three basic elements (see fig. 2):

- ***Roles*** (e.g. airport screeners) which represent a set of rights and duties as they can be assigned to persons, teams or organizations. Eventually, the characteristics of a role are based on the expectations of other roles. These kinds of reciprocal relationships are typical for social systems – roles are a means to introduce social aspects into the models.

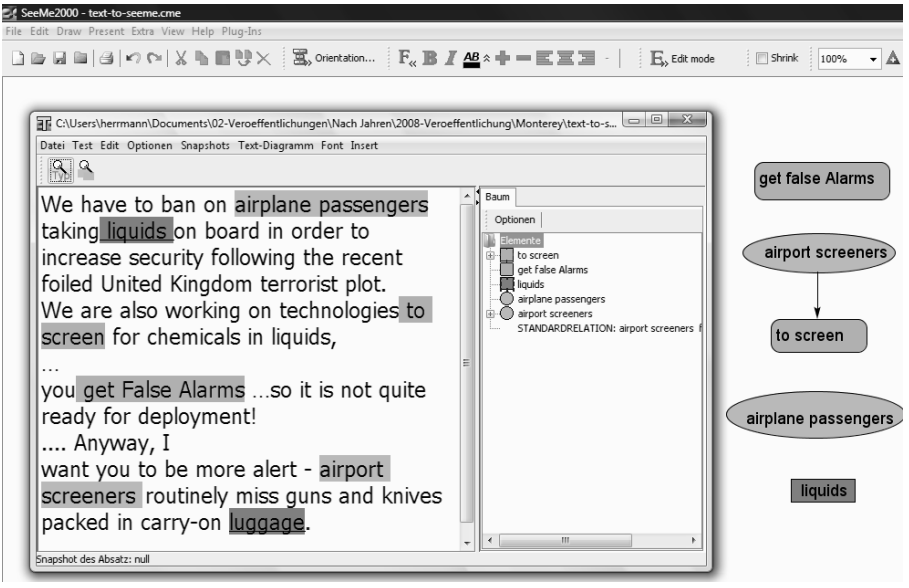


Fig. 2. Transformation support from Text to SeeMe

- ***Activities*** (e.g. screening) which are (usually) carried out by roles and stand for the dynamic aspects which represent change, such as the completing of tasks, functions etc.
- ***Entities*** (e.g. new technical support) representing passive phenomena; e.g. resources being used or modified by activities, such as documents, tools, programs, items of the physical world. They can represent containers (e.g. a box, a warehouse) or ephemeral phenomena (e.g. an utterance).

SeeMe offers nine standard relations depending on the types of elements being connected and on the relation's direction. Relations are depicted with directed arcs. They have a starting- and an ending-point which are anchored in basic elements. The "reading-direction" mirrors the direction of the arcs in the following exemplary definitions[1]:

- The role ***carries out*** [1] the activity;
- the activity ***influences*** [2] the role (e.g. the passengers);
- an entity (`luggage`) ***is used by*** [3] the activity;
- which ***produces*** *or* ***modifies*** [4] an entity (`technical support`);
- an activity ***is followed by*** [5] another one.

Relations can be connected to super-elements or to one of its sub-elements (that means crossing the border of the super-element). If a relation is pointing to a super-element, it is also referring to all of its sub-elements.

Relations can be **incompletely anchored to elements**: If it is not clear whether a relation should be connected to the whole super-element or only to a subset of its

[1] The numbers [1]-[5] in the explanation refer to the numbers labeling the related arcs in fig. 3.

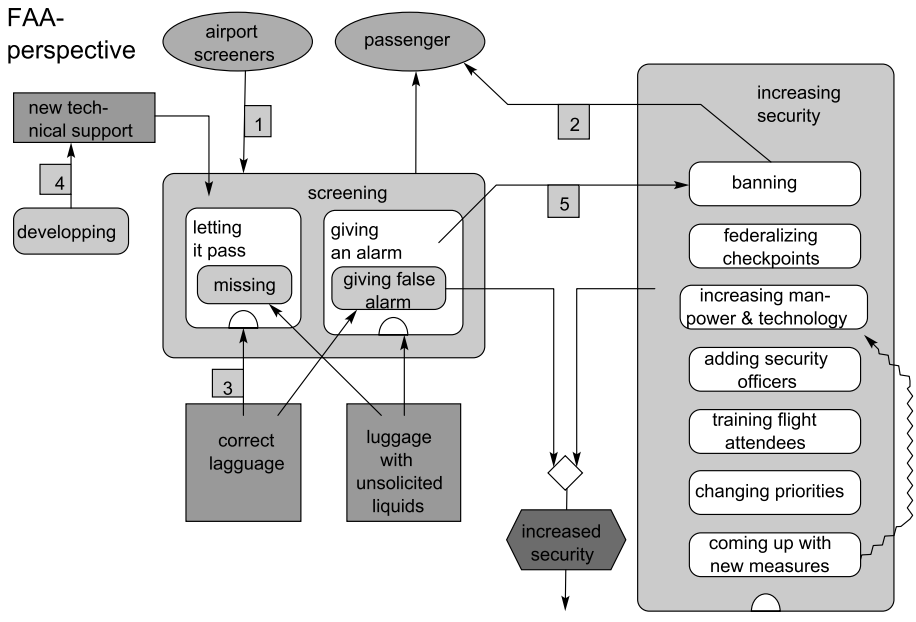


Fig. 3. Basic elements of SeeMe

sub-elements (and then to which of them), the **relation crosses the super-element** (increasing security) and is not connected to a distinctive sub-element. In the example (fig. 3), it is apparent by the crossing relation that not every case of "giving an alarm" leads to banning the passenger to take the liquid with him – the sub-activities which lead to the decision are not displayed.

Relations **can be left out**, e.g. between sub-activities if it is not clear in which sequence they occur. While "giving alarm" and "banning" are in a sequence (fig. 3), this is not the case with "letting it pass" and "giving an alarm". An arrow can also start or end in an undefined space if the element to which it is anchored is unknown or not represented in the diagram – e.g. to indicate the need to collect further information.

If two or more relations are assigned to the same element with their starting- or end-points, the question is how the interdependency between them can be described. These kinds of dependencies are expressed with logical **connectors**. Typical logical constellations are "or", "xor" or "and". However, the **logical type** of a connector can be left **unspecified** if its meaning is clear from the context of a diagram or if it is not reasonable to be more precise.

Segment lines are part of the notation with which one separates super-elements into segments which help to cluster sub-elements or attributes according to different perspectives. Different observers or stakeholders have different views of a system. None of them can be complete. In some cases it can be sensible to represent these different views in an integrated manner. Therefore, SeeMe allows a modeler to represent different perspectives of the decomposition of an element. Figure 3 shows the perspective of the FAA on activities, which "increase security". The perspective of the TSA could be added to the same super-element by including more sub-activities, which could then be separated by a segment line.

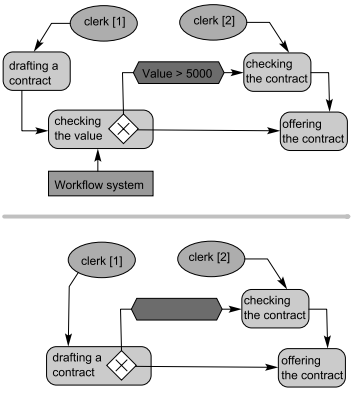


Fig. 4. Control vs. freedom of decision

If a connector represents a logical "OR" or "XOR", the question arises of under which **condition** a relation can be instantiated. In many cases, this decision can be clearly derived from the context. If this contextual specification is not clear enough for the relevant stakeholders, so called **modifiers can be annotated** (green hexagons in figure 4). **Modifiers can** also **be incomplete**: they can be empty if we only know that the instantiation of an element or relation depends on a condition but the condition is unknown or changes from case to case. A specific relevance has the usage of this kind of incompleteness to express **freedom of decision** as shown by the second case of figure 4. In the upper case, a `workflow system` decides whether a second `clerk` (the numbers in brackets indicate that different persons should instantiate the roles) will `check the contract`. In the second case, it is the `clerk`[1] who decides whether a checking of the contract is reasonable or not. The condition is left unspecified to express that it is specified ad hoc by the `clerk`[1].

SeeMe is constructed in such a way that it is flexible in both directions: it can be used to express vague, informal structures and it can support formal specifications which are similar to UML-activity diagrams, flow charts, eEPC [21] or Entity-relation-diagrams.

How SeeMe is used

SeeMe diagrams are usually employed in facilitated workshops to conduct walk-throughs. During these *socio-technical walkthroughs* (STWT, [9]), details of the diagrams are discussed and modified step-by-step. The facilitator has to prepare reasonable steps as well as questions which guide the critical inspection and discussion of the diagrams. The purpose of the workshops is to bring different stakeholders together and to integrate the varying viewpoints into one diagram which can serve as the large picture of the constellations that have to be taken into account during the requirements analysis and construction. It is obvious that the viewpoints of the participants can, for several reasons, not be completely represented – therefore the STWT-diagram contains imprecision which can be indicated with the SeeMe-symbols as described above.

SeeMe has been employed in more than 12 practical projects to support requirements engineering, software development, training of software users, improvement of

processes, documentation of technically supported processes as reference models, documentation of needs for adaptation and their realization, support of organizational change. The projects took place in areas such as knowledge management for services and manufacturing, printing industries, campus management and library software for universities, production and distribution of photographs. The average number of diagrams per case is 11,5 for those diagrams that have more than 20 elements. The practical cases were interwoven with phases of scientific reflection. The main achievements of this reflecting refer to the questions of how the diagrams can be used to support facilitation and incremental improvement of socio-technical systems, and of how to deal with imprecision.

There are different ways in which one can deal with incompleteness as it occurs in SeeMe diagrams:

- Incompleteness can be retained if it is clear that all the participants of a project know what is meant and were able to complete the missing data by themselves. Vagueness will be eliminated during one of the following workshops when the requirements construction has to be completed or when prototypes are available.
- Incompleteness will only be eliminated after the designed socio-technical system has been brought into reality and has been tested for a while.
- Incompleteness remains a part of the socio-technical system since the basis to overcome it changes from case to case so that decisions have to be made flexibly.

The main limits of SeeMe, which became apparent through empirical research where workshops and interviews were conducted, were the lack of support for programming and the readability of the diagrams by those individuals not supported by a facilitator. Although the STWT-workshops are a convenient way to integrate the viewpoints of several stakeholders, the situations where it can be used are limited for the following reasons:

- Too many perspectives or too many representatives of a certain viewpoint have to be taken into account (e.g. all the regional branches of a large company).
- The relevant stakeholders don't have time to meet during a time consuming workshop.
- It may be more advantageous if the diagrams are modeled in the immediate context of a workplace where all the aspects which have to be taken into account are co-present.
- Perspectives or diagrams may have to be taken into account, which have been created in the past by interviewing stakeholders who are no longer available.

In these cases it is very helpful and reasonable to employ automatic mechanisms which help to compare diagrams and help to extract the differences between them. The differences can either refer to divergences between concrete specifications or to cases where one stakeholder gives a detailed specification while another prefers to stay incomplete or imprecise with respect to a certain issue. The next section (4) will discuss how differing SeeMe-specifications can be compared.

Further perspectives

In the course of software development processes, SeeMe mainly provides support for the communication between varying stakeholders during the early phases of requirements analysis. In most cases where SeeMe has been used, software-engineers have

used consolidated SeeMe-diagrams for either immediately starting with prototyping or for manually creating UML-diagrams as a basis for the programming phases. It is a part of ongoing research - but not a focus of this paper - to develop semi-automatic mechanisms for transforming SeeMe-diagrams into more formal notations such as UML-Activity-diagrams or BPEL specifications, or EPC (c.f. stage 3 in fig. 1). The challenge here is to combine automatic transformation with the activities of a human modeler to complete information and to eliminate vagueness which is included in the SeeMe-diagrams.

4 Comparison of SeeMe Specifications

The basic idea to compare SeeMe specifications is to use the underlying graph-based data structure of SeeMe specifications. We use the general graph comparison procedure SiDiff [24], which uses two graphs to create a new graph. This graph contains the two original graphs plus additional edges identifying common and different parts of the original graphs. In fig. 5 we partially sketch the result of comparing two graphs. The various graphs are obtained by a special export of the SeeMe – editor. The actual tool chain we use will be explained below.

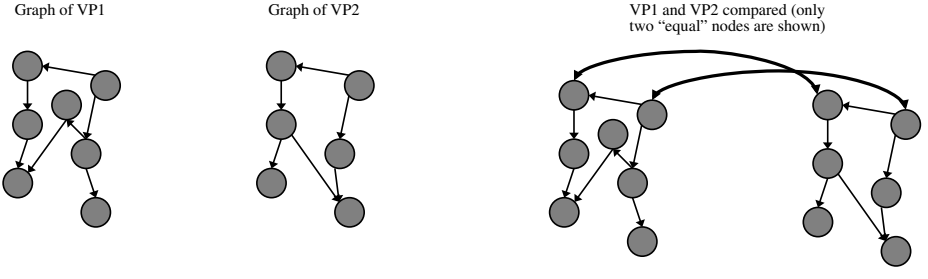


Fig. 5. Graph Differencing

The various nodes in the graphs have attributes and types allowing the identification of the elements of the corresponding SeeMe – diagram. In order to compute the actual specification which represents the common part, and the representation of the difference, we use the graph transformation tool AGG [22]. This tool offers powerful transformation capabilities to select parts of a graph and modify it according to a set of given rules. These rules define in a declarative way to manipulate a graph.

In our case we need a set of rules which extract common and different parts from the SiDiff comparison result. These new graphs are then transformed into the representation, which can then be used by the SeeMe-editor to visualize the result of the comparison in terms of the SeeMe – representation scheme. These steps realize the stages 1 and 2 of figure 1 above. An additional set of rules is needed to come up with a derivation of new representations like Use Case descriptions etc. (stage 3 in fig. 1).

In detail the following steps are necessary:

1. Export of the involved SeeMe models as XML-files
2. Production of the difference of the XML-represented SeeMe models using SiDiff

3. Import of the compare-graph into the AGG – transformation system in addition to the rule set
4. Application of the rules to extract the common part and the difference parts
5. Export of these parts in a SeeMe compatible format for further display

Later we will give more details of the procedure sketched above. In addition to steps 1. to 5., some auxiliary processing is necessary in order to optimize the various main comparison and transformation steps. We refer to the steps and involved tools sketched above as the *tool chain* comprising the SeeMe-editor, SiDiff graph differencing engine and AGG graph transformation system.

4.1 Comparison and Transformation Steps

This addresses the first two steps in the tool chain sketched above. The SeeMe-editor provides an export of a SeeMe-model as an XML-file. This XML-format is a one-to-one copy of the internal editor-data structure and contains all necessary data of a model, including geometry and layout information. This kind of model-related information, which is important for good tool usability, is not part of the semantics of SeeMe as a notation. Since this layout related information disturbs the graph differencing procedure, this part of the XML-file is removed. In a more sophisticated implementation of our procedure we will not throw this information away but hide it in a way (e.g. as a special attribute) so that it survives the transformation process. Later on the layout information can be recovered and used to produce a good layout when the SeeMe representation of the comparison and transformation result is displayed.

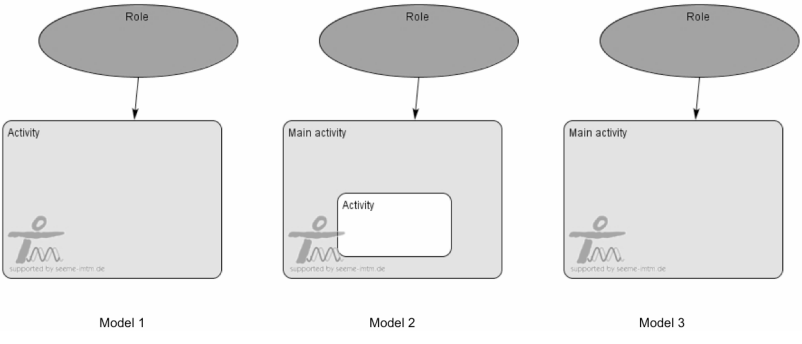


Fig. 6. Three simple SeeMe models

In order to show some details we use three simple[2] SeeMe diagrams which are transformed and compared. In figure 6, we see these simple models as SeeMe diagrams, while in figure 7 the XML-representation of Model 1 is partially given (parts of the XML-tree are hidden which is indicated by the "+"-sign at the respective XML-tag).

As can be seen in fig. 7, the elements of Model 1 (Role, Activity etc.) are represented as appropriate XML elements (e.g. <mBaseElement type=Role …>). There is

[2] These diagrams are very simple. However, since the related XML-files and graphs become very large with more realistic example specifications we will stick with such simple diagrams for the purpose of the presentation here.

the section in the XML document which is given by the `geometric` tag below the `extension` tag. This extension part is removed for the purpose of comparing the XML documents. In addition there are so called `somID`, which are unique identifiers for the elements of a model.

Fig. 7. XML representation of Model 1 of fig. 5 (some XML-Elements hidden)

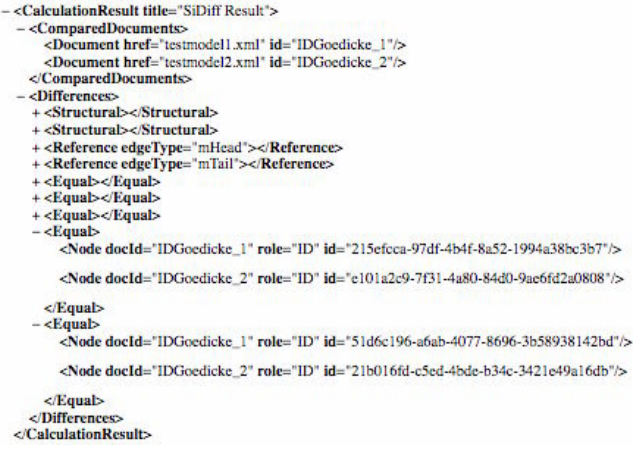


Fig. 8. Result of comparing Model 1 with Model 2

In figure 8 we see the result of comparing Model 1 with Model 2. Again only a part of the elements are shown but one can see that certain elements of the involved graphs have been identified as equal. Comparing Model 1 with Model 3 yields the result shown in figure 9 (SiDiff detects an *update* in the node `Main Activity` although this is not necessarily an update in the traditional sense since it may have arisen from parallel development and is simply *unequal*; but we will ignore this for the time being).

```
– <CalculationResult title="SiDiff Result">
  – <ComparedDocuments>
      <Document href="testmodel1.xml" id="IDGoedicke_1"/>
      <Document href="testmodel3.xml" id="IDGoedicke_2"/>
    </ComparedDocuments>
  – <Differences>
    – <Update attributeName="som.NAME">
        <Node docId="IDGoedicke_1" role="ID" id="215efcca-97df-4b4f-8a52-1994a38bc3b7"/>
        <Node docId="IDGoedicke_2" role="ID" id="08f66df1-f913-469e-bc62-aaae65a4be98"/>
      </Update>
    + <Equal></Equal>
    + <Equal></Equal>
    + <Equal></Equal>
    + <Equal></Equal>
    + <Equal></Equal>
    + <Equal></Equal>
    </Differences>
</CalculationResult>
```

Fig. 9. Result comparing Model 1 with Model 3

Based on this comparison result, further transformations can extract useful diagram parts which are then reported back to the stakeholders. This can then be used to drive the stakeholders' discussion forward, making commonalities and differences explicit at the diagram level. In figure 10, the AGG view [23] of the comparison result is depicted and is a direct representation of the XML-file of fig. 9. This is transformed using a simple set of rules into XML-files, which can be read in to the SeeMe editor. It can directly depict the comparison result at the level of SeeMe models. Figure 15 shows this result.

In order to explain the transformation process in more detail, we have selected some parts of the graph shown in fig. 10 which provides the entire graph as overview. These parts are indicated by the labeled ellipses 1, 2 and 3. Basically, these parts represent three kinds of sub graphs which can occur in the comparison result. Starting from the root node (labeled with `calculationResult`) in part 1 (fig. 11) a node (`ComparedDocuments`) identifies the compared documents (`Document`). This provides the pointers to the original SeeMe diagrams.

In fig. 12, part 2 of fig. 10 is given. This shows the update-node of the result graph. This identifies the two nodes in the SeeMe diagrams of Model 1 and Model 3 where the activity name update has taken place.

In fig. 13, part 3 of fig. 10 is provided. This shows a node which identifies the elements of the SeeMe diagrams Model 1 and Model 3 that have been identified by SiDiff as equal. As one can see in fig. 10, SiDiff has identified most of the elements of Model 1 and Model 3 as equal – which in this case of the constructed example is

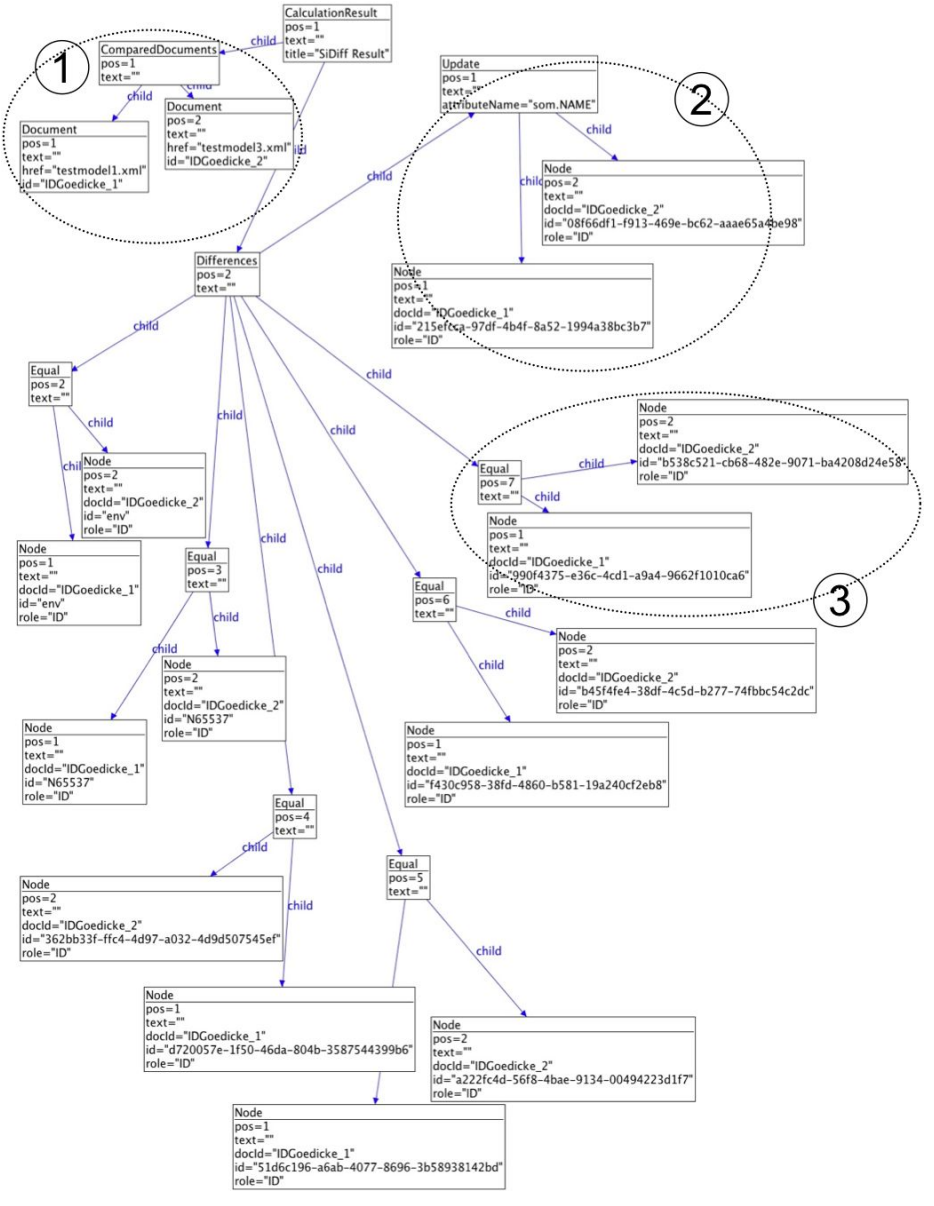


Fig. 10. AGG screenshot representing the comparison result shown in fig. 9

not a surprise. However, the SiDiff procedure applied to the more complex SeeMe specifications (fig 3, 16 and 17) show similarly good results and it is beyond the scope of this paper to present them here.

Based on such structures it is straightforward to provide graph transformation rules that use the comparison result of fig. 10, the graph representation of the two compared SeeMe specifications and produce an actual arc between the nodes of the diagrams

identified as “equal” by the SiDiff procedure. In fig. 14, a sample rule is given which creates an (double ended) arc between two elements of two diagrams. The notation used in fig. 14 is not exactly the input format for the graph transformation tool AGG, but comes close and only minor technical details have been left out to keep the presentation simple. The nodes labeled `Equal` and `Node` are from the result graph, while the nodes labeled `mBaseElement` are from the graphs of the two diagrams being compared. The notation `$1`, `$2` for the attributes `ID` and `somID` respectively defines that the attribute values for `ID` and `somID` are variable but have to be identical at the respective nodes.

As a result of applying such transformation rules as shown in fig. 14 a combined diagram shown in figure 15 is obtained.

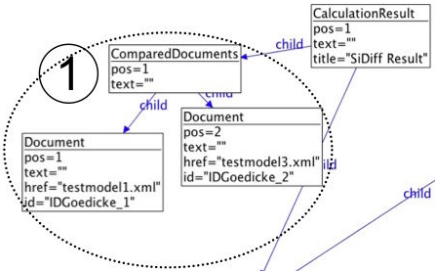


Fig. 11. Part 1 from fig. 9

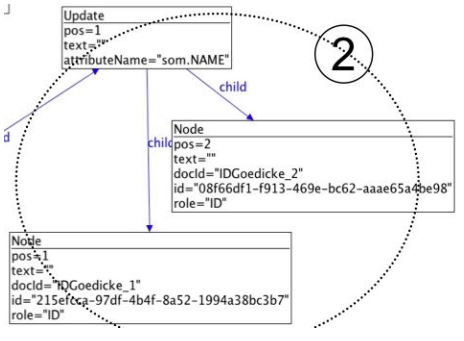


Fig. 12. Part 2 from fig. 9

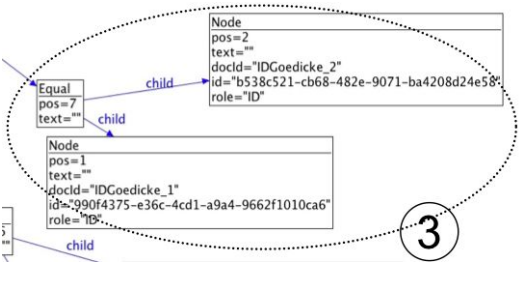


Fig. 13. Part 3 of fig. 9

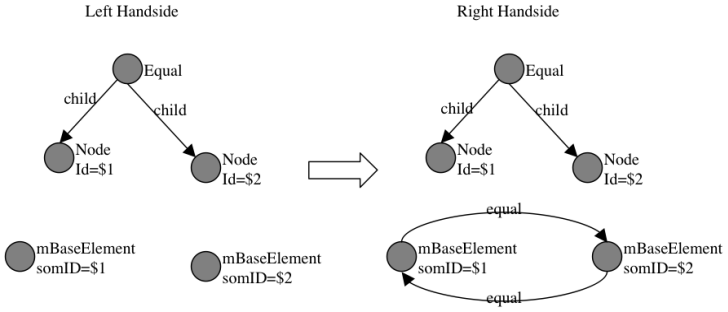


Fig. 14. Rule creating links between "equal" – elements of two diagrams

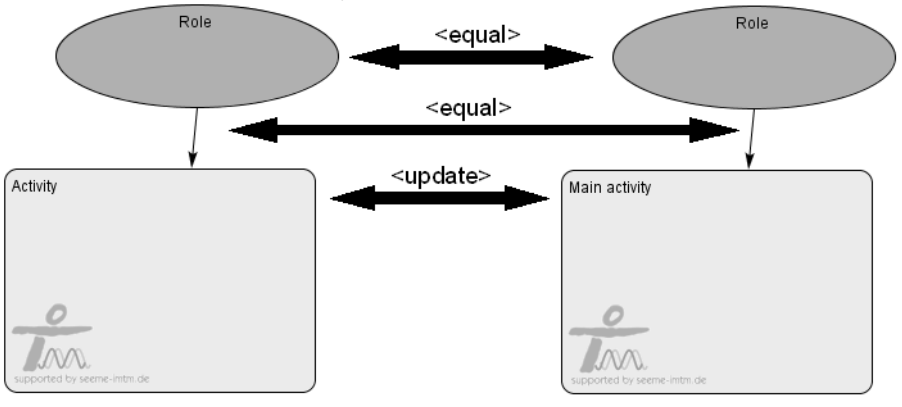


Fig. 15. Comparison result at the SeeMe model level comparing model 1 and model 3 of fig. 6

The result seems to be trivial but this is due to the simple examples used here in order to keep the overall presentation short enough. However, using the SeeMe diagrams representing some views of the Airport Security scenario also showed good results. Further aspects of tuning the graph differencing process and the issues related to the presentation of the comparison will be discussed later.

Once this state is reached the development process can be pursued using the various specifications either by going back to stage 1 (c.f. fig. 1) where additional discussions are performed to get more information regarding the project at hand, or by stepping forward to stage 3 in which the agreed results are used to derive further specification artifacts at a more detailed level towards a solution.

4.2 Comparison of Important SeeMe Fragments Regarding Ambiguity and Imprecision

Looking at the case study [13], three stakeholders are involved. The FAA ViewPoint is sketched in fig 3. The airport security agent's ViewPoint is depicted in fig 16 and the TSA's perspective is given in fig 17.

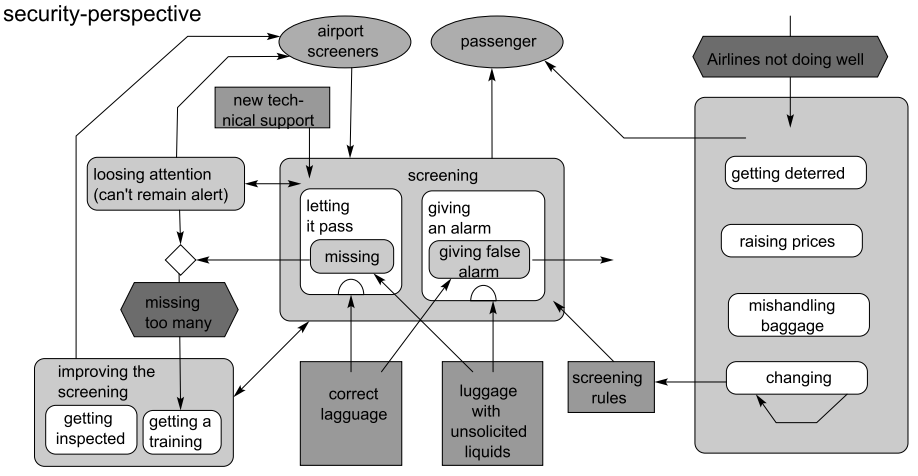


Fig. 16. Airport Security Agent SeeMe ViewPoint

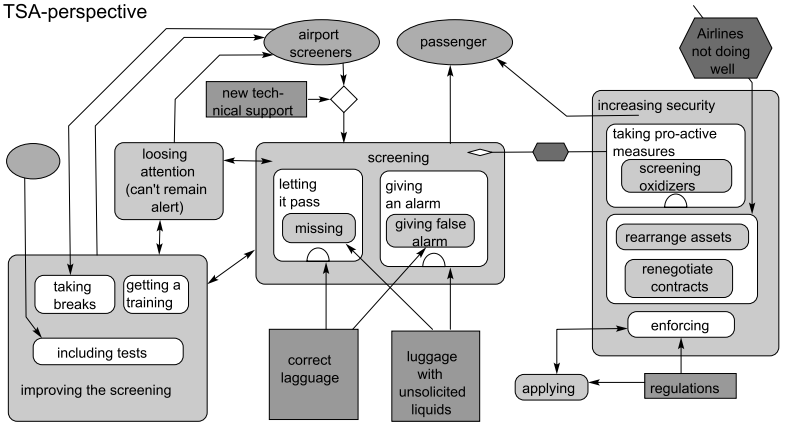


Fig. 17. Transport Security Agency's SeeMe ViewPoint

The diagrams of fig 3, 16 and 17 were derived by the analysis of a text which represents a dialogue between the different roles. The key concepts and statements which dealt with measures to improve security and the problems to be overcome were identified. In a further step it was decided whether to represent them as roles, activities or entities or as a combination thereof. Relations were either derived from the text or added if they were implicitly suggested by the meaning of the context. Symbols for vagueness were used if the dialogue did not reveal the needed information or if it just gave hints which pointed to further possibilities which were not explicitly explained. Sometimes super-elements were added to cluster elements and to provide a better overview (e.g. in fig. 16 to cluster "getting deterred", "raising prices" etc.).

The pair-wise comparison of the diagrams (fig. 3, 16 & 17) yields the screening activity as the common part of these SeeMe diagrams. Since a useful way to present such commonalities and differences at the SeeMe level is beyond the scope of this paper and subject to further research, we will not go into these details. However, graying out common parts and highlighting differences is a way to go forward. Another useful idea might be to keep the common parts static in a presentation and let the user flip through the differences (as in e.g. iTunes cover flow). The usability of an appropriate user interface for the SeeMe editor will also be crucial to creating an effective way to communicate differences and commonalities.

In principle, however, a limited number of different situations may arise. Due to this fact, a small set of graph transformation rules (e.g. in AGG) can deal with the task of extracting the common and different parts and provide the necessary additional information to show these parts in the context of the involved diagrams. The overall structure of the situation is sketched in fig. 5. There the dotted lines link corresponding nodes of the graph representation of two SeeMe models. Thus all common parts are flagged by such kinds of arc in the graph. The parts of the two sub graphs which are not linked by such "equal" arcs are displayed as mentioned above using different color, highlighting etc. The various types of combinations regarding the difference part are:

- Update of an attribute (as indicated by a special arc in the graph as well)
- Missing / different relations between SeeMe elements
- Missing / different SeeMe element (Entity, Activity)
- Different structures
- Different degrees of explicit imprecision and ambiguity
- Varying inclusion of different perspectives in the same diagram

The main question with respect to the identified differences is about the reasons for these differences (why-question) and how to deal with them (how-question). Depending on those kinds of elements *not* indicated as common, these "why"-questions can be refined. Therefore, strategies for representing the differences have to be implemented.

For example, figure 18 displays a comparison of the FAA (fig. 3), security (fig. 16) and TSA (fig. 17) perspectives. The diagram of fig. 18 is manually constructed to illustrate the potential outcome which should be achieved by an automatic comparison. Those areas, which are not different, are indicated with grey boxes. The specification of the sub elements of the activity "increasing security" is extended by including all three perspectives. Relations which could disturb the comparison are hidden by a hide-and-show mechanism provided by the SeeMe-editor. They can be retrieved with the grey residues which are left over if a relation is hidden. It becomes apparent that adding the different perspectives into one super-element ("increasing security") can help to reduce incompleteness, or can increase uncertainty about the question really represented by a super-element.

4.3 How to Deal with the Differences

It is natural to ask whether differently decomposed diagrams can be integrated into one large picture or whether the differing specifications do exclude each other and therefore require a decision on how the final specification should look. The more

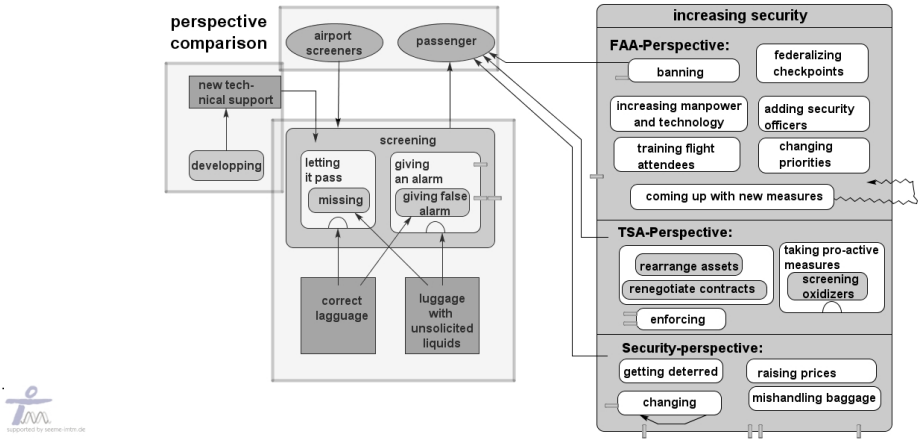


Fig. 18. Making commonalities and differences more comprehensible

difficult case is to deal with conflicts with where the decision to reduce ambiguity or uncertainty is necessary. This can take place in different phases of software development and therefore on different information bases:

- Differences and conflicting perspectives are consolidated during the early phase of conceptual design – usually in workshops where the differing specifications are discussed.
- The decision is only made when first prototypes can be evaluated and help to make the practical impacts of the differences more comprehensive.
- The differences cannot be consolidated before a first period of practical usage has taken place and has revealed the real problems behind the different views.
- The differences apply to different situations and the software has to be designed flexibly enough to deal with all kinds of differing specifications since they represent different situations which can happen in real life. For example it may be the case that the two different constellations represented in figure 3 (pre-programmed workflow vs. freedom of decision) can both be appropriate with respect to the tasks to be supported by the software – and therefore the software has to be flexibly configured depending on the cases to be dealt with.

This shall suffice to describe our approach to combine SeeMe and the ViewPoint concept. As we have been able to see, this is possible and has numerous opportunities for exploitation. We briefly summarize our contribution and provide some context with other work.

5 Assessment and Related Work

The work and the approach presented here address a number of areas. On the one hand, it is the decentralized view-oriented work that we use to capture different perspectives of different stakeholders which is a natural follow up to the research regarding the ViewPoint concept and similar work. On the other hand, the use of SeeMe as a

representation scheme within a ViewPoint introduces the explicit specification of imprecision and ambiguity in a specification statement. The entire approach is a combination of both ideas and presents a step beyond the "living with inconsistencies" paradigm of the ViewPoint concept. One can regard the introduction of explicit ways to state ambiguities and imprecise statements in a specification as a special way to use the idea of "living with inconsistencies".

By offering these explicit ways for specifying ambiguous and imprecise facts in a statement, new checks between ViewPoints become possible. The results of the checks provide additional insight into the specific development process and the involved stakeholders' view.

The research related to viewpoints [2] and [16] sparked a lot of further research since its intention was (and still is) to consider the decentralized nature of actual software development processes seriously. The related paradigm of "living with inconsistency" is a natural consequence of such a decentralized approach. Thus much of the work in this area has been on detecting inconsistencies by defining relationships between ViewPoints [17]. This kind of work has been tackled in various ways. One was to use various forms of logic and in an operational form first order logic as in the xlinkit-approach [15], or to use graphs and graph transformation [1], [4]. The main problem in these attempts was to find a generic way to express commonalities and differences in the specification part of a ViewPoint in an operational way. One is to use an executable form of first order logic as in xlinkit [15]. While this approach is powerful and efficient, it lacks the more general power of graph matching and graph transformation approaches. This is not so much in the sense of the theoretical expressiveness of graph transformation approaches but more in the sense of the way of expressing the involved rules. However, the well-founded mathematical graph transformation approaches as in AGG [22], which are based on algebra and category theory, imply the enumeration of all possible combinations of element types in their respective left hand side of the involved rules. This is, of course, possible but given the number of types of elements (as in SeeMe) it quickly becomes impractical to consider all the combinations of element types of realistic representation schemes.

Due to the progress in generic graph differencing engines as, for example, in SiDiff, new effective ways to compute commonalities and differences in graph-based representation schemes become available. Thus the combination of such techniques presented here is a step forward in using decentralized specification schemes successfully.

The strength of SeeMe – if it is compared with other methods – is the possibility to express and indicate vagueness. SeeMe is the only modeling notation which explicitly indicates and deals with vagueness from a qualitative point of few. It includes various means of depicting that a specification is incomplete or that its correctness is uncertain. SeeMe does not use probabilities or quantitative intervals to express imprecision. The quantitative modeling of uncertainty is the main subject of other approaches, which combine fuzziness with modeling see e.g. [12]. To compare SeeMe with quantitatively oriented approaches is not reasonable since they pursue different purposes. Other methods deal implicitly with qualitative imprecision: Use case diagrams are incomplete since they represent an overview or are on a higher level of abstraction. Other methods, like i* [26] represent dependency diagrams which offer the possibility to represent dependencies between goals, conditions, tasks etc. i* differentiates between goals and soft-goals. Using a symbol for a soft-goal can be considered as an

explicit indicator of imprecision. However, dependency diagrams are not process-oriented and are not a means to support a step-by-step refinement to derive functionality and modes of interaction with the technical system. SeeMe is not exclusively focused on the interaction with the technical system, as is the case with use-case diagrams in UML. By contrast, SeeMe supports the presentation of entire processes and work settings. Compared with methods which are similar to flowcharts, SeeMe has been extended by adding the possibilities for embedding sub-elements. All kinds of methods which use the nesting of elements (e.g. state charts, [7]) can omit details when they present embedded substructures. However, these approaches do not make explicit whether the modeler is aware of the incompleteness of a diagram, or not. The viewers cannot discern whether a lacking aspect has been intentionally or unintentionally left out. Furthermore, SeeMe is not restricted to only present a view on certain aspects such as functionality, data, organization or flows, but can also integrate these views. SeeMe is compatible with other, more formal methods, since it can mimic structures as they can be found in activity diagrams, eEPCs or in flow charts. Theoretically it can represent all structures needed for programming. However, in the practical projects it was not used for this purpose.

The additional aspect of explicitly including various forms of ambiguity and imprecision enriches the ViewPoint framework by important additional means of expression especially for the upstream software development activities where the identification and resolution of all kinds of inconsistencies and imprecision is vital.

6 Conclusions

So far it has only been possible to make incompleteness and uncertainty comprehensible with SeeMe if they are related to a single perspective. The representation of differences between varying perspectives was not systematically supported. We have presented a new way of addressing those kinds of ambiguity and imprecision which are related to varying stakeholder perspectives as they occur naturally in a distributed development process. Therefore we have employed two established approaches: SeeMe and ViewPoints. Based on the graph-based structure of SeeMe specification we have shown how the goal of joining both approaches without losing specification information during the various necessary transformation steps can be accomplished. Our contribution here has to be seen as a feasibility study of combining these two approaches and there are a number of details which have to be dealt with in order to arrive at a working tool and environment. However, we have shown that it is possible without involving great effort in terms of graph transformation or other low level tools.

In short we can say that this was a successful study which provided new insights into the use of SeeMe on the one hand and decentralized development processes using ViewPoints on the other. Thus we see the combination of both approaches using general graph differencing and graph transformation as underlying machinery as a good basis to realize new useful tools which support the specification process very early in the development process on a decentralized basis, with a good connection to traditional processes using UML-based notations and tools.

Technically there are a number of topics which can easily achieve a great deal of progress. This applies especially to the problem of comparing two SeeMe diagrams. The current procedure is still very generic and is not tuned to the special appearance and structure of the SeeMe diagrams. The SiDiff tool can be tuned to the kind of diagrams to be compared and this tuning will prove very helpful.. However, the current version already achieves good results.

In addition, a number of pre- and post-comparison transformations of the graphs exported by the SeeMe editor will also enhance the results of the overall process. Especially by hiding the geometry information as a pre-comparison measure and recovering and recalculating the geometries and layouts after the comparison of the diagrams with SiDiff will need techniques to present the comparison results in various useful ways be enabled.

The presentation of the diagram comparison is a major area where we have to find useful ways of representing the common parts and the differences between diagrams. While we have shown here that the necessary information can be provided through the transformation processes, the way the results are presented to stakeholders is not obvious. Various prototypes will be created to investigate this question.

Acknowledgements. We would like to thank Michael Striewe (University of Duisburg-Essen) who provided support in creating the diagrams in various versions and helped to conduct the study. We are also grateful to Udo Kelter and his group at the University of Siegen in providing SiDiff over the web in such a way that supported the peculiar structure of SeeMe diagrams and the fulfillment of our special requests.

References

1. Enders, B.E., Goedicke, M., Heverhagen, T., Tracht, R., Troepfner, P.: Towards an Integration of Different Specification Methods by Using the ViewPoint Framework. J. Integrated Design & Process Science 6(2), 1–23 (2002)
2. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. Intl. J. of Software Engineering and Knowledge Engineering 2(1), 31–57 (1992)
3. Finkelstein, A., Sommerville, I.: The Viewpoints FAQ. Software Engineering Journal 11, 2–4 (1996)
4. Goedicke, M., Enders-Sucrow, B., Meyer, T., Taentzer, G.: ViewPoint-oriented software development: Tool support for integrating multiple perspectives by distributed graph transformation. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 43–47. Springer, Heidelberg (2000)
5. Green, T., Benyon, D.: The skull beneath the skin: entity-relationship models of information artifacts. Int. J. Human-Computer Studies 44, 801–829 (1996)
6. Harel, D.: Statecharts: A Visual Formalism For Complex Systems. Science of Computer Programming 8, 231–274 (1987)
7. Harel, D.: On Visual Formalisms. CACM 31, 514–529 (1988)
8. Herrmann, T., Loser, K.-U.: Vagueness in models of socio-technical systems. Behaviour and Information Technology 18(5), 313–323 (1999)

9. Herrmann, T., Kunau, G., Loser, K.-U., Menold, N.: Sociotechnical Walkthrough: Designing Technology along Work Processes. In: Clement, A., Cindio, F., Oostveen, A., Schuler, D., van den Besselaar, P. (eds.) Artful Integration: Interweaving Media, Materials and Practices. Proc. 8th Participatory Design Conference, pp. 132–141. ACM, New York (2004)
10. Herrmann, Th.: SeeMee in a Nutshell, Technical Report Univ. Bochum (2006), https://web-imtm.iaw.ruhr-uni-bochum.de/pub/bscw.cgi/0/208299/30621/30621.pdf
11. Kienle, A., Herrmann, T.: Integration of Communication, Coordination and Learning Material – a Guide for the Functionality of Collaborative Learning Environments. In: Proc. 36th Annual Hawaii International Conference on System Sciences (HICSS 2003) - Track1, vol. 1, p. 33. IEEE Computer Society, Los Alamitos (2003)
12. Lee, J., Jong-Yih Kuo, J.-Y., Xue, N.-L.: A note on current approaches to extending fuzzy logic to object-oriented modeling. Intl. J. of Intelligent Systems. 16(7), 807–820 (2001)
13. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–25. Springer, Heidelberg (2007)
14. Moody, D.: Graphical Entity Relationship Models: Towards a more User understandable Representation of Data. In: Thalheim, B. (ed.) ER 1996. LNCS, vol. 1157, pp. 227–244. Springer, Heidelberg (1996)
15. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a Consistency Checking and Smart Link Generation Service. ACM Trans. On Internet Technology 2(2), 151–185 (2002)
16. Nuseibeh, B., Kramer, J., Finkelstein, A.: A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. IEEE Trans. on Software Engineering 20, 760–773 (1994)
17. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: Proc. 25th International Conference on Software Engineering (ICSE 2003), p. 676. IEEE CS Press, Los Alamitos (2003)
18. Oberquelle, H., Kupka, I., Maass, S.: A view of human-machine communication and cooperation. Intl. Journal of Man-Machine Studies 19, 309–333 (1983)
19. Piwetz, C.: Requirements Definitions for Groupware Systems – A View-Oriented Approach. PhD. Dissertation, Univ. Duisburg-Essen (2001)
20. Rational Software Corp. Unified Modelling Language. Documentation Set Version 1.0., Santa Clara, CA: Rational Software Cooperation (1997)
21. Scheer, A.-W.: Architecture of Integrated Information Systems: Foundations of Enterprise Modelling. Springer, Berlin (1992)
22. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062. Springer, Heidelberg (2004)
23. Taentzer, G., Toffetti Carughi, G.: A graph-based approach to transform XML documents. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, Springer, Heidelberg (2006)
24. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference Computation of Large Models. In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 295–304. ACM, New York (2007)
25. Yourdon, E.: Modern structured analysis. Yourdon Press, Englewood Cliffs (1989)
26. Yu, E., Mylopoulos, J.M.: Understanding "Why" in Software Process Modelling, Analysis and Design. In: Proc. 16th International Conference on Software Engineering, pp. 159–168. IEEE Computer Society Press, Los Alamitos (1994)

Towards Combining Ontologies and Model Weaving for the Evolution of Requirements Models

Allyson M. Hoss and Doris L. Carver

Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803, USA
{ahoss1,dcarver}@lsu.edu

Abstract. Software change resulting from new requirements, environmental modifications, and error detection creates numerous challenges for the maintenance of software products. While many software evolution strategies focus on code-to-modeling language analysis, few address software evolution at higher abstraction levels. Most lack the flexibility to incorporate multiple modeling languages. Not many consider the integration and reuse of domain knowledge with design knowledge. We address these challenges by combining ontologies and model weaving to assist in software evolution of abstract artifacts. Our goals are to: recover high-level artifacts such as requirements and design models defined using a variety of software modeling languages; simplify modification of those models; reuse software design and domain knowledge contained within models; and integrate those models with enhancements via a novel combination of ontological and model weaving concepts. Additional benefits to design recovery and software evolution include detecting high-level dependencies and identifying differences between evolved software and initial specifications.

Keywords: abstract artifact, design knowledge, domain knowledge, modeling languages, knowledge reuse, software evolution.

1 Introduction

Considerable research in requirements engineering focuses on new development while less research addresses integrating new technologies, such as pervasive devices and artificial intelligence software, with existing software systems. And yet, rapid changes in today's computing environments require software to evolve quickly or become obsolete and perish. When total replacement is not desirable or feasible, software must eventually evolve to incorporate new technologies. Additionally, unanticipated events often make the merging of new requirements into existing systems a priority. A simple scenario derived from the airport security software system case study presented for analysis at the 14th Monterey Workshop [1] demonstrates such needs. A new requirement banning passengers from carrying liquids on board aircrafts became a priority after an attempted terrorist plot to blow up an airplane in the United Kingdom. Now, in addition to screening carry-on luggage for dangerous items such as knives and guns, the scanning software should check for liquids existing in a

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 85–102, 2008.

variety of containers. This scenario highlights the need to address potential threats from as yet unanticipated sources and identify people and/or inanimate objects requiring secondary screening. Pervasive devices such as smart cards or biometric passports for travelers are an example of applying a new technology, biometric authentication, to address security concerns [2]. Similar smart cards can enable biometric authentication utilizing both fingerprint and iris data [3]. In addition to interacting with such pervasive devices, it will soon become paramount to integrate knowledge from several new technologies such as a machine-learning based software program, developed at the National University of Ireland at Galway, that can detect substances that might be used as explosives or illegal drugs [4, 5]. Other newly developed airport security devices include the use of sensors in conjunction with artificial intelligence based behavioral screening to detect atypical behavior indicating possible malicious intent [6]. For example, an erratically driven automobile with a weight far exceeding its normal range would warrant closer inspection. Software interaction will also be required with biometric security system devices such as cameras and other sensors detecting physical traits in addition to fingerprint and iris data [6].

Software systems interacting with such new technologies will need to process knowledge (data and rules associated with that data based on specific domain and context) in a format that facilitates reasoning to handle uncertainty, change, and interaction. This includes extracting and reusing software design and domain knowledge, manipulation of abstract artifacts, performing dependency analysis, and processing multiple modeling languages. Interaction with multiple external databases to analyze personal information is inevitable. Eventually, detection must include attempts to circumvent security measures, such as fake fingerprints. Ontological reasoning and domain knowledge incorporated into the evolved software will enable existing systems to better adapt to and reason about the new technologies with which they will interact. We elaborate on software evolution with regard to these challenges in Section 2, followed by a brief overview of ontologies in Section 3 and model weaving in Section 4. In Section 5 we present our initial research steps towards combining model weaving and ontologies as a promising solution to these challenges. Section 6 reviews related literature. Section 7 reviews the advantages of our approach and concludes with the next steps in our research.

2 Software Evolution Challenges for Changing Environments

Software evolution refers to changes a software product undergoes to meet changes in its environment and/or requirements. Modifications include adding, deleting, and/or modifying artifacts such as requirements, design, source code, and test cases. Software evolution involves reverse engineering and forward engineering. Reverse engineering has been defined as “the process of analyzing a subject system to 1) identify the system’s components and their interrelationships and 2) create representations of the system in another form or at a higher level of abstraction” [7]. Forward engineering after reverse engineering takes the re-created abstract representations of the existing system, incorporates new requirements, and produces the evolved system implementation. Software evolution techniques primarily address source code changes and include ad-hoc copy-and-modify, refactoring, visualization, generative,

and aspect-oriented approaches. One of the current challenges in software evolution is effectively evolving the more abstract artifacts of software development [8]. The software evolution approaches that do abstract above the code level are often "far too detailed to be helpful for talking about the design with someone else…working with too detailed a model is a trap" [9].

Design recovery is at the heart of reverse engineering; it "recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains" [7]. In essence, design recovery extracts knowledge. Common design abstractions recreated during design recovery include abstract syntax trees and control flow graphs. However, these abstractions do not typically incorporate domain knowledge or facilitate reasoning with both design and domain knowledge. Also, few design recovery techniques extract domain knowledge from the source or design models. The capabilities to extract such knowledge and reason with that knowledge are critical to addressing the pervasive computing need for context-awareness and ability to handle uncertainty, change, and interaction. Context-awareness involves "capturing and making sense of imprecise and sometimes conflicting data and uncertain physical worlds-different types of entities (software objects) in the environment must be able to reason about uncertainty" [10].

Design recovery includes dependency analysis. Analysis of dependencies usually occurs at the implementation level by reasoning with code level constructs such as variables, statements, and procedures; and, it often includes program dependence graphs. Architectural dependency analysis involves higher-level artifacts such as components, connections and ports, and more recently includes analysis of dependencies at the package level. Recovering designs for evolution into pervasive computing would benefit from analysis of dependencies at an abstract level dealing with high-level design and requirements constructs such as objects, behavior, relationships, goals, and constraints. Such high-level constructs facilitate the modeling of contextual and interaction dependencies. Modeling dependencies at an abstract level, such as with conceptual graphs, provides a "coherent and complete description of dependencies at the general level and explicitly delineates the characteristics of the dependency from any domain limitations" [11]. Dependency analysis may require assistance from people. There is a need for enhanced human interaction in reverse engineering techniques [12].

A common limitation in software evolution techniques today is a single language focus. Few techniques consider the plethora of modeling languages ranging from general-purpose languages to domain specific languages. It is not uncommon for software applications to incorporate several modeling languages in an effort to address interaction in the heterogeneous needs of pervasive computing. A "crucial, and largely neglected, aspect of software evolution research is the need to deal with multiple languages" [8].

Lastly, there is a lack of software design and domain knowledge reuse in software evolution. A recent description of model evolution activities includes change propagation, impact analysis, inconsistency management, model refactoring, code generation, reverse engineering, version control, and traceability management [13]. Missing from this list is the integration and reuse of software design and domain knowledge. And yet, capturing both the experience of software design engineers and domain knowledge

and then reusing such experience and knowledge would save time, effort, and cost of future development. Knowledge of a system is often buried within code and/or documentation that are not updated consistently. Due to inevitable employee turnover, designers must absorb details on partially completed systems to make incremental updates. Making software design and domain knowledge reusable would help address these problems.

This research uses three types of software design knowledge: design representation, design rationale, and design implementation. Design representation includes abstract descriptions of what a software system should do and how it should be done. Design representation includes software artifacts such as requirements, use cases, patterns, and design diagrams. Design rationale "is the explicit listing of decisions made during a design process and the reasons why those decisions were made" [14]. Design implementation includes the platform specific descriptions of the design representations (such as code and test plans). Considerable research focuses on knowledge reuse in design rationale and design implementation. While some knowledge reuse exists in design representation, such as software patterns, reuse of knowledge related to the syntactic and semantic rules governing the relationships among substructures of abstract design artifacts is very limited.

Software evolution techniques inevitably must address issues concerning unpredictable environments, change, adaptation, interaction, and context-awareness. Extracting design and domain knowledge from existing software systems, reasoning and identifying dependencies within that knowledge, and interacting with multiple modeling languages will become critical in handling such issues. Ontologies and model weaving offer encouraging potential for addressing these challenges.

3 Ontologies

Utilized for several years in philosophy, linguistics, and artificial intelligence, ontologies are now a popular knowledge representation model in a variety of software development areas such as multi-agent systems, natural language processing, information retrieval, and pervasive computing. An ontology consists of hierarchically arranged concepts, relationships among those concepts, and rules that govern those relationships. While no standard definition of ontology exists, a commonly accepted definition describes an ontology as a formal, explicit specification of a shared conceptualization [15, 16]. An ontology is, therefore, an abstract model of some area of knowledge used to share information regarding that knowledge area. It contains explicitly defined and generally understood concepts and constraints that are machine understandable.

Ontologies and metamodels are similar but not synonymous. Ontologies model real-world domains or systems and describe real-world entities while metamodels define modeling languages that in turn describe real-world domains or systems [17]. Instances of metamodels are models. Metamodels describe data structures while ontologies typically do not. Lastly, "a valid meta-model is an ontology, but not all ontologies are modeled explicitly as meta-models" [18].

Ontologies and ontology-based models offer numerous benefits to software development in pervasive computing. A recent survey [19] highlights these benefits including:

- representing and reasoning with context data;
- representing and managing privacy and trust issues;
- matching "producers and consumers of contextual information";
- facilitating design of interaction; and, modeling uncertainty.

Developers utilize automated reasoning software [20] to address validation, ambiguity, incompleteness, and infer new knowledge [21]. Several pervasive computing systems utilize ontologies such as the Context Broker Architecture (CoBrA) [22] to assist with context-aware computing. CoBrA represents pervasive computing concepts using the Standard Ontology for Ubiquitous and Pervasive Applications [23].

This research utilizes one ontology to represent design knowledge, described in Section 3.1, and a second ontology to represent domain knowledge, described in Section 3.2.

3.1 Representing Design Knowledge

The Ontology for Software Specification and Design (OSSD) Model [24] will be the basis to represent design knowledge. A partial view of this model is given in Figure 1. The OSSD Model is an ontology of software design and specification knowledge that consists of hierarchically arranged software development concepts, relationships, and rules. The Model integrates the structural and relationship knowledge acquired from multiple views of a software design and it utilizes ontological reasoning via rules associated with its properties to assist with both syntactic and semantic error detection among multiple design views and identify high-level dependency relationships among software design and requirements constructs.

The graphical notation of the OSSD Model includes rounded rectangles representing classes interconnected via solid lines implying "Is-a" relations, and dashed lines representing properties that describe additional details regarding classes and conceptually link related classes. The direction of the arrow at the end of a dashed line distinguishes a "from" class and "to" class. Class names are capitalized and written in italics. Property names are written in italics but are not capitalized. Instances of a class (not shown in the generic view of the OSSD Model in Figure 1) are indicated at the end of a double-headed arrow. The top level of the OSSD Model is a *Construct*, which is subdivided into nine subconstructs: *Object, Attribute, Behavior, Relation, State, Transition, Goal*, *Constraint,* and *Plan*. Each of these subconstructs is further subdivided. Properties within the OSSD Model depict both structural and behavior relationships between OSSD constructs and imply the "*has*" relationship unless otherwise labeled. The OSSD Model contains agent-oriented concepts of goal, belief, and intention. Beliefs portray knowledge that an agent has of its environment. They are represented in the OSSD Model via *Object, Relation, Attribute, State, Transition* and *Constraint*. Goals are the ultimate outcomes desired by an agent and are represented via *Goal*. Intentions are the goals that an agent is focusing on at a specific moment in time and are depicted via how the agent plans to work towards its selected

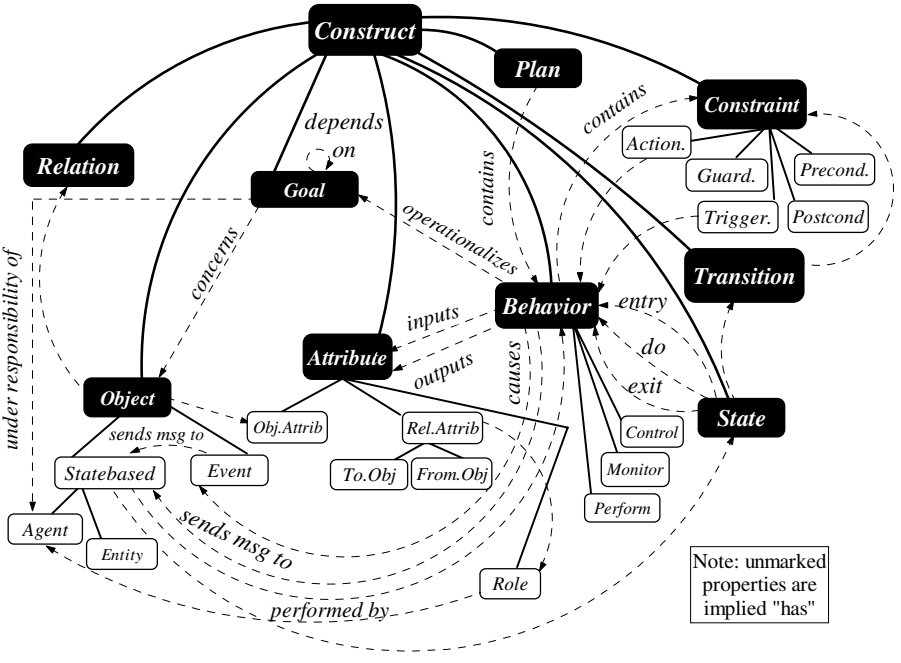


Fig. 1. Partial view of the OSSD Model

goals based on its current knowledge. Intentions are represented via *Plan*. An *Agent* is an *Object* that controls and/or monitors the behavior of other *Objects*. *Agents* interact with other *Agents*, control *Entities*, and react to *Events* based on sensory input from their environment. *Agents* execute their own thread of control, maintain their own internal state, and cannot be a subcomponent of another *Object*. *Agents* send messages to other *Objects* and receive responses from *Objects.* An *Entity* is an *Object* that does not control or monitor the behavior of other *Objects* unless those *Objects* are subcomponents of the *Entity. Entities* perform operations at the request of *Agents* and send messages to *Agents* indicating an operation has been performed. The internal state of an *Entity* can be changed as a result of receiving a message from another *Object.* An *Event* is an *Object* that has only one *State* with no significant duration of time. An *Event* can be as simple as a discrete change in an environment variable, including temporal variables, or the completion of a complex operation.

As an example, modeling the simple airport security scenario given in the Introduction using the OSSD Model, the passenger might be represented as an Agent whose Goal is to travel by flying (Behavior) via an airplane (Entity) but the Constraints on that Behavior requires identifying a thumbprint (Entity) before (Precondition) any airplane (Entity) is boarded (Behavior).

3.2 Representing Domain Knowledge

We will also utilize a second ontology, the Standard Upper Merged Ontology (SUMO) [25] as the basis for our ontological representation to store domain knowledge. “Upper

ontologies are quickly becoming a key technology for integrating heterogeneous knowledge coming from different sources" [26]. SUMO is a large formal ontology that is available to the public and is currently mapped to the complete WordNet lexicon [27]. WordNet is a lexical reference system for the English language that categorizes English words into parts of speech (noun, verb, adjective, adverb). It organizes words into sets of synonyms, referred to as synsets, gives definitions and provides semantic relations between the synsets. These relations include synonyms/antonyms, hypernyms/hyponyms (is-a relations with a broader and narrower definition), and meronyms/holonyms (similar to part/whole of the part-of or has-part relations). A partial view of the SUMO hierarchy is shown in Figure 2.

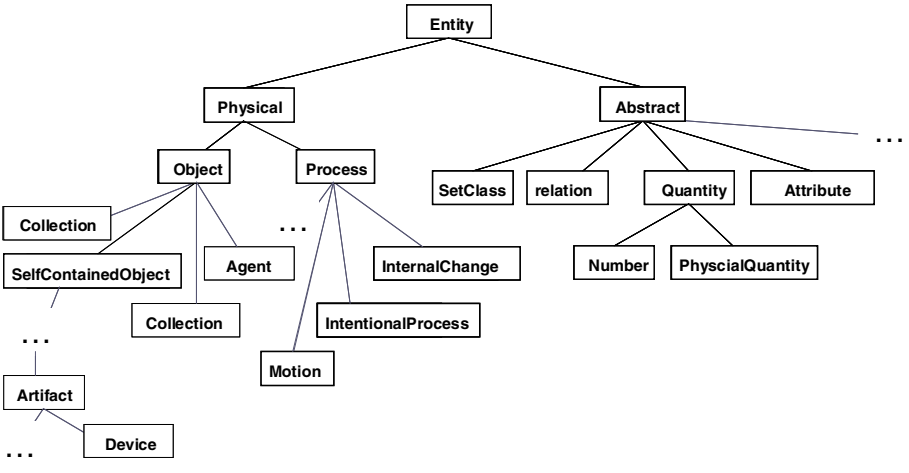


Fig. 2. Partial View of SUMO Hierarchy

Example SUMO classifications for domain knowledge obtained from the requirements of the airport security software system case study include:

Entity:Object:LivingThing:Organism:Person:Traveler:Passenger
Entity:CausalAgent:Person:Traveler:Passenger

Entity:Object:Artifact:Instrumentality:Container:Case:Luggage

Entity:Object:Artifact:Instrumentality:Device:Instrument:Weapon:Gun;

Entity:Abstract:Synset:Verb:Examine;Inspect;
Entity:Physical:Process:InternalChange:BiologicalProcess:…:Inspect;

Entity:Abstract:Relation:…Communication:…Ban
Entity:Physical:Process:IntentionalProcess:…:Communication:…Ban

These examples demonstrate SUMO representing knowledge that conveys both physical and abstract concepts.

There exist several advantages to basing our ontological representation of the domain knowledge on SUMO. Its broad range of knowledge is extended via several mid-level ontologies, such as communications and distributed computing. It can be further linked to new sub-ontologies thereby making it scalable. It is a mature ontology with extensive documentation that is "intended to be used for enabling data interoperability, information search and retrieval, automated inference, and natural language processing" [28]. Current examples of ontologies integrated with, based on, and/or derived from SUMO include:

- supply chain management ontology [29];
- ontology representing experimental design, methodology, results [30];
- context ontology for personal information management [31];
- ontology describing pervasive computing services [32]; and
- General Ontology for Linguistic Description (GOLD) [33].

4 Model Weaving

Model weaving is a form of model transformation. Stated simply, model transformation involves transforming a source model to a target model. Model transformation is at the heart of a variety of techniques including forward engineering from models to code, refinement and refactoring of models, transformation between models, and reverse engineering from code to models. The OMG's Model Driven Architecture (MDA) [34] defines guidelines for model definition and transformation. Model weaving utilizes a weaving model that has typed links containing user-defined semantics to map between model elements [35]. Figure 3 elaborates on existing model weaving

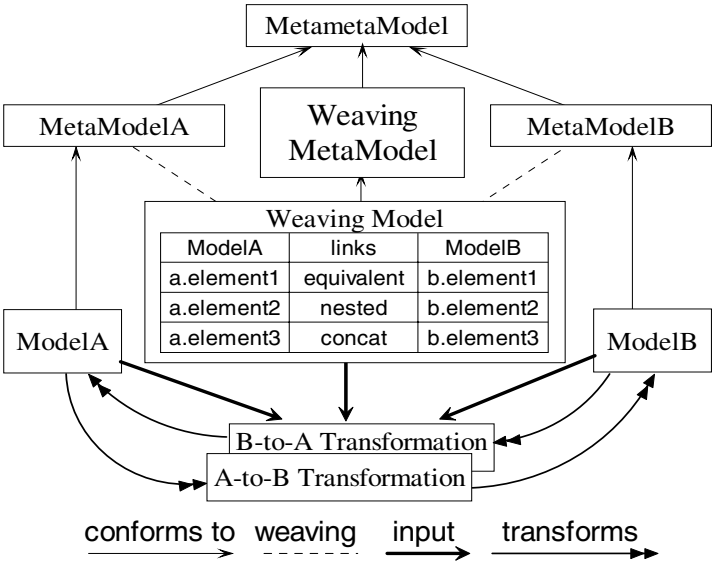


Fig. 3. Model weaving

examples [36, 37] to depict the model weaving concept. Weaving links specify the semantic relationships between source and target models above and beyond the one-to-one element matching of most model transformation approaches. The weaving model conforms to a predefined weaving metamodel that defines a variety of mapping capabilities. With model weaving, one element of a source model can be linked to a set of elements in the target model and vice versa. Complex mappings such as n:1, 1:m, and n:m are possible as well as expressions such as equality, equivalence, non-equivalence, and generality via metamodel extensions and mapping expressions [36]. The weaving model is incorporated into a transformation program that performs the actual model transformation. Creating a weaving model is a semi-automated process in which many similarities among model elements can be identified automatically, but manual refinement may be necessary.

Model weaving offers three advantages over other model transformation techniques [35]: links in a weaving model are bi-directional whereas most model transformation techniques produce unidirectional transformations; transformation patterns associated with weaving links are more reusable than the structure dependent coding patterns of most model transformation approaches; and changes to source and target metamodels propagate through weaving links with fewer modifications to the weaving metamodel than in other model transformation techniques in which source and target model changes require changes in the model transformation program.

5 Combining Model Weaving and Ontologies

Conceptually, this research weaves together concepts from artificial intelligence (ontologies representing both software design knowledge and domain knowledge), linguistics (analysis of word usage to assist knowledge extraction), and software engineering (design recovery and evolution). The top of Figure 4 shows initial requirements and design specifications of an airport security system transformed into two ontological representations, an Airport Design Ontology and an Airport Domain Ontology, via two corresponding Generic Ontologies. The Generic Design Ontology consists of properties, or rules, governing the relationships among software requirements and design constructs. We utilize the OSSD Model as basis for the Generic Design Ontology. The Generic Domain Ontology focuses on facts and rules within a given domain. We base the Generic Domain Ontology on SUMO. The DomainWM, DesignWM, and Design&DomainWM in Figure 4 represent the models we use to weave between a given metamodel and the Generic Domain Ontology, between a given metamodel and the Generic Design Ontology, and between the Generic Domain Ontology and the Generic Design Ontology respectively. Figure 4 also shows how the merging of the existing airport software system, new airport security requirements, and new knowledge from independent ontologies, such as a security ontology, into the ontological representations of the airport software system would occur. The right side of Figure 4 shows the resulting output as either the existing airport software design or the enhanced airport software design. The output format can be either in the same or different modeling language as the input format.

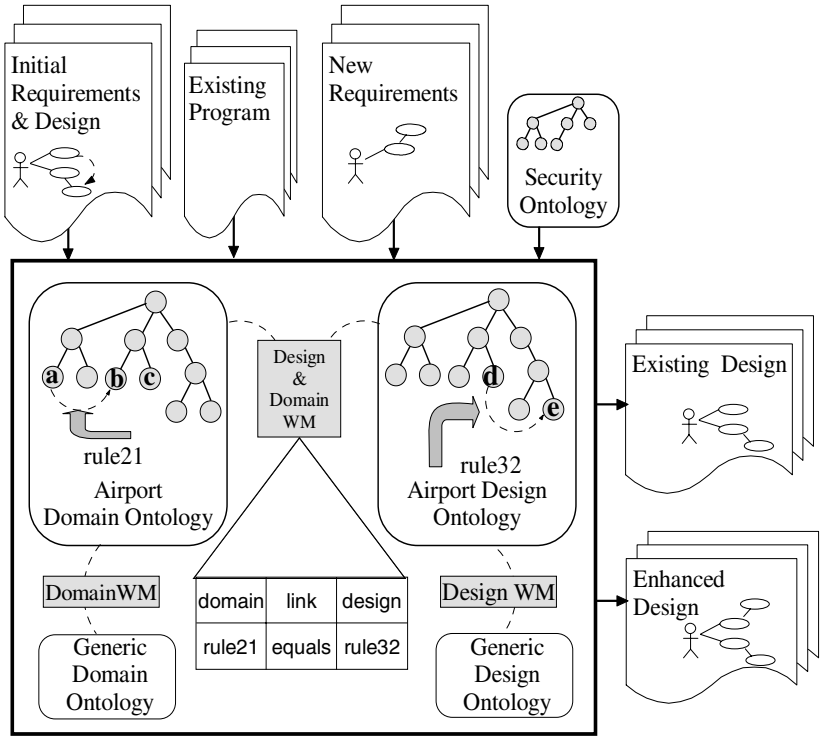


Fig. 4. Conceptual Overview

Our approach facilitates knowledge reuse by applying knowledge acquired during processing of one application to subsequent applications via our generic ontologies. We utilize the ontologies shown in Figure 4 to store initial requirements and design models that are later accessed during the evolution of those models thereby facilitating knowledge reuse and multi-language integration.

Referring to the motivating scenario presented earlier, we consider the evolution of an airport software system to incorporate new security requirements. Developers wish to identify how the existing system differs for the original design, current knowledge dependencies, and how to incorporate new requirements such as recognition of a customer's smartcard to facilitate biometric authentication.

To perform these tasks, we would perform a series of weaving transformations transforming the initial design for the airport software system into two ontological representations, one for design knowledge and one for domain knowledge as shown in Figure 4 via the Airport Design Ontology and Airport Domain Ontology respectively. Next, we would weave the current implementation, represented in Figure 4 as the "existing program", into ontological representations of the evolved Airport Design and Airport Domain Ontologies. At this point, we could utilize weaving models to output an "existing design" representation, in the same or different modeling language than the initial design. Or, we could utilize an ontology versioning program such as PROMPTDIFF [38] to assist in identifying the differences between the ontological representations of the initial design and the existing design. Alternatively, we could

weave new requirements and/or design knowledge concerning pervasive devices and authentication techniques into these ontological representations.

In Figure 4, we input new requirements, such as a rule21 that controls the relationship between domain constructs “a” and “b” which has a dependency affecting and/or producing rule32 that controls the relationship between design constructs “d” and “e”. We weave them both into the existing ontologies for the airport software system. Numerous ontology-merging programs exist, such as IPROMPT [39], to assist in merging the ontological representations of initial design and new requirements. We then utilize the weaving models to output an “enhanced design” representation in one or more modeling languages. We intend to provide interfaces to facilitate human interaction in manipulating the ontological representations, weaving models, and high-level inter-dependency rules.

We anticipate that weaving new knowledge such as thumbprint recognition into an exiting application will benefit from knowledge obtained from several ontologies developed to address a variety of domains such as pervasive computing services, network security, privacy, and access control. We will define weaving models to weave such knowledge into the Generic Domain Ontology and Generic Design Ontology and therefore make it accessible to future applications.

Figures 5a and 5b provide a more detailed but generic overview of our weaving logic. Figure 5a shows weaving of the initial design, Model A in the center of the diagram, into its ontological representation DesignOntologyA’, in the bottom right corner of the diagram. We would weave together MetaModel A and the Generic Domain Ontology to produce a weaving model, WMMMA2GenericDomain, which would become input with Model A to a transformation program that would produce the DomainOntologyA (shown in the left side of Figure 5a). Next, We would weave

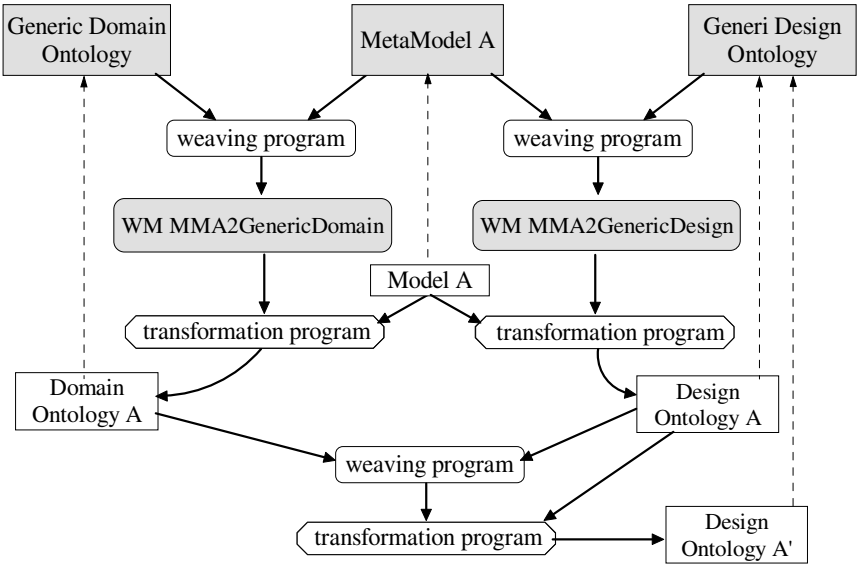


Fig. 5a. Weaving Initial Design A to Ontological Representation A’

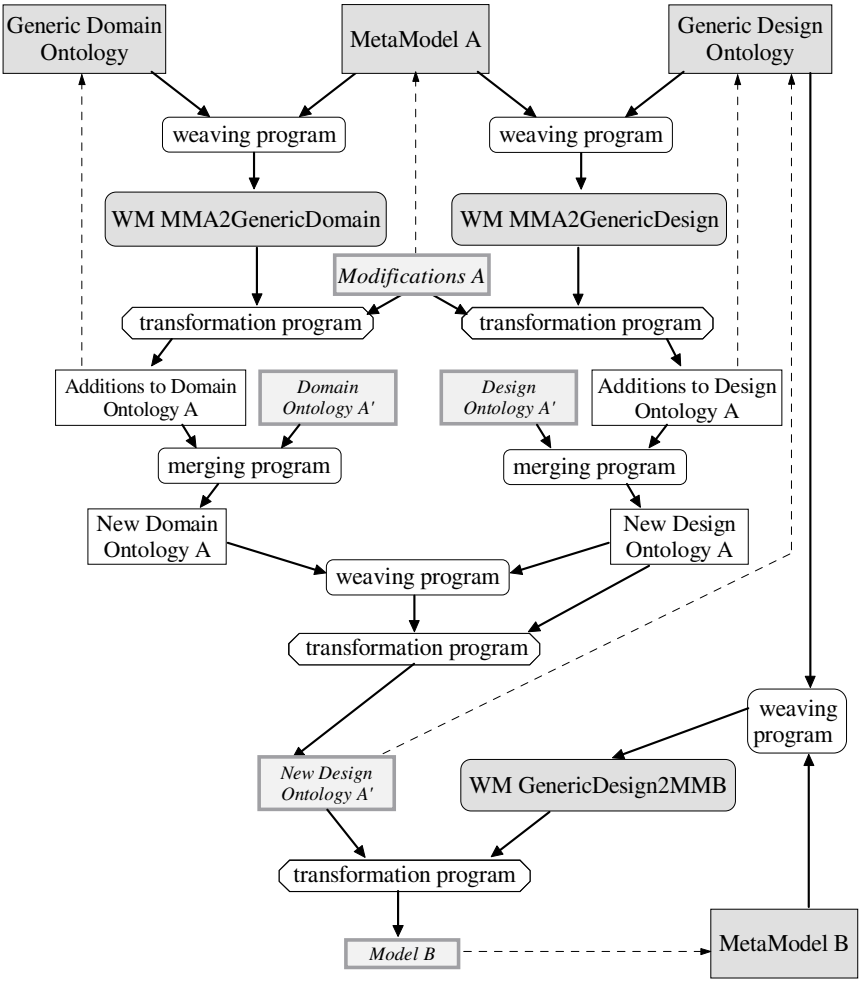


Fig. 5b. Weaving New Requirements Ontological Representation A'

together MetaModel A and the Generic Design Ontology to produce a weaving model, WMMMA2GenericDesign, which would become input with Model A to a transformation program that would produce the DesignOntologyA (shown in the right side of Figure 5a). Finally, we would weave together the DomainOntologyA and DesignOntologyA to produce the DesignOntologyA' which contains both design and domain knowledge woven together. This same process would be repeated to produce the ontological representations of the "existing program" as a second version of DesignOntologyA'. The two versions of OntologyDesignA' could be compared to identify the differences between the initial design and the existing design.

Figure 5b shows how new requirements, Modifications A shown in the center of the diagram, would be woven into the existing DesignOntologyA' and DomainOntologyA', producing the NewDesignOntologyA' located near the bottom center of the

diagram. The bottom right half of Figure 5b shows how NewDesignOntologyA' would be woven into a different modeling language representation, ModelB, based on MetaModel B.

In summary, this approach identifies differences between evolved software and initial requirements and design specifications, detects knowledge dependencies within a software design, and simplifies requirements and design modifications.

6 Related Research

Several ontological solutions are emerging to address software evolution. One approach assists the software maintenance processes by using ontologies and automated reasoning, via description logics, to represent heterogeneous software maintenance artifacts by creating separate ontologies for source code and documentation (such as requirements and design) and mapping between them providing query and reasoning capabilities [40]. This approach provides considerable analysis capabilities but does not produce evolved artifacts. Much of the related work utilizing ontologies in software maintenance focuses on representing software maintenance knowledge. For example, a software maintenance ontology [41] consisting of high-level maintenance concepts such as software system, modification processes, computer science skills, organizational structure, and application specific knowledge provides a unifying framework for software evolution tool interaction.

The most prevalent application of model weaving in software evolution involves the use of aspects. Aspect-oriented modeling weaves together design abstractions such as associations and behavior by extracting common features with the end goal of improved understanding and maintenance via separation of concerns such as security and mobility. XWeave [42] is a tool developed to support model evolution by weaving aspect models into non-aspect-oriented metamodels. As an example, XWeave weaves a FireDetection feature, represented via a FireDectectionSensor, into a metamodel of a smart home. XWeave can weave instances of the Eclipse Modeling Framework (EMF) Ecore Metametamodel [43]. The weaving process is based on matching element names or expressions in both models.

An ontology-based metamodel matching framework [17] addresses the integration of modeling languages by transforming metamodels to ontologies and then utilizing ontology matching tools to map between them. As a demonstration, metamodels based on Ecore are redefined as ontologies based on the Web Ontology Language (OWL) [44]. A weaving model is created based on a manual mapping process to derive model transformation rules that in turn automate the transformation between metamodels. The focus of the matching is with metamodel elements and not model elements.

A Problem Domain Ontology (PDO) [45] is at the heart of a methodology to extract, organize, and analyze knowledge from requirements documents for software-intensive systems written in natural language. The PDO facilitates analysis of interdependencies among problem domain concepts as well as ontological reasoning to infer knowledge concerning software assurance. These researchers extend interdependency analysis into an ontology-based information system and knowledge representation methodology [46] to identify and analyze intra-domain and cross-domain

interdependencies within critical infrastructures such as medical facilities, transportation systems, and communications systems.

Current software reuse research focuses on representing and retrieving software artifacts such as code, patterns, components, and experience. While patterns share design knowledge they do not facilitate reasoning with that knowledge nor address domain knowledge reuse. The KOntoR approach [47] provides both domain knowledge reuse and reasoning capabilities by storing software artifacts in a metadata repository and utilizing ontologies to represent both software design and domain knowledge. While the KOntoR approach also processes software artifacts specified in variety of formats, its reuse does not incorporate rule knowledge concerning the relationships among software design constructs. REBUILDER UML [48] facilitates reuse of software design knowledge utilizing ontologies and Case-Based Reasoning (CBR). This tool combines UML class diagrams with domain ontologies to provide users with a software design knowledge library of problem, solution, and outcome cases. It focuses on one software modeling language, knowledge reuse only at the object or class diagram level, and uses ontologies to represent only domain knowledge.

In contrast to the above research, our research:

- considers the evolution of both design and requirements models;
- includes the weaving of both metamodel and model constructs;
- extracts and utilizes domain knowledge buried within models;
- weaves domain and design knowledge; and, lastly,
- facilitates human interaction in the development of weaving models and therefore in the software evolution process.

7 Conclusion

We presented a method to combine ontologies with model weaving to facilitate the evolution of abstract software artifacts, such as requirements models, to meet the challenges of integration with new technologies. This research represents a part of our ongoing work to implement a system called the Evolution Weaver. Specifically, it focuses on evolving requirements and design models; reusing software design and domain knowledge; and integrating multiple software modeling languages for software evolution. It approaches these challenges by applying ontological representation and reasoning to improve the understanding and modeling of software systems; performing linguistic analysis to identify implicit knowledge of a software system embedded within software models; and utilizing model weaving concepts to facilitate incremental software development of models specified using multiple modeling languages.

Our approach extracts design and domain knowledge from source, design, and requirements models into ontological representations based on the Ontology for Software Specification and Design (OSSD) Model and the Suggested Upper Merged Ontology (SUMO). We create weaving models to integrate evolutionary development utilizing multiple software modeling languages. The ontological representations retain both design and domain knowledge between software versions, thereby facilitating knowledge reuse.

This research has the potential to provide benefits above and beyond those directly related to software evolution. It could facilitate the merging and reuse of knowledge obtained from multiple ontologies developed for a variety of domains. Its economic benefits include the potential to reduce the software evolution costs that must be incurred by organizations incorporating new technologies into existing systems. This ontological approach could facilitate scalable software development. At the core of this research, "ontologies inherently are extendable." [49]. Lastly, we intend to incorporate human interaction to guide the ontological reasoning and weaving logic and therefore facilitate human intellectual control during software evolution.

Our next steps include implementing a proof of concept, the Evolution Weaver, utilizing open source applications both as components of the software evolution processing and as test subjects for its verification. The utilization of open source technologies in this research provides benefits from both a developmental and educational standpoint. Concerning the former, the Open Source Initiative (OSI) [50] expounds upon the numerous advantages of open source development and continually strives to convince the commercial software development world via broadcasting its numerous successful examples. Open source tools and projects are also becoming recognized as beneficial in teaching software design [51]. "The use of open-source projects guarantees that the students will have an experience with a software system of realistic size and complexity" and it "prepares a student better for their future software engineering career" [52].

References

1. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
2. Maselli, J.: FAA Turns to Smart Cards to Increase Airport Security. InformationWeek (2002), http://www.informationweek.com/news/software/showArticle.jhtml?articleID=6501097
3. DeGuzman, M.-L.: Airport Thumbs Up on ID System. ComputerWorld Canada (2007), http://www.computerworldcanada-digital.com/computerworldcanada/20070302/?pg=18
4. O'Brien, C.: Irish Software to Detect Airline Threats. ElectricNews.Net Ltd (2006), http://www.electricnews.net/news.html?code=9821209
5. Ryder, A.: Analyze-IQ: Machine Learning Software. National University of Ireland, Galway (2008), http://www.nuigalway.ie/nanoscale/analyze_iq.html
6. NSF: New Technologies Could Make Airport Screening More Effective and Less Cumbersome. NSF Press Release 06-154 (2006), http://128.150.4.107/news/news_summ.jsp?cntn_id=108133&org=NSF
7. Chikofsky, E., Cross, J.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software 7, 13–17 (1990)
8. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in Software Evolution. In: 8th International Workshop on Principles of Software Evolution, CA, pp. 13–22. IEEE Computer Society, Los Alamitos (2005)
9. Berrisford, G.: Why IT Veterans are Sceptical about MDA. In: 2nd European Workshop on Model Driven Architecture, pp. 125–135, University of Kent, Canterbury, (2004)

10. Ranganathan, A., Al-Muhtadi, J., Campbell, R.: Reasoning about Uncertain Contexts in Pervasive Computing Environments. Pervasive Computing 3, 62–70 (2004)
11. Cox, L., Delugach, H.: Dependency Analysis Using Conceptual Graphs. In: 9th International Conference on Conceptual Structures, pp. 117–130, University Laval, Quebec, (2001), http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-41/Cox.pdf
12. Canfora, G., Di Penta, M.: New Frontiers of Reverse Engineering. In: 29th International Conference on Software Engineering, Washington, DC, pp. 326–341. IEEE Computer Society, Los Alamitos (2007)
13. Mens, T., Van Der Straeten, R.: On the Use of Formal Techniques to Support Model Evolution. In: 1ères Journées sur l'Ingénierie Dirigée par les Modèles, pp. 115–124. Sébastien Gérard, Jean-Marie Favre, Pierre-Alain Muller, Xavier Blanc (2005)
14. Jarczyk, A., Loeffler, P., Shipman, I.F.: Design Rationale for Software Engineering: A Survey. In: 25th Annual IEEE Computer Society Hawaii Conference on System Sciences, pp. 577–586. IEEE, Los Alamitos (1992)
15. Gruber, T.: A Translation Approach to Portable Ontology Specifications. In: Knowledge Acquisition, vol. 5, pp. 199–220. Academic Press, London (1993)
16. Borst, W.: Construction of Engineering Ontologies. Ph.D. Dissertation, University of Twente, Enschede (1997)
17. Kappel, G., Kargl, H., Kramler, G., Schauerhuber, A., Seidl, M., Strommer, M., Wimmer, M.: Matching Metamodels with Semantic Systems – An Experience Report. In: Workshop Model Management und Metadaten-Verwaltung, pp. 38–52. Verlag Mainz (2007)
18. Woody, P.: What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?, Metamodel.com (2003), http://www.metamodel.com/article.php?story=20030115211223271
19. Ye, J., Coyle, L., Dobson, S., Nixon, P.: Ontology-based models in pervasive computing systems. In: The Knowledge Engineering Review, vol. 22, pp. 315–347. Cambridge University Press, Cambridge (2007)
20. Lutz, C., Baader, F., Franconi, E., Lembo, D., Möller, R., Rosati, R., Sattler, U., Suntisrivaraporn, B., Tessaris, S.: Reasoning Support for Ontology Design. In: Coence Grau, B., Hitzler, P., Shankey, C., Wallace, E. (eds.) 2nd International Workshop OWL: Experiences and Directions (2006)
21. Gaitanou, P.: Ontology Semantics and Applications. In: 2nd International Conference on Metadata and Semantics Research. MTSR Organizing Committee, Corfu (2007)
22. Chen, H., Finin, T., Joshi, A.: An Intelligent Broker for Context-Aware Systems. In: Ubicomp 2003, pp. 183–194,Ubicomp, (2003)
23. Chen, H., Perich, F., Finin, T., Joshi, A.: SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In: International Conference on Mobile and Ubiquitous Systems: Networking and Services, pp. 258–267. IEEE Computer Society, Los Alamitos (2004)
24. Hoss, A., Carver, D.: Ontological Approach to Improving Design Quality. In: IEEE Aerospace Conference. IEEE, Los Alamitos (2006)
25. Niles, I., Pease, A.: Toward a standard upper ontology. In: 2nd International Conference on Formal Ontology in Information Systems. ACM Press, New York (2001)
26. Mascardi, V., Cordì, V., Rosso, P.: Comparison of Upper Ontologies. In: Baldoni, M., Boccalatte, A., De Paoli, F., Martelli, M., Mascardi, V. (eds.) Conf. on Agenti e industria: Applicazioni tecnologiche degli agenti software, pp. 55–64 (2007)
27. Miller, G.: WordNet: A Lexical Database for English. Communications of the ACM 38, 39–41 (1995)

28. Semy, S., Pulvermacher, M., Obrst, L.: Toward the Use of an Upper Ontology for U.S. Government and U.S. Military Domains: An Evaluation, MITR Corporation (2004), http://www.mitre.org/work/tech_papers/tech_papers_05/04_1175/04_1175.pdf
29. Haller, A., Gontarczyk, J., Kotinurmi, P.: Towards a complete SCM Ontology – The Case of ontologising RosettaNet. In: 23rd Annual ACM Symposium on Applied Computing, pp. 1467–1473. ACM, New York (2008)
30. Soldatova, L., King, R.: An Ontology of Scientific Experiments. Journal of the Royal Society Interface 3, 795–803 (2006)
31. Latif, K., Tjoa, A.: Combining Context Ontology and Landmarks for Personal Information Management. In: IEEE International Conference on Computing & Informatics. IEEE, Los Alamitos (2006)
32. Weeds, J., Keller, B., Weir, D., Wakeman, I., Rimmer, J., Owen, T.: Natural Language Expression of User Policies in Pervasive Computing Environments. In: OntoLex 2004, LREC Workshop on Ontologies and Lexical Resources in Distributed Environments. ACM, New York (2004)
33. Farrar, S., Langendoen, T.: A Linguistic Ontology for the Semantic Web. GLOT International 7, 97–100 (2003)
34. Object Management Group: Model Driven Architecture, V1.0.1, OMG (2003), http://www.omg.org/docs/omg/03-06-01.pdf
35. Del Fabro, M., Jouault, F.: Model Transformation and Weaving in the AMMA Platform. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 71–77. Springer, Heidelberg (2006)
36. Del Fabro, M., Bezivin, J., Valduriez, P.: Weaving Models with the Eclipse AMW Plugin. Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen (2006), http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium2_WeavingModels.pdf
37. Smolik, P.: MAMBO Metamodeling Environment. Ph.D. dissertation. Brno University of Technology, Brno (2006), http://www.mambomde.info/MamboMDE.pdf
38. Noy, N.F., Kunnatur, S., Klein, M., Musen, M.A.: Tracking changes during ontology evolution. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 259–273. Springer, Heidelberg (2004)
39. Noy, N.: Ontology Management with the Prompt Pplugin. In: 7th International Protégé Conference. Stanford Center for Biomedical Informatics Research, CA (2004), http://protege.stanford.edu/conference/2004/abstracts/Noy.pdf
40. Witte, R., Zhang, Y., Rilling, J.: Empowering software maintainers with semantic web technologies. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 37–52. Springer, Heidelberg (2007)
41. Anquetil, N., de Oliveira, K., Dias, M.: Software Maintenance Ontology. In: Ontologies for Software Engineering and Software Technology, pp. 153–173. Springer, Heidelberg (2006)
42. Groher, I., Voelter, M.: XWeave: Models and Aspects in Concert. In: 10th International Workshop on Aspect-oriented Modeling, pp. 35–40. ACM Press, New York (2007)
43. The Eclipse Foundation: Eclipse Modeling Framework (2008), http://www.eclipse.org/modeling/emf/
44. W3C: Web Ontology Language, OWL (2004), http://www.w3.org/2004/OWL/
45. Lee, S.-W., Muthurajan, D., Gandhi, R., Yavagal, D., Ahn, G.-J.: Building Decision Support Problem Domain Ontology from Natural Language Requirements for Software Assurance. International Journal of Software Engineering and Knowledge Engineering 16, 851–884 (2006)

46. McNally, R.K., Lee, S.-W., Yavagal, D., Xiang, W.-N.: Learning the critical infrastructure interdependencies through an ontology-based information system. Environment and Planning B: Planning and Design 34, 1103–1124 (2007)
47. Happel, H., Korthaus, A., Seedorf, S., Tomczyk, P.: KOntoR: An Ontology-enabled Approach to Software Reuse. In: 18th International Conference on Software Engineering and Knowledge Engineering, pp. 329–344, Knowledge Systems Institute, IL, (2006)
48. Gomes, P., Leitão, A.P.: A tool for management and reuse of software design knowledge. In: Staab, S., Svátek, V. (eds.) EKAW 2006. LNCS, vol. 4248, pp. 381–388. Springer, Heidelberg (2006)
49. de Bruijn, J.: Using Ontologies: Enabling Knowledge Sharing and Reuse on the Semantic Web. Technical Report DERI-2003-10-29, DERI – Digital Enterprise Research Institute (2003)
50. Open Source Initiative (OSI): Creative Commons Attribution 2.5 (2008), http://www.opensource.org/
51. Fuhrman, C.: Exploiting Open-source Projects to Study Software Design. Informatics in Education 6, 53–66 (2007)
52. Buchta, J., Petrenko, M., Poshyvanyk, D., Vaclav, R.: Teaching Evolution of Open-Source Projects in Software Engineering Courses. In: 22nd IEEE International Conference on Software Maintenance 2006, pp. 136–144. IEEE Computer Society, Los Alamitos (2006)

Reducing Ambiguities in Requirements Specifications Via Automatically Created Object-Oriented Models

Daniel Popescu[1], Spencer Rugaber[2], Nenad Medvidovic[1], and Daniel M. Berry[3]

[1] Computer Science Department, University of Southern California, Los Angeles, CA, USA
{dpopescu,neno}@usc.edu
[2] College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
spencer@cc.gatech.edu
[3] Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
dberry@uwaterloo.ca

Abstract. In industry, reviews and inspections are the primary methods to identify ambiguities, inconsistencies, and under specifications in natural language (NL) software requirements specifications (SRSs). However, humans have difficulties identifying ambiguities and tend to overlook inconsistencies in a large NL SRS. This paper presents a three-step, semi-automatic method, supported by a prototype tool, for identifying inconsistencies and ambiguities in NL SRSs. The method combines the strengths of automation and human reasoning to overcome difficulties with reviews and inspections. First, the tool parses a NL SRS according to a constraining grammar. Second, from relationships exposed in the parse, the tool creates the classes, methods, variables, and associations of an object-oriented analysis model of the specified system. Third, the model is diagrammed so that a human reviewer can use the model to detect ambiguities and inconsistencies. Since a human finds the problems, the tool has to have neither perfect recall nor perfect precision. The effectiveness of the approach is demonstrated by applying it and the tool to a widely published example NL SRS. A separate study evaluates the tool's domain-specific term detection.

1 Introduction

The typical industrial software specifier writes software requirements specifications (SRSs) in a natural language (NL). Even if a final SRS is written in a formal language, its first draft is usually written in a NL. A NL SRS enhances the communication between all the stakeholders. However, on the downside, often a NL SRS is imprecise and ambiguous [3].

Many an organization follows a three-step review process to assess the quality of a NL SRS and to identify ambiguities and other defects in the NL SRS [35]. First, assigned reviewers try to find defects in the document. Second, in a meeting of the reviewers, all found defects are collected and rated according to their severities. Third, the reviewed NL SRS and the collected defects are sent back to the authors for corrections.

In this process, the quality of a review of a document is dependent mainly upon how effectively each human reviewer is able to find ambiguities and other defects in the document. However, a human reviewer and even a group of them have difficulties

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 103–124, 2008.

identifying all ambiguities, even when using aids such as checklists. A human reviewer might overlook some defects while reading a SRS, because he might assume that the first interpretation of the document that came to his mind is the intended interpretation, unaware of other possible understandings. In other words, he unconsciously disambiguates an ambiguous document [10].

When a NL SRS is large, some ambiguities might remain undetected, because ambiguities caused by interaction of distant parts of the NL SRS are difficult to detect. In any one review sessions, only an excerpt of the NL SRS can be reviewed. Any ambiguity in the reviewed excerpt that is caused by interaction with a part of the NL SRS outside the excerpt may not be detectable. Moreover, when a NL SRS is large, lack of time may prevent some parts of the NL SRS from ever being reviewed. Having a faster method for conducting reviews would permit larger chunks of the whole NL SRS to be reviewed at once. It would permit also faster reviews so that more reviews and, thus, greater coverage of the NL SRS would be possible in any duration.

A controlled, grammar-constrained NL [9, 8] helps to reduce ambiguities by constraining what can be said in the NL. One possible grammar rule eliminates passive voice and, therefore, ensures that the doer of an action is always known. However, a controlled NL can address only syntactic ambiguity. Semantic ambiguity is beyond its control.

Because a semantic model omits unnecessary details, it helps to reduce complexity, and it helps the user to visualize important aspects. Therefore, with the help of a semantic model, a human reviewer can validate larger system descriptions than otherwise. Moreover, the human reviewer can focus on conceptual correctness and does not need to worry about the consistent use of concepts and correct grammar usage. Therefore, if a semantic model can be created of the system specified by a NL SRS, a thorough review of the NL SRS becomes easier.

Of course, constructing a model bears the risks of introducing new defects and of the model's not representing the original NL SRS. Researchers have tried to mitigate these risks by using automatic approaches and NL processing (NLP) [32, 25, 12] techniques, ranging in complexity from simple lexical processing [e.g., 7, 41, 40], through syntactic processing [e.g., 26, 34], all the way through language understanding [e.g., 11]. A software tool can scan, search, browse, and tag huge text documents much faster than a human analyst can. Furthermore, a tool works rigorously, decreasing the risk of overlooked defects and unconscious disambiguation. Each of some of the approaches tries to build a model from a NL source and may even try to reason about the content, possibly with assistance from the human user.

However, no automatic NLP tool is perfect. For any NLP tool T, there are utterances in its NL for which T's underlying formalism is not powerful enough to process the utterances correctly. Generally, the more complex the processing in T, the greater the chances for not processing an utterance correctly. The measures of correct processing are *recall* and *precision*, which are defined in Section 4.2. For now, recall is a measure of how much of what T should find it does find, and precision is a measure of how much of what T does find it should find.

Certainly, the recall and precision of T's parser bound T's quality. In addition, the recall and precision of T's domain-specific term (DST) identification bound T's quality.

The typical domain has its own terms, abbreviations, and vocabulary. Therefore, a tool must detect these to create a correct model. Each fully automated tool has difficulty to create a correct model because it relies on a semantic network that is based on a predefined domain dictionary. For many a domain, a domain dictionary does not exist. A semi-automated tool requires its human users to build a domain dictionary while writing the NL SRS. Clearly, not every requirements engineering (RE) process is so mature that in it, a domain dictionary is built. Even in a mature process, some domain terms might be forgotten, because the analysts assume that these terms are widely understood and recognized.

We have created an approach for helping a specification writer or reviewer identify ambiguities in a NL SRS, in which the approach tries to address all of the above mentioned problems. Hereinafter, the new approach is called *our approach* to distinguish it from other approaches. For our approach, we have built a prototype dowsing[1] tool, called *Dowser*. Dowser is based on one controlled NL. It can create from any NL SRS an object-oriented (OO) diagram, which can then be assessed by human reviewers.

In the first step, Dowser parses a NL SRS and extracts the classes, methods, and associations of a textual class model from the NL SRS. In the second step, Dowser diagrams the constructed textual class model. In the third step, a human reviewer can check the generated diagram for signs of defects in the NL SRS. Dowser and our approach are based on NL processing (NLP) and not on NL understanding. Dowser cannot judge whether or not the produced model describes a good set of requirements or classes. This judgement requires understanding. Therefore, in our approach, a human is the final arbiter of ambiguity. Because the human is in the loop, Dowser has to have neither perfect recall nor perfect precision.

To evaluate the effectiveness of our approach, we have implemented a prototype of Dowser and have tested it on a widely used example SRS, describing an elevator system [15]. The case study demonstrates the effectiveness of the approach. Since identifying domain terminology is required for any successful NLP-based approach, we conducted separate studies to evaluate Dowser's DST detection. The studies show that our approach is capable of achieving high recall and precision when detecting DSTs in a UNIX manual page.

Section 2 discusses related work. Section 3 describes out approach and all of its components. Section 4 describes validating case studies, and Section 5 concludes the paper by discussing the results and future work.

2 Related Work

Related work can be divided into four parts: (1) NLP on SRSs, (2) controlled languages, (3) automatic OO analysis model (OOAM) extraction, and (4) domain-specific term (DST) extraction.

NLP on SRSs: Kof describes a case study of the application of NLP to extract and classify terms and then to build a domain ontology [20]. This work is the most similar

[1] A dowser is a tool that makes use of domain knowledge in understanding software artifacts [5].

to our approach. The built domain ontology consists of nouns and verbs, which constitute the domain's concepts. In the end, the domain ontology helps to detect weaknesses in the requirements specification. Gervasi and Nuseibeh describe their experiences using lightweight formal methods for the partial validation of NL SRSs [12]. They check properties of models obtained by shallow parsing of natural language requirements. Furthermore, they demonstrate scalability of their approach with a NASA SRS.

Controlled languages: Fuchs and Schwitter developed Attempto Controlled English (ACE) [9], a sublanguage of English whose utterances can be unambiguously translated into first-order logic. Over the years, ACE has evolved into a mature controlled language, which is used mainly for reasoning about SRSs [8]. Juristo et al. developed other controlled languages, SUL and DUL [17]. For these languages, they defined a correspondence between linguistic patterns and conceptual patterns. After a SRS has been written in SUL and DUL, an OOAM can be created using the correspondence.

OOAM Extraction: Several tools exist that automatically transform a SRS into an OOAM. Mich's NL-OOPS [26] tool first transforms a parsed SRS into a semantic network. Afterwards, the tool derives an OOAM from the semantic network. Delisle, Barker, and Biskri implemented a tool that uses only syntactic extraction rules [6]. Harmain and Gaizauskas implemented another syntax-based tool, and they introduce a method to evaluate the performance of any such tool [15]. As in our approach, Nanduri and Rugaber [29] have used the same parser and had the same initial idea of using syntactic knowledge to transform a NL SRS into an OOAM. The objective of their approach was to validate a manually constructed OOAM. Our approach's main objective is to identify ambiguity, inconsistency, and underspecification in a NL SRS. The more restricted objectives of our approach enables a more detailed discussion of the problem space and contributes (1) a constraining grammar, (2) analysis interpretation guidelines, (3) additional transformation rules, and (4) DST extraction.

DST Extraction: Mollà et al. developed a method for answering questions in any specified technical domain. This work recognizes the importance of dealing with specified technical terminologies in NLP tools that are applied to SRSs [28, 11].

3 Our Approach

The goal of our approach is to reduce the number of ambiguities, inconsistencies, and underspecifications in a NL SRS through automation. Assuming that automation will not be perfect, i.e., it will have less than 100% recall and less than 100% precision, we let a human make the final decision about a potential ambiguity, inconsistency, or underspecification.

While reading through a NL SRS, an engineer usually builds a mental model of the described system and reasons about the correct relations of the SRS's concepts. If an engineer could only analyze the correctness of a model, instead of having also to create it, write it down, and then analyze it, he could use his skills and time more effectively.

Considering that a reviewer could more effectively inspect a model than the complete NL SRS, we developed an automatic approach based on the following observations:

- Each of most software companies uses a NL SRS [24] to describe a software system regardless of its domain.
- An OOAM is able to show the most important concepts and relations among the concepts of a system to build.
- Many an OO design method suggests building an OOAM of a sentence by identifying the parts of the sentence and creating a class from the subject, attributes from the adjectives, and methods and associations from the verb [1, 30].

Consider the example transformation of the sentence[2] The audio player shall play the music list. into an OOAM; audio player is the subject of the sentence, play is the verb, and music list is the direct object. This sentence could therefore be modeled as the diagram:

In this example, the adjectives audio and music are not broken out, because each is part of a DST.

Using this syntax-based method, the typical functional requirement sentence of a NL SRS can be transformed into an OOAM. Since this heuristic suggests using mostly syntactic information, the transformation can be automated. A NL parser can create parse trees of any NL text [34]. The OO design literature gives many rules and heuristics to transform many a syntactic construct into a textual OOAM. Off-the-shelf software exists to diagram textual OOAMs [36].

Since NL SRSs are written for a wide range of domains such as medical, technical or judicial domains, a successful approach must be robust in identifying DSTs. Our approach addresses this need by using syntactic information and a robust parser with guessing capability.

The overall quality of any NL SRS can be improved by enforcing the use of a constraining grammar. A constraining grammar reduces the possibilities of ambiguities by constraining the allowed language constructs. At the same time, it increases the quality of parsing, reduces the number of parses, and results in more precise OOAMs.

Therefore, by using and extending existing technology, we can create a tool that automatically transforms a NL SRS into an OOAM that helps a human being to identify ambiguities, inconsistencies, and under specifications in the NL SRS.

[2] A sans serif typeface is used for example text, except in a parse tree. Beware of punctuation, also typeset in the sans serif typeface, at the end of any example. It should not be considered as punctuation in the containing sentence, which is typeset in the serifed typeface. Sometimes, two consecutive punctuation symbols appear; the first, typeset in the sans serif typeface ends an example, and the second, typeset in the serifed typeface is part of the sentence containing the example. A typewriter typeface is used for example text in any parse tree, in which monospacing is essential for the correct display of the tree.

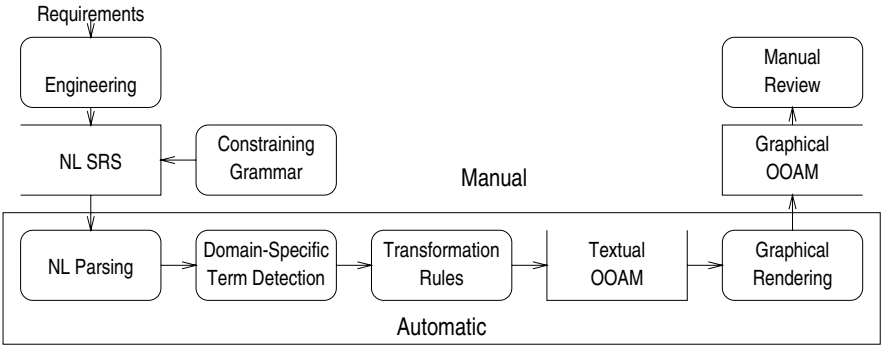


Fig. 1. Flow of the Approach

Figure 1 shows the flow of our approach. First, Dowser parses a NL SRS according to a constraining grammar. Second, from relationships exposed in the parse, Dowser creates the classes, methods, variables, and associations of an OOAM of the specified system. Third, the OOAM is diagrammed so that a human reviewer can use the model to detect ambiguities, inconsistencies, and underspecifications.

One issue that arises from basing the model building, a semantic process, on a parser, a syntactic process, is "Why not have its parser report also syntactice ambiguity?", especially since whatever parser is used probably can report all parses it finds for any sentence. In the end, an integrated tool might very well report both syntactic and semantic ambiguities. However, given the wealth of work on syntactic ambiguity [e.g., 4, 7, 18, 25, 41] and the dearth of work on semantic ambiguity, the focus of this work is on semantic ambiguity. If and when a completely satisfactory approach to identifying semantic ambiguity is found, if it is based on a parser, then certainly that same parser can be used as the basis for identifying syntactic ambiguity in a single integrated ambiguity identification tool.

3.1 Constraining Grammar

Any NL allows expressing the same concept in different ways using different syntactic structures. For example, a sentence in active voice can be translated into passive voice without changing the semantics or pragmatics. However, passive voice can encourage ambiguities. For example, the sentence The illumination of the button is activated. leaves room for different interpretations, because it is not clear who holds the responsibility for activating the illumination. Alternatively, the sentence could be describing a state. As a consequence, a constraining grammar can be introduced to decrease the possibility of ambiguity. A constraining grammar enables formal reasoning without the disadvantages of a fully formal language [8]. A constraining grammar has the other advantage that it is more amenable to parsing, and extraction rules based on it can be created more easily.

Observe that the constraining grammar used may bear no relationship with the internal grammar of the parser used other than sharing the same NL. The goals of the two grammars are different. A constraining grammar tries to reduce the number of ways

to say something and to be uniguous[3]. The goal of a NL parser's grammar is to be as general as possible and to recognize any legitimate sentence in the NL. That is, a NL parser's grammar is designed to be as ambiguous as is the NL itself.

Our approach uses a constraining grammar that is derived from Juristo et al.'s grammar [17]. They have developed two context-free grammars and an unambiguous mapping from these grammars to OOAMs. This mapping is explicitly defined and allows better model creation than with commonly used heuristics that are justified only intuitively. Moreover, the explicit definition enables automation.

Using a constraining grammar influences the style of a NL SRS, because a constraint grammar enforces simple sentences. The typical sentence has a basic structure consisting of subject, verb and object. Furthermore, only simple subclause constructions are allowed, such as conditional clauses, using if, when, velc.[4] Therefore, a NL SRS will contain many short sentences if it is written according to the developed controlled grammar. Shorter, simpler sentences tend to be less ambiguous, because at the very least, they avoid some coordination and scope ambiguities.

3.2 Natural Language Parsing

Since an OOAM is created automatically from syntactic information, we needed a parser to extract this information from the NL SRS. The parser we used was developed by Sleator and Temperley (S&T) at Carnegie-Mellon University [34]. Sutcliffe and McElligott showed that the S&T parser is robust and accurate for parsing software manuals [38]. Since software manuals are similar to SRSs [2], the S&T parser looked promising for our approach. Additionally, the S&T parser was chosen because it is able to guess the grammatical role of unknown words. This capability is used for the DST detection, which is described in Section 3.4. However, in principle, any other parser e.g., The Stanford Parser [19] could be used. Dowser would have to be adjusted to work with the parser's output.

The parser is based on the theory of link grammars, which define easy-to-understand rule-based grammar systems. A link grammar consists of a set of words, i.e., the terminal symbols of the grammar, each of which has one or more linking requirements. A sequence of words is a sentence of the language defined by the grammar if there exists a way to assign to the words some links that satisfy the following three conditions:

1. Planarity: the links do not cross;
2. Connectivity: the links suffice to connect all the words of the sequence together; and
3. Satisfaction: the links satisfy the linking requirements of each word in the sequence.

The link grammar parser produces the links for every such sentence. After parsing, the links can be accessed through the API of the link grammar parser.

Each established link in a sentence has a link type, which defines the grammatical usage of the word at the source of the link. The sentence The elevator illuminates the button. shows three different link types:

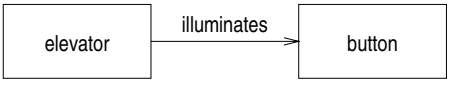
[3] The coined term "uniguous" means "not ambiguous".

[4] "velc." means "or others" and is to "vel cetera" as "etc." is to "et cetera".

```
     +-------Os------+
+---Ds--+-----Ss----+         +--Ds--+
|       |           |         |      |
the elevator.n illuminates.v the button.n .
```

A D link connects a determiner to a noun, an S link connects a subject noun to its finite verb, and an O link connects a transitive verb to its object. A small s after the type of a link indicates that the target of the link is a singular noun.

From this example sentence, the first extraction rule can be derived. If a sentence contains an S link and an O link, then create a class from the subject noun and one from the object noun. Afterwards, create a directed association from the subject class to the object class, which is named by the verb:



A directed association was chosen over a simple class method, because a directed association shows also who invokes the action. If the action were modeled as a class method, the information about who causes the action would have been lost. Using this rule, Dowser would extract the classes elevator and button and a directed association illuminates from the elevator class to the button class.

To avoid having two different classes created for elevator and elevators, our approach incorporates the lexical reference system *WordNet* [27] to find the stems of nouns and verbs. Therefore, the name of each created class is the stem of a noun.

One might think that the use of a constrained language would reduce the number of parses and might even ensure that there is only one per sentence. However, the language constraints only encourage and do not guarantee uniguity. In general, the link parser returns multiple parses for any input sentence. For example, When an elevator has not to service any requests, the elevator remains at its final destination and the doors of the elevator are closed. returns 16 parses. However, as the link grammar homepage [39] says, "If there is more than one satisfactory linkage, the parser orders them according to certain simple heuristics." In our experience, these heuristics selected as first our preferred parse. Of courser, these simple heuristics cannot be expected to always select the parse that the writer intended, because syntactic ambiguity is a hard problem. Even humans have to rely on semantic understanding and context to resolve difficult cases of syntactic ambiguity, e.g., the classic The boy saw the man with the telescope..

3.3 Transformation Rules

Transformation rules bridge the gap between the extracted syntactic sentence information and the targeted OOAM. Each transformation rule describes how a combination of words of identified grammatical roles can be transformed into classes, associations, attributes, and methods.

The transformation rules Dowser uses were derived from Juristo *et al.*'s grammar [17], the OO methods literature [29, 30], and conducted experiments. In total, Dowser uses 13 transformation rules. The five most frequently used are the following:

1. The most frequently applicable rule is the first extraction rule, described in Section 3.2. If a parsed sentence contains a subject and an object link, then create two classes with a directed association named after the verb.
2. Aggregations are an important notion in UML class diagrams. One rule for extracting aggregations is similar to the first rule. The major difference is the verb. This rule is applicable only if the parsed sentence contains a subject and an object link and the verb stem is one of have, possess, contain, or include. In that case, the object is aggregated to the subject.
3. Sometimes, a subclause describes a system action without the need of an object, particularly, if the system reacts to a given event, e.g., If the user presses the button, the elevator moves.. An event clause starts with an if or when. If Dowser detects an event clause, and the main clause has only a subject link, then a class from the subject link noun is created and the verb is added to the new class as a method.
4. A genitive attribute indicates two classes with an aggregation, e.g., The system stores the name of the customer. or The system stores the customer's name.. If Dowser detects a genitive, it creates two classes with one linking aggregation. For either example, Dowser would create a class customer, and it would aggregate the class name to the class customer; name could have been modeled as an attribute. However, other sentences in the specification would add methods to the class name later. Therefore, with the syntactic information of only one sentence, it cannot be decided if name is an attribute or an aggregated class. This rule needs to be constrained by semantic information. For example, Dowser should not apply this rule to the sentence The user enters the amount of money.. Although amount is a noun, the class amount is not desired in this case.
5. Although active clauses are preferred in NL SRSs, passive clauses are still needed. They are used to describe relations and states, e.g. as in A husband is married to his wife.. From this sentence, two classes are created, from the subject noun and the noun of the prepositional phrase. The passive verb and the connecting word to link the prepositional phrase described with the association.

Dowser applies two post-processing rules after it executes all possible transformation rules.

The first post-processing rule converts all classes that are aggregated to another class into attributes of that other class. Only a class that lacks any attribute, method, or incoming or outgoing association is transformed. For example, one rule described above extracts two classes and one aggregation from the sentence The system stores the name of the customer.. The rule creates a class name and a class customer. However, the class name has probably no method or association. Therefore, if a class contains no method after all rules have been applied, it is transformed into an attribute of the class customer.

The second post-processing rule removes the class system from the OOAM, since all other classes together form the system; system is not a class, because it cannot be a subpart of itself.

The full set of rules, a user's manual for Dowser, and other details may be found at the Web site, `http://www.cc.gatech.edu/projects/dowser/`.

3.4 Domain-Specific Terms

In order to build the correct classes into the OOAM of a NL SRS, our approach has to be able to detect DSTs. If Dowser were to extract only the concept button from elevator button, Dowser would be identifying an incorrect term.

To achieve high DST recall, the parser could access a special domain data dictionary. However, for each of most domains, such a data dictionary does not exist. Software is built for financial, medical, technical, and other domains. Creating a domain dictionary for each of these rich domains is complex and difficult. Additionally, the customer of a software application might prefer to use her own terms for describing her product. A product could have arbitrarily chosen names, such as DNAComp07. Therefore, even if a domain dictionary exists for a NL SRS, DST detection remains a challenge.

The link types of the link grammar parse can be used to identify DSTs. The typical DST happens to be built from an attributive noun or a proper noun. Therefore, in a link grammar parse, the `AN` link, denoting an attributive noun, and the `G` link, denoting a proper noun, help to identify DSTs.

Consider the link grammar parse of the sentence The RS7 business plan is due in May.:

```
+---------Ds---------+
|    +-------AN-------+
|    |        +---AN---+--Ss-+-Pa-+-MVp+-IN+
|    |        |        |     |    |    |   |
The RS7 business.n plan.n is.v due.a in May
```

Thus, RS7 business plan is a domain-specific term.

A parser typically has problems parsing words that are not in its internal dictionary. However, the S&T link grammar parser has a guessing mode, in which it can guess the syntactic role of an unknown term. Therefore, it is often able to guess the syntactic role of an unknown term, improving its DST recall.

Since DST detection is essential for transforming a NL SRS into an OOAM, we conducted a study, described in Section 4.2, about the recall and precision of our approach.

3.5 Diagramming OOAMs

The previous steps are able to create a textual OOAM. However, it is easier for a human to understand a graphical OOAM than a textual OOAM. Using the approach described by Spinellis [36], an extracted textual OOAM is diagrammed. The tool *UMLGraph* [37] transforms a textual description into a *dot file*, which the tool *Graphviz* [13] can transform into any of some popular graphic formats, such as JPEG, GIF, or PNG.

3.6 Interpretation of OOAM

In the last step of our approach, a human analyst checks the created diagram for ambiguities.

Some ideas that a human analyst can use to find defects in an OOAM are:

- An association is a hint for possible ambiguities. For example, suppose that each of two different classes sends a message to the same target class. The analyst should check that the two different classes actually are to communicate with the same target class. If a motion sensor activates one type of display, and a smoke detector activates another type of display, then the class diagram should reflect this situation with two different display classes.
- Each class should reflect one and only one concept. For example, the analyst should check that book and textbook are really two different classes when Dowser does not create a generalization of these two classes.
- If a class has an attribute, but the attribute is not of a primitive type, such as string or number, then the definition of the attribute might be missing in the original text. After a definition is added, the attribute should be represented by its own class.
- If a class has no association, then the class might be underspecified, as there are no relations or interactions between the class and other classes.

3.7 Limitations of Method

Observe that the OOAM is a model of only static relationships among the concepts mentioned in the parsed NL SRS. We have not attempted to apply our approach to modeling behavior.

4 Studies

This section describes the case studies in which we evaluated the effectiveness of our approach in helping an analyst to identify ambiguities, inconsistencies, and underspecifications in a NL SRS and in which we evaluated Dowser's effectiveness at DST identification.

4.1 Elevator Case Study

To evaluate the effectiveness of our approach, we implemented Dowser and applied it to an example NL SRS that we call "the ESD". The ESD describes the control software for an elevator system [16]. The ESD was chosen, because it could be the NL SRS of a real industrial system. At the same time, the ESD is short enough to completely described in this paper. Moreover, the ESD happens to contain enough defects that it illustrates the defect types that can be revealed with the help of Dowser.

The original ESD was:

An n elevator system is to be installed in a building with m floors. The elevators and the control mechanism are supplied by a manufacturer The internal mechanisms of these are assumed (given) in this problem.

Design the logic to move elevators between floors in the building according to the following rules:

1. Each elevator has a set of buttons, one button for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited (i.e., stopped at) by the elevator.
2. Each floor has two buttons (except ground and top), one to request an up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The buttons are cancelled when an elevator visits the floor and is either travelling the desired direction, or visiting a floor with no requests outstanding. In the latter case, if both floor request buttons are illuminated, only one should be cancelled. The algorithm used to decide which to serve first should minimize the waiting time for both requests.
3. When an elevator has no requests to service, it should remain at its final destination with its doors closed and await further requests (or model a "holding" floor).
4. All requests for elevators from floors must be serviced eventually, with all floors given equal priority.
5. All requests for floors within elevators must be serviced eventually, with floors being serviced sequentially in the direction of travel.

First, we applied Dowser tool to the original unmodified ESD, which was not written according to any constraining grammar. Since the transformation rules have been created assuming that the analyzed text conforms to a constraining grammar, applying Dowser to the original ESD resulted in a diagram with only five classes such as these and set. None of these classes describes any domain concepts of the ESD.

To successfully apply Dowser to the ESD, the ESD had to be rewritten sentence-by-sentence to conform to the constraining grammar. No information was added or removed from the original ESD during the rewriting. Therefore, the rewriting did not introduce any new defects, which would have adulterated the results of the case study.

The rewritten ESD is:

An n elevator system is to be installed in a building with m floors.

1. Each elevator has buttons. Each elevator has one button for each floor. When a user presses a button, the elevator illuminates the button and the elevator visits the corresponding floor. When the elevator visits a floor, the elevator cancels the corresponding illumination.
2. Each floor has two buttons. (except ground and top). If the user presses the up-button, an up-elevator is requested. If the user presses the down-button, a down-elevator is requested. If the user presses a button, this button becomes illuminated. When an elevator visits a floor, the elevator cancels the corresponding illumination of the button in the desired direction. The system minimizes the waiting time.
3. When an elevator has not to service any requests, the elevator remains at its final destination and the doors of the elevator are closed. The elevator then awaits further requests.

4. The elevators service all requests from floors with equal priority eventually.
5. If a user presses a button within the elevator, the elevator services this request eventually in the direction of travel.

Applying Dowser to the new ESD resulted in the diagram of Figure 2. Dowser created an OOAM containing 14 partially connected classes with attributes and methods.

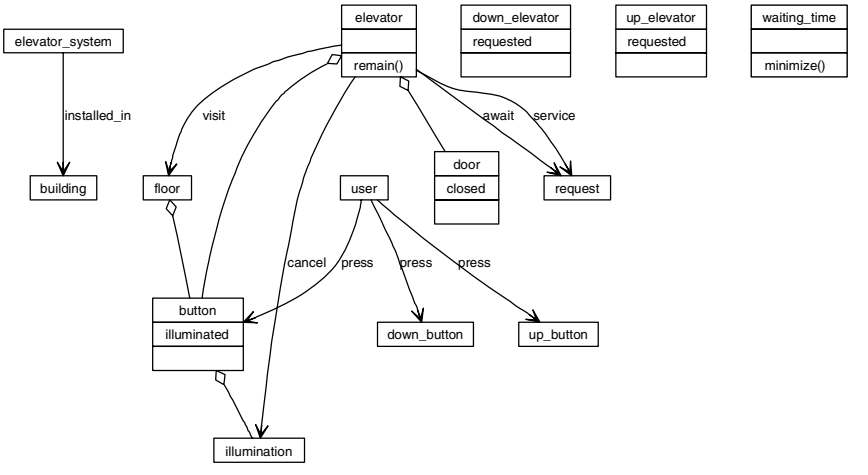


Fig. 2. OOAM of the ESD

The graphically rendered OOAM does not reveal defects on its own. By applying the guidelines described in Section 3.6, we identified four conceptual defects in the OOAM, which could be traced back to the original ESD.

1. The diagram shows classes up-elevator and down-elevator. Neither class has any connection to any other class, and neither has any association to the class elevator. Furthermore, the up-elevator class has an attribute requested, while elevator serves a request indicates that each of up-elevator and down-elevator is a specialization of elevator. All of this information indicates that neither concept, up-elevator nor down-elevator, is defined enough in the original ESD.
2. The class door in the diagram contains the attribute closed. However, the class has no method to manipulate this state. If closing and opening the door are within the scope of the system, then it is clear that the concept door is not defined enough in the original ESD.
3. In the ESD, each of the floor and the elevator has buttons. Therefore, each of the class elevator and the class floor should have an aggregated class button in the OOAM. However, the diagram indicates that both have the same type of button. Since a button in an elevator and a button in a floor have different behaviors, it is unlikely that the same class describes both types of buttons. Generalizing both types to a single class button could therefore lead to misinterpretations. Defining concepts elevator button and floor button would resolve this ambiguity and enhance the clarity of the ESD.

4. Each of the classes up-button and down-button is connected to only the class user in the OOAM. Since a user is an actor in the system, the diagram does not clarify where button belongs. The location can be derived from the ESD, because button is mentioned in the paragraph that mentions floor. However, it should not be necessary to use this fact. Therefore, each concept should be specified in more detail in the ESD to reduce the ambiguity in the specification.

This case study shows how Dowser can help an analyst identify defects in a NL SRS. If a constraining grammar is used to write a NL SRS, our approach can help detect ambiguities, inconsistencies, and underspecifications.

4.2 DST Detection Quality

The case study in Section 4.1 shows the importance of detecting DSTs. For the ESD, Dowser needed to be able to detect DSTs such as floor button, elevator button, or down elevator. If Dowser were to extract only the terms button and elevator from the ESD, it would create wrong classes and associations.

Section 3.4 explains how Dowser relies on syntactic information to identify DSTs. To measure Dowser's DST detection capability, we conducted a separate study. To measure Dowser's DST detection, the metrics *recall* and *precision* were calculated of Dowser's extraction of DSTs from a user's manual. These metrics are used to evaluate the success of many NLP tools [14] :

– *Recall* measures how well a tool identifies desired pieces of information in a source:

$$Recall = \frac{I_{correct}}{I_{total}} \tag{1}$$

Here $I_{correct}$ is the number of correctly identified desired pieces of information in the source text and I_{total} is the total number of desired pieces of information in the source text.

– *Precision* measures how accurately a tool identifies desired pieces of information in a source:

$$Precision = \frac{I_{correct}}{I_{correct} + I_{incorrect}} \tag{2}$$

Here $I_{correct}$ is as for *Recall* and $I_{incorrect}$ is the number of incorrectly identified desired pieces of information in the source text.

We chose the *intro man page* of the *Cygwin environment* with which to measure recall and precision of Dowser's DST detection. A manual page seems to be a suitable experiment source, because it is a technical document with a large number of DSTs.

The steps of the experiments are as follows.

1. We manually identified every DST, a noun or a compound noun, from the *intro man page*. The *intro man page* contains 52 terms like Cygwin, Linux-like environment, Linux API emulation layer, and POSIX/SUSv2. The 52 terms consists of 33 single-noun terms and 19 compound-noun terms.

2. For the first experiment, the link grammar parser extracted every term out of the *intro man page* without the capability of extracting compound-noun terms. It recognized 31 of the single-noun terms and none of the compound-noun terms. Therefore, it reached a recall value of 59.6% for all terms and of 93.9% for single noun terms.
3. For the second experiment, compound-noun term detection was added to the link grammar parser. After this, the tool recognized 10 compound-noun terms. As the single-noun terms detection rate stayed the same, the tool recognized 41 terms. Therefore, it reached a recall value of 78.8%.

Afterwards, the undetected terms were examined. It turned out that five terms were undetected because they appeared in grammatically obscure or wrong parts in the sentence. Correcting these sentence, increased the detected terms to 46 and the recall value to 88.46%.

The five not identified terms were (1) case-insensitive file system; (2) intro man page; (3) Linux look and feel; (4) Red Hat, Inc.; and (5) User's Guide. The term Red Hat, Inc. is not recognized because of the comma, User's Guide cannot be detected syntactically, because if every genitive were part of a term, it would lead to an over-generation of terms. Linux look and feel is not recognized because of the conjunction and in the term. Case-insensitive file system and intro man page can be only partially detected, because case-insensitive is an adjective, which is only sometimes part of a term. Another example demonstrates the difficulties caused by adjectives. In readable manual, only manual is a term in most cases. Using every adjective as part of the term would lead to an overgeneration of terms. The term intro man page is not recognized because the link grammar parser guesses that intro is an adjective. However, if it is planned that case-insensitive file system is a concept within a SRS, then writing it with initial upper-case letters would allow the link grammar parser to detect it as a proper noun, and thus as a DST.

Dowser extracted seven wrong terms, since it created wrong terminology for the incompletely detected terms, e.g., it extracted the term man page instead of intro man page. Overall, Dowser reached a precision value of 86.79% on the *intro man page*.

The DST detection experiment shows that using only syntactic information from the link grammar parser allows a fairly high DST detection rate.

4.3 Monterey Workshop Airport Security Case Study

As a second case study, we have applied Dowser to the workshop case study, here called WSCS, taken from the 2007 *Monterey Workshop on Innovations for Requirements Analysis: From Stakeholder Needs to Formal Designs* [22]. Unlike the case study reported in Section 4.1, the WSCS was not written as a SRS. The WSCS is a transcript of a discussion about airport security by three principal stakeholders: the Transportation Security Administration (TSA), the Federal Aviation Administration (FAA), and Airport Screening and Security (ASS). No stakeholder in the discussion even attempts to be correct, complete, and consistent. While no real SRS is correct, complete, and consistent, at least an SRS's authors try to make it so. One's goals in any writing effort affect the output.

The first step was rewriting the sentences of the WSCS discussion transcript to meet the constraints of the constraining grammar. The first main change was to make each sentence active [31]. The WSCS's domain is familiar enough to the man-in-the-street that we had no problems identifying each doer. In the normal industrial case, we would have asked the client for this information. The second main change was to remove each modal verb, such as can in can deal that consists of a modal verb followed by a main verb. A modal verb has no real effect on the OOAM to be built; the main verb carries all the meaning. There were some so-called sentences of the transcript that had to be made into more than one sentence. There were portions of the transcript that defied rewriting. They were simply left out.

The revised transcript had 65 sentences, and they were submitted to Dowser. The Dowser tool was able to parse and extract information out of 58 of these sentences, for an 89% acceptance rate, leaving 7 unaccepted sentences.

Since Dowser listed each unaccepted sentence, we were able to perform an additional detailed manual analysis. By creating the link-grammar parse of each sentence, we were able to compare the created links to the rules and found the reason for Dowser's rejection of the sentence. The reasons can be categorized into three types.

1. The link-grammar parser was not able to guess the correct syntactic role of a word in the sentence,
2. Dowser's rules were not refined enough to handle the sentence, or
3. the sentence was underspecified.

We rewrote each of the 7 sentences, keeping the semantics consistent with sentence's intent while clarifying underspecfications. We chose to rewrite the not accepted sentences, because this strategy is the easiest way for an analyst to achieve a 100% acceptance rate with Dowser.

The final list of sentences from which Dowser can parse and extract information is given by the transcript below. Note that neither the first 5 lines nor any line with a three-letter acronym for a stakeholder followed by a colon were submitted to Dowser. These help the reader see the correspondence to the original transcript.

Case Study: Air Traveling Requirements Updated (Blog scenario)
Participants:
Transportation Security Administration (TSA)
Federal Aviation Administration (FAA)
Airport screening and security (ASS)

FAA:
Airline passengers not take liquids on board.
We increase security following the recent foiled UK terrorist plot.
We develop technologies that screen for chemicals in liquids.
You know backscatter.

ASS:
Technologies that work in laboratories cause false alarms when used dozens of times in daily screening.

Technologies are not ready for deployment.

FAA:
False positives help us to stay alive.
You be more alert.
ASS miss guns and knives packed in carry-on luggage.

ASS:
ASS moves 2 million passengers through US airports daily.
ASS not remain alert to rare events.

TSA:
TSA deal with it.
You take frequent breaks.
As a test, TSA will impose the image of a weapon on a normal bag.
ASS learns that the image of a weapon appear on a normal bag.
Solutions will be a combination of machine intelligence and human intelligence.

ASS:
ASS take breaks.
ASS get inspected.
ASS not get annual surprise tests.
ASS gets tests every day.
If a screener misses too many tests consistently, he is sent to training.

TAS:
TSA and ASS and FAA take proactive steps to prevent another attack.
We only react. We do not anticipate.
If someone uses a box cutter to hijack a plane, then passengers not take box cutters on board.
If someone hides explosives in his shoes, then ASS x-ray everyone's shoes and ban matches.

FAA:
For each dollar an attacker spends, FAA spends a thousand dollars.
There are no easy solutions.
FAA federalizes checkpoints.
FAA brings in more manpower.
FAA brings in more technology.

TSA:
We plan responses in advance.
Nobody needs a metal object to bring down an airliner.
Nobody needs even explosives to bring down an airliner.
Everything on an airplane burns.

Passengers not burn.
Technologies detect oxydizers.
Screeners learntodetect oxydizers.

FAA:
Airlines take the lead on aviation security.
Airlines marketed cheap tickets.
Airlines passed security off on the federal government.
Security officers be on every flight.
Retrain flight attendants as security officers.
We cannot give passengers soda and peanuts.
Passing around soda and peanuts should be secondary.

ASS:
A lot of airlines are not doing well.
A lot of airlines are on government assistance.
Airlines raise prices.
Airlines mishandle baggage.
The TSA changes screening rules constantly.
Anything radical costs a lot of money.
Anything radical deters people.
An economic threat is also a threat.

TSA:
TSA enforce consistency in regulations.
TSA enforce consistency in regulations' application.
Airline bankruptcy has advantages.
Bankruptcy makes it easier to rearrange company assets.
Bankruptcy makes it easier to renegotiate vendor contracts.
Bankruptcy makes it easier to renegotiate supplier contracts.

FAA:
TSA, FAA, and ASS have productive discussions.
TSA, FAA, and ASS get back to work.
TSA, FAA, and ASS come up with concrete measures.
TSA, FAA, and ASS generate some ROI.
The above examples are not all-inclusive.

In the end, after obtaining sentences that are acceptable to Dowser, we decided not to try to analyze the generated OOAM. In any case, the sentences do not even begin to form a correct, complete, and consistent SRS. So there is little point in discovering ambiguities. The set of sentences itself is ambiguous because there are unbounded ways to complete them into a SRS. Indeed, Dowser was never intended to be used in the part of RE in which this sort of raw requirements information is analyzed.

Nevertheless, the exercise of rewriting the sentences to be acceptable to Dowser was valuable in making the intent of the sentences clearer to any reader. It forced us to identify the doers of each activity that was specified in a passive sentence. It forced us to grapple with the meaning of many a phrase that is used in normal conversation that really have no effect on any OOAM. It forced us to find simpler ways to express things that had been expressed in the normal sloppy way of everyday conversation. The result was, in our opinion, a much cleaner set of sentences. Learning how to write these cleaner sentences may be the most valuable effect of the use of Dowser.

This exercise exposed some limitations of the current version of Dowser. Note that there are two sources of limitations, (1) what the link grammar can parse and (2) what the Dowser rules can handle to construct an OOAM from the results of the parse. It was fairly easy to rewrite the sentences so that they could be accepted by the parser. It was more difficult to get Dowser to handle parsed sentences. In each case that Dowser did not accept a construct, the choices were (a) to modify existing Dowser rules or to add new Dowser rules to deal with the construct or (b) to change the containing sentence to avoid the construct. If we could see the effect on OOAMs of the construct, we took the first choice. If not, we took the second choice.

The two main limitations that we discovered are of the second kind. The current version of Dowser cannot deal with modal verbs and with negations, e.g., not. Because we could not see their effect on OOAMs, we would take the second choice response. Because we did not analyze the model, this response was applied only for the modal verbs in the last version of the sentence. That is, the nots are still there. One possible treatment of a not is to build the normal graph with the verb labeling the arc and putting a not by the verb to indicate a misuse case [33].

5 Discussion and Conclusion

Dowser, a tool built mostly out of existing software, is able to help a human analyst to identify ambiguity, inconsistency, and underspecification in a NL SRS.

Of course, Dowser's lack of perfection, particularly in the construction of an OOAM and in the DST recall, says that Dowser can be used as only one of an array of approaches and tools for identifying DSTs and for detecting ambiguity, inconsistency, and underspecification in NL SRSs. However, because of the inherent difficulty of these tasks for humans, every little bit helps!

One drawback of the approach is that for best results, the input should be written in the constrained language. If the actual input is not written in the constrained language, it must be rewritten. This rewriting necessity might be considered a reason not to use Dowser. However, one could argue that the rewriting is part of the whole process of eliminating ambiguity, which ultimately the human carries out.

5.1 Future Work

The lack of perfection says that more work is needed:

– How does Dowser perform on larger, industrial-strength NL SRSs? Answering this question would help to explore the problem space and to find new unsolved research questions.

- The current Dowser cannot resolve anaphora. An anaphor is a linguistic unit, such as a pronoun, that refers to a previous unit. In The customer can buy text books and return them., them is an example of an anaphor, which must be resolved to text books. While Dowser can identify anaphora, it cannot resolve them. A simple solution would be to have Dowser flag all anaphora in its input text, so a human analyst could change each to its resolution.
- DST identification can be improved. As mentioned above, syntactic information is not sufficient to detect all the DSTs within a document. Therefore, frequency analysis or baseline text analysis [21] might improve DST identification.
- Additional semantic knowledge could improve the capability of Dowser. For example, the *WordNet* lexicon contains information about hypernyms, which can indicate superclasses, and meronyms, which can indicate aggregations. This information could be used to supply missing links in an OOAM. However, although *WordNet* is a large online lexicon, it lacks DSTs and therefore might be only a little help. Extending Dowser's dictionary with DSTs could reduce this problem.
- The current Dowser is not applicable to a NL SRS that has a functional language style, i.e., with sentences such as, The system must provide the functionality of.... Handling such sentences would require a different grammar. Future work could examine which grammar is the most suitable for class extraction.
- The UML offers a set of different diagram types. NLP could be used to create sequence, state, or other diagrams. For example, Juristo *et al.* [17] developed also a controlled grammar for specifying dynamic behavior.
- Other work has found different sources of ambiguities in NL SRS. Since there seems not to be a single perfect approach, different approaches (e.g. [7, 23, 41]) could be integrated into a single framework for validating NL SRSs. This integration could lead to a new method of developing NL SRSs. Indeed, this sort of integration would deal with any syntactic ambiguities found by the parser that is used.

Acknowledgments

The authors thank the anonymous referees for some really helpful suggestions. Daniel Berry's, work was supported in part by Canadian NSERC Grant Number NSERC-RGPIN227055-00.

References

[1] Abbott, R.J.: Program Design by Informal English Descriptions. Comm. ACM 26, 882–894 (1983)

[2] Berry, D.M., Daudjee, K., Dong, J., Fainchtein, I., Nelson, M.A., Nelson, T.: Users' Manual as a Requirements Specification: Case Studies. Requir. Eng. J. 9, 67–82 (2004)

[3] Berry, D.M., Kamsties, E.: Ambiguity in Requirements Specification. In: Leite, J.C.S.P., Doorn, J. (eds.) Perspectives on Requirements Engineering, pp. 7–44. Kluwer, Boston (2004)

[4] Bucchiarone, A., Gnesi, S., Pierini, P.: Quality Analysis of NL Requirements: An Industrial Case Study. In: 13th IEEE International Conference on Requirements Engineering (RE 2005), San Diego, CA, USA, pp. 390–394. IEEE Comp. Soc. Press, Los Alamitos (2005)

[5] Clayton, R., Rugaber, S., Wills, L.: Dowsing: A Tools Framework for Domain-Oriented Browsing Software Artifacts. In: Automated Software Engineering Conference, Honolulu, HI, USA, pp. 204–207 (1998)

[6] Delisle, S., Barker, D., Biskri, K.: Object-oriented Analysis: Getting Help from Robust Computational Linguistic Tools. In: Friedl, G., Mayr, H.C. (eds.) Application of Natural Language to Information Systems, Oesterreichische Computer Gesellschaft, Vienna, Austria, pp. 167–172 (1999)

[7] Fabbrini, F., Fusani, M., Gnesi, S., Lami, G.: The Linguistic Approach to the Natural Language Requirements, Quality: Benefits of the use of an Automatic Tool. In: 26th Annual IEEE Computer Society-NASA GSFC Software Engineering Workshop, San Diego, CA, USA, pp. 97–105. IEEE Comp. Soc. Press, Los Alamitos (2001)

[8] Fuchs, N.E., Schwertel, U., Schwitter, R.: Attempto Controlled English (ACE) Language Manual, Version 3.0. Tech. Rept. 99.03, Dept. Computer Science, U. Zurich (1999)

[9] Fuchs, N.E., Schwitter, R.: Attempto Controlled English (ACE). In: 1st International Workshop On Controlled Language Applications (CLAW), Leuven, Belgium, pp. 124–136 (1996)

[10] Gause, D., Weinberg, G.: Exploring Requirements: Quality Before Design. Dorset House, New York (1989)

[11] Gemini Natural-Language Understanding System, `http://www.ai.sri.com/natural-language/projects/arpa-sls/nat-lang.html`

[12] Gervasi, V., Nuseibeh, B.: Lightweight Validation of Natural Language Requirements. Softw., Pract. & Exper. 32, 113–133 (2002)

[13] Graphviz–Graph Visualization Software Home Page, `http://www.graphviz.org/Credits.php`

[14] Grishman, R.: Information Extraction: Techniques and Challenges. In: Pazienza, M.T. (ed.) SCIE 1997. LNCS, vol. 1299. Springer, Heidelberg (1997)

[15] Harmain, H.M., Gaizauskas, R.J.: CM-Builder: A Natural Language-Based CASE Tool for Object-Oriented Analysis. Autom. Softw. Eng. 10, 157–181 (2003)

[16] Heimdahl, M.: An Example: The Lift (Elevator) Problem, `http://www-users.cs.umn.edu/heimdahl/formalmodels/elevator.htm`

[17] Juristo, N., Moreno, A.M., Lopez, M.: How to Use Linguistic Instruments For Object-Oriented Analysis. IEEE Softw. 17, 80–89 (2000)

[18] Kiyavitskaya, N., Zeni, N., Mich, L., Berry, D.M.: Requirements for Tools for Ambiguity Identification and Measurement in Natural Language Requirements Specifications. Requir. Eng. J. 13, 207–239 (2008)

[19] Klein, D., Manning, C.D.: Accurate Unlexicalized Parsing. In: 41st Meeting of the Association for Computational Linguistics, pp. 423–430. Assoc. for Computational Linguistics, Morristown (2003)

[20] Kof, L.: Natural Language Processing for Requirements Engineering: Applicability to Large Requirements Documents. In: Russo, A., Garcez, A., Menzies, T. (eds.) Workshop on Automated Software Engineering, Linz, Austria (2004)

[21] Lecoeuche, R.: Finding Comparatively Important Concepts between Texts. In: 15th IEEE international Conference on Automated Software Engineering, pp. 55–60. IEEE Comp. Soc. Press, San Diego (2000)

[22] Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
[23] Mich, L.: On the Use of Ambiguity Measures in Requirements Analysis. In: Moreno, A., van de Riet, R. (eds.) 6th International Conference on Applications of Natural Language to Information Systems (NLDB 2001). LNI, vol. 3, pp. 143–152. Gesellschaft für Informatik, Bonn (2001)
[24] Mich, L., Franch, M., Novi Inverardi, L.: Market Research for Requirements Analysis Using Linguistic Tools. Requir. Eng. J. 9, 151 (2004)
[25] Mich, L., Garigliano, R.: Ambiguity Measures in Requirements Engineering. In: International Conference on Software–Theory and Practice (ICS2000), 16th IFIP World Computer Congress, pp. 39—48, Publishing House of Electronics Industry, Beijing, China (2000)
[26] Mich, L., Garigliano, R.: NL-OOPS: A Requirements Analysis Tool Based on Natural Language Processing. In: 3rd International Conference on Data Mining Methods and Databases for Engineering. Witpress, Southampton, UK (2002)
[27] Miller, G.A., Felbaum, C., et al.: WordNet Web Site. Princeton U., Princeton, NJ, USA, http://wordnet.princeton.edu/
[28] Mollá, D., Schwitter, R., Rinaldi, F., Dowdall, J., Hess, M.: ExtrAns: Extracting Answers from Technical Texts. IEEE Intelligent Syst 18, 12–17 (2003)
[29] Nanduri, S., Rugaber, S.: Requirements Validation via Automated Natural Language Parsing. J. Mgmt. Inf. Syst. 12, 9–19 (1996)
[30] Rumbaugh, J.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (1991)
[31] Rupp, C., Goetz, R.: Linguistic Methods of Requirements-Engineering (NLP). In: European Software Process Improvement Conference (EuroSPI), Copenhagen, Denmark (2000), http://www.iscn.com/publications/#eurospi2000

Innovations in Natural Language Document Processing for Requirements Engineering*

Valdis Berzins, Craig Martell, Luqi, and Paige Adams

Naval Postgraduate School, 1411 Cunningham Road, Monterey, California 93943
{berzins,cmartell,luqi,phadams}@nps.edu

Abstract. This paper evaluates the potential contributions of natural language processing to requirements engineering. We present a selective history of the relationship between requirements engineering (RE) and natural-language processing (NLP), and briefly summarize relevant recent trends in NLP. The paper outlines basic issues in RE and how they relate to interactions between a NLP front end and system-development processes. We suggest some improvements to NLP that may be possible in the context of RE and conclude with an assessment of what should be done to improve likelihood of practical impact in this direction.

Keywords: Requirements, Natural Language, Ambiguity, Gaps, Domain-Specific Methods.

1 Introduction

A major challenge in requirements engineering is dealing with changes, especially in the context of systems of systems with correspondingly complex stakeholder communities and critical systems with stringent dependability requirements. Documentation driven development (DDD) is a recently developed approach for addressing these issues that seeks to simultaneously improve agility and dependability via computer assistance centered on a variety of documents [1,2]. The approach is based on a new view of documents as computationally active knowledge bases that support computer aid for many software engineering tasks from requirements engineering to system evolution, which is quite different from the traditional view of documents as passive pieces of paper. Value added comes from automatically materializing views of the documents suitable for supporting different stakeholders and different automated processes, as well as transformations that connect different levels of abstraction and representation. The sheer size and complexity of enterprise-wide systems makes such automation support a necessary condition for reliability rather than a convenience. The body of documents that embody the requirements of such systems is encyclopedic in size and scope, and consequently impossible for a single person to understand in detail. Assuring absence of contradictions or other non-local quality properties on such scales is practically impossible for unaided humans.

* This work was supported in part by ARO under project P-45614-CI.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 125–146, 2008.

At the level of requirements engineering, a central problem is related to bridging the gap between stakeholders, who communicate in natural language, and software tools, which depend on a variety of formal representations. A prominent problem is resolving ambiguity, which is typical of natural language and to a somewhat lesser degree the popular informal design notations such as UML. If ambiguities in stakeholder needs statements are transferred into system specifications without being accurately resolved, they are likely to produce system faults. This is because the world view and tacit assumptions and priorities of system developers usually differ from those of prospective system clients. Others include finding implied but unstated requirements, detecting conflicts between needs of different stakeholders, and resolving such conflicts. Communication gets increasingly difficult as systems scale up. Stakeholders are typically comprised of diverse groups, each of which has its own specialized domain knowledge, jargon, and unique tacit understanding of the problem. Bridging the gaps becomes key to success as complexity increases because each group typically has only a partial understanding of the issues, constraints, possible solutions, and cost implications. [3,4] For large systems the gaps between communities can be so extreme that different stakeholders experience different realities. For example, during analysis of an avionics software fault that would cause an airplane to turn upside down when it crosses the equator, it was suggested that this was a severe problem that should be fixed right away. A fighter pilot disagreed, saying that the pilot could just turn the plane right side up again and go on. A later reaction from a helicopter pilot was that if it happened to him, he would die as a result.

Progress on increasing flexibility without damaging reliability depends on computer aid within an end-to-end process that includes requirements engineering. This leads to a need for natural language processing that can help bridge the gap between natural stakeholder communication and unambiguous requirements models such as those embodied in the DDD view of documents. Ever present changes in requirements imply that this gap must be bridged repeatedly. This in turn implies that incremental methods that can take advantage of knowledge gained in previous iterations would be helpful.

In the 1970s the automatic programming group at MIT headed by Prof. Bill Martin sought to create an end-to-end system that went from user requirement documents to running code for business information systems. The project made progress at the top and bottom levels of this process, but the two ends were never integrated together.

The capabilities of natural language processing (NLP) software and our understanding of requirements engineering (RE) have improved substantially over the past 30 years. This paper re-examines how the current state of NLP can contribute to requirements engineering, how close is it to making a practical impact in this context, and what needs to be improved to enable widespread adoption. We examine the connection between a hypothetical NLP front end and requirements engineering processes that would follow, and identify some of the differences between generic NLP and domain-specific NLP embedded in a requirements engineering process.

1.1 Challenges of NLP for Requirements Engineering

Requirements engineering is a critical part of the system development process because requirement errors cost roughly 100 times less to correct during requirement engineering than after system delivery [5]. This imposes extreme constraints on the accuracy of NLP that we might use to derive system requirements. However, NLP accuracies are currently in 90%-92% range, at best (see section 2). Therefore NLP must be augmented with other methods for removing residual errors, and accuracy must be greatly improved if it is to be seriously used for RE.

1.2 Why All Is Not Yet Lost

NLP in the context of RE should be more tractable than generic NLP, because it has the usual advantages of a domain-specific approach: scope is narrower, more is known about the context, and specialized methods may apply. In particular, much more is known about the intentions of the speaker and the context, such as typical goals and surrounding tasks.

1.3 Overview

Section 2 presents a selective history of the relationship between RE and NLP. Section 3 briefly summarizes recent trends in NLP. Section 4 outlines basic issues in requirements engineering and how they relate to interactions between a hypothetical NLP front end and system development processes and tools that follow. Both aspects have been simplified to help bridge the gap between the two communities; our apologies in advance to experts in both domains for leaving out some of the subtleties of each area. Section 5 outlines some improvements to NLP that may be possible in the context of RE. Section 6 concludes with an assessment of what should be done to improve likelihood of practical impact in this direction.

2 A Selective History of the Relationship between RE and NLP

The desire to use natural language in software engineering has existed nearly as long as the discipline itself. Indeed the invention of the compiler was an attempt to express machine code in a higher-level language, one more closely resembling human communication. Since the introduction of the FORTRAN compiler in 1954, computer scientists and programmers have sought ways to interact more naturally with the computer and eliminate the burden of translating required tasks into machine code that could be directly executed. This section is an overview of natural language processing (NLP) influence in the software development process, with an emphasis on requirements engineering. It is not intended to be an exhaustive overview; however, it is an attempt to illustrate representative works over the last four decades that have utilized NLP techniques.

After the development of high level programming languages such as FORTRAN and COBOL, "automatic programming" was one of the first attempts to bring natural language into the software engineering process. Ruth wrote that "automatic programming systems are simply the next logical step in the progression that has taken us from writing in machine language to using assemblers to using compilers."

As summarized by Balzer after 15 years of related work, components of automatic programming included: 1) a means of acquiring a high-level specification (requirements), 2) a mechanism for requirements validation, 3) a means of translating the high-level specification into a low-level specification, and 4) an automatic compiler for compilation of the low-level specification [6]. Automating the software development process was traditionally viewed as a compilation problem, since that was where the majority of development effort had been concentrated, but Balzer realized that it was a specification problem as well.

Those who sought true end-to-end automated software development eventually realized that the challenges were more difficult, and the goal more elusive, than originally anticipated. This was primarily due to four factors:

1. Insufficient computing power
2. Immaturity of the field of NLP
3. Insufficient understanding of the substance and difficulty of requirements engineering
4. Increasing complexity of software and software development forced new techniques in software engineering at different levels of abstraction

As a means of tackling the specifications element of the automatic programming problem, Balzer et al. developed the Gist specification language. Gist was one of the first attempts to render higher-level specifications in a pseudo-natural language. The idea behind the concept was that requirements could be captured in a human-readable form, which could then be automatically translated into lower-level specifications. These would, in turn, be automatically compiled into executable code. In addition to facilitating the requirements and verification process, the aim was also to correct what Balzer saw as a flawed step in the software development life cycle: maintenance. Prior to this work, maintenance had primarily involved directly editing the implementation when changing needs dictated, rather than updating the specification, then the implementation. This shortcut was typically motivated by cost and schedule pressures, and gained short term benefit at the expense of increased long term maintenance costs due to the loss of specification information. Gist sought to address this by necessitating only that the specification be changed; the system would then be able to automatically generate the new implementation based on the updated specification. While Gist showed some success, and was used as a specification language for USC software engineering courses, it still fell short of achieving the ultimate goal of an end-to-end solution for specification to implementation. Some of its shortcomings were that, despite its high level, it was still found to be unreadable

(a paraphraser was subsequently developed that partially alleviated this) and it was not possible to automatically translate Gist into a compilable form. [6]

In 1974, Heidorn described a system that used English as a *very high-level language* (VHLL) in simulation programming. The underlying program was written in FORTRAN and was implemented under CP/CMS on the IBM 360/67. Programming the simulation took the form of describing the problem statement in natural language English phrases in a dialogue session with the computer. The computer had the ability to query the user when additional information or clarification was needed, and likewise, the user could ask questions of the computer if a particular response was unclear. After the problem statement was entered to the system's satisfaction, it would notify the user and an English-language description of the problem could be produced for verification purposes. The language processing facility was based on sets of decoding rules that were input into the system and were interpreted "in the fashion of a bottom-up, parallel-processing syntax-directed compiler." The system used approximately 300 rules, which included rules for tasks such as stemming and verb-phrase transformations. Using Balzer's four-phase automatic programming paradigm, we can describe Heidhorn as having envisioned that the problem acquisition phase was the one for which this system had the most to offer. [7]

By 1978, researchers had come to realize the magnitude of the NLP problem. Martin remarked, "Making computers comprehend natural language has turned out to be a very difficult task, not clearly distinguishable from the general problem of creating artificial intelligence." His insight, however, was that a useful database query system could be developed by solving a part of the NLP problem. Some of the concepts he introduced included a loosening of formal syntactic rules, whereby a system could parse a query if it was understandable according to part-of-speech even if it may violate a syntax rule (e.g., "He picked up her." versus "He picked her up.") and an assumption that users would ask questions to which they wanted informative answers (e.g., "Do you know the departure time of flight 32?" would elicit an actual time, not just a "yes"). Martin's EQS system competed with several other database query languages of the time, including LADDER, ROBOT, and PLANES, but unlike those systems, EQS used natural language parsing, and was able to both capture and produce more information. Its advantages were that it could acquire additional domain-specific syntactic details from the user without requiring explicit knowledge of English syntax, it could accept multiple phrasings of a query, it could be extended with new words and phrases, it could be programmed to assist the user in adding new semantic knowledge, and pronoun reference resolution was easier. On the negative side, EQS was computationally intensive and would "waste time splitting hairs in cases that don't matter as well as in those that do." [8]

Desire for natural language programming gave birth to higher and higher-level languages. The Business Definition Language (BDL), introduced in 1977, was another early example of what was to be known as a very high-level language (VHLL). It was developed specifically to reduce the amount of manual labor involved in specifying business problems and using these specifications to develop

applications. Since the operations of most businesses (particularly at the time) revolved around paper-based forms, BDL had three component sublanguages: one to define the business forms, one to describe the organization, and one for defining calculations. [9]

Many researchers began to explore pseudo-natural language specification languages. In 1995, Lu et al. proposed BIDL (Business Information Description Language) as a component of their PROMIS knowledge-based tool for automatically prototyping management information systems. BIDL draws on three primary knowledge bases for requirements analysis: a domain dictionary, a domain generic model, and software rules. These operate in conjunction with an interactive requirement analyzer to produce a system specification for design and implementation. Although BIDL is English-like, the system requires an analyst to work with end-users to mark up the requirements into a BIDL document. [10]

Many VHLLs were pseudo-natural languages, which meant that they resembled specific natural languages, but had an unambiguous syntax and semantics, just like typical programming languages. This enabled reliable automated processing and translation. They appeared as stylized natural-language text that could be read and understood with some effort by untrained people. However, successfully writing well-formed descriptions in VHLLs was still difficult and required skills similar to programming.

By the early 1990s, requirements engineering had become a full-fledged discipline in its own right and researchers sought to apply natural language tools and techniques to the requirements process in combination with other emerging ideas. Rolland and Proix defined requirements engineering as the part of the "development cycle that involves investigating the problems and requirements of the user community and developing a conceptual specification of the future system." They proposed that a linguistic approach be used to develop a CASE tool for requirements engineering support. Using this tool, unambiguous specifications would be derived from natural language descriptions of the problem space, and for validation, natural language text would be generated from the specification. [11]

In 1993, considering the difficulties inherent in a pure natural language approach to requirements engineering, Kaindl proposed RETH (Requirements Engineering Through Hypertext), a hypertext-based approach to bridge natural language with a formal representation. [12] The Internet, and hypertextual information, was increasing in popularity, therefore it seemed a natural progression to apply this technology toward the requirements process, which Kaindl recognized as "one of the most important and least supported parts of the software life cycle." For example, terms from a domain-specific jargon can be hyperlinked to their definitions to warn non-specialist readers of special meanings and provide and easy way to look up definitions of unfamiliar terms. Kaindl's approach incorporated elements of methodologies and fields such as object-orientation and artificial intelligence. In particular, it treated requirements as objects which could be classified and further refinements derived via inheritance. The stated goal of

Kaindl's work was not to supplant other formal representation techniques, rather to complement them.

In 1993, Ryan criticized previous NLP approaches to RE as being fraught with "many unrealistic suppositions and presumptions." His main argument was that the desire to produce systems requirements as a result of "natural, or 'near natural' conversation" did not make the process easier, nor did it result in more accurate requirements, both of which were objectives in using NLP. In [13] he detailed his argument and offered as an alternative some areas and tasks in the requirements engineering process where he believed NLP could be realistically and usefully applied. His supporting claim was that the requirements process was not merely one of inter-language translation; it was also incumbent upon the requirements analyst to understand the unstated assumptions of those with domain knowledge and be able to model this "common sense" knowledge – an AI problem well beyond the ability of any known machine to solve.

Assuming that the information to be analyzed for requirements was already in some textual form (and not diagrams or other non-textual form), Ryan suggested that automated techniques to scan, search, browse, and tag large bodies of text could be of some use. His view was that the system would have some value as a purely clerical machine, without having to achieve any level of understanding of the text. Another area in which Ryan believed NLP could play a useful role was in that of a "refinement guard." The idea behind this was that, in the requirements process, some user requirements may be difficult to quantify and to translate into a specification language, therefore they may be at risk and likely to be "refined out." He proposed a "requirements tracing facility" to tag requirements early on in the process and allow them to be represented in some form (e.g., comments or links) in the formal specification system. Finally, he describes two approaches to requirements verification where NLP could play a role. In the first approach, the system would generate test scripts containing extreme and average cases based on the formal specification for the client's approval. In the second approach, the system would use an iterative critiquing strategy based, perhaps, on a question-answer method to compare test schemas to the developing specification.

Ryan's critique ended by stating his belief that the requirements process was an organic, social construct, and NLP and other techniques would be of better service in a supporting role, rather than one of replacement.

Nanduri and Rugaber, in 1995, proposed a requirements validation approach using NLP to support an Object Oriented Analysis (OOA) method [14]. Their study involved using a natural language parser to extract candidate objects and associations from a requirements document and construct an object model diagram. The results were tested against the results of a manual OOA process.

Sleator and Temperley's publicly available link grammar parser [15] was used as natural language parser in the Nanduri and Rugaber study. Guidelines were created for creating an object model from the specifications text and were used as rules in text post-processor. Since the parser dealt with each sentence independently, Nanduri and Rugaber modified their tool to accumulate knowledge between the parsed sentences. This entailed using empirical rules for tasks such

as anaphora resolution, which is the task of deciding to what a pronoun, for example, refers[1].

The tool was tested with example high-level specifications for four different applications: a helicopter landing, an automatic teller machine (ATM), an elevator, and an employment database. The results obtained by the process were comparable to the models constructed by hand. In the instances where the tool failed, Nanduri and Ragaber identified the following as causes: parser inadequacy, ambiguous or incomplete specifications in the original requirements documents, lack of domain knowledge, and inadequacy of guidelines. The conclusion reached by the study was that a fully-automated process of model generation using NLP was still not achievable, but that there was value in pursuing further research in an attempt to overcome some of the limitations encountered.

Attempts to turn NL into software requirements were by no means limited to the English language. In 1995, Ohnishi proposed CARD (Computer Aided Requirements Definition), which was a software requirements environment that accepted both Japanese-based textual language and visual language as input and delivered a software requirements specification (SRS) as output. CARD was designed in response to a desire to tackle five elements of the software development process: 1) requirements analysis, 2) requirements description, 3) SRS verification, 4) SRS execution, and 5) preliminary software design. The design goal of CARD was to achieve a quality SRS as measured by correctness, testability, traceability, feasibility, and usability. [16]

In the mid-to-late 1990s, full-fledged requirements engineering environments and tool suites began to emerge. One of the most promising NL-focused, requirements-engineering applications was the Circe environment introduced by Ambriola and Gervasi in 1997. [17] Circe was described as "a Web-based environment for aiding in natural language requirements gathering, elicitation, selection, and validation." It employed a NL recognition engine that takes as input a set of requirements, glossaries (predefined and system specific), and a set of model-action-substitution (MAS) rules that employ fuzzy matching. The output of Cico, the natural language recognition engine of Circe, is a set of abstract requirements, which can be viewed in different user-selectable forms (DFD, E-R, etc.). Notable features of Circe include flexibility, customizability, and extensibility. More work is being done to extend Circe to new domains (e.g., temporal). While Circe is able to detect limited classes of conflicts in modeled data, it does not tackle the task of conflict resolution.

Policy analysis and the derivation of requirements from organizational policy has become a focus area, particularly over the last decade. For large organizations with massive policy bases, this is a complex problem, and the mapping from policy to requirements, or even between policies, can be difficult. Tools that can analyze the language of policies, derive requirements, and check for consistencies, overlap and redundancy, gaps, or inaccuracies are especially sought after. Michael et al. have described the architecture of a natural language input-processing tool (NLIPT) as part of a policy workbench. This tool maps natural

[1] See Section 3, below.

language policy statements to a computationally equivalent form that can be used in a workbench to reason about, maintain, and further develop policy. The tool consists of an extractor, an index-term generator, a structural modeler, and a logic modeler. The prototype tool achieved a 96% accuracy in parsing 99 Naval Postgraduate School security policy statements. [18] This supports our hypothesis that domain-specific natural-language processing tools can potentially attain higher accuracy.

Denger et al. discussed the use of natural language patterns in eliminating imprecision and ambiguity in high-level requirements [19]. Focusing on requirements for embedded systems – those in which high precision is often required to prevent catastrophic failure – the research involved examining language patterns in documentation for elements such as events, conditions, systems reactions, etc. Sentence patterns are then generated and combined into scenarios for complete specifications. This modular approach stresses flexibility so that more precise requirements may be formed by giving the author more expressive freedom during the process. Authoring rules were also developed, which were used in conjunction with the patterns. The authoring rules were designed to describe how natural language could be used to reduce ambiguity. The results of applying this approach showed that the system was able to analyze requirements and rewrite them to reduce ambiguity or include missing information, however, the rewritten requirements tended to be longer and grammatically more clumsy that those written by hand. Additional manual effort was required to clean up the writing to enhance readability.

The development of XML and other structured markup languages inspired some researchers to consider them as an alternative to using natural language in requirements engineering. In 2003, Durán et al. proposed XML/XSLT as a tool in requirements verification. Their justification was that the "lighter" technology provided sufficient flexibility and adequate results without the demand on computer resources that NLP-based approaches imposed. [20]

In 2004, Lee and Bryant stated that natural language was a preferred approach for systems engineering because users must be involved throughout the software development lifecycle to obtain good results. The challenges in using NL in this context were twofold: 1) the natural ambiguity in NL, and 2) the different levels of formalism between the NL domain and the formal specification domain. The project entailed the development of a system that assisted analysts in converting parts of a requirements document written in NL to a formal specification language via linguistic and formal specification techniques. These issues were addressed by using Contextual Natural Language Processing (CNLP) to undertake the ambiguity problem and Two Level Grammar (TLG) to deal with the differing levels of formalism. The research showed that, in some cases, efficient executable code (in a high-level language such as Java or C++) could be generated by using the output of the CLNP-TLG system as input to a formal specification system, the Vienna Development Method–VDM++, which provides analysis tools and code generation capabilities. This process required manual transformation of the text and construction of a problem specific model.

It is still to be seen if this technique will scale up to larger, practical problems. No evaluation of the accuracy of this approach has been provided in [21].

There is still much work to do in natural-language-based requirements engineering research. Focus areas include NLP support in requirements elicitation (NL-based question/answer tools), requirements modeling (developing heuristics for formalizing natural language policies and inferring abstractions and preliminary models from natural-language requirements texts), and validation/verification (comparison of final specification to requirements) [22]. It is clear from examining previous and ongoing work, and considering current NLP capabilities, that the largest gain of productivity in requirements engineering would be realized through the development of supporting toolsets that can partially automate or support a manual process. The focus should be less on developing NLP systems that can understand every nuance of natural language and independently create perfect, implementable specification documents, and more on special purpose tools that can assist professionals in managing large requirement sets in specialized domains through the use of parsing, vocabulary matching, tagging, etc.

Since requirements develop in an iterative process of validation, diagnosis and improvement, tools that can reduce the need for repeating unchanged parts of manual processes would be valuable for RE.

3 Summary of Recent Trends in Natural Language Processing

Natural Language Processing (NLP) is a cross-cutting discipline that includes computer science, linguistics, artificial intelligence and cognitive science, as well as statistics and information theory. The objective of NLP is automated understanding and generation of written natural languages (NL). Challenges of NLP include: the complexity and ambiguity of language constructs; the fact that understanding a natural language often requires representation of one's knowledge about the outside world (tacit knowledge); and the fact that non-linguistic context might also need to be considered, since it often helps to improve the interpretation of speaker intentions. Table 1 provides some concrete examples of the problems just mentioned.

On the one hand, research in NLP still struggles with conceptual difficulties such as context modeling or formalization of speaker intentions [24]. On the other hand, the initial period of excessive optimism in the field was followed by mature statistical analysis and creation of extensive linguistic resources that have helped foster excellent progress in many NLP domains, e.g., part-of-speech-tagging and parsing.

One of the important methodological developments in NLP research was identifying different levels of representation and processing, each with their own set of relevant entities, statistical relations, problems and solutions. NLP distinguishes at least four processing levels: lexical, syntactic, semantic, and pragmatic. Each level has its own patterns of ambiguity (see Table 1) and corresponding

Table 1. Challenges of NLP

Problem	Examples
1. Ambiguity of word meaning and scope	The word *plot* can refer to either a secret scheme or a graphical representation of data. (See blog posting 8 in the workshop case study [23])
	Natural languages usually don't specify which word a phrase or an adjective modifies. For example, in the sentence *Someone hijacks a plane with a box cutter*, does *with a box cutter* refer to *Someone* or to *a plane*? (See blog posting 7 in the workshop case study [23]).
2. Computational Complexity	For determination of grammaticality, it is possible that an exponential number of parse trees might need to be checked.
3. Tacit knowledge and anaphora resolution	The sentences *We gave the passengers the seats because they were waiting* and *We gave the passengers the seats because they were empty* have the same surface grammatical structure. However, in the former the word *they* refers to the passengers, in the latter it refers to the seats: the reference cannot be resolved properly without knowledge of the properties and behavior of passengers and seats.
4. Non-linguistic Context	Includes stakeholder's role, attitude, exaggeration to make a point, domain knowledge, facial expression, gestures, disfluencies, time of the year/day, recent events, etc.

processing methods. As a rule of thumb, the higher the level, the longer the contextual dependencies that have to be taken into account. Importantly, processing at each level is not generally independent. For example, knowing semantics of a sentence may help to disambiguate the part of speech for a particular word.

NLP can be viewed as a sequence of processing steps that starts from a raw text and proceeds through each higher level of representation. Under this approach the output of a lower level is the input for a higher level. Though there are some interdependencies, for simplicity each level is most often considered independently. This assumption greatly facilitates the identification of specific features at each level. It is also important to note that while processing on lexical and syntactical levels is relatively well defined, the higher levels of NLP are not standardized in terms of their objectives or output formats. This is due to the overall complexity of the processing on higher levels and in the extra-linguistic features involved. For example, in order to define pragmatic content for a text one needs to know the intentions of the reader or writer, which typically are not the part of a text.

Since many NLP techniques rely on statistical dependencies in the text, the construction of large-scale, comprehensive data sets, or corpora, has become an important thrust in NLP research. These corpora are composed of a set of texts with words tagged with various labels (e.g. part of speech (POS), semantic, syntactic and role-based ones). Table 2 gives some examples. These corpora provide a rich source for probabilistic modeling of languages. However, each corpus is limited to a specific domain of a particular language (e.g. English

Table 2. Levels of NLP

Level	Problems	Methods/KB
Lexical	Part of speech (POS) tagging	Part of speech tagger Corpora: WSJ, Brown Corpus
Syntactic	Generation of parse-trees representing syntactic structure of sentence	Probabilistic parsers Corpora: WSJ, Brown Corpus
Semantic	Context modeling; Word-Sense Disambiguation	Semantic parsers, WSD Classifiers; Corpora: FrameNet, Senseval
Pragmatic	Goal, content or topic of a text or discourse; Anaphora Resolution	Discourse Analyzers; Corpora: Penn Discourse Treebank

novels and news). The problem of adjusting either corpora or tools to another domain is yet to be solved, although progress is being made [25,26,27,28,29]. Below we briefly explain each step of our simplified NLP model and provide a description of corpora that are used to derive statistical dependencies.

The first step in processing texts is finding word boundaries, called tokenization, and assigning each word a part of speech (e.g. noun, verb, adjective or adverb; quite surprisingly, there are around 40 different POS categories in the most common scheme). This process is called "part of speech tagging" (POS-tagging) and it provides important information for all following stages [30,31]. Usually, POS-tagging is carried out iteratively using short contextual dependencies that specify how a POS of a given word depends on the POS of the previous word. These dependencies are described by a set of conditional probabilities of the form $P(POS1|POS2)$ where $POS1$ is part of speech we are interested in and $POS2$ is the part of speech of the previous word. Contemporary methods of POS-tagging achieve tagging precision above around 97%.

The second step of NLP analyzes larger chunks of a sentence than individual words. In particular, it identifies Noun Phrases (NP), Verb Phrases (VP), Prepositional Phrases (PP), etc. The corresponding method, called syntactic parsing, outputs syntactic trees that provide both labels and the hierarchical structure of a sentence. Most modern parsers are at least partly statistical; that is, they rely on a corpus of training data which has already been annotated (parsed by hand). In short, they use POS information from a previous level but within a larger context to figure out the conditional probabilities of syntactic constituents. Parsing methods condition probabilities not just on POSs but also on the words themselves. State of the art precision in parsing is currently around 92% [28,32,33]. One of the challenges of syntactic parsing is that each sentence can have multiple valid parse trees. Note that parsing difficulties can come from propagation of inaccuracies from a previous stage of processing (Figure 1).

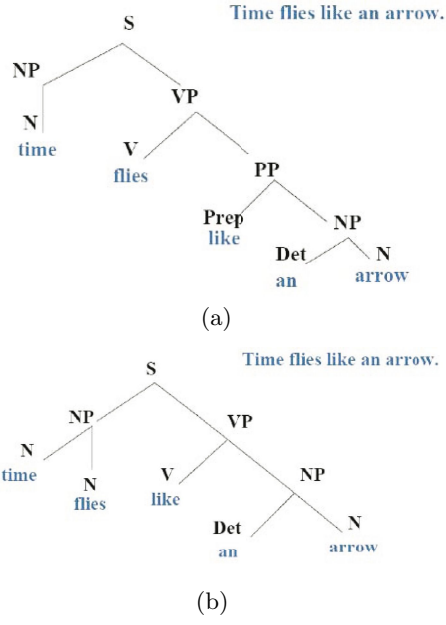


Fig. 1. Two alternative parsing trees of the same sentence. In this example, the ambiguity in parsing comes from the wrong assignments of different POS-tags in a previous stage of processing. Even with the correct POS assignment, syntactic trees can vary depending on the attachment of prepositional phrases and other factors.

The next step in our simplified NLP model is semantic processing. Issues here concern how to represent the meaning of a sentence, how to make linguistic inferences, as well as word-sense disambiguation (WSD). WSD is the problem of determining in which sense a word is used in a given context [34]. For example, consider the word *bass* that has two distinct senses: a type of fish and a tone of low frequency. In the two following sentences it is clear to a human which senses are used:

1. The bass part of the song is very moving.
2. I went fishing for some sea bass.

However, for machines WSD is a difficult task. Compared to POS tagging, which requires a fairly short context, WSD might involve much longer dependencies. Successful contemporary implementations of WSD use Kernel methods such as SVM trained on the SemCor knowledge base (which contains 352 texts). Most of the texts are annotated with POS, lemma, and WordNet synset. The performance is usually much worse than in POS tagging with precision around 75% for English [35,34]. Such low performance may suggest that contemporary linguistic representations developed for statistical classifiers are not adequate enough to model word senses.

One of the solutions is to use better structured input representations that incorporate relations between words such as the ones included in the WordNet [36].

This knowledge base, developed at Princeton University, addresses not only POS and synsets but also such relationships as synonymy/antonymy, meronymy/holonymy (part/whole), hypernymy/hyponym (super and subclasses). While WordNet describes possible word meanings by corresponding synsets, example sentences, and a rich set of relations there is still a need to automatically identify meaning in a given context. There have been several attempts to systematically analyze meaning of words, for example, using argument structure. Levin [37] proposed that verbs' semantic classes correlate with their syntactic and morphological structure. This allowed her to classify verbs in groups such as Put Verbs (mount, place, put) or Correspond Verbs (agree, argue, clash, collaborate, communicate, etc.).

However, more detailed examination of Levin's classes revealed that better classification should be at least partially semantically motivated. This started the FrameNet project at Berkeley University. In FrameNet, not only verbs but also other POSs are assigned role frames. The FrameNet lexical database currently contains more than 10,000 lexical units (e.g. "traffic light", "take care of", "by the way"), more than 6,100 of which are fully annotated, in more than 825 semantic frames, exemplified in more than 135,000 annotated sentences. The basic idea of FrameNet is that one cannot fully understand the meaning of a single word without access to all the essential knowledge that relates to that word. For example, one would not be able to understand the word "sell" without knowing anything about the situation of commercial transfer, which also involves, among other things, a seller, a buyer, goods, money, the relation between the money and the goods, and the money and so on. Thus, a word activates, or evokes, a frame of semantic knowledge relating to the specific concept it refers to, or highlights, in frame semantic terminology.

Finally, we turn briefly to pragmatics, which is concerned with understanding the relationships between language and context. For example, an important aspect of this level of analysis is anaphora resolution. Simply put, anaphora resolution is concerned with the problem of resolving what a pronoun or a noun phrase refers to. For example, consider the following two cases:

1. John helped Mary. He was kind.
2. There were dresses of several different colors and styles. They were all pretty and labeled with price tags. Sally chose a blue one. Mary chose a skimpy one.

In case 1, "He" clearly refers to John. But to what does "one" refer in case 2? Humans have no problem understanding that the mentions of "one" in the third and forth sentences refer to "they" in the second sentence, which, in turn, refers to "dresses" in the first sentence. However, for a machine, the mentions of "one" could also have referred to "price tags" in the second sentence. For more complex computer communication, like blogs or on-line chat, anaphora resolution is even harder. Other forms of discourse analysis include understanding the discourse structure—i.e., what role does a sentence play in the discourse—and speaker turn-taking.

4 NLP in the Context of RE

NLP in the context of RE differs from general purpose NLP because the inputs and outputs are different, as illustrated in Figures 2 and 3. The result of the NLP front end should be a model of the requirements. Although there are a variety of notations and formalisms for requirements, we believe that the structure summarized in this section provides a useful reference model that is close to the mark. For a detailed description and examples see [38]. The requirements are most usefully conceptualized as a database containing structured information, or an instance of an object model of the requirements rather than as a text document. Abstractly, the requirements database consists of:

1. *Problem ontology,* which is called an environment model in [38]. This provides an unambiguous vocabulary for defining the requirements: each symbol denotes a unique concept with a well-defined meaning. Although any distinct symbols will do for mathematical analysis or processing by software, compound symbols composed of multiple words are often used to enhance human understanding. For example, the two word senses in the example in section 3 could be denoted by the symbols *bass_fish* and *bass_tone*. In our specific framework, a concept can be a type, relationship, attribute, or constant (distinguished instance of a type). Related concepts are generally grouped into modules, often related to types, and are subject to specialization and multiple inheritance that combines constraints by conjunction. Meaning of concepts is described by associated natural language texts, logical formulae, real-world measurement processes, or links to other defining documents. In particular, concepts can be uniquely mapped into symbols of a typed logic or other formalism to support further analysis, transformation, and simplification. Concepts correspond roughly to the semantic frames mentioned in the previous section, although in this context they are domain-specific and sometimes application specific. New specializations of previously known concepts are often acquired as part of RE. There appears to be a relatively small set of core concepts related to typical RE processes and common properties of problem domains for which software solutions are desired. Approximately 140 such concepts are identified in [38]. This number is small enough to suggest that special case methods for recognizing them may be affordable.

2. *Requirements hierarchy.* Each node in the hierarchy represents a requirement, which is a constraint that the proposed system will have to satisfy. Nodes can have many views, such as natural language descriptions, diagrams, mathematical formulae, etc. Higher level nodes are more abstract and may leave many details unspecified. Lower level nodes refine the meaning of their parent node by specifying additional details related to the parent requirement. Thus the hierarchy is a representation that supports and documents the process of resolving ambiguities and imprecise statements. The representation supports a process of iterative refinement that gradually sharpens the intended meaning of a stakeholders' statements and reduces ambiguity. This

sharpening of meaning goes beyond NLP processes that seek to determine which of several possible interpretations is the correct one for a given piece of text. It also involves requirements validation processes, such as prototyping, which help stakeholders understand the implications of their choices. This will help them finalize and sometimes reformulate their decisions. This process is currently carried out by human experts. In a completed hierarchy, leaf nodes are defined in terms of the vocabulary of the problem ontology, and are unambiguous in the sense that they do not contain references to undefined concepts. If the requirements are to be used as the basis for automated testing of the system under development, then the concepts used in the requirements must all be measurable or computable from measurable concepts. Achieving this level of clarity with high confidence of validity is the Holy Grail of RE. Conversely, a typical recurrent nightmare is the possibility of a catastrophic system failure due to failure to discover a critical unstated requirement.

3. *System model.* Later stages of requirements engineering generally produce a model of the proposed system at some level of detail. At a minimum, interfaces and externally visible behavior of the proposed system must be modeled, along with its interactions with its context: the (human) stakeholders of the proposed system and external systems it communicates with. There are a variety of notations for this type of model, including use cases, UML, many formal modeling languages, as well as architecture description languages.

We conjecture that the structures identified above and the associated processes can be exploited to improve all aspects of RE, including NLP applied to statements from stakeholders. For example, [38] identifies heuristics for eliciting missing needed information related to requirements. These heuristics can be represented as questions to the stakeholders that are linked to reusable concepts in the common core of the problem ontology. When statements from stakeholders are linked to such concepts, the associated questions are triggered. This structure can aid the associated NLP systems in the following ways:

1. Prior knowledge of the question that was asked can help NLP processes to correctly interpret the response by conditioning the probabilities of the the various possible interpretations;
2. Previously triggered problem-domain concepts can be linked to various domains in the ontologies by NLP processes, thereby conditioning the probabilities of other terms/senses associated with the domain. This kind of information should help with word-sense disambiguation as well as parsing.

The Worldwide Web Consortium's (W3C) efforts to enable the Semantic Web have resulted in developments such as OWL (Web Ontology Language)[2] and

[2] http://www.w3.org/2004/OWL/

RDF (Resource Description Framework)[3]. These tools hold potential in structuring the requirements hierarchy and providing a bridge between that and the system model. In particular, software object frameworks expressed using these may expose a richer semantic representation that could enable automated "reasoning" about model composition and support the preceding NLP processes for eliciting information from the stakeholder as well as the transformation of stakeholder information into a formal representation.

The requirements should serve both as guidance for system developers and as a reference standard on which system quality assurance is based. In highly automated processes that current software engineering research is seeking to enable, the information in the requirements should be sufficiently complete and precise to enable automatic generation of at least the software that can test a system implementation to determine whether or not it meets the requirements to within a given statistical confidence level. In some visions of model-based domain specific development, information in the requirements may also be used to directly generate parts of the deliverable code. Such code generation processes use models of domain-specific software structures, known as reference architectures, and sets of rules for tailoring known solution methods to specific problem characteristics extracted from the requirements. Both the reference architecture and the generation rules are constructed for each problem domain by skilled software designers.

In any case, the delivered system is unlikely to be any better than the requirements, reinforcing the mantra that accuracy of the requirements has great importance. Existing manual processes for deriving requirements from informal stakeholder statements therefore incorporate a variety of checking procedures that include reviews, storyboarding, simulation and prototype demonstration, dependency tracing, consistency checking, and many others. NLP in the context of RE must be integrated with such checking procedures to achieve needed accuracy.

Other processes that must be supported after formalization of the stakeholder input include detecting and resolving conflicts between needs of different stakeholders, finding errors of omission, and finding cases where different stakeholders may agree on the wording of a requirement but not on its meaning. This last case is significant in large systems because they typically involve stakeholders from a variety of different specializations and communities.

5 How NLP Can Be Improved in the Context of RE

Generic NLP, as illustrated in Figure 2, has only one set of inputs, the natural language text and the accompanying general linguistic resources. In the context of RE there should be additional information: identification of the source of the text, including the author's identity, role in the process, expertise areas, etc., as shown in Figure 3. There are also other sources of relevant information, including general-purpose information about requirements engineering processes, system

[3] http://www.w3.org/RDF/

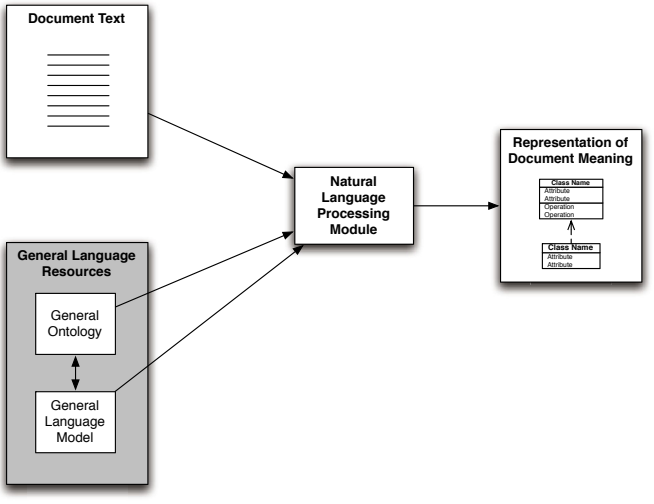


Fig. 2. Generic Natural Language Processing

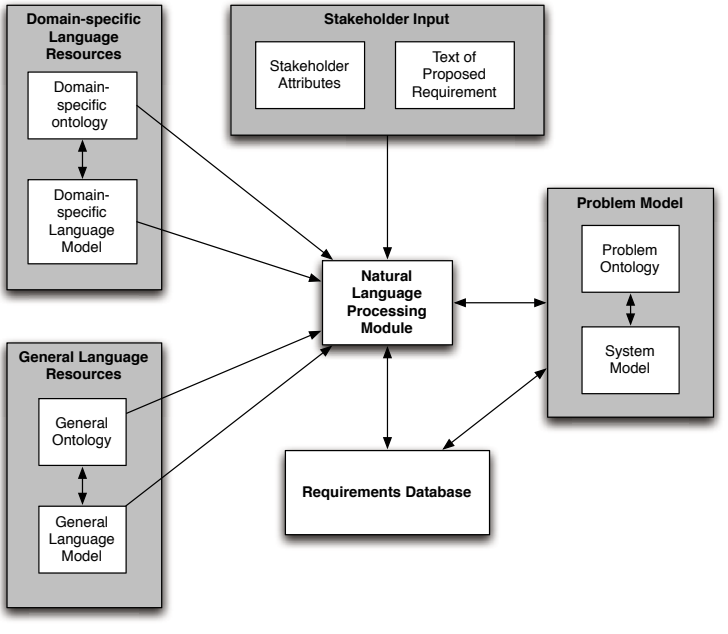


Fig. 3. Natural Language Processing for Requirements Engineering

development processes, and typical problem domain concepts and jargon as well as information about the kind of system to be developed in each particular project. All of this information can be used to limit the search space for the NLP, condition the probabilities of possible word senses, and provide models of

the context of the discourse that can provide the basis for judging likelihood of interpretations for much larger bits of text than individual words or phrases. This information can drive different post-processing that seeks to identify particular types of errors or just to identify and question the generated interpretations that have weak evidence. For example, the following ambiguous sentence from the example blog can be resolved only when we know that the person speaking is an airport security agent: *And people can't remain alert to rare events, so they slip by.* Viz, it is rare for someone to smuggle dangerous liquids in their carry-on luggage; consequently, it is difficult for screening agents to continually be alert, and the event goes unnoticed.

6 Conclusions

It appears that NLP is getting close to the point where it can contribute to requirements engineering, but it cannot do so in a vacuum. The results must be checked and reviewed, and existing methods must be improved by using more aspects of the context of the process to improve accuracy.

Even approximate NLP could facilitate text analysis and reduce workload by prioritizing documents, using context for effective search, making summaries, and classifying texts or their fragments even if accuracy of the process is insufficient to support requirements engineering based solely on the raw output of the NLP. The difference from fully automated processing is that NLP methods will typically give users several options and it will be their responsibility to select the right one. Thus currently the most safe and effective use of NLP is to integrate its methods with human processing as it is conceptualized in Human System Integration (HSI) framework. The value added would be that, the automated processing could identify some weaknesses that unaided humans might miss [39,40].

The issues that will determine whether or not NLP enters widespread use in requirements engineering are economic: it must cost less and produce more accurate results than corresponding manual processes that rely on human experts to interpret and model the raw statements from the stakeholders. This is a challenging goal that reaches beyond the traditional bounds of NLP to include social, organizational and psychological issues.

References

1. Luqi, Zhang, L., Berzins, V., Qiao, Y., Qiao, Y.: Documentation driven development for complex real-time systems. IEEE Transactions on Software Engineering 30(12), 936–952 (2004)
2. Luqi, Zhang, L.: Documentation driven development for complex systems. In: Proceedings of Workshop on Advances in Computer Science and Engineering, Berkeley, CA, pp. 141–170 (2006)
3. Stone, A., Sawyer, P.: Identifying tacit knowledge-based requirements. Software, IEE Proceedings 153(6), 211–218 (2006)
4. Kelly, D.F.: A software chasm: Software engineering and scientific computing. IEEE Software 24(6), 119–120 (2007)

5. Boehm, B.W.: Software Engineering Economics. Prentice Hall PTR, Upper Saddle River (1981)
6. Balzer, R.: A 15 year perspective on automatic programming. IEEE Transactions on Software Engineering SE-11(11), 1257–1268 (1985)
7. Heidorn, G.E.: English as a very high level language for simulation programming. In: Proceedings of the ACM SIGPLAN symposium on Very high level languages, pp. 91–100. ACM Press, New York (1974)
8. Martin, W.A.: Some comments on eqs, a near term natural language data base query system. In: ACM 1978: Proceedings of the 1978 annual conference, pp. 156–164. ACM Press, New York (1978)
9. Hammer, M., Howe, W.G., Kruskal, V.J., Wladawsky, I.: A very high level programming language for data processing applications. Commun. ACM 20(11), 832–840 (1977)
10. Lu, R., Jin, Z., Wan, R.: Requirement specification in pseudo-natural language in promis. In: Proceedings of the Nineteenth Annual InternationalComputer Software and Applications Conference, COMPSAC 1995, pp. 96–101 (1995)
11. Rolland, C., Proix, C.: A Natural Language Approach For Requirements Engineering. In: Loucopoulos, P. (ed.) CAiSE 1992. LNCS, vol. 593, pp. 257–277. Springer, Heidelberg (1992)
12. Kaindl, H.: The missing link in requirements engineering. SIGSOFT Softw. Eng. Notes 18(2), 30–39 (1993)
13. Ryan, K.: The role of natural language in requirements engineering. In: Proceedings of the IEEE International Symposium on Requirements Engineering (1993)
14. Nanduri, S., Rugaber, S.: Requirements validation via automated natural language parsing. In: Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, vol. 3, pp. 362–368 (1995)
15. Sleator, D., Temperley, D.: Parsing English with a link grammar. In: Proceeedings, Third International Workshop on Parsing Technologies, Tilburg, The Netherlands/ Durbuy, Belgium (1993)
16. Ohnishi, A.: CARD: An environment for software requirements definition. In: Proceedings of APSEC, Asia-Pacific Software Engineering Conference, pp. 420–429 (1995)
17. Ambriola, V., Gervasi, V.: Processing natural language requirements. In: 12th IEEE International Conference on Automated Software Engineering, pp. 36–45 (1997)
18. Michael, J.B., Ong, V.L., Rowe, N.C.: Natural-language processing support for developing policy-governed software systems. In: 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, TOOLS 39, pp. 263–274 (2001)
19. Denger, C., Berry, D.M., Kamsties, E.: Higher quality requirements specifications through natural language patterns. In: Proceedings of IEEE International Conference on Software: Science, Technology and Engineering, SwSTE 2003, pp. 80–90 (2003)
20. Durán, A., Ruiz, A., Bernárdez, B., Toro, M.: Verifying software requirements with xslt. SIGSOFT Softw. Eng. Notes 27(1), 39–44 (2002)
21. Lee, B.-S., Bryant, B.R.: Automation of software system development using natural language processing and two-level grammar. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 219–233. Springer, Heidelberg (2004)
22. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: FOSE 2007: 2007 Future of Software Engineering, Washington, DC, USA, pp. 285–303. IEEE Computer Society (2007)

23. Luqi, Kordon, F.: Advances in requirements engineering: Bridging the gap between stakeholders' needs and formal designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
24. Iwanska, L., Zadrozny, W.: Introduction to the special issue on context in natural language processing. Computational Intelligence 13(3), 301–308 (1997)
25. Hwa, R.: Supervised grammar induction using training data with limited constituent information. In: Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics, Morristown, NJ, USA, pp. 73–79. Association for Computational Linguistics (1999)
26. Steedman, M., Hwa, R., Clark, S., Osborne, M., Sarkar, A., Hockenmaier, J., Ruhlen, P., Baker, S., Crim, J.: Example selection for bootstrapping statistical parsers. In: NAACL 2003: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, pp. 157–164. Association for Computational Linguistics, Morristown, NJ, USA (2003)
27. Xi, C., Hwa, R.: A backoff model for bootstrapping resources for non-english languages. In: HLT 2005: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing, pp. 851–858. Association for Computational Linguistics, Morristown, NJ, USA (2005)
28. McClosky, D., Charniak, E., Johnson, M.: Effective self-training for parsing. In: Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, pp. 152–159. Association for Computational Linguistics, Morristown, NJ, USA (2006)
29. Steedman, M., Osborne, M., Sarkar, A., Clark, S., Hwa, R., Hockenmaier, J., Ruhlen, P., Baker, S., Crim, J.: Bootstrapping statistical parsers from small datasets. In: Proceedings of EACL 2003, Budapest, Hungary, pp. 331–338 (2003)
30. Charniak, E.: Statistical techniques for natural language parsing. AI Magazine 18(4), 33–44 (1997)
31. van Halteren, H., Zavrel, J., Daelemans, W.: Improving accuracy in wordclass tagging through combination of machine learning systems. In: Proceedings of the ANLP-NAACL, Seattle, Washington, Morgan Kaufman (2000)
32. Collins, M.: Three generative, lexicalized models for statistical parsing. In: Cohen, P.R., Wahlster, W. (eds.) Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics, Somerset, New Jersey, pp. 16–23. Association for Computational Linguistics (1997)
33. Collins, M.: Discriminative reranking for natural language parsing. In: ICML 2000: Proceedings of the Seventeenth International Conference on Machine Learning, pp. 175–182. Morgan Kaufmann Publishers Inc., San Francisco (2000)
34. Mihalcea, R.: Knowledge Based Methods for Word Sense Disambiguation. In: Word Sense Disambiguation: Algorithms, Applications, and Trends. Kluwer, Dordrecht (2006)
35. Mihalcea, R.: Unsupervised large-vocabulary word sense disambiguation with graph-based algorithms for sequence data labeling. In: HLT 2005: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing, pp. 411–418. Association for Computational Linguistics, Morristown, NJ, USA (2005)
36. Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.J.: Introduction to WordNet: An on-line lexical database. International Journal of Lexicography 3(4), 235–244 (1990)

37. Levin, B.: English Verb Classes and Alternations: a preliminary investigation, Chicago and London. University of Chicago Press, Chicago (1993)
38. Berzins, V., Luqi: Software engineering with abstractions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1991)
39. Plummer, S.: Memorandum: Awareness of human-systems integration (HSI) in Air Force acquisitions (2000)
40. Blasch, E.: Assembling a distributed fused information-based human-computer cognitive decision making tool. Aerospace and Electronic Systems Magazine 15(5), 11–17 (2000)
41. Helbig, H.: Knowledge Representation and the Semantics of Natural Language. Springer, Berlin (2006)
42. Hartstrumpf, S.: Coreference resolution with syntactico-semantic rules and corpus statistics. In: Proceedings of the Fifth Computational Natural Language Learning Workshop (CoNLL-2001), Toulouse, France, pp. 137–144 (2001)
43. Jurafsky, D., Martin, J.H.: Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall PTR, Upper Saddle River (2000)
44. Manning, C.D., Schütze, H.: Foundations of Statistical Natural Language Processing, Massachusetts. The MIT Press, Cambridge (1999)
45. Luqi: Transforming documents to evolve high-confidence systems. In: Proceedings of Workshop on Advances in Computer Science and Engineering, Berkeley, CA, pp. 71–72 (2006)
46. Erk, K., Pado, S.: Shalmaneser - a flexible toolbox for semantic role assignment. In: Proceedings of LREC 2006, Genoa, Italy (2006)
47. Ge, N., Hale, J., Charniak, E.: A statistical approach to anaphora resolution. In: Proceedings of the Sixth Workshop on Very Large Corpora, pp. 161–170 (1998)
48. Merialdo, B.: Tagging english text with a probabilistic model. Computational Linguistics 20(2), 155–171 (1994)
49. Kupiec, J.: Robust part-of-speech tagging using a hidden markov model. Computer Speech and Language 6, 225–242 (1992)
50. Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of english: The penn treebank. Computational Linguistics 19(2), 313–330 (1994)
51. Hwa, R., Osborne, M., Sarkar, A., Steedman, M.: Corrected co-training for statistical parsers. In: Proceedings of the Workshop on the Continuum from Labeled to Unlabeled Data in Machine Learning and Data Mining, International Conference of Machine Learning, Washington, DC (2003)
52. Lee, B.S.: Automated conversion from a requirements document to an executable formal specification. In: Proceedings of the 16th Annual International Conference on Automated Software Engineering, ASE 2001, p. 437 (2001)
53. Ruth, G.R.: Automatic programming: Automating the software system development process. In: ACM 1977: Proceedings of the 1977 annual conference, pp. 174–180. ACM Press, New York (1977)

Logic-Based Regulatory Conformance Checking

Nikhil Dinesh, Aravind K. Joshi, Insup Lee, and Oleg Sokolsky

Department of Computer Science
University of Pennsylvania
Philadelphia, PA - 19104, USA
{nikhild,joshi,lee,sokolsky}@seas.upenn.edu

Abstract. In this paper, we describe an approach to formally assess whether an organization conforms to a body of regulation. Conformance is cast as a model checking question where the regulation is represented in a logic that is evaluated against an abstract model representing the operations of an organization. Regulatory bases are large and complex, and the long term goal of our work is to be able to use natural language processing (NLP) to assist in the translation of regulation to logic.

We argue that the translation of regulation to logic should proceed one sentence at a time. A challenge in taking this approach arises from the fact that sentences is regulation often refer to others. We motivate the need for a formal representation of regulation to accomodate references between statements. We briefy describe a logic in which statements can refer to and reason about others. We then discuss preliminary work on using NLP to assist in the translation of regulatory sentences into logic.

1 Introduction

Regulations, laws and policies that affect many aspects of our lives are represented predominantly as documents in natural language. For example, the Food and Drug Administration's Code of Federal Regulations[1] (FDA CFR) governs the operations of American bloodbanks. The CFR is framed by experts in the field of medicine, and regulates the tests that need to be performed on donations of blood before they are used. In such safety-critical scenarios, it is desirable to assess formally whether an organization (bloodbank) conforms to the regulation (CFR).

Conformance checking is a relatively new problem in requirements engineering, which has been gaining attention in industry and academia [1]. A key difference between regulations and other sources of informal requirements is in determining the source of a requirement. The requirements used to design a system often arise from varied places, such as interviews with customers and discussions with domain experts. This makes the identification of requirements a difficult problem. However, since there are consequences associated with disobeying the law, law-makers spend considerable effort in articulating the requirements

[1] http://www.gpoaccess.gov/cfr/index.html

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 147–160, 2008.

(as normative sentences). As a result, one can informally associate a requirement with a sentence or discourse.

The challenge in conformance checking is that the task of formalizing the requirements is difficult, due to the large size and complexity of regulations. The long term goal of our work is to use natural language processing (NLP) techniques to aid in the formalization of regulation. From the perspective of using NLP for requirements engineering, this area is epecially interesting due to the availability of large corpora of regulations that can serve as a test-bed for NLP techniques.

We approach the problem of formally determining conformance to regulation as a model-checking question. The regulation is translated to statements in a logic which are evaluated against a model representing the operations of an organization. The result of evaluation is either an affirmative answer to conformance, or a counterexample representing a subset of the operations of the organization and the specific law that is violated. A similar approach is adopted by several systems [1,2,3].

When a violation is detected, the problem could be in one of three places: (a) the organization's operations, (b) the regulation or (c) the translation of the regulation to the logic. To aid in determining the source of the problem, there needs to be a notion of correspondence between the sentences of regulation in natural language and logic. We attempt to maintain a correspondence by translating regulation to logic one sentence at a time. An added benefit of doing this is to be able to focus our NLP efforts at the sentence level.

In this paper, we discuss two related parts of our approach. The first part deals with the issue of designing a logic into which we can translate regulation one sentence at a time. The main difficulty that we encountered in doing this is the problem of *references to other laws*. A common phenomenon in regulatory texts is for sentences to function as conditions or exceptions to others. This function of sentences makes them dependent on others for their interpretation, and makes the translation to logic difficult. In Section 2, we argue (using examples and lexical occurence statistics) that a logic to represent regulation should provide mechanisms for statements to refer to others, and to make inferences from the sentences referred to.[2] In Section 3, we briefly describe the logic that we use to represent regulation.

In the second part of the paper (Section 4), we turn our attention to the problem of using NLP to assist in the translation of sentences of regulation into logic. Section 5 concludes.

2 The Problem of References to Other Laws

In this section, we argue that a logic to represent regulation should provide a mechanism for sentences to refer to others. The discussion is divided into two parts. In Section 2.1, we discuss examples of the phenomenon that we are

[2] A study in [1] suggests that such references between sentences are common in privacy regulation as well.

interested in and how they may be represented in a logic with no mechanism for sentences to refer to others. We then contrast the distribution of some lexical categories in the CFR with newspaper text, which suggest that references to sentences are an important way of expressing relationships between sentences in regulation (Section 2.2).

2.1 Examples

The examples in this section are shortened versions of sentences from the CFR Section 610.40, which we will use through the course of the paper. Consider the following sentences:

(1) Except as specified in (2), every donation of blood or blood component must be tested for evidence of infection due to Hepatitis B.
(2) You are not required to test donations of source plasma for evidence of infection due to Hepatitis B.

(1) conveys an obligation to test donations of blood or blood component for Hepatitis B, and (2) conveys a permission not to test a donation of source plasma (a blood component) for Hepatitis B. To assess an organization's conformance to (1) and (2), it suffices to check whether "all non-source plasma donations are tested for Hepatitis B". In other words, (1) and (2) imply the following obligation:

(3) Every non-source plasma donation must be tested for evidence of infection due to Hepatitis B.

There are a variety of logics in which one can capture the interpretation of (3), as needed for conformance. For example, in first-order logic, one can write $\forall x : (d(x) \wedge \neg sp(x)) \Rightarrow test(x)$, where $d(x)$ is true iff x is a donation, $sp(x)$ is true iff x is a source plasma donation, and $test(x)$ is true iff x is tested for Hepatitis B. Thus, to represent (1) and (2) formally, we inferred that they implied (3) and (3) could be represented more directly in a logic.

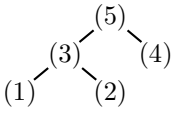
Now suppose we have a sentence that refers to (1):

(4) To test for Hepatitis B, you must use a screening test that the FDA has approved for such use.

The reference is more indirect here, but the interpretation is: "if (1) requires a test, then the test must be performed using an appropriate screening test kit". A bloodbank is not prevented from using a different kind of test for source plasma donations. (4) can be represented by first producing (3), and then inferring that (3) and (4) imply the following:

(5) Every non-source plasma donation must be tested for evidence of infection due to Hepatitis B using a screening test that the FDA has approved for such use.

It is easy to represent the interpretation of (5) directly in a logic. However, (5) has a complex relationship to the sentences from which it was derived, i.e., (1), (2) and (4). The derivation takes the form of a tree:



The examples we have considered are simplified versions of the sentences in the CFR 610.40. In the CFR, (1) has a total of six exceptions, and the exceptions have statements that qualify them further. This process of producing a derived obligation and translating it becomes extremely difficult.

References to other laws are not always hierarchical or acyclic. There are two kinds of circularities that can arise. The first is a syntactic circularity which arises due to vague references. For example, two occurences of the phrase "required testing under this section" can give rise to a cycle if one interprets "this section" as "all the other sentences in this section". However, such phrases typically appear in paragraphs where no tests are required and the cycle can be broken by restricting the references to paragraphs where tests are required. The second kind of circularity is a semantic circularity which can make the regulation paradoxical, e.g., with self referential sentences. Fortunately, we have not observed such circularities.

To summarize, if one wishes to use a logic with no support for referring to other sentences, translating regulation to the logic would involve the following steps: (a) resolving circularites to construct a hierarchy of references, (b) creating derived obligations by moving up the hierarchy, until a set of derived obligations with no references are obtained, and (c) translating the final set of derived obligations to logic.

This procedure would not be problematic if there are few cases of references. In the following section, we discuss the distribution of some lexical categories in the CFR which suggest that this is a very common case. This makes the procedure impractical in terms of the effort that would be involved. The logic that we describe in later sections lets one express references directly, and the resolution of cirularites and creation of derived obligations happen as part of the semantics.

2.2 Distribution of Lexical Categories

In the previous section, we saw several examples of how sentences in regulation refer to others. Natural language offers a variety of devices to relate sentences to others. A large class of such devices fall under the rubric of *anaphora*, which is a means of linking a sentence to the prior discourse. Common examples of such anaphoric items are pronouns and adverbial connectives, e.g., however, instead, furthermore, etc.[3]

[3] Not all uses of pronouns are anaphoric. Some pronouns are bound by quantifiers, e.g., *every one loves their mother*. We report counts based on occurence of strings and do not distinguinsh between different uses.

Table 1. Differences in the distribution of some anaphoric lexical items in the Wall Street Journal (WSJ) corpus and the CFR. Both the WSJ and the CFR have approximately 1M words.

Lexical Item	WSJ	CFR
he, she, him, her	8564	297
it, its	15168	2502
they, their	4500	862
ADV1	3162	2402
ADV2	2453	349
such	662	3028
References to other laws	_	18509

Table 1 contrasts the distribution of potentially anaphoric items in the Wall Street Journal (WSJ) corpus, with the CFR. The first three rows show counts of pronouns, and the CFR has a markedly lower number of pronouns than the WSJ. The next two rows show counts of adverbial connectives. ADV1 comprises of the connectives *also, however, in addition, otherwise, for example, therefore, previously, later, earlier, until* and *still*. These connectives have specialized uses in the CFR and tend to be quite frequent, with *otherwise* being the most frequent in the CFR (517 cases). ADV2 is a set of 48 adverbial connectives annotated by the Penn Discourse Treebank [4] excluding those in ADV1, e.g., *instead, as a result, nevertheless*. The connectives in ADV2 are significantly more frequent in the WSJ than in the CFR.

The last two rows in Table 1 show two common ways of establishing relationships between sentences in the CFR. The adjective *such* is a common way of refering to a set discussed in an immediately preceding law, e.g., *such tests*. The last row counts explicit references to other law, by searching for phrases like *this section*, or references to section and paragraph identifiers. Of the categories we considered this is by far the most frequent in the CFR.

We now describe the logic that we use to handle references. The other frequently occuring anaphora (ADV1 and *such*) are typically accompanied by references (e.g., *however* and *otherwise* give exceptions to other laws), and similar mechanisms can be used to express them. Formalizing the remaining anaphora is a subject of future work.

3 A Logic That Allows References between Laws

In this section, we describe the logic that we attempt to translate the regulation into. The description in this section is brief and informal, and introduces only the machinery needed to clarify the discussion in Section 4. We refer the reader to [5,6] for a formal account of the semantics and the computational issues. Consider our examples again:

(6) Except as specified in (7), every donation of blood or blood component must be tested for evidence of infection due to Hepatitis B.

(7) You are not required to test donations of source plasma for evidence of infection due to Hepatitis B.

(6) and (7) are represented as follows:

- 6.**o**: $d(x) \wedge \neg \text{by}_7(\neg\Diamond test(x)) \rightsquigarrow \Diamond test(x)$ and
- 7.**p**: $d(y) \wedge sp(y) \rightsquigarrow \neg\Diamond test(y)$

First, consider the formula 7.**p**: $d(y) \wedge sp(y) \rightsquigarrow \neg\Diamond test(y)$. This is read as "It is permitted that if y is a donation of source plasma, then it is not tested eventually". The letter **p** denotes permission, $d(y)$ asserts that y is a donation, $sp(y)$ asserts that y consists of source plasma, $test(y)$ asserts that y is tested, and $\Diamond$is the linear temporal logic (LTL) operator eventually. The connective $\rightsquigarrow$ is a variant of implication which we will discuss in what follows.

Now consider the subformula $\text{by}_7(\neg\Diamond test(x))$. This is read as "By the law (7), x is not tested eventually". We note that this subformula should hold iff y is a donation of source plasma. And finally, 6.**o**: $d(x) \wedge \neg \text{by}_7(\neg\Diamond test(x)) \rightsquigarrow \Diamond test(x)$ can be paraphrased as "It is obligated that if x is a donation and it is not the case (7) doesn't permit that x is not tested eventually, then x must be tested eventually". The letter **o** denotes obligation. Formulas in the logic are evaluated with respect to sequences of states of an implementation (in a manner similar to LTL). Each state is associated with a set of objects and a way of evaluating predicates.

Table 2. A run and its annotations

Time	Objects	Predicates	Annotations
1	o_1	$d(o_1)$, $sp(o_1)$, $\neg test(o_1)$	2: $\neg\Diamond test(o_1)$
2	o_1	$d(o_1)$, $sp(o_1)$, $\neg test(o_1)$	2: $\neg\Diamond test(o_1)$
	o_2	$d(o_2)$, $\neg sp(o_2)$, $\neg test(o_2)$	1: $\Diamond test(o_2)$
3	o_1	$d(o_1)$, $sp(o_1)$, $test(o_1)$	2: $\neg\Diamond test(o_1)$
	o_2	$d(o_2)$, $\neg sp(o_2)$, $\neg test(o_2)$	1: $\Diamond test(o_2)$

Table 2 shows a possible run of a bloodbank. First, an object o_1 is entered into the system. o_1 is a donation of source plasma ($d(o_1)$ and $sp(o_1)$ are true). When a donation is added, its test predicate is initially false. Then, an object o_2 is added, which is a donation but not of source plasma. In the third step, the object o_1 is tested. Unless the run is extended to test o_2, the bloodbank doesn't conform to the statements (6) and (7). We now discuss how the annotations are arrived at, and used to assess the regulation.

We first evaluate 7.**p**: $d(y) \wedge sp(y) \rightsquigarrow \neg\Diamond test(y)$ with respect to all variable assignments. When y is assigned the value o_1, the precondition $d(y) \wedge sp(y)$ is true, and we *annotate* the state with 7: $\neg\Diamond test(o_1)$. This annotation happens regardless of whether $\neg\Diamond test(o_1)$ is true or false under the variable assignment.

Next, we evaluate 6.**o**: $d(x) \wedge \neg \text{by}_7(\neg\Diamond test(x)) \rightsquigarrow \Diamond test(x)$. When x is assigned the value o_1, $d(x)$ is true. To evaluate $\text{by}_7(\neg\Diamond test(x))$ we check if there

is an annotation (ψ) on the state such that $\psi \Rightarrow \neg\Diamond test(o_1)$ is valid, i.e., a theorem in LTL. Since, $\neg\Diamond test(o_1)$ is an annotation this is an appropriate candidate for ψ and we conclude that $\text{by}_7(\neg\Diamond test(x))$ is true. Hence the precondition $d(x) \wedge \neg\text{by}_7(\neg\Diamond test(x))$ is false, and the obligation is vacuously satisfied.

When considering a non-source plasma donation (o_2), no annotation is provided by 7.**p**: $d(y) \wedge sp(y) \rightsquigarrow \neg\Diamond test(y)$. We will not be able to find ψ such that $\psi \Rightarrow \neg\Diamond test(o_2)$ is valid. This will make the precondition $d(x) \wedge \neg\text{by}_7(\neg\Diamond test(x))$ true, and if a test is not performed eventually, a violation will be detected. Violations are detected only with respect to obligations. Permissons do not produce violations and are relevant to conformance only via references from an obligation.

Complexity: In [5], we show that conformance checking is hard for EXPTIME. The high complexity is due to the satisfiability tests that are needed to evaluate references. A case study of the FDA CFR motivated a restriction, called *the single copy property*, which allows us to compile out the satisfiability tests [6]. For acyclic regulations, the compilation procedure yields first-order temporal logic statements. Conformance checking with first-order logic is PSPACE-complete. However, the exponential factor is determined by the maximum predicate-arity, which tends to be small. [6] describes algorithms for checking conformance at runtime, and an evaluation using a prototype implementation.

Related Work: The logic describe here is a starting point in adding references to systemes such as [2,3]. [2] represents business contracts as SQL queries, and [3] uses first-order logic augmented with real time operators. References can be added to these systems, provided that the existential quantification is relativized to either the preconditions or the postconditions. However, restrictions are needed to ensure that the satisfiability tests remain decidable. [1] discusses the importance of anlayzing references, but do not provide a formalization.

4 NLP as an Aid in Formalizing Regulation

In this section, we discuss preliminary work on using natural language processing (NLP) to aid in creating a logic-based representation of regulation. We emphasize that we use NLP purely as an assistive technology and do not attempt to replace the human user.[4]

We approach the problem using the supervised learning methodology, which has been used for a variety of tasks in NLP, e.g., parsing, computing predicate-argument structure, named-entity recognition etc. The supervised learning methodology proceeds as follows:

1. Define a representation (logic) to be computed from the text
2. Manually describe/annotate how units of text correspond to units of the desired representation. The number of examples manually annotated depends on the needs of the application.
3. Train a (statistical) learning algorithm to compute the representation.

[4] The intended users of our system are designers of software in the organization being regulated.

Our focus upto this point has been on Steps 1 and 2, i.e., designing the logic and formulating an annotation scheme to associate natural language and logic. In order for the methodology to be successful, it should be possible for a human to describe how she went from natural language to logic. Such a description would take the form of an annotation guideline. For example, [7] gives guidelines for annotating phrase structure on sentences, and [8] gives guidelines for annotating discourse relations. The process of formulating guidelines is typically one of iterative refinement. We begin by fixing a representation, and then annotating a few sentences with this representation. The problematic cases are analysed, resulting in revisions to the guidelines, and the process repeats.

We have made three annotation passes over 100 sentences and are in the process of refining guidelines. The rest of this section describes what we are attempting to annotate and the difficulties encountered. In Section 4.1, we decompose the annotation process into three steps. Some of the key problems encountered are discussed in Sections 4.2 and 4.3. Section 4.4 discusses related work. A preliminary discussion of our approach appears in [9]. We have since extended the logic and annotation guidelines.

4.1 Translating Regulatory Sentences to Logic

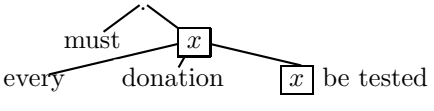
Many logics are semantically adequate for the application of conformance checking. However, to be able to describe or annotate how a statement in logic is obtained from natural language, the logic and natural language need to be syntactically isomorphic. (8) and (9) are examples of what we mean by syntactically isomorphic statements in natural language and logic.

(8) Every donation must be tested

(9) 8.**o**: $donation(x) \rightsquigarrow tested(x)$

We note that predicates such as $tested(x)$ need to be refined to accomodate references between laws, and we discuss this issue in what follows. We now sketch the procedure for associating (8) and (9).

First, (8) is mapped to the *abstract syntax tree* (AST):



The AST is obtained from (8) by moving the modal *must*, followed by moving the phrase *every donation* to the front of the sentence. While moving a word or phrase, a variable is optionally left behind as a placeholder.

The second step is to associate leaf nodes which are not the leftmost child of their parent with components of the formula. *donation* is associated with the predicate $donation(x)$, and *x must be tested* is associated with the predicate $tested(x)$.

Given the associations for non-leftmost leaves, the leftmost leaves are associated with operations that combine the associations of their siblings to create an

association for the parent. *every* is associated with an operation that combines the associations of its siblings to associate $donation(x) \rightsquigarrow tested(x)$ with its parent. Finally, *must* is associated with an operation that takes $donation(x) \rightsquigarrow tested(x)$ and associates 8.**o**: $donation(x) \rightsquigarrow tested(x)$ with its parent. This procedure for translating natural language to logic can be broken into three steps:

1. Converting a sentence to an AST
2. Associating the non-leftmost leaves of the AST with components of the logic
3. Associating the leftmost leaves with combination operations

This decomposition of the problem is the one adopted (modulo terminology) in theoretical linguistics [10]. Of these steps, our goal is to achieve automation with a good level of accuracy for Step 1. For Steps 2 and 3, we can only envision partial automation in the immediate future. The goal is to design appropriate interfaces to assist the user in performing these steps. We now discuss some of the challenges in associating regulatory sentences with ASTs (Section 4.2). We then turn to a discussion of some issues related to Steps 2 and 3 in Section 4.3.

4.2 Annotating Sentences with ASTs

The AST produced from a sentence is a resolution of scope ambiguities. The sentence (8) above is simple in comparison to the sentences that one encounters in regulatory text, where a sentence has multiple noun phrases and modalities. Consider the following sentence from CFR 610.1 (the AST is shown in Figure 1):

(10) No lot of any licensed product shall be released by the manufacturer prior to the completion of tests for conformity with standards applicable to such product.

(11) 10.**o**: $licensedProduct(x) \wedge lotOf(y,x) \wedge priorTo(\varphi) \wedge manufacturer(z) \rightsquigarrow \neg releasedBy(y,z)$

For simplicity, we omit some details from (11). The phrase *the completion of tests for conformity with standards applicable to such product* involves a reference to other laws, i.e., the applicable standards appear in various places in Part 610. The subformula $priorTo(\varphi)$ in (11) can be formalized using a variant of the technique discussed in Section 3. We now discuss some issues related to (10), (11) and the AST in Figure 1.

Consider again the phrase *the completion of tests for conformity with standards applicable to such product*. While we can give this phrase an internal stucture in the AST, we do not know how to associate it structurally to its formal interpretation. In other words, from the perspective of translation, the phrase has to be treated as an idiom of sorts. In annotating a sentence with its AST, we give such phrases an internal structure and leave the problem of treating it as an idiom to subsequent steps.

Another issue is the question of what to move. In many linguistic theories, only quantificational noun phrases, e.g., *any product*, are treated as candidates

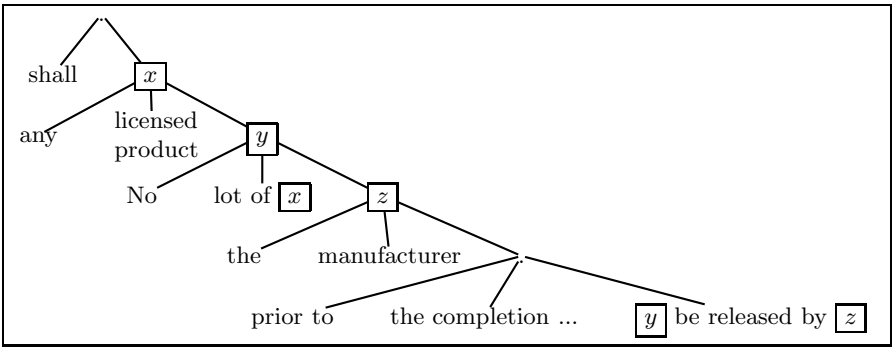


Fig. 1. AST for (10). The structure for the noun phrase *the completion of tests for conformity with standards applicable to such product* is not shown.

for movement. In our annotation scheme, the constructs that are moved are noun phrases, coordinated and subordinated phrases/clauses, relative clauses, and some modals and adverbs. This lets use describe the scopal interaction of these constructions without having to construct a separate phrase structure tree, thus saving annotation effort.

A difficulty in annotating ASTs is that there are many constructions in natural language which we do not know how to formalize. Consider the following statement:

(12) You must perform one or more such tests, as necessary, to reduce adequately and appropriately the risk of transmission of communciable disease.

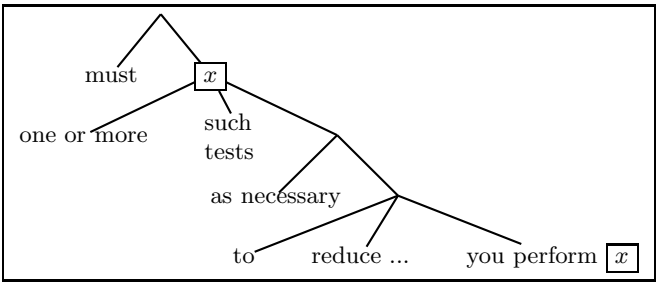


Fig. 2. AST for (12). The structure for the clause *reduce adequately and appropriately the risk of transmission of communciable disease* is not shown.

In (12) it is unclear how to order *one or more such tests*, *as necessary* and *to reduce adequately and appropriately the risk of transmission of communciable disease*. The intended interpretation of this sentence (which is also unclear) is "if a test is required, then it must be performed repeatedly until a conclusive

result is obtained".[5] In such cases, we construct an AST following the surface order of the phrases, as shown in Figure 2.

The point to take from this discussion is that not all the structure provided in an AST can be mapped directly to logic. On occasion one has to "undo" some of the movements in order to perform the association. An analogous situation arises in the problem of alignment in machine translation (between natural languages). One cannot always find a syntactically isomorphic translation of an English sentence into French. Certain constructions have to be treated idiomatically. In translating to logic, the number of constructions that we have to treat idiomatically give us a way to evaluate the syntactic expressive power of the logic. If there are many such constructions, it would suggest that the logic needs to be extended. We now discuss issues in associating the leaves of the AST with formulas in logic.

4.3 Associating the Leaves of ASTs with Logic

In Section 4.1, we gave the AST for the sentence "every donation must be tested". The leaf node "x be tested" was associated with the predicate $tested(x)$. In order to accomodate references between laws, we need to be able to infer, for example, that "if no tests are required, then a test for Hepatitis B is not required". Such inferences would not succeed if we used predicates such as $tested(x)$ and $testedForHepatitisB(x)$ (we need $\neg tested(x) \Rightarrow \neg testedForHepatitisB(x)$ to be valid). To handle such cases, we approach the definition of the set of predicates in two steps, which we describe below.

The first step is creating a schema. A schema is a set of class definitions. A class definition consists of a set of attribute definitions, and an attribute definition is a name associated with a type. The types of attributes are taken to be either atomic values (numbers or strings), references or sets of references. For example, the class *Donation* has an attribute named *tests* which is a set of references to objects in the class *Test*.

Predicates are treated as assertions over instances of the schema and are defined using a description logic. The logic that we use combines graded modal logic (modal logic with counting quantifiers), and hybrid logic (which allows one to refer to particular objects). The predicate "$testedForHepatitisB(x)$" is formalized as: $@_x(\exists tests : (purpose = \text{Hepatitis B}))$, read as "at the object referred to by x, there is an object referenced in the *tests* attribute and the *purpose* attribute of the test object has the value "Hepatitis B".

Given a set of documents, there are many ways one could create a schema, depending on domain knowledge and taste. Designing NLP-based interfaces to aid in the extraction of schemas has been explored in the past [11,12]. Our goal is to adapt previous work to the regulatory domain. Both creating the schema and defining the predicates will require significant manual intervention.

The key challenge in translating natural language to logic (for our application) is being able to decompose the problem into steps that depend mostly on

[5] The intended interpretation was clarified in a memo released by the FDA.

the text, and steps that depend mostly on domain knowledge. We believe that the computation of ASTs and the creation of schemas can be tied closely to the text, and as more documents are formalized better accuracy can be achieved. The problem of creating predicate definitions would benefit from further decomposition, and to our knowledge, this is an open problem.

4.4 Related Work

The problem of translating natural language to logic has received much attention in theoretical linguistics [13]. There are many problems both in the design of logic and the translation procedure that have yet to be resolved. More recently, there have been efforts in NLP to automate this translation for some applications.

[14,15] show that a good degree of automation can be achieved when the text is constrained. The sentences considered are queries to geographical database, e.g., "Which states does the river Delaware run through?". The specific corpus considered associates each sentence to logic. The associations between components of a sentence and logic are computed during the learning phase. While this approach reduces the annotation effort, the inference of associations during the learning step becomes more difficult.

[16] describes a corpus annotated using a manually crafted Head-driven Phrase Structure Grammar (HPSG). In addition to parse trees, a translation to logic is associated. The logic produced is similar in spirit to the ASTs that we annotate. We do not adopt this approach directly for two reasons. First, the annotation of ASTs avoids the overhead of creating phrase structure trees. Second, the logic produced in [16] introduces a large number of predicates (approximately one per word), and this makes the formulas large and difficult to refine. The leaves of the AST are typically phrases, and we have found in case studies that it is easier to define predicates at this level of granularity.

[17] discusses an approach to computing wide-coverage semantic interpretation. The goal is to be able to produce approximate translations in first-order logic and carry out inferences. Similar problems arise in the definition of predicates. The envisioned applications are those for which some errors in the logic produced are tolerable. For our application, while it may be impossible to avoid errors, the goal is to provide a correct translation of a sentence. This involves a careful analysis of modalities, which is not possible in current wide-coverage techniques.

5 Conclusions and Future Work

We have motivated the need for a formal representation of regulation to accomodate references between laws (Section 2). We described, in Section 3, a logic that accomodates certain kinds of references, i.e., those appearing in preconditions. There is also the need for reference in postconditions, to express naturally cases where one law cancels obligations and permissions given by another. We are currently working on extending the logic to allow such references.

In Section 4, we described preliminary work on using NLP to assist in creating the formal representation of regulation. In NLP, the focus has been on computing information tied to the surface structure of the sentence, such as parse trees and predicate-argument structure. However, in formalizing requirements, we are often interested in inferences drawn from sentences and the context. Relating these inferences back to the surface structure of a sentence poses interesting challenges to both NLP and formal methods.

We have focussed entirely on regulatory requirements in this paper, and designed machinery to accomodate its peculiarities. The logic that we have developed is useful for expressing rules with a large number of exceptions. Since the logic and the annotation of ASTs are not independent, there will be challenges in adapting the approach to different kinds of requirements. A particularly challenging aspect is the large number of modalities in natural language. In the regulatory texts that we have examined, *time* and *obligation* are the salient modalities, but this may not be the case for other kinds of requirements. A study of requirements in different domains is a topic for further research.

References

1. Breaux, T.D., Vail, M.W., Anton, A.I.: Towards regulatory compliance: Extracting rights and obligations to align requirements with regulations. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (2006)
2. Abrahams, A.: Developing and Executing Electronic Commerce Applications with Occurrences. PhD thesis, Univeristy of Cambridge (2002)
3. Giblin, C., Liu, A., Muller, S., Pfitzmann, B., Zhou, X.: Regulations Expressed as Logical Models (REALM). In: Moens, M.-F., Spyns, P. (eds.) Legal Knowledge and Information Systems (2005)
4. Miltsakaki, E., Prasad, R., Joshi, A., Webber, B.: The Penn Discourse Treebank. In: LREC (2004)
5. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: Reasoning about conditions and exceptions to laws in regulatory conformance checking. (in submission) (2008), `http://www.cis.upenn.edu/~nikhild/reasoning.pdf`
6. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: Checking traces for regulatory conformance. In: Proceedings of the Workshop on Runtime Verification (to appear, 2008)
7. Bies, A., Ferguson, M., Katz, K., MacIntyre, R.: Bracketing guidelines for Treebank II style Penn Treebank Project (1995), `ftp://ftp.cis.upenn.edu/pub/treebank/doc/manual/root.ps.gz`
8. The PDTB Group: The Penn Discourse Treebank 1.0 Annotation Manual. Technical Report IRCS-06-01, IRCS (2006)
9. Dinesh, N., Joshi, A.K., Lee, I., Webber, B.: Extracting formal specifications from natural language regulatory documents. In: Proceedings of the Fifth International Workshop on Inference in Computational Semantics (2006)
10. May, R.: Logical Form: Its structure and derivation. MIT Press, Cambridge (1985)
11. Overmeyer, S.P., Lavoie, B., Rambow, O.: Conceptual modeling through linguistic analysis using lida. In: 23rd International conference on Software Engineering, pp. 401–410 (2001)

12. Bryant, B.R.: Object-oriented natural language requirements specification. In: ACSC 2000, The 23rd Australasian Computer Science Conference (January 2000)
13. Heim, I., Kratzer, A.: Semantics in Generative Grammar. Blackwell, Malden (1998)
14. Zettlemoyer, L.S., Collins, M.: Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In: Proceedings of UAI (2005)
15. Wong, Y.W., Mooney, R.J.: Learning synchronous grammars for semantic parsing with lambda calculus. In: Proceedings of ACL (2007)
16. Oepen, S., Flickinger, D., Toutanova, K., Manning, C.: LinGO Redwoods: A rich dynamic treebank for HPSG. In: Proceedings of the workshop on treebanks and linguistic theories (2002)
17. Bos, J., Clark, S., Steedman, M., Curran, J.R., Hockenmaier, J.: Wide-coverage semantic representations from a CCG parser. In: Proceedings of COLING (2004)

On the Identification of Goals in Stakeholders' Dialogs

Leonid Kof

Fakultät für Informatik, Technische Universität München,
Boltzmannstr. 3, D-85748 Garching bei München, Germany
kof@informatik.tu-muenchen.de

Abstract. Contradictions in requirements are inevitable in early project stages. To resolve these contradictions, it is necessary to know the rationale (goals) that lead to the particular requirements. In early project stages one stakeholder rarely knows the goals of the others. Sometimes the stakeholders cannot explicitly state even their own goals. Thus, the goals have to be elaborated in the process of requirements elicitation and negotiation.

This paper shows how the goals can be derived by systematic analysis of stakeholders dialogs. The derived goals have to be presented to the stakeholders for validation. Then, when the goals are explicitly stated and validated, it becomes easier to resolve requirements contradictions.

1 Introduction

When a complex system is developed, there exist many different stakeholders or stakeholder groups whose interests should be taken into account. For example, if we build a drive-by-wire system, the obvious stakeholders would be the car manufacturer itself (OEM, original equipment manufacturer), prospective drivers, service staff, and the legislator. Every stakeholder has his own goals. These goals can be conflicting. For example, one of the OEM's goals may be cost reduction. Cost reduction can be achieved, for example, by reducing the brake system to the rear wheels only. This would conflict with the legislator's goal to provide road safety.

The above example of goal conflict should be obviously resolved before the goals are refined. In the case that the goal conflicts are less apparent, the goals could be refined to finer subgoals, before the conflict becomes apparent. For example, the legislator's goal to provide road safety does not conflict with the road maintainer's goal to minimize wearing down of the road surface by the cars. However, reasonable refinements of these goals can become conflicting: "provide road safety" can be refined to "use caterpillars instead of wheels", whereas "minimize wearing down of the road surface" can be refined to "prohibit caterpillars". This conflict cannot be resolved, unless we retreat to the original goals and look for alternative goal refinements.

The problem of conflicting goal refinements is not really a problem, as long as every stakeholder can explicitly state his top-level goals. The normal project situation is, however, that stakeholders themselves have rather vague ideas about their own goals. In this case they can intuitively identify requirements that are problematic (i.e., conflicting with their goals). Conflict resolution, however, results in looking for a requirements set satisfying every stakeholder. This can be a rather tedious business, particularly when goals are not explicitly specified.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 161–181, 2008.

To facilitate the whole requirements engineering process, it is important to identify the stakeholder goals as early as possible. This paper discusses possibilities of goal identification on the basis of stakeholders' dialogues transcripts.

The remainder of the paper has 6 sections. Section 2 introduces the case study used to illustrate the approach. Section 3 gives an overview of goal-oriented requirements engineering, including rules of thumb to identify goals. Section 4 shows how the goals can be manually identified in the case study. Section 5 gives an overview of available approaches to natural language processing (NLP) and their applications to requirements engineering. Here, the idea is to use NLP for goal identification. Then, Section 6 shows how goal identification could be automated. Finally, Section 7 summarizes the whole paper.

2 Case Study

The procedure for goal identification, presented in this paper, is evaluated on a small case study on an airport screening system. The case study is just a two-page document, representing an online stakeholder discussion [1]. This document does not contain any explicitly stated requirements. To give a flavor of the document, Table 1 presents the first three paragraphs of the document.

There are three stakeholders participating in the discussion: a representative of the Transportation Security Administration (TSA), a representative of the Federal Aviation Administration (FAA), and a representative of the airport screening and security staff. The case study represents a rather intense discussion, where none of the stakeholders explicitly states his goals. They all agree on the goal that air traffic security should be improved, but they see different problems and propose different solutions to the common goal. Altogether, each writes just 4-5 paragraphs, which is surely not enough to identify all requirements. However, the statements of every stakeholder are motivated by his goals, which makes the case study a good example to demonstrate goal extraction.

Table 1. Stakeholders' dialogue, excerpt

Federal Aviation Administration: We have to ban on airplane passengers taking liquids on board in order to increase security following the recent foiled United Kingdom terrorist plot. We are also working on technologies to screen for chemicals in liquids, backscatter, you know...
Airport Screening and Security: Technologies that could help might work well in a lab, but when you use it dozens of times daily screening everything from squeeze cheese to Channel No. 5 [sic] you get False Alarms... so it is not quite ready for deployment!
Federal Aviation Administration: Come on! Generating false positives helped us stay alive; maybe that wasn't a lion that your ancestor saw, but it was better to be safe than sorry. Anyway, I want you to be more alert - airport screeners routinely miss guns and knives packed in carry-on luggage.

3 Goal-Oriented Requirements Engineering

Software development involves different stakeholders, and conflicts among stakeholders are common. To resolve the conflicts, it is vital to know not only the position of

every stakeholder, but also the rationale for the position, originating from some goal. This idea is the basis for the Win-Win negotiation approach [2].

A *goal* in requirements engineering is "an objective the system under consideration should achieve" [3]. In order to satisfy some goals, cooperation of several active components, or *agents* can be required. For example, to achieve the goal "safe air transportation" it is necessary that the administrative authorities and the airport screening staff cooperate.

Goals can be refined to subgoals in two ways. There exist AND and OR refinements. If some goal is AND-refined to a set of subgoals, it is necessary to satisfy all the subgoals to satisfy the original goal. For example, the goal "safe air transportation" can be AND-refined to the goals "proper aircraft maintenance" and "no terrorists on board", which have both to be satisfied in oder to achieve "safe air transportation". If some goal is OR-refined to a set of subgoals, it is sufficient to satisfy one of the subgoals to satisfy the original goal. For example, the goal "no explosives in carry-on luggage" can be OR-refined to "do not allow any carry-on luggage" and "screen carry-on luggage".

To identify goals, two key questions can be applied: "WHY" and "HOW". An answer to a "HOW"-question for a goal gives a possible refinement of the goal. An answer to a "WHY"-question for a goal identifies its superior goals. For example, if we ask "WHY" the goal "screen carry-on luggage", we get that "there be no explosives in carry-on luggage" and, perhaps, that "there be no sharp items in carry-on luggage" and that "there be no liquids in carry-on luggage".

Apart form asking "WHY"- and "HOW" questions, there are two further way to identify goals:

- List the problems of the existing system. The negation of every problem becomes a goal of the system to be built.
- Look for goal-indicating expressions in the requirements document, like "purpose", "objective", "concern", "intent", "in order to", etc.

Van Lamsweerde provides a much more thorough introduction to goal-oriented requirements engineering [3].

4 Case Study: Manual Goal Identification

In the ideal world, every stakeholder could explicitly state his goals and identify contradictions to other stakeholders' goals. The small case study, treated in this paper, shows that this is not the case in the real world. In the stakeholders' dialog, the goals are mostly implicit, they manifest themselves in proposals that a stakeholder makes and in objections to the proposals made by others. For example, in the case study the FAA officer opens the discussion with the statement that "We have to ban on airplane passengers taking liquids on board *in order to increase security following the recent foiled United Kingdom terrorist plot*." In this sentence, a goal is explicitly stated, introduced by the phrase "in order to". The reaction to this statement shows the goal of the airport screening staff, rather indirectly: "Technologies that could help might work well in a lab, but when you use it dozens of times daily screening everything from squeeze cheese to Channel No. 5 [sic] you get False Alarms... *so it is not quite ready for deployment*!"

The actual goal is the application of screening techniques in day-to-day operation, not distinguishing squeeze cheese from explosives.

In the case study, we can identify the goals by asking the question for each statement, why the statement was made by its uttering stakeholder. In this way, we can identify the following goals of the stakeholders:

- Goals of the Federal Aviation Administration:
  improvement of security: "We have to ban on airplane passengers taking liquids on board in order to increase security following the recent foiled United Kingdom terrorist plot"
  effectiveness: "We are trying to federalize checkpoints and to bring in more manpower and technology"
- Goals of the Transportation Security Administration:
  improvement of security:
    pro-active thinking: "We have yet to take a significant pro-active step in preventing another attack everything to this point has been reactive"
    consistency in regulations: "I think that enforcing consistency in our regulations and especially in their application will be a good thing to do"
- Goals of the airport screening and security staff:
  application of the rules in everyday operation: "Technologies that could help might work well in a lab, ..., so it is not quite ready for deployment", "It's not easy to move 2 million passengers through U.S. airports daily"
  cost effectiveness for the airlines: "I mean an economic threat is also a threat"
  consistency in rules: "There are constant changes in screening rules - liquids/no liquids/3-1-1 rule"

These goals are not contradiction-free. By analyzing the document, it is possible to identify following contradictions:

- proactive thinking, which is a TSA goal, vs. cost effectiveness, which is an FAA goal. Actually, this is not necessarily a contradiction, but it sounds like a contradiction in the dialog.
- responsibility for the security checks: airlines become responsible, which is an FAA goal, vs. the authority currently performing the checks remains responsible.
- acceptability of false positives: acceptable for FAA, not acceptable for the screening staff

Probably due to the fact that each stakeholder considers his own goals as obvious, no one ever explicitly states them. Instead, each stakeholder presents solutions that seem adequate to him and explains why he thinks the solutions proposed by others are problematic. This observation about indirect goal statements will be used in Section 6 in order to systematize and potentially automate the identification of goals.

5 Natural Language Processing in Requirements Engineering

Traditionally, natural language processing is considered as taking place at four layers: lexical, syntactic, semantic, and pragmatic. Analysis tasks and result types for every kind of analysis are sketched in Table 2.

Table 2. Classification of text analysis techniques

Approach type	Analysis tasks	Analysis results
lexical	identify and validate the terms	set of terms used in the text
syntactic	identify and classify terms, build and validate a domain model	set of terms used in the text and a model of the system described in the text
semantic	build a semantic representation of every sentence	logical representation of every sentence, formulae
pragmatic	build a representation of the text, including links between sentences	logical representation of the whole text, formulae

For all layers except pragmatic there exist analysis techniques, either potentially automatable or already automated. Lexical techniques are the simplest. They consider each sentence as a character or word sequence, without taking further sentence structure into account. Due to this simplicity lexical techniques are extremely robust. The flip side of this robustness is that lexical methods are limited to pure term extraction. Syntactic approaches, as opposed to lexical ones, take also sentence structure into consideration. Based on this sentence structure, they extract not only the terminology, but also some domain model. Semantic approaches achieve more than the previous two classes: they produce a formal representation of the text. It is mostly a kind of first order predicate logic, but the concrete representation may differ. This task is surely very demanding, which poses severe limitations on the text for the approaches to work. As for pragmatics analysis, there is no automated procedure yet. There exist, however, a logic capable of capturing pragmatics-motivated relations between sentences.

The remainder of this section describes different kinds of text analysis approaches in more detail: Section 5.1 introduces the lexical approaches, Section 5.2 introduces the syntactic approaches and Section 5.3 introduces the semantic approaches. Section 5.4 presents the logic to capture pragmatic relations between sentences. Finally, Section 5.5 discusses the applicability of different kinds of analysis to goal identification.

5.1 Lexical Approaches: Analyzing the Document Vocabulary

The goal of lexical approaches is to identify concepts used in the requirements document. They do not classify the identified terms or build a domain model. The common feature of these techniques is that they analyze the document as only a sequence of characters or words. Berry [4] lists several approaches applying lexical techniques to requirements engineering. To give the flavor of lexical approaches, the following will be considered here: AbstFinder by Goldin and Berry [5], lexical affinities by Maarek and Berry [6] and documents comparison by Lecoeuche [7].

AbstFinder [5] works in the following way: it considers each sentence simply as a character sequence. Such character sequences are compared pairwise to find common subsequences. These subsequences are assumed to be potential domain concepts to be approved by the user. For example, consider two sentences taken from the steam boiler case study [8]:

> The steam-boiler is characterized by the following elements:

and

> Above m2 the steam-boiler would be in danger after five seconds, if the pumps continued to supply the steam-boiler with water without possibility to evacuate the steam.

The first sentence is shorter and it is augmented with spaces before the start of the search for common character subsequences. Then one of the sentences is rotated character-wise and for each rotated position AbstFinder controls whether there are aligned common subsequences. Rotation of the sentences is necessary to identify character chunks placed differently, like "flight" and "book" from "The flights are booked" and "He is booking a flight". (This example is taken from the AbstFinder article [5].) Such analysis is performed for all sentence pairs.

For the steam boiler example introduced above, the aligned position would look like

```
          The steam-boiler is characterized by...
Above m2 the steam-boiler would be in danger...
```

In this case AbstFinder would identify "the steam-boiler" as a concept contained in the document.

However, when considering two other sentences from the steam boiler specification, like

> Below m1 the steam-boiler would be in danger after five seconds, if the steam continued to come out at its maximum quantity without supply of water from the pumps

and

> Above m2 the steam-boiler would be in danger after five seconds, if the pumps continued to supply the steam-boiler with water without possibility to evacuate the steam

AbstFinder would identify "the steam-boiler would be in danger after five seconds, if the" as a common concept. This is not a concept that can be used to model the application domain. To decide which extracted sequences of characters really represent application-specific concepts, human analyst has to approve the extracted concepts.

The approach by Maarek [6] identifies concepts as word pairs where the appearances of these two words in the same sentence correlate. For example, "steam" and "boiler" often co-occur in the steam boiler specification [8], so this approach would identify "steam boiler" as an application concept.

Both Goldin and Berry and Maarek assume that the most important terms can be identified as the most frequent ones. Thus, they would miss an important term that is used only once, e.g. in the title or in the once-mentioned explanation of an acronym.

The approach by Lecoeuche [7] is more selective, in the sense that it not only extracts concepts, but also measures their importance and neglects concepts whose importance does not reach the manually set threshold. The approach compares the frequency of

the concept in the analyzed document with the frequency of the same concept in some baseline document. Let F_a be the number of occurrences of some term in the analyzed document and F_b the number of occurrences of the same concept in the baseline document. Then, the importance measure of a concept is defined as $imp = \frac{F_a}{F_a+F_b}$. High importance measure can imply that the concept is mentioned just few times in the baseline document (for example in the definitions), but is mentioned many times in the analyzed document. Concepts with a high importance measure are identified as application domain concepts.

Sawyer et al. [9] apply a similar idea to identify application domain concepts. The difference lies in the definition of the baseline documents: For the Lecoeuche's approach, the baseline document has to be provided by the user, whereas Sawyer et al. compare term frequency in the analyzed document with the term frequency in everyday usage. A term whose frequency in the analyzed document significantly differs from the frequency in everyday usage is considered as an important application domain concept.

It is easy to use lexical analysis to identify many potential goals. Van Lamsweerde suggests in [3], for example, to identify potential goals in requirements documents by means of certain key phrases, like "purpose", "objective", "concern", "intent", "in order to", etc. This technique can be used in our case study as well (cf. Section 6).

5.2 Syntactic Approaches: Identifying Terms and Relations

Syntactic approaches, presented in this section, promise more than pure vocabulary analysis. These approaches became widely known in the field of object-oriented analysis, as they allow for easy mapping of extracted concepts to classes, objects, attributes and methods. Some of these approaches do not offer any automation in their original versions, but they could be partially automated using linguistic techniques available now. Complete automation is still not possible, both due to low precision of the available tools and due to necessity to adapt the tools to every concrete document to analyze.

One of the first approaches aiming at analysis of specification texts is the one by Abbott [10]. The goal of Abbot's approach is to

> "... identify the data types, objects, operators and control structures by looking at the English words and phrases in the informal strategy"

Abbott takes the following types of words and phrases into consideration during model building:

- common nouns
- proper nouns and other forms of direct reference
- verbs and attributes

These word types are used in the following way during model building:[1]

1. A common noun in the informal strategy suggests a data type.
2. A proper noun or a direct reference suggests an object.

[1] This list and the examples are taken from Abbott's paper [10].

3. A verb, predicate or descriptive expression suggests an operator.
4. The control structures are implied in a straightforward way by the English.

This strategy works in the following way: given the specification text like

> If the two given DATEs are in the same MONTH, the NUMBER_OF_DAYS between them is the difference between their DAYs of MONTH,

Abbott identifies the common nouns (capitalized in the above example) as data types. A similar strategy is applicable to objects: in a phrase like

> Determine the number of days between THE_EARLIER_DATE to the end of its month. Keep track of this THAT_NUMBER in the variable called "DAY_COUNTER"

there are direct references "THE_EARLIER_DATE" and "THAT_NUMBER", marked by "the"/"that" and a proper noun "DAY_COUNTER". They are identified as program objects.

The third kind of concepts translated from text to program, the operators, are identified either as verbs or as attributes or descriptive expressions. For example, in the sentence

> If the two given dates ARE_IN_THE_SAME_MONTH, THE_NUMBER_OF_DAYS between them is the DIFFERENCE_BETWEEN their DAYS_OF_MONTH,

there is a predicate "ARE_IN_THE_SAME_MONTH" and descriptive expressions "THE_NUMBER_OF_DAYS", "DIFFERENCE_BETWEEN" and "DAYS_OF_MONTH", which become program operators.

Abbott's procedure gives some guidelines for translating the specification text into a program, but these guidelines are not completely automatable. Given a part-of-speech (POS) tagger, attaching a POS-tag to every word, it would be possible to identify nouns, verbs, etc. Such taggers were not available at the time Abbott wrote the paper but are available now. The precision of currently available taggers lies at about 97% [11,12]. Even the most precise tagger does not achieve a 100% precision and can become an error source.

A POS tagger would allow to identify common and proper nouns: We could say that a common noun is any word assigned the noun tag. Identification of proper nouns is a bit more complex. There exist approaches to recognize standard classes of proper names, i.e. names of people, places and organizations [13]. These approaches can also be transferred to other classes of proper names, as for example shown by Witte et al. [14] for programming concepts, i.e. variables, classes, and objects. However, to apply these techniques, it is necessary to manually define the set of domain-specific keywords. For example, Witte et al. introduce the keyword "variable" for variables and then recognize names like "variable X" as variable names. To apply Abbott's rules to the above example, we would have to manually define "counter" as a keyword. Then, we could identify "DAY_COUNTER" as well as other counters as program objects.

Abbott's third rule is really difficult to automate: Abbott himself gives examples of operators expressed by a verb, a noun phrase, or a prepositional phrase. However, he

does not provide guidelines how to distinguish phrases representing an operator from non-operator phrases.

Chen's method of building entity-relationship (ER) diagrams [15] is similar to Abbott's approach in that each maps natural language texts to application domain models. Chen defines a set of rules for translating English text to ER diagrams. The first two rules coincide with Abbott's ones:

1. A common noun corresponds to an entity type.
2. A transitive verb corresponds to a relationship type.

Other rules are specific to the ER-representation:

3. An adjective in English corresponds to an attribute of an entity in the ER-diagram.
4. An adverb in English corresponds to an attribute of a relationship in an ER-diagram.
8. The objects of algebraic or numeric operations can be considered as attributes.
9. A gerund in English corresponds to a relationship-converted entity type in ER-diagrams.

The remaining rules address firm expression patterns:

5. If the sentence has the form: "There are ...X in Y", we can convert it into the equivalent form "Y has ...X"
6. If the English sentence has the form "The X of Y is Z" and if Z is a proper noun, we may treat X as a relationship between Y and Z. In this case, both Y and Z represent entities.
7. If the English sentence has the form "The X of Y is Z" and if Z is not a proper noun, we may treat X as an attribute of Y. In this case, Y represents an entity (or a group of entities), and Z represents a value.

It is easy to see that Rules 1–4 and 8–9 are very similar to Abbott's rules. They just target at another representation form as Abbott's rules (ER-diagrams instead of Ada programs). Rules 5–7 create additional relations by analyzing firm expression patterns.

Saeki et al. [16] designed a tool aimed at automation of the approaches introduced above. They extract nouns and verbs from the text and build a noun table and a verb table. Then they select actions and action relations from the verb table. Although they aim at constructing an object-oriented model from a specification text, they do not perform any concept classification, which would yield a class hierarchy, but produce a flat model. An approach that performs not only concept extraction, but also classification, is presented below.

Ontology Building Technique: Syntactic text analysis techniques can be used to build an application domain ontology as well. In computer science, an ontology consists of a concept hierarchy, also called taxonomy, augmented with some more general, other than "is-a", relations. A taxonomy, in turn, consists of a term list and the "is-a"–relation, also called specialization or sub-typing. Thus, extraction of a domain-specific ontology consists of three basic steps:

1. term extraction
2. term clustering and taxonomy building, finding "is-a" relations
3. finding associations between extracted terms

These steps are explained below in detail.

Extraction of terms from requirements documents: To extract terms, each sentence is parsed and the resulting parse tree is decomposed. Noun phrases that are related to the verb of the sentence are extracted as domain concepts. For example, from the sentence "The control unit sends an alarm message in a critical situation" "send" is extracted as the main verb, "control unit" as the subject and "alarm message" as the direct object.

Term clustering: The second step clusters related concepts. Two concepts are considered as related and put into the same cluster if they occur in the same grammatical context. I.e., two terms are related in the following cases:

- They are subjects of the same verb.
- They are direct objects of the same verb.
- They are indirect objects of the same verb and are used with the same preposition.

For example, if the document contains two sentences like

1. "The control unit sends an alarm message in a critical situation"
2. "The measurement unit sends measurements results every 5 seconds",

the concepts "control unit" and "measurement unit" are considered as related, as well as "alarm message" and "measurements results".

Taxonomy building: Concept clusters constructed in the previous step are used to build the taxonomy by joining overlapping concept clusters. The emerging larger clusters represent more general concepts. For example, the two clusters "{alarm message, measurements results}" and "{control message, measurements results}" are joined into the larger cluster

{alarm message, control message, measurements results}

because they share the common concept "measurement results". The new joint cluster represents the more general concept of possible messages.

This step also aids in identifying synonyms[2] because synonyms are often contained in the same cluster. For example, if a cluster contains both "signal" and "message", the domain analyst performing the ontology construction can identify them as synonyms.

In the original approach [17], the tool ASIUM [18] was used to cluster terms and build a taxonomy. Other clustering approaches are possible as well [19].

Associations/relations mining: There is a potential association between two concepts if they occur in the same sentence. Each potential association then has to be validated by the requirements engineer before being recorded as an association between concepts.

Note, that validation of the association proposed by the association mining tool automatically implies validation of the requirements document. If the tool suggests an association that *cannot* be valid, for example a pair containing completely unrelated concepts, then we have detected an evidence that the requirements document

[2] Different names for the same concept.

contains some inconsistent noise that must be eliminated. The tool KAON [20] can be used for this step. Maedche and Staab [21] give an in–depth treatment of association mining.

More details on the ontology extraction approach sketched above can be found in [17].

5.3 Interpreting Sentences: Semantic Approaches to Text Analysis

Semantic approaches are the most demanding on the formulation. In return, they extract the most information from text. As the name says, these approaches build a semantic representation as their results. Each of these approaches uses one of two kinds of semantic representations: discourse representation structures or mapping of verbs to predicates with their arguments.

Discourse representation structure (DRS) is a kind of first order predicate logic with explicit introduction of variables and definitions of variable scopes and accessibility. An example DRS, taken from Blackburn et al. [22], is shown in Figure 1. This DRS consists of one large scope box with two subordinate scope boxes. Each of the subordinate scopes contains some object references, represented by the variables x and y, and statements about these objects. For example, the left box introduces the object x and states $woman(x)$. The right box introduces a new object y and states $boxer(y)$ and $loves(x, y)$. The whole DRS represents the sentence "Every woman loves a boxer" and is equivalent to the formula

$$\forall x.woman(x) \Rightarrow \exists y.boxer(y) \wedge loves(x, y). \tag{1}$$

(See the technical report by Blackburn et al. [22] for the translation rules between DRSs and formulae and for other details.)

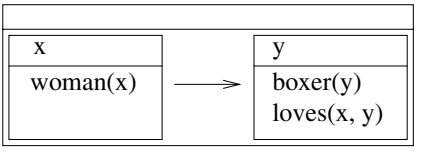


Fig. 1. Discourse Representation Structure (DRS) for "Every woman loves a boxer"

To compute the semantics-DRS, Blackburn et al. [22] define a calculus for such structures. This calculus defines operations on DRSs, like merging, conjunction, negation, and so on. In order to translate a sentence to the representing DRS, a DRS–λ–expression[3] is associated with every word, all the word–λ–expressions are chained to one sentence–λ–expression and then this large λ–expression is evaluated according to the reduction rules of the λ–calculus.

The following example shows semantics calculation with ordinary first order formulae, but a very similar calculation can be done with discourse representation structures.

[3] For an introduction to λ–calculus see, for example, Baader and Nipkow [23].

The example uses ordinary first order formulae just not to over-complicate the matters. First of all, λ–expressions are introduced for every word class:

$$\begin{array}{lll}
\text{Proper names:} & Alice & = \lambda P.(P\ Alice) \\
\text{Common names:} & woman & = \lambda y.(woman(y)) \\
\text{Intransitive verbs:} & walks & = \lambda x.(walk(x)) \\
\text{Transitive verbs:} & loves & = \lambda X.(\lambda z.(X\ (\lambda x.love(z,\ x)))) \\
\text{“every”:} & every & = \lambda P.(\lambda Q.(\forall x.((P\ x) \rightarrow (Q\ x)))) \\
\text{“a”:} & a & = \lambda P.(\lambda Q.(\exists y.((P\ y) \wedge (Q\ y))))
\end{array} \tag{2}$$

It is possible to calculate the sentence semantics just by replacing every word with its λ–expression and performing standard reductions defined in the λ–calculus. For example, the semantics of “Alice loves a man” is calculated as follows:

$$\begin{array}{l}
Alice\ loves\ a\ man = \\
= \lambda_{Alice}\ (\lambda_{loves}\ (\lambda_a\ (\lambda_{man}))) \\
= \lambda_{Alice}\ (\lambda_{loves}\ ((\lambda P.(\lambda Q.(\exists y.((P\ y) \wedge (Q\ y)))))\ (\lambda y.(man(y))))) \\
= \lambda_{Alice}\ (\lambda_{loves}\ (\lambda Q.(\exists y.(((\lambda y.man(y))\ y) \wedge (Q\ y))))) \\
= \lambda_{Alice}\ (\lambda_{loves}\ (\lambda Q.(\exists y.((man(y)) \wedge (Q\ y))))) \\
= \lambda_{Alice}\ ((\lambda X.(\lambda z.(X\ (\lambda x.love(z,\ x)))))\ (\lambda Q.(\exists y.((man(y)) \wedge (Q\ y))))) \\
= \lambda_{Alice}\ (\lambda z.((\lambda Q.(\exists y.((man(y)) \wedge (Q\ y))))\ (\lambda x.love(z,x)))) \\
= \lambda_{Alice}\ (\lambda z.(\exists y.(man(y) \wedge (\lambda x.love(z,x))\ y))) \\
= \lambda_{Alice}\ (\lambda z.(\exists y.(man(y) \wedge love(z,y)))) \\
= (\lambda P.(P\ Alice))\ (\lambda z.(\exists y.(man(y) \wedge love(z,y)))) \\
= (\lambda z.(\exists y.(man(y) \wedge love(z,y))))\ Alice \\
= \exists y.(man(y) \wedge love(Alice,y))
\end{array} \tag{3}$$

As the above example shows, the semantics calculation is quite complicated. Furthermore, introduction of additional words in the sentence would add additional λ–expressions to the computation and would disturb it. This makes approaches of this kind extremely fragile. They are applicable to only restricted specification languages with fixed grammars.

To make this approach applicable to document analysis, it is necessary to restrict the natural language. Fuchs et al. [24], for example, introduced a controlled specification language (ACE, Attempto Controlled English). The language is restricted in the following way:

Vocabulary: The vocabulary of ACE comprises
- predefined function words, e.g. determiners, conjunctions, prepositions
- user-defined, domain-specific content words, e.g. nouns, verbs, adjectives, adverbs

Sentences: There are
- simple sentences,

- composite sentences,
- query sentences.

Simple sentences have the form $subject + verb + complements + adjuncts$.

Firm sentence structure and the necessity to explicitly define the vocabulary in advance restrict the applicability of ACE and other λ–calculus based approaches to real requirements documents.

The other group of semantic approaches uses verb subcategorization frames for semantics representation. A verb subcategorization frame is a verb with its arguments, namely its subject and its objects. For example, for the verb "send", possible arguments are: sender, receiver, sent object. When interpreting the sentence "Component X sends message Y to component Z", in the semantic representation "component X" becomes the sender, "component Z" the receiver and "message Y" the sent object.

This idea is used by Hoppenbrouwers et al. [25] to identify domain concepts and relations between them. Hoppenbrouwers et al. define a set of roles, or semantic tags, like $agent$, $action$, $patient$ etc. The analyst marks the relevant words with these tags. For example, the sentence "Component X sends message Y to component Z" can be manually tagged as

$$(\texttt{Component X})/agent\ \texttt{sends}/action\ (\texttt{message Y})/patient\ \texttt{to}\ (\texttt{component Z})/other.$$

Sentences marked in such a way are used to find $agents$, $actions$, and $patients$.

Ambriola and Gervasi [26] go further than Hoppenbrouwers et al. and build a semantic tree representation of a sentence. To build the semantic representation, they start with a glossary. Each term in the glossary is manually furnished with an associated list of tags. These tags are then used to automatically mark every word of a sentence. For example, the sentence

The terminal sends the password to the server can be canonized as

$$\texttt{terminal}/IN/OUT\ \texttt{sends password}/INF\ \texttt{to server}/IN/OUT/ELAB$$

The applied tags are domain-specific.

After the tagging, a set of transformation rules is applied to marked sentences, translating the tagged sentence to a semantic tree. Figure 2, taken from Ambriola and Gervasi [26], shows an example semantic tree. It shows the representation of the sentence

> When the server receives from the terminal the password, the server stores the signature of the password in the system log.

This tree shows dependencies between actions, namely that the left subtree depends on the right one. Furthermore, the tree shows the semantics of every action. This rich representation allows for extraction of abstract state machines, ER diagrams and other formalisms [27,28].

The drawback of this approach is obvious: the approach is able to analyze only sentences that fit into the predefined transformation rules. The transformation rules are defined manually and it is almost impossible to cater for all the potential constructions that can occur in a real requirements document.

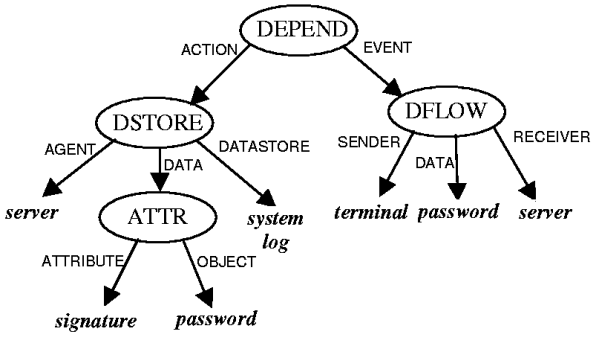


Fig. 2. Semantic tree according to Ambriola and Gervasi [26]

The above approach was further developed and improved by Gervasi and Zowghi [29]. In the improved approach the tool became interactive: for the words not contained in the user's glossary, the user has to specify first, to which category the new word belongs, for example "sender", "receiver", "message", ... Then, when the category of every word is known, the tool translates every sentence to a first-order-logic formula, based on a parse tree like the tree shown in Figure 2. Then, a theorem prover is applied to check whether the set of formulae obtained for the whole text is satisfiable, i.e., contradiction-free.

Rolland and Ben Achour [30] apply the idea of case frames, which is very similar to the approach by Ambriola and Gervasi, introduced above, to whole sequences of sentences to build the semantics of a use case description. As in the approaches by Ambriola and Gervasi and by Gervasi and Zowghi, only firm expression patterns are supported. They also define a set of expressions for temporal relations between individual sentences.

Although interesting in itself, semantics representation is not necessarily the final goal of document analysis. Vadeira and Meziane [31] use semantic text analysis and formulae representation to produce a VDM [32] model. They start with a set of logical formulae and translate them to an ER model first. To build the ER model, they assume that the predicates that build up the formulae are the relationships and predicate arguments are the entities. Then they use a set of heuristics to determine multiplicity of the relations in the basis of formulae. The final step in their approach is the translation of the ER-diagram to the formal specification language VDM.

Although the idea of semantics analysis is very promising for the step from a requirements document to a system model, the approaches are not really mature yet. They are applicable solely to sentences with restricted grammar. What is lacking is a semantic broad-domain parser, putting no restrictions on allowed expression forms and able to cope with sentences that are not completely grammatically correct. It is an open question whether such a parser will ever become possible.

5.4 Logic to Capture Pragmatics

To capture pragmatics, it is necessary to understand links between sentences. To model these links, Asher and Lascarides [33] introduce seven rhetorical relations: narration,

Table 3. Rhetorical relations according to Asher and Lascarides [33]

Narration	Max fell. John helped him up.
Elaboration	He had a great meal. He ate salmon. He devoured lots of cheese.
Explanation	Max fell. John pushed him.
Result	Max switched off the light. He drew the blinds. The room became dark.
Background	John moved from Brixton to St. John's Wood. The rent was less expensive.
Contrast	-Max owns several classic cars. -No, he does not. -He owns two 1967 Alpha spiders.
Parallel	John said that Mary cried. Sam did too.

elaboration, explanation, result, background, contrast, parallel. Table 3 shows their examples for each rhetorical relation. For every relation, they introduce logical operations combining DRS representations for every sentence, of the type described in Section 5.3, to a DRS representation of the discourse.

The "contrast" relation may be especially useful in the context of goal identification. As stated in Section 3, negation of the problems with the existing system is a potential source of the goals for the system to be built. The "contrast" relation potentially identifies problems. For example, in the dialogue excerpt shown in Table 1, there is a "contrast" relation between the phrase "Technologies that could help might work well in a lab…" and the previous statement by the FAA representative. The negation of the problem, namely "Technologies that could help should work not only in the lab", identifies the goal. Unfortunately, automated recognition of the relation types is not possible at the moment.

5.5 Applicability of Different Kinds of Analysis to Goal Identification

The discussion in Sections 5.1– 5.4 makes clear that goal identification takes place on the lexical and pragmatic analysis levels. This discussion makes also obvious that the higher the analysis level, the lower the precision, and automation, as sketched in

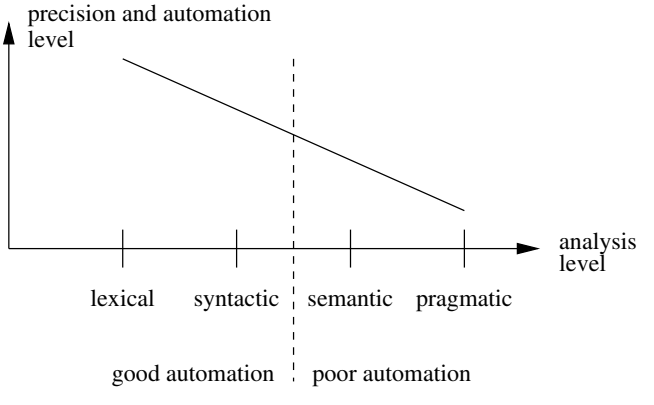


Fig. 3. Precision of different analysis levels

Figure 3. For lexical and syntactic analysis, 100% recall is possible, in the sense that there exist tools that assign a POS tag to every word and assign a parse tree to every sentence, even if the sentence is not completely grammatically correct. If we restrict lexical analysis to search for certain keywords, 100% precision is possible, with a `grep`-like tool. For syntactic analysis, there are taggers available with the precision of about 97% [12], and parsers with the precision of about 80% [34]. Beyond syntactic analysis, we either have to abandon the idea to handle broad domain language and trade recall for precision, like in Attempto Controlled English [24] and similar languages, or fall back to manual analysis. The situation gets even worse if we try to analyze pragmatics. Due to these problems it is not very likely that full-fledged goal identification be automated in the near future, or even in the far.

6 Case Study, Goal Identification by Means of Natural Language Processing

Section 4 shows how to identify goals in a text by close inspection of the text. Now we want to systematize the inspection procedure. To systematize the analysis, we apply two observations to every paragraph, motivated by the goal identification rules by van Lamsweerde (cf. Section 3):

- Phrases like "have to" and "in order to" may directly show a goal.
- If the first sentence of a paragraph does not contain any of the above phrases, the first sentence states the reason why the previous paragraph is problematic. In this case, the negation of this sentence shows the stakeholder's goal.

6.1 Evaluation of the Rule Application

Table 4 shows the results of the application of the above rules to the case study. The application was performed manually by adhering to the rules as strictly as possible. This means that in some cases not the first sentence of the paragraph but the first meaningful one was taken into consideration. For example, statements like "come on", "well…", "we can deal with it" were ignored, as they do not contribute to the identification of the goals. For this reason, Table 4 sometimes lists other than the first sentence of the paragraph.

It is important to emphasize that the negations listed in Table 4 were not constructed by purely textual deletion or addition of "not" at some position in the sentence. Furthermore, negations had to be generalized. For example, "It's not easy to move 2 million passengers…", statement from paragraph 4, was negated to "It should be easy to move 2 million passengers…" and then generalized to "The screening system has to handle 2 million passengers daily". In a similar way, "On each dollar that a potential attacker spends on his plot we had to spend $1000 to protect" was negated to "On each dollar that a potential attacker spends on his plot we should spend much less than $1000 to protect", and generalized to "The screening procedure should remain affordable". The negation performed for the second sentence, resulting in "On each dollar … we should spend much less than $1000 to protect", cannot be performed on the semantic level, let

Table 4. Application of the hypothesis to the case study

	Sentence	State of the art/Goal	Evaluation
1	We have to ban on airplane passengers taking liquids on board in order to increase security following the recent foiled United Kingdom terrorist plot.	State of the art: we do not ban passengers taking liquids, terrorist plot like in the UK is possible. Goals: ban passengers taking liquids, increase security	
2	Technologies that could help might work well in a lab, but when you use it dozens of times daily screening everything from squeeze cheese to Channel No. 5 [sic] you get False Alarms ...	Goals: technologies should work not only in the lab, and the proportion of false alarms in daily screening should not lie above some threshold	Goal correctly identified
3	Generating false positives helped us stay alive; maybe that wasn't a lion that your ancestor saw, but it was better to be safe than sorry.	No goal identifiable. However, this sentence is not useless: It states that the threshold mentioned above is not necessarily zero.	—
4	It's not easy to move 2 million passengers through U.S. airports daily.	Goal: the screening system has to handle 2 million passengers daily	Goal correctly identified
5	We can deal with it. What if you guys take frequent breaks?	No goal identifiable	—
6	Sounds good though we do take breaks and are getting inspected.	No goal identifiable	—
7	We have yet to take a significant pro-active step in preventing another attack everything to this point has been reactive.	State of the art: We do not take pro-active steps. Goal: We have yet to take pro-active steps	Goal correctly identified
8	On each dollar that a potential attacker spends on his plot we had to spend $1000 to protect.	Goal: we should not spend too much on the screening procedure, it should remain affordable	Goal correctly identified
9	We need to think ahead. For instance, nobody needs a metal object to bring down an airliner, not even explosives.	Goal: identify other types of objects to be banned	Goal correctly identified
10	Airlines need to take the lead on aviation security.	Goal: Airlines need to take the lead on aviation security, not FAA.	Goal correctly identified
11	Sir, a lot of airlines are not doing well and are on the Government assistance.	Goal: Airlines should not be responsible for additional cost-intensive tasks.	Goal correctly identified
12	I think that enforcing consistency in our regulations and especially in their application will be a good thing to do.	State of the art: regulations are inconsistent Goal: regulations should be consistent.	Goal correctly identified
13	Ok, we had very productive discussion	No goal identifiable	—

alone the syntactic and lexical ones. A negation on the semantic level would result in "We should not spend $1000 to protect". This negation is correct too, but it still allows unintended interpretations like "We should spend more than $1000 to protect" or "We should spend $999 to protect". Building sensible negation on pragmatic level, like "We should spend much less than $1000", requires knowledge going beyond pure sentence semantics. This knowledge is absolutely obvious for humans and extremely difficult to capture in AI applications.

It is easy to see that Table 4 contains all the goals identified by ad-hoc analysis in Section 4. However, it is necessary to bear in mind that the case study was rather small and that both analysis runs, ad-hoc and systematic, were performed by the same person, which makes the results potentially biased. Thus, to properly evaluate the rules for goal identification, a controlled experiment is necessary. In the experiment, one group of people would have to identify goals using the introduced rules, and the other group would have to identify the goals ad-hoc.

6.2 Possible Implementation

To implement the introduced procedure for goal identification, it is necessary to solve at least two problems:

- It is necessary to define what a meaningful sentence is, in order to analyze the first meaningful sentence of every paragraph.
- Negation is not always possible by simple deletion or addition of "not". Furthermore, negated sentences have to be generalized. Generalization can be seen also as the application of the "WHY"-question to the negation. (I.e., we would permanently ask the question "why is it really a problem?")

The first problem is relatively simple from the point of view of computational linguistics: We could eliminate sentences without grammatical subject, like "come on" and "well…", as well as questions, like "What do you suggest?" in the case study document. This would work for most paragraphs of the considered case study, but still not for all. To achieve high precision, manual post-processing would be necessary even for this step.

The second problem, the negation, is much more difficult. Purely syntactic negation is obviously insufficient, as we would negate "it's not easy to move 2 million passengers…" to "it's easy to move 2 million passengers…", that does not really state the goal "it should be easy to move 2 million passengers…". To go beyond pure syntactic analysis, we could represent the sentence to be negated as a discourse representation

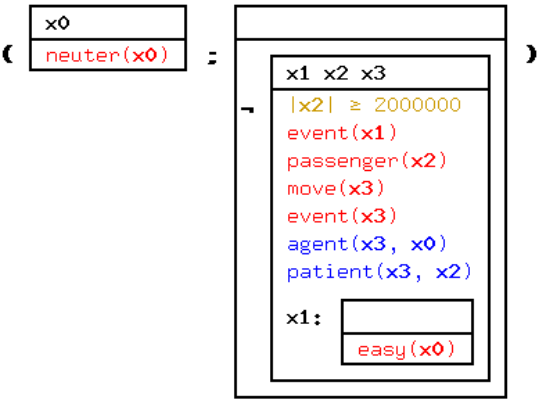


Fig. 4. DRS for the sentence "it's not easy to move 2 million passengers"

structure (DRS) (cf. Section 5.3 and [35,36]). For the sentence "'it's not easy to move 2 million passengers…" this would result in the representation shown in Figure 4. This representation is created by the DRS tool Boxer available as a component of the C&C tool suite [37]. Then we can take a negation on the DRS level. This would be equivalent to representation of the sentence as a formula in first order logic and then taking a negation on the logical level. However, this results, again, in removing the negation from the second box ("¬"-sign) and, therefore, in the sentence "it's easy to move 2 million passengers…".

Thus, even semantic negation is not sufficient to obtain the goals and we have to move to negation on the pragmatics level. Negation on the pragmatics level would include profound knowledge of real world and knowledge of motivation for certain statements. Then we can get, for example, from "On each dollar that a potential attacker spends on his plot we had to spend $1000 to protect" to "On each dollar that a potential attacker spends on his plot we should spend much less than $1000 to protect". On this level we could also implement generalization. For example, in the case study we had to generalize "On each dollar that a potential attacker spends on his plot we should spend much less than $1000 to protect" to "The screening procedure should remain affordable". Unfortunately, this is far beyond the capabilities of state-of-the-art linguistic tools.

7 Summary

In this paper a method for identification of stakeholders' goals by analyzing stakeholders' dialogs was introduced. This method is based on two key assumptions:

- A sentence containing certain keywords directly represents a goal.
- Otherwise, if a sentence is the first meaningful sentence of its paragraph, the negation of this sentence represents a goal.

The second rule used in this paper, the negation rule, can also be seen as an application of the WHY-rule of Section 3 to the dialog: We are just asking the question, why a particular statement was made. One of the reasons to start a new dialog segment is a stakeholder's disagreement with the last statement of his opponent. In this case, the negation of the first statement of the new dialog segment shows the reason for the disagreement, which is some goal of the stakeholder.

Explicit goal identification is important for several reasons. Goals serve to achieve requirements completeness and pertinence, managing requirements conflicts, etc. [3]. The presented approach is especially suitable to manage requirements conflicts when negotiating requirements: In the Win-Win negotiation approach [2], requirements conflicts are resolved in such a way that the *goals* of every stakeholder remain satisfied. In the case of goal conflicts, such a resolution is impossible. Thus, identification of goals and goal conflicts, as in the presented paper, contributes to identification of potential problems early in the development process.

Acknowledgments. I am very grateful to Daniel Berry and two anonymous reviewers. They helped a lot to improve the paper.

References

1. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
2. Grünbacher, P., Boehm, B.W., Briggs, R.O.: EasyWinWin: A groupware-supported methodology for requirements negotiation, http://sunset.usc.edu/research/WINWIN/EasyWinWin/index.html
3. van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: Proceedings of the 5th IEEE International Symposium on Requirements Engineering, pp. 249–263. IEEE Computer Society, Los Alamitos (2001)
4. Berry, D.: Natural language and requirements engineering - nu? In: International Workshop on Requirements Engineering, Imperial College, London, April 25 (2001), http://www.ifi.unizh.ch/groups/req/IWRE/papers&presentations/Berry.pdf
5. Goldin, L., Berry, D.M.: AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation. Automated Software Eng 4, 375–412 (1997)
6. Maarek, Y.S., Berry, D.M.: The use of lexical affinities in requirements extraction. In: Proceedings of the 5th international workshop on Software specification and design, pp. 196–202. ACM Press, New York (1989)
7. Lecoeuche, R.: Finding comparatively important concepts between texts. In: The Fifteenth IEEE International Conference on Automated Software Engineering, Grenoble, France, pp. 55–60. IEEE Computer Society Press, Los Alamitos (2000)
8. Abrial, J.R., Börger, E., Langmaack, H.: The steam boiler case study: Competition of formal program specification and development methods. In: Abrial, J.R., Borger, E., Langmaack, H. (eds.) Dagstuhl Seminar 1995. LNCS, vol. 1165, pp. 1–12. Springer, Heidelberg (1996)
9. Sawyer, P., Rayson, P., Cosh, K.: Shallow knowledge as an aid to deep understanding in early phase requirements engineering. IEEE Trans. Softw. Eng. 31, 969–981 (2005)
10. Abbott, R.J.: Program design by informal English descriptions. Communications of the ACM 26, 882–894 (1983)
11. Ratnaparkhi, A.: A maximum entropy model for part-of-speech tagging. In Brill, E., Church, K., eds.: Proceedings of the Conference on Empirical Methods in Natural Language Processing (Somerset, New Jersey), pp. 133–142. Association for Computational Linguistics, Morristown, NJ, USA (1996)
12. Curran, J.R., Clark, S., Vadas, D.: Multi-tagging for lexicalized-grammar parsing. In: 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Morristown, NJ, USA, 17-21 July, pp. 697–704 (2006)
13. Language-Independent Named Entity Recognition, http://www.cnts.ua.ac.be/conll2003/ner/
14. Witte, R., Li, Q., Zhang, Y., Rilling, J.: Ontological text mining of software documents. In: Kedad, Z., Lammari, N., Métais, E., Meziane, F., Rezgui, Y. (eds.) NLDB 2007. LNCS, vol. 4592, pp. 168–180. Springer, Heidelberg (2007)
15. Chen, P.: English sentence structure and entity-relationship diagram. Information Sciences 1, 127–149 (1983)
16. Saeki, M., Horai, H., Enomoto, H.: Software development process from natural language specification. In: Proceedings of the 11th international conference on Software engineering, pp. 64–73. ACM Press, New York (1989)
17. Kof, L.: Text Analysis for Requirements Engineering. Ph.D thesis, Technische Universität München (2005)

18. Faure, D., Nédellec, C.: ASIUM: Learning subcategorization frames and restrictions of selection. In: Nédellec, C., Rouveirol, C. (eds.) ECML 1998. LNCS, vol. 1398. Springer, Heidelberg (1998)
19. Nenadić, G., Spasić, I., Ananiadou, S.: Automatic discovery of term similarities using pattern mining. In: Proceedings of CompuTerm 2002, pp. 43–49. Association for Computational Linguistics, Morristown (2002)
20. Welcome to KAON, http://kaon.semanticweb.org/
21. Maedche, A., Staab, S.: Discovering conceptual relations from text. In: Horn, W. (ed.) ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence, pp. 321–325. IOS Press, Amsterdam (2000)
22. Blackburn, P., Bos, J., Kohlhase, M., de Nivelle, H.: Inference and computational semantics. CLAUS-Report 106, Universität des Saarlandes, Saarbrücken (1998)
23. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1999)
24. Fuchs, N.E., Schwertel, U., Schwitter, R.: Attempto Controlled English (ACE) language manual, version 3.0. Technical Report 99.03, Department of Computer Science, University of Zurich (1999), http://www.ifi.unizh.ch/attempto/publications/papers/ace3_manual.pdf
25. Hoppenbrouwers, J., van der Vos, B., Hoppenbrouwers, S.: NL structures and conceptual modelling: grammalizing for KISS. Data Knowl. Eng. 23, 79–92 (1997)
26. Ambriola, V., Gervasi, V.: Experiences with domain-based parsing of natural language requirements. In: Fliedl, G., Mayr, H.C. (eds.) Proc. of the 4th International Conference on Applications of Natural Language to Information Systems. OCG Schriftenreihe (Lecture Notes), vol. 129, pp. 145–148. Oesterreichische Computer Gesellschaft (1999)
27. Ambriola, V., Gervasi, V.: The Circe approach to the systematic analysis of NL requirements. Technical Report TR-03-05, University of Pisa, Dipartimento di Informatica (2003)
28. Gervasi, V.: Synthesizing ASMs from natural language requirements. In: Proc. of the 8th EUROCAST Workshop on Abstract State Machines, pp. 212–215. Universidad de Las Palmas (2001)
29. Gervasi, V., Zowghi, D.: Reasoning about inconsistencies in natural language requirements. ACM Trans. Softw. Eng. Methodol. 14, 277–330 (2005)
30. Rolland, C., Ben Achour, C.: Guiding the construction of textual use case specifications. Data & Knowledge Engineering Journal 25, 125–160 (1998)
31. Vadera, S., Meziane, F.: From English to formal specifications. The Computer Journal 37, 753–763 (1994)
32. Jones, C.B.: Systematic Software Development using VDM. Prentice-Hall, Upper Saddle River (1990)
33. Asher, N., Lascarides, A.: Logics of Conversation. Cambridge University Press, Cambridge (2003)
34. Clark, S., Curran, J.R.: Wide-coverage efficient statistical parsing with ccg and log-linear models. Comput. Linguist. 33, 493–552 (2007)
35. Bos, J., Clark, S., Steedman, M., Curran, J.R., Hockenmaier, J.: Wide-coverage semantic representations from a CCG parser. In: COLING 2004: Proceedings of the 20th international conference on Computational Linguistics, pp. 1240–1246. Association for Computational Linguistics, Morristown (2004)
36. Bos, J.: Towards wide-coverage semantic interpretation. In: Proceedings of the 6th International Workshop on Computational Semantics (IWCS 6), pp. 42–53 (2005)
37. C&C Tools, http://svn.ask.it.usyd.edu.au/trac/candc

Text Classification and Machine Learning Support for Requirements Analysis Using Blogs

Douglas S. Lange

Space and Naval Warfare Systems Center
San Diego, CA 92152
doug.lange@navy.mil

Abstract. Text classification and machine learning technologies are being investigated for use in supporting knowledge management requirements in military command centers. Military communities of interest are beginning to use blogs and related tools for information sharing, providing a comparable environment to the use of blogs for system requirement discussions. This paper describes the work in the area being performed under the Personalized Assistant that Learns (PAL) program sponsored by the Defense Advanced Research Projects Agency. Comparisons are then made to how the technology could provide similar capabilities for a requirements analysis environment. An additional discussion of how the task learning capabilities from PAL could also benefit requirements analysis in a rapid prototyping process is provided.

1 Introduction

The United States Military has adopted several network based communications mechanisms. During the second Gulf War, *chat* was an important method of communications, reducing the need for voice circuits. *E-mail* protocols have been used heavily since the early '90s for longer more structured messages in the place of old teletype methods. Now, *blogs* and *wikis* have come into use for knowledge sharing purposes [1].

The U.S. Strategic Command has developed and uses heavily a capability that can best be described as a hybrid of wikis and blogs. The Strategic Knowledge Integration Web (SKIWeb) allows users to post information about key events, and allows other users to add comments and edit the information. Events can be linked to other events, and lists of events are used to provide key information to various communities of interest [2]. While SKIWeb is structured differently than most blogging capabilities, the information within it can, with small transformations, be represented as being structured exactly like a collection of blogs.

With sponsorship by the Defense Advanced Research Projects Agency (DARPA), the Space and Naval Warfare Systems Center (SSC) along with SRI International and Northrop-Grumman is working to transition machine learning technology into both SKIWeb and a blog/Really Simple Syndication (RSS) capability being developed for U.S. Navy command and control. It is envisioned that the learning technology can not only aid the bloggers, reducing the labor costs of publishing information, but can also

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 182–195, 2008.

extract information for other purposes. It is this second feature that is most closely aligned with the goal of extracting software requirements from blogs.

2 PAL Blogs

Machine learning from the DARPA Personalized Assistant that Learns (PAL) program is being used in several ways in conjunction with blogs. These approaches will help both those who publish information on blogs and those who subscribe to receive the information over Really Simple Syndication (RSS) feeds. Figure 1 depicts an envisioned Navy Composeable FORCEnet (CFn) PAL Blog capability. The two sides of the picture depict the publishing activities (left) and the subscription activities (right). Sections 2.1 and 2.2 describe how machine learning contributes to both sides. The figure shows the situation envisioned for CFn use.

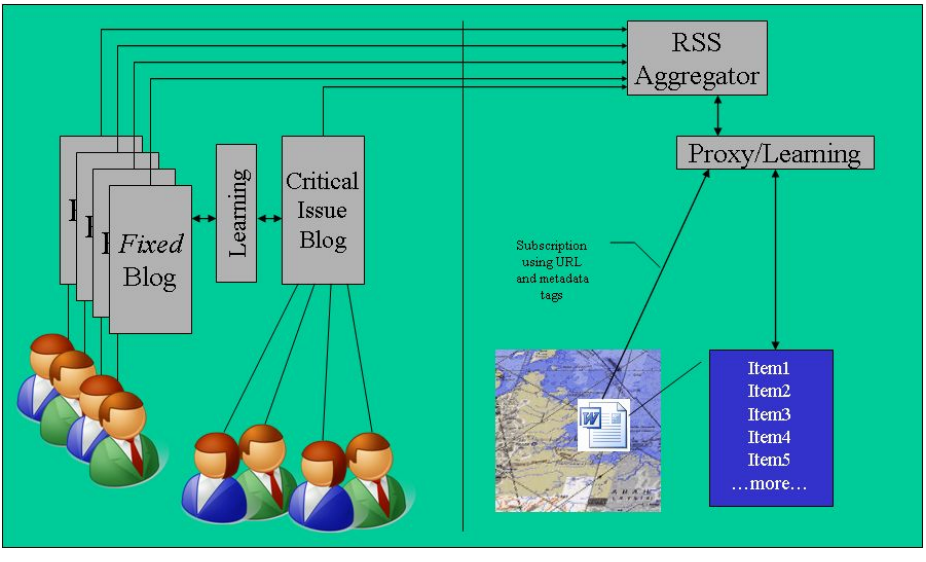


Fig. 1. PAL Blog

On the left side, each information publisher would have their own "fixed" blog. Someone managing parts orders for aircraft would publish information above and beyond what fixed databases allow. On the other hand, when a crisis occurs, multiple users may contribute to a blog about a particular critical issue. These blogs would be utilized during the life of the crisis. The goal of adding a learning capability to support publishing is to help authors find additional information held that relates to posts already made in their blogs. If an author posts information about wing parts for a particular jet, were there emails, received blog articles, documents, etc. that related to the posts. Perhaps the automated assistant can help the author publish more relevant information faster.

The right side of the diagram depicts an RSS Aggregator subscribing to a series of blogs that it has learned may contain information relevant to the user. Additional

information is extracted from the display as configured by the user and documents that fall within the relevance criteria for the user. In Figure 1, items relating to a document posted on the map display are found by comparing the text within the document and the topic models learned concerning the user's interests and needs.

For SKIWeb, SKIPAL (the SKIWeb version of PAL) provides similar benefits. Using topic modeling [9], SKIPAL learns to recognize events of interest to each user and provides a recommended reading list. Text classifiers are taught to recognize particular topics of interest and SKIPAL will identify the people most likely connected to the events and other events that are most closely related. SKIPAL will also learn to gather additional information about an event and learn the tasks that must be accomplished when particular kinds of events occur.

A simple illustration of how SKIPAL communicates with SKIWeb is in Figure 2 below.

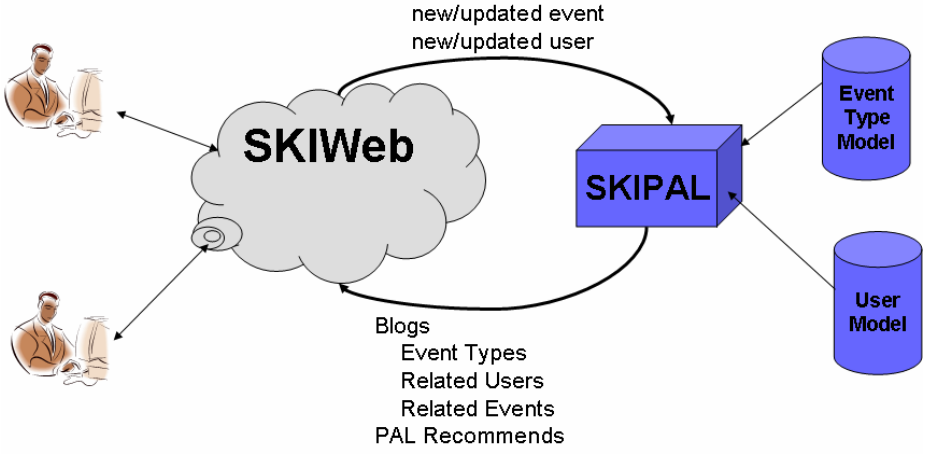


Fig. 2. SKIPAL - SKIWeb Architecture

Users enter through the *SKIPAL Recommends* page as shown in Figure 3 below.

Selecting an event brings up a SKIWeb page that has been supplemented by SKIPAL. SKIPAL shows a list of related people, related events, indicates if the event was recognized by a trained classifier, and provides amplifying links and tasks. A sample is shown in Figure 4.

2.1 Learning on the Publishing Side

Two technologies are applicable to the publishing side in the CFn capability. The first is statistical text classification. Statistical text classifiers group documents based on learning the probabilities of documents within a category containing particular words [10]. Various learning techniques can be applied to text to help map the topics found in a corpus. Various algorithms can be used [3] and the selection can depend on the characteristics of the text, and the mode by which the classifier is to be trained and used. Experience with naïve Bayesian classifiers is discussed in section 4.1.

Fig. 3. SKIPAL Recommends

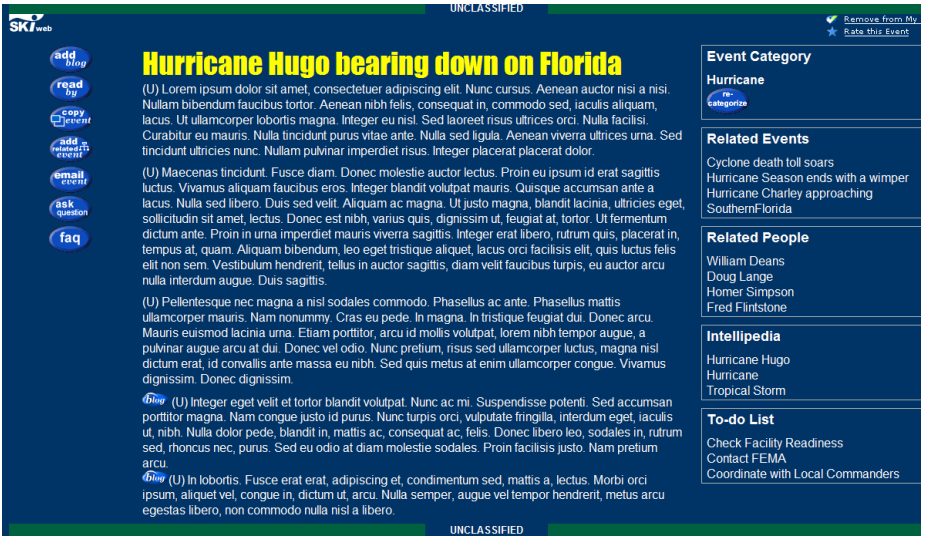


Fig. 4. SKIPAL Supplemented Event

Text classifiers occupy the *Learning* module in Fig. 1. By subscribing to the blog of a user, text classifiers can determine what topics the user writes about. If a user frequently reports the status of aircraft in his/her blog, a model of the writer's interests can reflect that. Further, through the use of intelligent search and indexing that employs the same classification capabilities, PAL may find new email, documents, chat,

or other information sources that have new information on aircraft status and suggest to the user that the information be added to the blog. In Section 3, it will be argued that this capability can be useful in defining system or software requirements from blogs.

A second technology applicable for the PAL Blog provides *task learning* [4]. Another approach to helping the user publish is for PAL to learn and generalize common tasks for the user. If the user frequently gets email about the status of aircraft, he or she may typically choose to do additional research before publishing the results. Perhaps a database of parts needs to be queried and a decision aid to calculate delivery times must be run. The PAL task learning technology allows a user to teach PAL that when such email arrives in the future, PAL is to go through those steps and report the results in a particular manner on the blog, thereby relieving the user of the need to perform the task. PAL. This technology is not directly applicable to an effort to use natural language tools such as blogs as requirements sources, but task learning itself can be a means to requirements gathering by using the task models that are generated as representative of the capabilities required.

2.2 Learning on the Subscription Side

On the subscription side, the goal of learning is to model the topics being published and in turn used by the reader in order to predict what information the user would like to see and find related material. In the PAL Blog, learning is being used to observe the reading habits of users and suggest RSS feeds to subscribe to, and even which entries from a feed to treat with higher priority. This will be done with text classification methods, mapping of topics, and even social networking clues such as which bloggers provide more authoritative information [9]. The features here are nearly identical to those being fielded in SKIPAL.

In SKIPAL, events are recommended to users based on learning what types of events the user authors, comments on, and reads. Explicit feedback is utilized as well. The user can direct SKIPAL to provide more or fewer of a particular kind of event. We are experimenting with using text classifiers and a more sophisticated topic modeler for this purpose. The topic modeling software (iLink) is discussed in the next section. In section 4, results from recent experiments will be provided in order to discuss how well the tools are able to accomplish these goals.

2.3 Models of Expertise

The third technique being applied to blogging activities is social network analysis. The iLink [10] capability, developed by SRI International as part of the PAL program, learns to attribute different levels of expertise on a topic to each member of a network. This is based on the ability of participants to respond to questions satisfactorily, the likelihood that other participants in the network will route messages to them and the numbers of messages written and read on the topic. The way iLink is structured, when a user poses a question through iLink it is distributed to those who are known to have some expertise in the area. Recipients are able to answer the question or forward it to others who they feel might know the answer. When answers do come, those who have provided useful information and those who referred the question to them have their scores raised. Those who provide poor information or cannot answer have their scores lowered. In this way, expertise is determined by the quality of information

rather than by simple claims in social network metadata. Social network analysis can show that the real organization of an entity may be very different than what the organization chart shows. Likewise, the expertise may not reside in those who are advertised as the experts on a topic. Similar results may show that the key stakeholders regarding particular requirements may be different than those believed.

3 Applications for Requirements Engineering

If we consider that the text of individual blogs contains information important to the requirements of a system being developed, then the use of machine learning capabilities in text classification, topic modeling, and social networking tools may provide a means to organize the statements, record arguments for and against a capability, identify critical stakeholders, and even provide some sense of priority. In this section, we will look at each of the activities being performed by machine learning technologies in the SKIPAL capability and compare them to the domain of software requirements and in particular to requirements being gathered based on blog posts.

3.1 Text Classifiers

In [7] there are only three people represented in the discussion on requirements. The power of using blogs or other network communications tools is that the number of people who could express an opinion or provide information can grow much larger. Text classification can group the posts into categories useful for requirements engineering. Those posts that relate to the "carrying of liquids" would get grouped together, much like SKIWeb events about particular subjects are grouped to help decision makers in a command find information.

Classifiers require training [10], so the method process for requirements engineering might be similar to the following:

1. Hold a limited conversation as was done in the case study, or extract a subset of the comments made.
2. Through human analysis determine the categories represented by the posts and label the text in preparation for using the classifier.
3. Train the classifier to recognize the categories using the labeled posts.
4. Run the remaining posts through the classifier and have the classifier do the labeling.
5. Use the output to see if stakeholders in general held similar or dissimilar views. In posts from outside the small initial group, if the posts were about the same topic, were the conclusions the same?
6. Investigate posts that failed to be recognized by the classifier. These may be sources of requirements not thought of by the initial group. Use these to start new discussion threads that can be analyzed by repeating the process.

Statistical methods are at their best when there are large quantities of data to work on. Classifiers would not be useful in the small case study of 3 subjects and 13 posts. But consider how blogs would allow large portions of the stakeholders to comment on many different issues relevant to the new development.

Text classification is also targeted as a technology to support the publishers of information in the PAL Blog. If each of the stakeholders involved in the discussion had access to publisher-side learning tools, they could quickly provide amplifying information for their opinions in their posts. An FAA official who had many emails indicating the weakness of a particular process or technology, or concerns about specific threats, may want to provide that information in the blog posts. The process for using the technology in this way might be the following:

1. The author posts comments on a particular issue.
2. The classifier (already trained earlier...see above), recognizes the categories that the post fits into and scans the users disk and accessible network locations for documents (letter, email, chat, etc.) that fit within the same category.
3. The user is shown the candidate items and can choose whether to publish them in the blog to support previous statements.

3.2 Topic Model Filtering

Which stakeholders have a particular interest in which topics? If we continue to assume that one reason to use blogs or other similar technology is that we want to reach a large community and we want in depth discussion of many topics, we probably want to help the stakeholders filter which posts they pay attention to.

The typical approach would be to define a priori what issues we are going to discuss with each stakeholder. The machine learning approach used in SKIPAL uses a different strategy. Everything is available to the user. SKIPAL learns based on what the user writes, reads, bookmarks and focuses the user's attention on the most relevant posts. Users are still free to look at others and by doing so improve the model of the user's interests. This allows the stakeholders to be involved in the areas they care most about.

The topic model further serves the requirements engineer by ensuring that those most concerned with a topic see questions and comments posted about it. Well before classifiers are trained and used, we can ensure that related posts are being put together and are noticed by the right people.

3.3 Social Network Analysis and Expertise Modeling

Through social network analysis, as blog entries are analyzed for good requirement content, judgments can be made about the quality of the input by engineers and used to learn the level of authority that should be attributed to individual authors both through the judgment of the engineers and by the level of agreement with those that are already judged to be authoritative.

That we expand our discussion to include tens, hundreds, or thousands of stakeholders, doesn't mean that we weight all opinions equally. The iLink capability used within SKIPAL develops a model of the expertise levels of every user on every topic. iLink is structured to treat posts as either questions or answers, but the questions don't have to be in proper question form. These could be statements and responses to statements just as easily. iLink is not recognizing the English structure of the sentences, but learning to build statistical models based on the words used.

In SKIPAL, a reader of an event may decide to post a question. After the user types the question, iLink determines the people with the highest expertise level for the topic and provides them as candidates for receiving the question. The user then chooses which people the question should be sent to. Recipients of the question, can

- Answer the question
- Fail to answer the question
- Forward the question on to somebody they believe can answer the question.

When the question is answered, the person posting the question can rate the response. The expertise model is updated to reflect the new information on who can answer questions on this topic or at least knows who to go to.

Utilizing the Q&A capability of iLink might be done in the following way:

1. A requirements engineer posts a question or states an issue relevant to the requirements being explored.
2. If we are early in the process and the model is ill-defined, the engineer can choose from among those who would normally be selected using a priori knowledge. In parallel, the same question could be posted to the blogs to allow all users to notice the question.
3. If the answers come back satisfactorily, those people will have their expertise ratings increased. The users may pull others in by forwarding the question and the model benefits from increased information.
4. Meanwhile the topic and expertise models are being built up by the posts and reading habits of stakeholders. When we next ask a question about the topic some new stakeholders may be suggested to the engineer.
5. When answers are returned, the engineers should judge them for the quality of the information rather than whether they agree or disagree with the position. This will lead to developing a model of the social network that points towards those stakeholders who can provide quality information to future questions.
6. When the requirements engineers use the classifier to pull out and group comments on issues, those posts by people with higher expertise values for the topic might be given more weight. In fact, tools could be developed that would sort the comments by the expertise of the poster.

When we expand the number of stakeholders we want to reach, the use of an expertise model allows us to understand the real social network among the stakeholders rather than just the advertised social network from organization charts. This should lead to better requirements by surfacing information from the real experts.

3.4 Task Learning

Finally, a major capability within the PAL program, but only indirectly related to text and blogging capabilities, is *task learning*. There are several mechanisms by which PAL derived systems can learn tasks that vary in the amount of interaction required by the user. The results of the learning produce a rich task model that describes the actions that must be performed, conditions and events that influence the tasks, and probabilistic information of the likelihood of success and duration of a particular step [5].

SKIPAL will use task learning in later spirals to manage tasks associated with categories of events. SKIPAL will be taught that when an event of a particular category is found that a certain set of tasks must be completed, that those with high levels of expertise from the models described above need to be sought out to perform some of them, and that SKIPAL must use what it has been taught to complete the rest. The PAL Blog proposes to use task learning in a similar fashion. When a message (in any form) is received by the user, a set of learned tasks are initiated that result in posts being either automatically or semi-automatically being made to the blog.

While some related set of processes may be possible for the requirements engineering blogs described in the case study, the utility isn't directly evident. However, tt may be possible to look at the task learning being done in PAL as a way to manage prototyping efforts.

If we view prototyping as shown in Figure 5 below, task learning can provide a valuable contribution to rapid prototyping. If the user working with a learning system can create a task that fulfills his/her needs, the resulting task model can become a requirements specification for the step in the figure labeled "Construct production system". This obviously will only work for requirements that can be described within an information system, but it may have merit nonetheless.

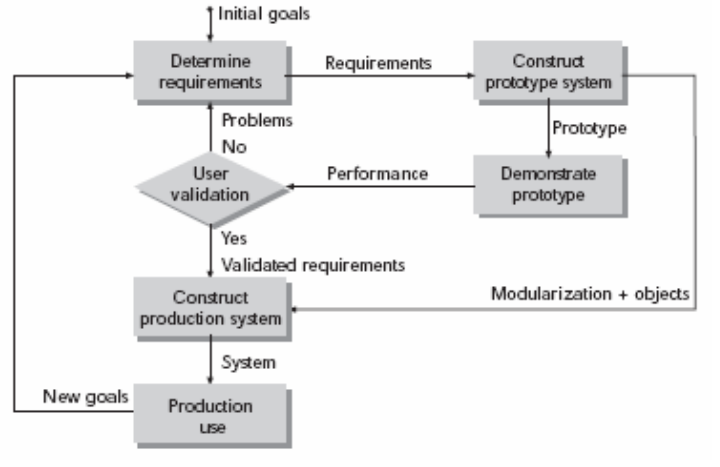


Fig. 5. Rapid Prototyping [From 6]

4 Results from Experiments with Blogs

During the summer of 2007, three experiments were done in analyzing text in the SKIPAL environment. Each of these experiments related to the problem presented in the case study [7] in ways described in Section 3.

4.1 Experiment 1 – Performance of the Classifier

In our first experiment, we trained a *Naïve Bayes* classifier to recognize events of different types using archival SKIWeb data. Table 1 below provides the results from the experiment.

Table 1. Results from Classifier Experiment

Category	# Events in Category	# Event Used for Training	# Events Used for Test	True Positive (TP)	False Positive (FP)	False Negative (FN)
Category #1	17	11	6	4	0	2
Category #2	84	56	28	28	0	0
Category #3	123	82	41	39	0	2
Category #4	39	25	13	10	1	3
Other	1387	909	471	470	7	1

In what is a fairly standard protocol, two-thirds of the events that belonged to the categories were labeled with category names. One third was left unlabeled and the classifier was asked to label them.

While the data in SKIWeb corresponded to discussions about events in the real world and in military exercises for which decisions needed to be made, they represent well the situation that could occur in requirements analysis. Using the case study [7], categories might have been labeled *breaks, screening tests, liquids,* or other representative topic labels, and the classifier could have sorted them out into different discussion threads. Likewise, the *other* category contains everything that wasn't judged to be in one of the labeled categories. This represents a source of new topic categories as well as place for less important messages to end up.

From Table 1, we see very small error rates even relative to the military knowledge management mission. These small numbers of errors would not be difficult for an engineering activity to work with. Prior to utilizing a naïve Bayes classifier for requirements engineering as discussed in Section 3, it would be worth investigating how small the training set could get before the error rate was unacceptable. It is possible that with only 10-20 blog posts labeled, acceptable classification could occur.

4.2 Experiment 2 – Relevance

In the second experiment, we trained the topic modeling engine from PAL and four other algorithms on one year of reading, writing, and book-marking habits of 6 users of the SKIWeb system. Five different recommendation engines, developed from the five algorithms, recommend thirty events from a new set of events the user should read. We then asked the users to judge whether the events recommended were relevant to them or not. The results, shown in Table 2 below, demonstrated that all of the techniques used for topic modeling were successful with some users, but other users were more difficult to characterize.

Review of these users showed that as expected that the topic modeling was able to recommend for active bloggers more easily than those who merely read. The hypothesis (which still needs to be checked) is that active participants focus their authorship in areas that are of more direct interest to them, but readers will read a wide variety of information. Most classifier based methods beat the topic modeling in this batch mode experiment, which put the topic modeling at a disadvantage due to its need for greater signal (expertise model from the questions and answers along with explicit feedback).

In future experiments we will hope to measure the difference in performance provided by the preferred environment for topic modeling which is the same environment for the requirements engineering blogs, and active discussion with dynamic social network. iLink is already being used in dynamic environments with user proclaimed success [9], but without metrics available.

The "Precision at n" measure is a standard used in information retrieval but can be problematic in recommender applications [8]. We provided a list of 30 recommendations and therefore measured "Precision at 30". SKIPAL actually produces a list of all events ordered in relevance order, but for the purposes of this experiment we cut the recommendations off at 30 so that users would only have to review 150 events each. It is possible that there were fewer than 30 relevant events to provide, in which case the algorithms were being penalized by the user interface. It is for this reason that it is only useful as a comparison among the algorithms and not as an absolute indication of value.

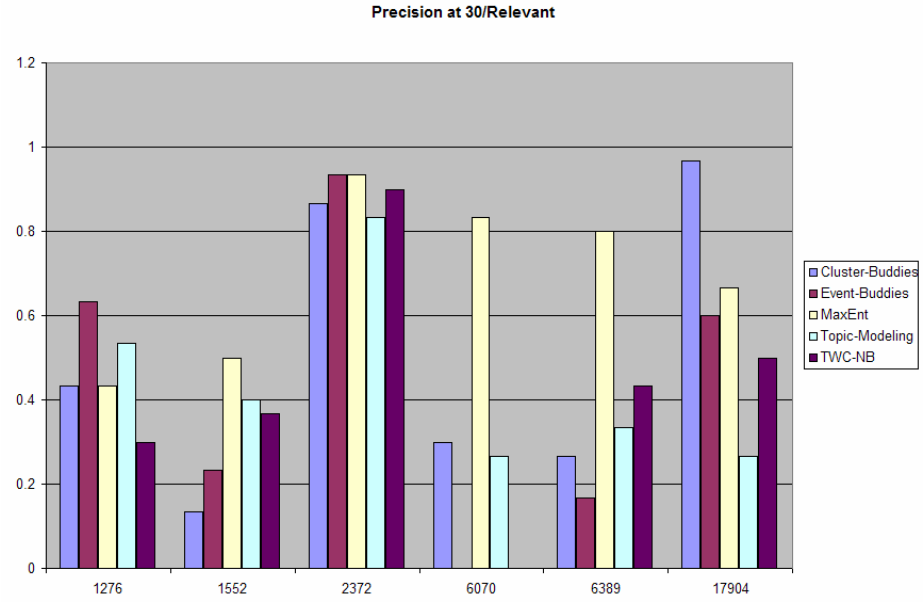


Fig. 6. Precision at 30 for Six Users of SKIWeb

Precision is defined as the number of relevant items in the list of 30, divided by the number of items recommended (i.e., 30). Recall (another information retrieval standard metric) was not possible to compute without asking the users to review all possible events from every day of the test, which was not feasible.

The precision measure indicates how well the recommender is doing in providing relevant recommendations within the list of 30 events relative to other methods providing the same number.

4.3 Experiment 3 – Relevance Based on 'Read-By' Data

In a third experiment, one year of SKIWeb data was again used to train five recommendation engine algorithms. These included:

- Topic modeling from iLink
- Naïve Bayes (labeld as TWCNB for transformed weight-normalized compliment naïve Bayes).
- Maximum Entropy Modeling (another classification approach)
- Event Buddies (using TWCNB to correlate events and users, then recommend events that most closely related users read)
- Cluster Buddies (events are clustered into word groups and users are associated with word groups based on events read, again using TWCNB).

This time, 50 users were selected whose reading habits were closest to the median number of events read, but without exceeding the median number. This time, two weeks of data was introduced one day at a time, and the recommendation engines were asked to provide a recommendation for the users' daily read. We then compared this to what the users had actually read assuming that it would be an indication of relevance. Again, signals for training included events read, authored, blogged comments, and bookmarks. The purpose was to help select an algorithm for use, so the metrics used again were mostly comparative. However, the following graphs show that all of the algorithms were successful at predicting what a user would read, and some significantly so. The metric used was a ratio of the area under a ROC curve [8] to the area under a ROC curve that represents a perfect ordering of the recommendations, where all relevant (read) events are above all irrelevant (unread) events. The results for each algorithm are represented by Gaussian distributions and graphed in Figure 7 below.

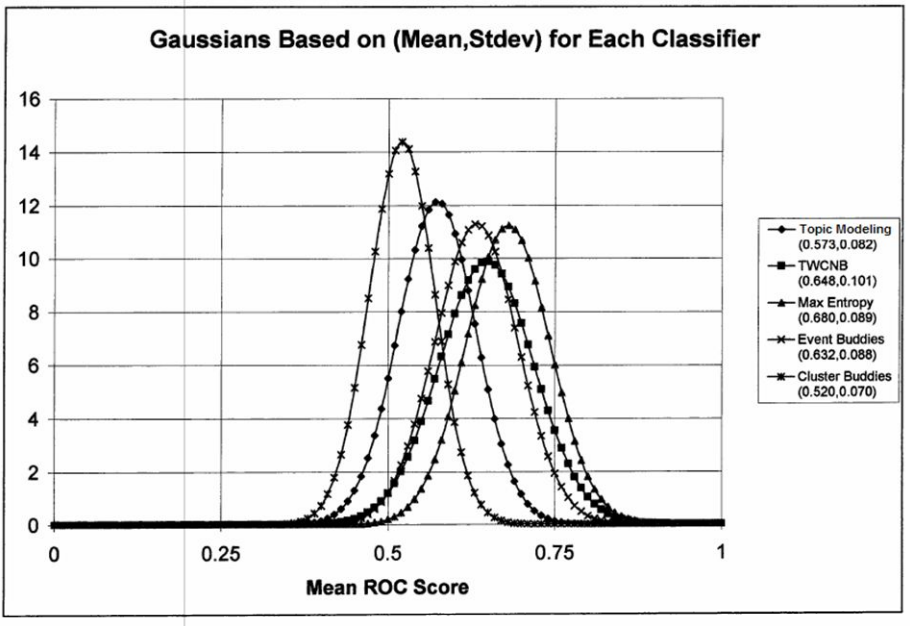


Fig. 7. Experiment Three Results

Again, the limitation of this experiment was its "batch" nature favoring classifiers which are more easily designed for such use. The topic modeling was therefore at a disadvantage as it requires interactive use to get all required signals.

4.4 Conclusions from Experimental Results as Related to Case Study

Classifiers and recommendation systems may well have a contribution to make if blogs are used to facilitate the discussion of system requirements. Classifiers can help find the entries that are relevant to particular requirements or issues. Likewise, recommendation systems that use topic modeling can identify the key stakeholders for which a requirement is important and allow the analysis to be focused on the needs of these stakeholders. The stakeholders are served through allowing them to focus on blog discussions relevant to their interests.

5 Conclusions

The problem posed by the case study is that we often would like to elicit requirements from a large and diverse population of stakeholders. Our current methods and tools require us to focus on a small number of stakeholder representatives. The case study by suggesting the use of blogs in defining requirements for a large enterprise like airport safety hints at something beyond 13 posts by 3 users.

SKIWeb and other military knowledge management capabilities serve exactly this same sort of population. Large commands themselves consist of hundreds or thousands of people, working in a variety of specialties. Distributed collaboration tools network many of these large commands together along with many smaller ones. SKIWeb was created out of a recognition that in a networked world, information and knowledge management was not a hierarchical process as was formerly the case in the military. Execution still has hierarchical components, but information doesn't need to be bound to such a structure.

Requirements elicitation and engineering in a large distributed enterprise doesn't differ very much from the primary activity of large military commands, deciding what actions need to be taken. Many staff members are acting as analysts. They collect information, develop a model of the environment, and suggest what actions are required.

Therefore, there are two aspects of these organizations that are related to the posed problem. First, we need capabilities that improve the ability of large numbers of people to communicate and share knowledge. Second, we need tools that allow analysts to ask questions or propose hypotheses, and get responses back in a way that doesn't overwhelm them.

The case study starts with an issue being raised and responses being proffered. SKIPAL will use iLink to perform this task. In using the statistical models of expertise and topic interests, users who raise issues have the ability to ensure that those with the greatest likelihood of responding authoritatively will see the questions. SKIPAL will evolve a model of who the experts are on any topic raised. Similarly, in a large enterprise, engineers need the ability to understand the real social network and find the true authorities. In a large diverse enterprise, limiting elicitation to a small number of proffered experts may yield poor results.

When we include the entire enterprise in on all topics, users need tools to allow them to filter what they read and focus on what matters to them. This is vital in the command and control environment of U.S. Strategic Command and the capability to focus on relevant information will help staff members avoid being overwhelmed with information. Similarly, if we want feedback from experts all over TSA, the FAA, and individual airports, then we must give them the tools to focus in on the issues that are important to them and that they feel they have something to contribute to. The topic models within SKIPAL perform this task, and the initial experiments were discussed in Section 4. User feedback on the performance of these tools now fielded are allowing the algorithms to be tuned and improved.

Text classifiers are useful in the SKIPAL environment to pick out events of particular interest. In the requirements engineering domain, classifiers can group posts by issue and allow requirements engineers to see all the arguments for an issue grouped together. By seeing what falls out of the classifiers into an "other" bin, new issues can be identified.

The machine learning tools applicable to large distributed command and control enterprises appear to offer many benefits to requirements engineering if we wish to elicit information from a large diverse population of stakeholders using network communication tools like blogs.

References

1. Seymour, G., Cowen, M.: A Review of Team Collaboration Tools Used in the Military and Government, http://www.onr.navy.mil/sci_tech/34/341/docs/cki_review_team_collaboration.doc
2. Boland, R.: Network Centricity Requires More than Circuits and Wires. Signal 61(1), 83–100 (2006)
3. Lange, D.: Boot Camp for Cognitive Systems: A Model for Preparing Systems with Machine Learning For Deployment. Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA (2007)
4. Conley, K., Carpenter, J.: Towel: Towards an Intelligent To-Do List. Technical Report, SRI International, Menlo Park, CA (2006)
5. Myers, K.: Building an Intelligent Personal Assistant. AAAI, Menlo Park (2006) (Invited Talk)
6. Goguen, P., Luqi: Formal Methods: Promises and Problems. IEEE Software 14(1), 73–85 (1997)
7. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
8. Herlocker, J., Konstan, J., Terveen, L., Riedl, J.: Evaluating Collaborative Filtering Recommender Systems. Trans. Information Systems 22(1), 5–53 (2004)
9. Davitz, J., Yu, J., Basu, S., Gutelius, D., Harris, A.: iLink: Search and Routing in Social Networks. In: 13th International Conference on Knowledge Discovery and Data Mining, pp. 931–940. ACM, New York (2007)
10. Segaran, T.: Programming Collective Intelligence. O'Reilly, Sebastopol (2007)

Profiling and Tracing Stakeholder Needs

Pete Sawyer, Ricardo Gacitua, and Andrew Stone

Lancaster University, Lancaster, UK. LA1 4WA
{sawyer,gacitur1}@comp.lancs.ac.uk, a.stone1@lancs.ac.uk

Abstract. The first stage in transitioning from stakeholders' needs to formal designs is the synthesis of user requirements from information elicited from the stakeholders. In this paper we show how shallow natural language techniques can be used to assist analysis of the elicited information and so inform the synthesis of the user requirements. We also show how related techniques can be used for the subsequent management of requirements and even help detect the absence of requirements' motivation by identifying unprovenanced requirements.

Keywords: Requirements engineering, natural language processing, information retrieval, tacit knowledge.

1 Introduction

Bidirectional traceability that spans implementation through design to the specification of requirements, and even all the way back to the origin of the requirements in the characteristics of the application domain is needed for the effective management of complex software projects and products [1]. The need for traceability is in part a result of the very wide difference that exists between the formality of program code and the informality with which stakeholders typically express their requirements. Much of software engineering has concerned itself with closing this gap by:

- raising the levels of abstraction with which software engineers can reason about a system, and allowing selectivity about the level of detail and the various behavioural and structural properties with which the engineers need to concern themselves;
- formalizing the expression of requirements so that software engineers can more easily reason about them and verify their attributes, such as completeness, consistency and so on.

The results of these two strands of work is manifested in the myriad of formal, semi-formal, structured, and other notations that have emerged over the last thirty years. Despite these developments, it is significant that representations of requirements are still predominantly informal.

The fact that "the majority of requirements are given in natural language, either written or orally expressed" [2] is still true is a fundamental inhibitor to requirements' early formalization. Natural language is usefully expressive and available to almost everyone, regardless of their background. As a medium for precise description, however, it has serious deficiencies. For example, in a language such as English, it is easy

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 196–213, 2008.

to unwittingly introduce ambiguity, and complex concepts are hard to express succinctly. Moreover, the rules of grammar are complex, contain many exceptions and are poorly representative of vernacular usage. Considerable skill is needed by a requirements analyst to extract the key information from stakeholders' expressions of requirements, synthesize requirements that specify the best solution to the underlying business problem given the different stakeholders' needs and the available resources, and to do so concisely and precisely. This synthesis is far more than a linguistic transformation. It involves developing an understanding of complex and often conflicting pieces of information expressed in a medium (natural language) that does not support conceptual reasoning in the way that formal languages do.

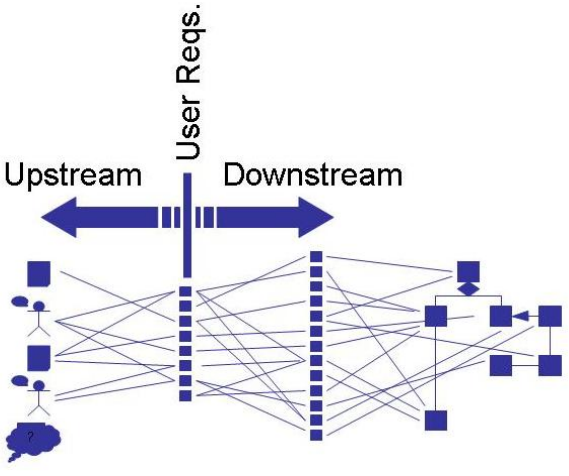


Fig. 1. Upstream vs. Downstream Requirements

The challenges posed by natural language have not deterred researchers in the field of natural language processing (NLP). Inevitably, a number of researchers in requirements engineering (RE) have investigated the application of techniques developed for NLP and the related field of information retrieval (IR) to natural language requirements. This work has achieved some successes, but many of the successes have been focused on processing the products of the RE process. By RE products we mean analyst-synthesized requirements or use cases. In Fig. 1, RE products are represented by the user requirements and the downstream artifacts; derived requirements, models, architecture, design, test cases and ultimately code. Upstream of the user requirements are the stakeholders, documents and other sources of requirements and contextualizing domain knowledge, some more tangible than others.

Downstream requirements artifacts can be relatively tractable to a range of NLP techniques if they are authored to conform to good requirements expression practice, or even conform to a controlled subset of (for example) English. Arguably, the most successful applications of NLP to downstream requirements have been the detecting requirements' defects or the inference of relationships between requirements. Despite these successes, the level of deep understanding of semantics and pragmatics needed

to automate the tasks performed by a requirements analyst continue to far exceed the capabilities of language engineering techniques.

The linguistic problems are even more marked in the artifacts upstream from the user requirements. While the analyst-synthesized requirements *may* conform to good requirements expression practice, the information sources from which the requirements are synthesized are always messy and unstructured. Despite these problems, NLP techniques can provide assistance to analysts provided the limits to what is achievable are clearly understood.

In this paper we are concerned with problems posed by upstream requirements artifacts and the potential for NLP and IR techniques for assisting the analyst. The contribution of work is to posit how the use of *shallow* NLP techniques may aid the analyst in the early stages of transitioning from stakeholders' needs to formal designs; the synthesis of user requirements that are informed by information elicited from the stakeholders and the subsequent management of this information. We also consider the conundrum posed by missing or suppressed information and the perhaps paradoxical potential for shallow techniques to detect the absence of information.

The remainder of the paper is structured as follows. In section two we describe our approach for concept identification which is a key, early activity of the requirements synthesis process. We illustrate this strand of our work with an analysis of the airport security case study from this volume of proceedings [3]. Section three describes the use of IR techniques for tracing between requirements and their sources and, again, applies the techniques we have developed to analysis of the airport security case study. In section four we discuss the problems of unprovenanced requirements and show how the same IR techniques used for upstream tracing can be useful for inferring the presence of tacit knowledge. Section five surveys related work and section six concludes the paper.

2 Assisting the Synthesis of User Requirements

Among the most challenging applications of NLP in RE have been problems where the language used is uncontrolled [4]. Uncontrolled language is characteristic of the upstream phases of RE [5] where the stakeholders not only hold different perspectives on the problem domain but express their needs in ways that often fail to conform to conventions of language use. The three bloggers in the airport security case study illustrate this well. Even ignoring the divergence of semantics and pragmatics of their perspectives on the problem, a number of lexical and syntactic characteristics of the text pose real natural language processing problems, such as idioms ("come on!"), implicit context ("we can deal with it.") and grammatical errors and typos ("We have to ban on ..", "Channel No. 5").

The characteristics illustrated by the airport security blog illustrate why the automatic synthesis of user requirements is way beyond the current state-of-the-art. However, NLP techniques exist that are relatively tolerant of some of the linguistic defects illustrated by the airport security blog. We refer to these as "shallow" NLP techniques. Such techniques are essentially lexical and morphological so are incapable of inferring required properties of an airport security system. Rather, they provide data about text that is about airport security, from which a human analyst may infer

properties of an airport security system. We describe such techniques as shallow because they are not based on models of language structure but are "characterized by the use of analytic techniques which depend on statistical properties of language structure rather than reliance on absolute logical rules" [6]. Because of shallow techniques' basis in statistics, they work best when there is a large corpus of text from which to infer properties.

A requirements analyst has to develop an understanding of the problem domain. A fundamental component of domain understanding is identification of the domain concepts. A number of researchers have investigated the identification of domain concepts by analysis of the text using, for example, frequency profiling [7] and lexical affinities [8]. Such work can serve to help identify entities in the problem domain and their relationships, reveal key terms and populate glossaries. Ultimately, they may be organized ontologically, perhaps in structural and behavioral models. In our work [4, 9, 10], we have investigated the combination of a number of statistical and corpus-based NLP techniques for concept identification. The techniques we have experimented with include part-of-speech (PoS) and semantic (or word-sense) tagging [11], lemmatization, frequency profiling, and collocation analysis (which is the same as lexical affinity identification).

The columns A and B in Table 1 show the most significant results of applying PoS tagging, lemmatization and frequency profiling to the airport security blog. The PoS tags assigned by our toolset's tagger (CLAWS [12]) are needed by several other types of processing. Although the CLAWS tagset uses over 160 different parts of speech, for simplicity in the remainder of this paper we refer only to basic parts of speech (noun, verb, adjective, etc.).

The PoS tags are used by the lemmatizer to collapse words with the same PoS to a base form called a lemma that, in contrast to the action taken by the more commonly

Table 1. The 10 Most Over-Represented Words in the Blog and Domain Corpus. Column A shows the 10 most over-represented lemmatized terms in the blog where each term can have any part of speech. Column B shows the 10 most over-represented verbs in the blog. Columns C and D show the same as columns A and B respectively, but apply to the blog combined with a corpus of documents on airport security. The most over-represented term is at the top of each column.

Blog only		*Blog and domain corpus*	
A: All terms	***B: Verbs only***	***C: All terms***	***D: Verbs only***
screener	screen	airport	access
security	ban	security	screen
airport	backscatter	passenger	check
oxidizer	federalize	capta	profile
screening	spend	surveillance	identify
faa	miss	flight	travel
airline	retrain	luggage	selfdiscipline
cutter	foil	traveler	carry
liquid	deter	screening	capture
dozen	renegotiate	liquid	pack

used technique of *stemming*, takes account of words' parts of speech. Hence, while a stemmer would have reduced the noun "screeners" and the verb "screens" to a common stem "screen", the lemmatizer distinguishes between them because they have different PoS, reducing them to the lemmas "screener" and "screen", respectively. The benefits of stemming versus lemmatization are arguable but for concept identification it helps to distinguish (for example) the role or actor signified by "screener" from the action or candidate use case "screen".

Following lemmatization, we performed frequency profiling on the lemmatized words. Frequency profiling can be done in a number of ways. The simplest is to simply rank words in order of their frequency of occurrence in a document. However, a more telling result can be achieved by comparing the frequency of occurrence of a word in a document against the frequency of occurrence in a *normative corpus*. With corpus-based frequency profiling, concepts that have particular significance to a domain can be revealed because the terms that signify them tend to appear to be over-represented. We compared the frequency of occurrence of each lemmatized word in the blog against the frequency of occurrence predicted by the British National Corpus (BNC). The BNC is one of the largest corpora of English text that has been compiled by linguists in order to understand the language. It contains text from a number of genres, both formal and informal, written and spoken, so is broadly representative of British English as a modern, living language. Its utility is in the fact that, as a normative corpus, it forms a benchmark against which other documents of English text can be compared. This is nicely illustrated by "oxidizer". Although oxidizer appears only twice (once in singular and once in plural form) in the 603 word blog, twice is still significantly more frequently than predicted by its rate of occurrence in the BNC.

The lemmatized words in column A are ranked according to how over-represented they are in the blog compared to their appearance in the BNC. Only the ten most over-represented words are shown, ranked from top to bottom. Hence, "screener" is the most over-represented word, strongly suggesting that it represents a significant concept within the bloggers' problem domain. In general, the further the analyst looks down the list, the less over-represented the words become. In [4], we show that the likelihood of a word in a frequency list representing a significant domain concept (that is, the *precision*) decays as the analyst traverses the list from top to bottom. Conversely, the likelihood of identifying all the significant domain concepts present in the list (that is, the *recall*) increases as the list is traversed, but at a decaying rate. This means that the analyst typically experiences the law of diminishing returns the further from the top of the list they go in search of domain concepts. How far down the list they should search is dependent upon the size of the list and the extent to which the terms deviate from the predicted frequency of occurrence. The density of potential domain concepts in the ten terms listed in column A in Table 1 suggests that even for such a short document, it would repay looking a bit further down the list.

It is interesting to note that each word in column A in Table 1 has been tagged as a noun. It appears to be a common feature of frequency profiling that nouns dominate the top of the ranked list. We hypothesize that this is a linguistic quirk that serves to distort the true significance of words' frequency of occurrence. To overcome this problem it is possible to filter the list on PoS. In the column B in Table 1, we have filtered the frequency list to show only the verbs. Potentially genuine goals

("foil", "deter") and use cases ("screen") are in evidence. The density of potential concepts within the verb-filtered frequency list, and therefore the precision, is fairly low, however.

Note that all NLP techniques are fallible so 100% recall or precision is almost never achieved. This fallibility is illustrated by Table 1 in which the elided words represent errors. "Capta" and "selfdiscipline" are formatting errors but "screening" and "backscatter" are errors produced by the PoS tagger. Consider "Screening" which occurs four times:

- once as a verb: "…you use it dozens of times daily screening everything from squeeze cheese…")
- three times as an adjective: "screening system", "screening devices" and "screening rules".

However, CLAWS has tagged "screening" as a noun in each case. This has in turn confused the lemmatizer, which should have converted the verb occurrence into the lemma "screen", perhaps promoting the ranking of "screen" up the frequency list.

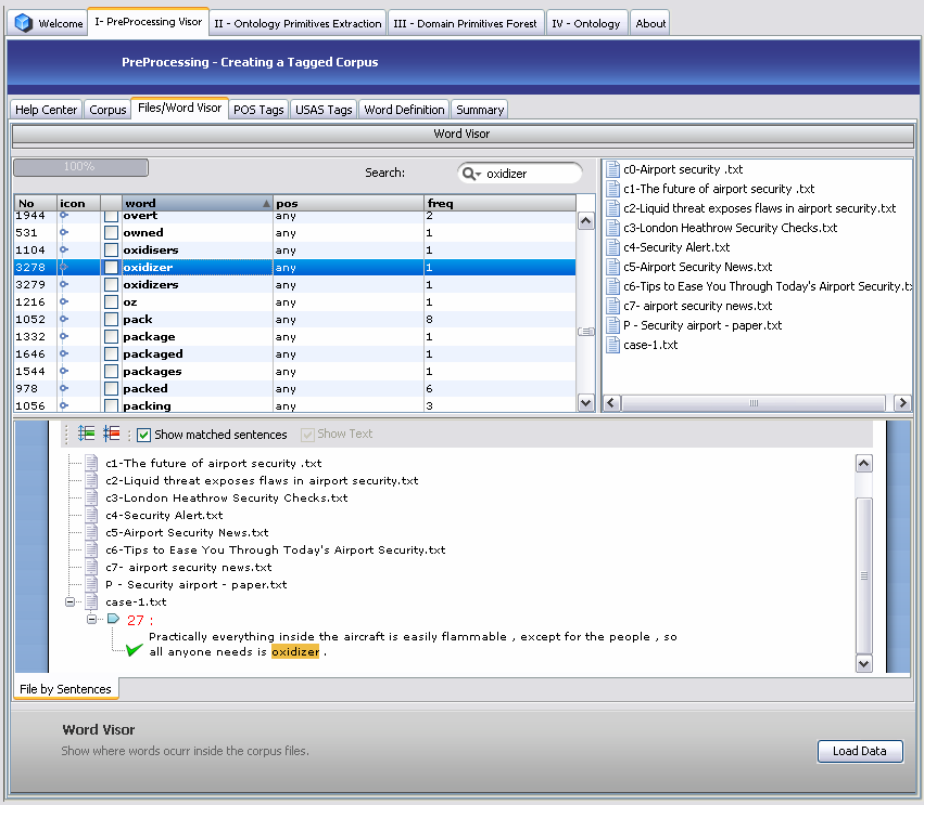


Fig. 2. Keyword in Context Viewer in *OntoLancs* ([9, 10])

A more subtle source of errors derives from the fact that corpus-based NLP techniques tend to work best when the volume of text to be processed is sufficient to yield results that are not distorting. This is again well illustrated by "oxidizer", ranked fourth in column A. The fact that a single blogger mentioned the term twice does not *per se* mean that it represents a significant concept within the bloggers' universe of discourse. That it *might* be significant can only be determined by a human analyst. To assist the analyst's decision-making keyword in context (KWIC) tools can be provided to reveal a word's occurrences in the text. A KWIC tool is illustrated in Fig. 2 in which the two occurrences of "oxidizer" are revealed by selecting the word in the frequency list.

Like columns A and B, columns C and D in Table 1 are unfiltered and verb-filtered frequency lists. C and D are frequency lists constructed from a small corpus of documents containing approximately 8000 words. This corpus was compiled from a mixture of press reports about airport security and advice on security published on travel websites, as well as from the text of the blog. We cannot claim that the corpus is truly representative of the domain. However, it is interesting to compare columns A and C, and B and D to help understand the focus of the blog within the general domain of airport security. If we had more confidence in the relevance and degree of consensus represented by our airport security corpus, we could use the results of the analysis as the starting point for the construction of a domain ontology that could be used for the reuse of knowledge across airport security applications. Given the degree of uncertainty over the veracity of our corpus, the most we can claim in this instance is that it reveals some of the general context of the bloggers' conversation.

Table 2. The Most Significant Lexical Affinities with the Top-Ranked Words in the Blog and Domain Corpus.The relative strength of each *Seed word/Collocated word* pair is depicted by its position in the table, with *Luggage/Hand* being the strongest.

Seed word	***Collocated word***
Luggage	Hand
Luggage	Carry-on
Passenger	Screening
Security	Department
Security	Aviation
Screening	Passenger
Luggage	Check-in
Security	Transportation
Security	Transport
Luggage	Check
Screening	Profiling
Screening	Security
Security	Safety
Security	Canadian
Security	Private
Airport	International
Surveillance	Nature
Capta	Biometric

Security	Heathrow
Passenger	Data
Security	Current
Luggage	Check
Security	Extra
Security	Air
Passenger	Airline
Luggage	Items
Security	Service
Security	Canada
Airport	Police
Security	Screening
Screening	System
Airports	Security
Passenger	Airport
Screening	Airport
Security	Baggage
Security	Item

In our earlier work [4], we elaborated on the use of statistical techniques for analyzing elicited requirements information. We analyzed the use of corpus-based frequency profiling, filtering on part-of-speech and on semantic class, and the use of lexical affinities. We concluded that frequency profiling yields the best performance of any individual technique. However, while generally performing relatively poorly when used in isolation, the other techniques tended to complement frequency profiling. Hence, for example, not all of the significant concepts present in the corpus of domain documents are likely to be so over-represented as to appear near the top of the ranked frequency list. The further down the list they are, the more likely they are to be overlooked by the analyst. However, at least some of these unidentified concepts are likely to co-occur with the concepts near the top of the frequency list. If they co-occur sufficiently often, they will show up as lexical affinities.

Table 2 illustrates this by showing lexical affinities with the ten most highly ranked words in the frequency list of the blog plus the domain corpus. A significant collocation is defined by Oakes [6] as "the probability of one lexical item co-occurring with another word or phrase within a specified linear distance or span being greater than might be expected from pure chance.". Table 2 is ordered with the most statistically significant collocations at the top. In a word span of one, collocations represent adjacencies which may indicate multi-word terms such as “boarding pass”. Non-adjacent collocations (word span > 1) may indicate different domain concepts that participate in some relationship. The lexical affinities in Table 2 are ranked in order of significance, using a word span of 5. Many of the lexical affinities come from adjacencies. Hence, the list shows compound terms such as “hand luggage” and “carry-on luggage”. However, it also shows non-adjacencies such as “department *of homeland* security”.

We have experimented with other shallow NLP techniques that also offer a useful complement to the techniques described or another way of viewing the text. Semantic taggers provide a shallow form of semantic analysis, sometimes called *word-sense resolution*, that can be used to classify words or groups of words according to a set of defined semantic categories. As with PoS tags, the analyst can use semantic tags to filter information and infer the meaning of phrases and passages in which they occur. A semantic tagset is derived from a taxonomy of semantic categories. Perhaps the best known word-sense taxonomy is WordNet [13] although our tools use a tagset derived from McArthur’s classification [14]. For example, semantic tagging could have been used to collect together synonyms of “luggage” to reveal “baggage” and “bag”. Had we used the underlying semantic classification of luggage/baggage/bag as the seed for lexical affinity analysis instead of the actual term, collocation analysis would have revealed “transparent bag” as a significant compound term. This would have reflected the fact that our domain corpus was compiled at a time when passengers were being required to place various items of hand luggage in transparent plastic bags.

The discussion above has stressed combinations of shallow NLP techniques. In our most recent work [9, 10], we have developed a tool, called *OntoLancs*, for investigating how best to combine individual techniques as ensembles. OntoLancs provides a protocol that enables NLP techniques to be treated as plug-ins. Once integrated, different techniques can be combined using a graphical language (Fig. 3). OntoLancs also provides support for organizing the discovered concepts into a domain ontology and encoded using the OWL ontology language. If a domain ontology already exists,

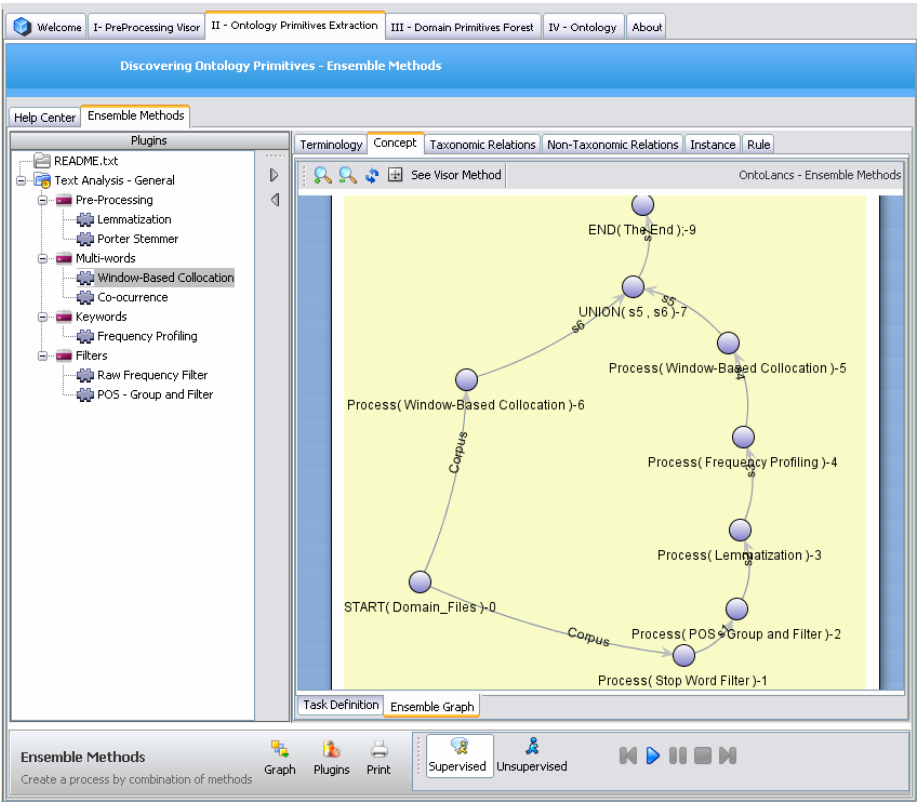


Fig. 3. Defining NLP Technique Ensembles with *OntoLancs*

the results of applying NLP techniques can be compared against it and the techniques' performance can be benchmarked. We are using this feature to evaluate the relative performance of different ensembles of NLP techniques in order to guide their application in analysis problems.

NLP techniques will never be capable of automating the derivation of requirements. Despite this, they have a role in assisting the human analyst's task of making sense of the myriad sources of information needed to inform the synthesis of user requirements. Whether NLP-assisted or not, information can be lost during the synthesis process, particularly when that information never existed in explicit form. The next section examines the role that shallow NLP techniques can play in recovering this lost information.

3 Upstream Trace Recovery

The process of user requirements synthesis is the first step in transitioning from the informal to the formal, although it is far from a simple activity and may involve (for example) goal modeling, scenario derivation, brainstorming and much else. Given the

complexity of the process, it is good practice to record the synthesized requirements' motivation since maintenance of an explicit record helps inform trade-offs and allows backwards tracing to the stakeholders or information sources that motivated the requirements. Such upstream or pre-requirements specification tracing [1] is, for a variety of reasons, commonly neglected.

Downstream tracing or post-requirements specification tracing is also commonly neglected, despite the ready availability of commercial requirements management (RM) tools that directly support downstream tracing. This failure of basic RM practice has motivated several researchers to investigate automatic downstream trace recovery. Techniques borrowed from information retrieval (IR) have been shown to be capable of inferring relationships between requirements at different levels of elaboration [15, 16, 17]. When benchmarked using sets of requirements for which a manual trace record already existed, downstream trace recovery [16, 17] can achieve approximately 90% recall, at about 20% precision. Such a balance of recall and precision appears to be acceptable to analysts, perhaps because errors of commission are generally easier to deal with than errors of omission. It is worth noting that the alternative to automatic trace recovery is manual trace recovery. Manual trace recovery is very expensive so imperfect recall of automatic trace recovery tools at the 90% level is easily justified. In all practical terms, the alternative to automatic trace recovery is that *no* requirements traces will be recovered.

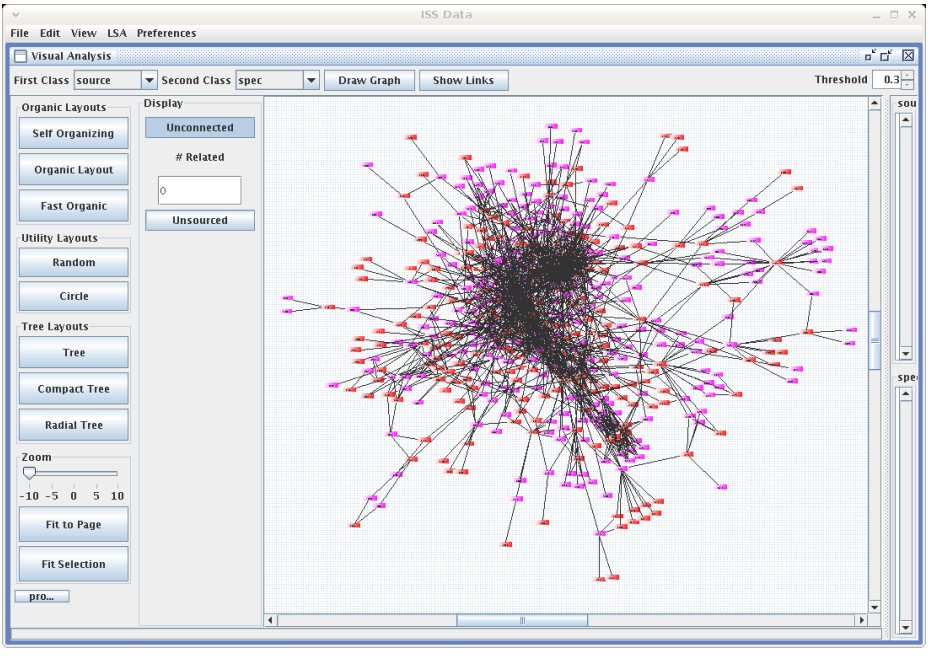


Fig. 4. An Organic Layout Algorithm Used to Display Pre-RST *Derives* Relationships

In our work, we have applied IR techniques to upstream trace recovery using our *Prospect* tool [18]. Prospect is designed to infer *derives* relationships between user requirements and the elicited knowledge that motivated them. Our hypothesis is that a user requirement that participates in a semantic relationship with passages of elicited text is likely to have been motivated in some way by the information embodied by the elicited text. Fig. 4. illustrates the results of using Prospect. It shows a cluster of several hundred requirements and passages of text from the information elicited from the stakeholders in a project. The scale is too small to see clearly here but requirements and "source" passages are represented as nodes. The arcs represent inferred derives relationships between requirements and their sources and the tight clusters reveal where the multiplicity of relationships is high.

Automatic trace recovery tools use a measure of the lexical similarity between two requirements statements to infer semantic relatedness. The IR technique used by Prospect, LSA [19], also measures lexical similarities but is able to do more than count the number of co-occurring words. Prospect can infer a relationship between (say) a user requirement and passages of elicited information even when the terms used are somewhat dissimilar, provided that the terms that are used occur sufficiently commonly in similar contexts for LSA to infer synonymy or polysemy. Hence, LSA can recognize concepts than underlie lexical signifiers. For example, "airline" and "carrier" are often used as synonyms of the same underlying concept, and LSA offers a mechanism to recognize this without requiring the manual construction of a glossary.

The results of our evaluations suggest that Prospect is capable of upstream trace recovery with a similar level of performance to downstream trace recovery. This claim needs to be qualified by the fact that we have been unable to identify any public-domain manually-traced upstream requirements data sets against which to benchmarking Prospect's performance. In our case studies, therefore, we have had to generate benchmarking data by manually reverse-engineering traces between user requirements and transcripts of elicitation exercises. Such a procedure represents a threat to the validity of estimates of prospect's performance. It is also important to recognize that there are other variables that have to be taken into account that are peculiar to upstream trace recovery and which make the achievable performance context-dependent. These variables include the completeness of the source text and the granularity with which monolithic source documents are partitioned.

Although the airline security blog was not intended to illustrate trace recovery, we used it as input to Prospect. The blog entries are not requirements, although they contain information that an analyst might use to inform the synthesis of requirements. Similarly, the corpus of domain documents is not representative of material an analyst would elicit from stakeholders, but it might plausibly embody knowledge that an analyst could use to develop an understanding the problem domain. Relationships detected between blog entries and passages of corpus text do not represent the *derives* relationships that Prospect was designed to identify, but other forms of relationship might be expected to exist. Prospect detected significant semantic relationships between six of the thirteen blog entries and passages of text from the corpus. A significant number of passages from the corpus showed a relationship with the first blog entry:

"We have to ban on airplane passengers taking liquids on board in order to increase security following the recent foiled United Kingdom terrorist plot. We are also working on technologies to screen for chemicals in liquids, backscatter, you know?"

One of the interesting things about this entry, and the blog as a whole, is that there is no explicit rationale for why passengers should be prevented from carrying liquids on board an airplane. However, the rationale *is* provided by several of the corpus passages that Prospect linked with the blog entry, including, for example:

"Claims that terrorists were plotting to use liquid explosives suggest they understood the limitations of current bomb detection methods, experts say."

Note that the common occurrence of "liquid"/"liquids" in the blog and corpus passages suggests that the relationship was inferred from lexical similarity only. This supports the good performance reported by techniques based purely on lexical similarity, such as [15] but offers no insight into the advantages of using the more computationally-intensive LSA.

LSA's tolerance of inconsistent vocabulary can be tested by our earlier observation that we would expect the synonymy of "airline" and "carrier" to be recognized. The corpus contains four passages of text that use the term "carrier" and many more that use "airline". Prospect identified relationships between two of the passages that used "carrier" with passages using "airline". That the recall was less than 100% reflects the fact that the weight that LSA attaches to two passages of text is proportional to the number of terms and concepts they share. A single shared concept such as that represented by "airline"/"carrier" is often insufficient in itself to show up as a strong degree of relatedness. This is not a failing of LSA; the fact that two passages of text mention either "airline" or "carrier" need not mean that they share deep semantic meaning. An example of genuine semantic relatedness is illustrated by the following two passages of text that Prospect correctly inferred a relationship between:

"Travelers are urged to check with airlines in advance."

"Passengers are strongly advised to check the website of their carrier or airport before travelling."

Note that the semantic relatedness is revealed not only by the synonyms "airline" and "carrier", but also by the shared term "check" and another pair of synonyms: "traveler" and "passenger".

Our simple experiment confirmed our hypothesis that interesting semantic relationships would exist between blog entries and the corpus. In addition to offering an insight into the advantages of LSA over techniques based on purely lexical similarities, the experiment suggests that the utility of tools like Prospect might extend beyond trace recovery to provide more general assistance for analysts. However, there is one utility of tools such as Prospect that is revealed by performing trace recovery and which we explore in the next section.

4 Unprovenanced Requirements

An interesting phenomenon that is commonly revealed by applying Prospect to upstream trace recovery is that of *unprovenanced* requirements. If the elicited information exists in text form, Prospect is typically able to infer derives relationships between user

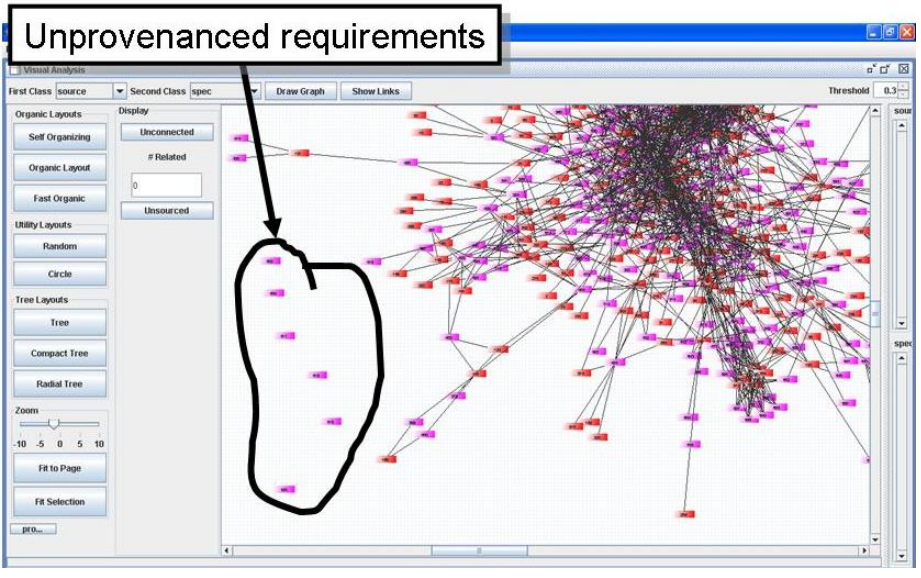


Fig. 5. Unprovenanced Requirements Revealed by *Prospect*

requirements and passages of the elicited text. The strength of a relationship between a user requirement and passages of elicited text can vary according to the lexical similarities that exist between them, but with the tolerance of synonymy and polysemy that LSA affords. In the case studies conducted so far, a minority of requirements appear to have no relationship with the elicited text. This is illustrated in Fig. 5 in which a number of apparently unprovenanced requirements can be seen, indicated by the fact, in contrast to the other visible requirements, that they are not connected to any passages of source text by an inferred derives relationship.

The largest of our case studies was conducted on a live project and we were able to interview the analysts to validate the results. Their responses showed a strong correlation between requirements identified by Prospect as unprovenanced and those where the requirements had been invented by application of the analysts' domain knowledge.

Clearly, invention is part of the job of an analyst because they must use their knowledge and experience to creatively add value to the needs stated by the stakeholders. One common reason for the need for invention is that the information elicited from the stakeholders is incomplete. Incompleteness can be due to a number of reasons, but one is that the stakeholders hold information that they don't articulate either through deliberately withholding it or (we assume, more commonly) unconsciously withholding it. Knowledge that is never articulated, either because it is hard to articulate, or is so integral to the holder's model of the world that they don't feel the need to make it explicit is *tacit* [20, 21, 22].

A number of elicitation methods exist that help cope with tacit knowledge or concealed information [23]. EasyWinWin [24], for example, is designed to identify, refine and reach consensus on the requirements for a system over a series of steps.

These steps are carefully structured using prompts and the staged revelation of stakeholders' requirements and priorities to tease out concealed information. We hypothesize that techniques such as LSA could enhance tool support for such methods by, for example, tracing the evolution of stakeholders' requirements over stages in the elicitation process and helping highlight discontinuities that might be revealing of concealed information or tacit knowledge. Hence, in addition to helping detect the effect of tacit knowledge in existing requirements, LSA may be useful in drawing tacit knowledge and concealed information out of stakeholders during requirements elicitation.

5 Related Work

For upstream requirements engineering, any NLP technique that is intolerant of deviations from grammar rules and other defects will fail. Symbolic NLP techniques, which depend on a model of the syntax and semantics of language, are too brittle when confronted with the poorly-conformant language and inconsistent vocabulary that is characteristic of many upstream requirements sources. Hence, while work such as [25, 26, 27, 28] make valuable contributions to downstream RE, the NLP techniques they are based upon don't work well upstream.

Recognizing this, Daniel Berry and collaborators [8, 29, 30, 31] have experimented with a range of techniques to help the analyst identify domain concepts. Identifying and classifying domain concepts imposes a high cognitive load on the analyst, particularly if the volume of text they need to analyze is high. To support upstream RE, therefore, "… the desire is for a clerical tool that helps with the tedious, error-prone steps of what a human elicitor does …" [31]. The challenge is to match or exceed human analysts' performance in recall yet achieve a level of precision in which verifying the abstractions and eliminating the false positives consumes significantly less effort than a manual analysis of the text.

Findphrases [30] and AbstFinder [31] were both based on the idea that domain concepts would recur frequently as repeated words within the target document. In both, a ranked list of frequently occurring words and phrases is returned. Stop words such as conjunctions and articles act as noise and have to be filtered. In case studies, AbstFinder out-performed human analysts for recall and achieved recall of approximately 25%. The authors argued that 25% precision was an underestimate because the remaining 75% contained some valid domain concepts that had not been included in the manual analysis document that was used to benchmark the performance. Precision of 25%+ represents good performance but still risks genuine domain concepts being overlooked in the noise. The ideas behind Findphrases and AbstFinder were refined by Lecœuche [7] who, in parallel with our work pioneered the use of corpus-based frequency profiling in RE.

Lexical affinities [8] represent collocations of words within text. As a stand-alone technique, Maarek found that Lexical affinities performed somewhat worse than AbstFinder on recall but achieved slightly better precision.

Our own early work has explored the use of a range of shallow, primarily corpus-based, NLP techniques for domain concepts, concluding that corpus-based frequency profiling offered the best performance [4]. One of our conclusions from this work was that by combining frequency profiling with other, individually poorer-performing

techniques, would help improve performance with little additional overhead. Our recent work [9, 10] is designed to test this hypothesis.

In the discussion above, we suggested that shallow NLP techniques are primarily useful for upstream RE. Recently however, Chantree *et al.* [32] have demonstrated that such techniques may also be useful for downstream RE problems, using corpus linguistics to identify ambiguities in requirements. Not all ambiguities in requirements are damaging and Chantree *et al.* focus their work on detecting ambiguities that *are* damaging; what they term *nocuous* ambiguities.

A different set of techniques have been found useful for identifying relationships between requirements and between requirements and other textual artifacts. Here, techniques developed by the IR research community for discovering *document similarity* have proved most useful. A number of researchers [15, 16, 17] have realized that textual requirements can be subjected to IR techniques to infer requirements similarity in order to infer *derives* relationships for trace recovery. In experiments using data sets of manually traced requirements as benchmarking data, they show that IR-based trace-recovery tools are capable of discovering up to, and sometimes over, 90% of the downstream trace relationships with generally reasonable precision.

One of the most useful families of IR techniques for trace recovery has proven to be that based on the *vector-space* model [33]. In the vector space model, each document is represented as a vector. The number of dimensions of the space that contains the vectors is proportional to the number of unique words in the combined vocabulary of all documents being compared. The magnitude of each vector (i.e. each document) in each dimension (i.e. each unique term) shows the frequency of each word in each document. Several existing trace recovery tools, including ReqSimile [15] and RETRO [16], use a vector space model. However, most variants of the vector space model are unable to cope with particular challenges posed by upstream tracing. Ravichandar *et al.* [34] conceptualize the upstream trace recovery problem in terms of coupling and cohesion. They argue that the loose coupling and high cohesion exhibited by a well-formed requirements specification is characteristically absent in the requirements sources. Our concern here is only with a subset of the problems identified by Ravichandar *et al.*, those that are linguistic, particularly the problems of inconsistent vocabulary.

One solution to linguistic inconsistency is to identify the inconsistent vocabulary in advance, so synonyms and polysemes in the text can be replaced with consistent terminology before the vector space is constructed. This can be achieved by manually constructing a thesaurus that is used to preprocess the traceable artifacts before candidate link generation. Thesaurus construction has been used to good effect in post-requirements trace recovery [16] but it comes at the cost of requiring analyst effort. This threatens the scaleability of upstream trace recovery where significant variation is the norm.

LSA offers an alternative, computational solution to inconsistent vocabulary by using Singular Value Decomposition (SVD) to post-processes the vector space. Despite being strictly statistical, SVD takes account of documents' vocabularies to arrange their vectors by patterns of word usage. The effect of SVD is to reduce the effects of synonymy and polysemy considerably, although it is also the computationally intensive step

of LSA. The extra computational cost of LSA over "vanilla" vector space model-based techniques makes LSA an expensive solution for downstream trace recovery that appears to offer few substantial benefits. Our work has shown that its real value in RE is to upstream trace recovery.

6 Conclusions

In [35], Kevin Ryan offered a critique of the application of natural language processing techniques to requirements engineering problems. Among Ryan's key observations was that it was both unfeasible and undesirable to automate the derivation of requirements from natural language text. Fourteen years later, Ryan's view still holds. Instead, work has focused on using NLP techniques as a tool to aid the human analyst. We argue that in the early stages of RE where the language is inevitably uncontrolled, shallow NLP techniques hold real promise as the basis for viable analysts' tools.

Before considering the application of NLP techniques to RE, it is crucial to understand the limits to what they can achieve. The automatic derivation of requirements from information elicited about the problem domain is unfeasible and will always remain so. However, help for the identification of domain concepts that the analyst can use for the manual construction of analysis models is feasible. Our research indicates that it is difficult to achieve adequate performance with any individual NLP technique. However, ensembles of techniques, combined in an appropriate way, can achieve sufficiently high levels of performance to make them genuinely useful.

One of the reasons why the automation of the analyst's task is unfeasible and undesirable is that much of the information that the analyst needs in order to formulate appropriate requirements is likely to be unstated. We have described how latent semantic analysis, when applied to upstream trace recovery can highlight disconnects between the formulated requirements and the information elicited from stakeholders. It appears that this disconnect is sometimes a symptom of missing or incomplete information, which in turn can be caused by stakeholders failing to articulate their knowledge. We believe that the ability to detect evidence of tacit knowledge is useful in itself and may form a component in a toolset for improving how tacit knowledge is handled within RE.

References

1. Gotel, O., Finkelstein, A.: An analysis of the requirements traceability problem. In: 1st International Conference on Requirements Engineering (ICRE 1994), pp. 94–101. IEEE Computer Society Press, Los Alamitos (1994)
2. http://fabrice.kordon.free.fr/Monterey2007/home.html
3. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
4. Sawyer, P., Rayson, P., Cosh, K.: Shallow Knowledge as an Aid to Deep Understanding in Early-Phase Requirements Engineering. IEEE Trans. Software Engineering 31(11), 969–981 (2005)

5. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In: 3rd IEEE International Symposium on Requirements Engineering (RE 1997), pp. 226–235. IEEE Computer Society Press, Los Alamitos (1997)
6. Oakes, M.: Statistics for Corpus Linguistics. Edinburgh University Press, Edinburgh (1998)
7. Lecœuche, R.: Finding comparatively important concepts between texts. In: 15th IEEE International Conference on Automated Software Engineering (ASE 2000), pp. 55–60. IEEE Computer Society Press, Los Alamitos (2000)
8. Maarek, Y., Berry, D.: The Use of Lexical Affinities in Requirements Extraction. In: 5th International Workshop on Software Specifications and Design, pp. 196–202. ACM, New York (1989)
9. Gacitua, R., Sawyer, P., Rayson, P.: A Flexible Framework to Experiment with Ontology Learning Techniques. Knowledge-Based Systems 21(3), 192–199 (2007)
10. Gacitua, R., Sawyer, P.: Ensemble Methods for Ontology Learning - An Empirical Experiment to Evaluate Combinations of Concept Acquisition Techniques. In: 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008), pp. 328–333. IEEE Computer Society Press, Los Alamitos (2008)
11. Archer, D., Rayson, P., Piao, S., McEnery, T.: Comparing the UCREL Semantic Annotation Scheme with Lexicographical Taxonomies. In: 11th EURALEX International Congress (Euralex 2004), pp. 817–827, Université de Bretragne Sud (2004)
12. Garside, R., Smith, N.: A Hybrid Grammatical Tagger: CLAWS4. In: Corpus Annotation: Linguistic Information from Computer Text Corpora, pp. 102–121. Longman, London (1997)
13. Miller, G.: WordNet: a lexical database for English. Comms. ACM. 38(11), 39–41 (1995)
14. McArther, T.: Longman Lexicon of Contemporary English. Longman, London (1981)
15. Natt och Dag, J., Regnell, B., Carlshamre, P., Andersson, M., Karlsson, J.: A Feasibility Study of Automated Support for Similarity Analysis of Natural Language Requirements in Market-Driven Development. Requirements Engineering 7(1), 20–33 (2002)
16. Huffman-Hayes, J., Dekhtyar, A., Karthikeyan Sundaram, S.: Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. IEEE Trans. Software Engineering. 32(1), 4–19 (2006)
17. Cleland-Huang, J., Settimi, R., Romanova, E., Berenbach, B., Clark, S.: Best Practices for Automated Traceability. IEEE Computer 40(6), 27–35 (2007)
18. Stone, A., Sawyer, P.: Identifying Tacit Knowledge-Based Requirements. IEE Proc. Software. 153(6), 211–218 (2006)
19. Deerwester, S., Dumais, S., Furnas, G., Landauer, T., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. For Inf. Sci. 41(6), 391–407 (1990)
20. Polanyi, M.: The Tacit Dimension. Peter Smith, Gloucester, Ma (1983)
21. Nonaja, I.: A dynamic theory of organizational knowledge creation. Organization Science 5(1), 14–37 (1994)
22. Busch, P., Richards, D., Dampney, C.: The graphical interpretation of plausible tacit knowledge flows. In: Asia-Pacific symposium on Information visualization (APVis 2003), pp. 37–46. Australian Computer Society (2003)
23. Collins, H.: What is tacit knowledge. In: The practice turn in contemporary theory, pp. 115–128. Routledge, London (2001)
24. Grünbacher, P., Briggs, R.: Surfacing Tacit Knowledge in Requirements Negotiation: Experiences using EasyWinWin. In: 34th Hawaii International Conference on System Sciences, pp. 8–15. IEEE Computer Society Press, Los Alamitos (2001)

25. Fabrini, F., Fusani, M., Gnesi, S., Lami, G.: An automatic quality evaluation for natural language requirements. In: 7th International Workshop on Requirements Engineering: Foundations for Software Quality (REFSQ 2001), Essener Informatik Beiträge, Essen, Germany (2001)
26. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Applications of linguistic techniques for use case analysis. Requirements Engineering 8(9), 161–170 (2003)
27. Mich, L., Mylopoulos, J., Zeni, N.: Improving the quality of conceptual models with NLP tools: an experiment. Technical Report DIT-02-0047, University of Trento (2002)
28. Garigliano, R., Morgan, R., Smith, M.: The LOLITA system as a contents scanning tool. In: 13th International Conference on Artificial Intelligence, Expert Systems and Natural Language Processing, Avignon, France (1994)
29. Berry, D., Yavne, N., Yavne, M.: Application of Program Design Language Tools to Abbott's method of Program Design by Informal Natural Language Descriptions. J. Systems and Software. 7(3), 221–247 (1987)
30. Aguilera, C., Berry, D.: The Use of a Repeated Phrase Finder in Requirements Extraction. J. Systems and Software. 13(9), 209–230 (1990)
31. Goldin, L., Berry, D.: AbstFinder, A Prototype Natural Language Text Abstraction Finder for Use in Requirements Elicitation. Automated Software Engineering 4(4), 375–412 (1997)
32. Chantree, F., Nuseibeh, B., de Roeck, A., Willis, A.: Identifying Nocuous Ambiguities in Natural Language Requirements. In: 14th IEEE International Conference on Requirements Engineering (RE 2006), pp. 56–65. IEEE Computer Society Press, Los Alamitos (2006)
33. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Comms. ACM. 18(11), 613–622 (1975)
34. Ravichandar, R., Arthur, J., Pérez-Quiñones, M.: Pre-Requirement Specification Traceability: Bridging the Complexity Gap through Capabilities. In: International Symposium on Grand Challenges in Traceability, TEFSE/GCT 2007 (2007)
35. Ryan, K.: The Role of Natural Language in Requirements Engineering. In: 1st IEEE International Symposium on Requirements Engineering (RE 2003), pp. 240–242. IEEE Computer Society Press, Los Alamitos (1993)

Author Index