UCOT: SEMIAUTOMATIC GENERATION OF CONCEPTUAL MODELS FROM USE CASE DESCRIPTIONS

Tommi Kärkkäinen, Miika Nurminen, Panu Suominen, Tuomo Pieniluoma, Ilari Liukko
Department of Mathematical Information Technology
University of Jyväskylä, Finland
{tka,minurmin}@mit.jyu.fi, panu.suominen@iki.fi, {tujupien,ilanliuk}@cc.jyu.fi

ABSTRACT

This paper describes a prototype of a system to automatically analyze use cases and create a conceptual model based on the analysis. Grammatical parser and Abbott's heuristic are used to process the use cases. User can modify the conceptual model by refining entities and relations, as well as roles for the entities. The model can be exported to be further utilized with object oriented analysis or domain-specific modeling. The system is evaluated based on the analysis of use cases used to describe the system itself.

The quality of models depends essentially on the quality and writing conventions of the use cases. If the use cases are written using subject-predicate-object structure and usage of synonyms is minimized, then the system can produce appropriate conceptual model of the use cases, facilitating requirements analysis and domain engineering.

KEY WORDS

Use case, conceptual model, Abbott's heuristic, natural language processing, domain engineering.

1 Introduction

The purpose of requirements analysis is to understand the needs of a customer towards a new application in its domain, and document this understanding in a clear way for the customer to approve and for developers to realize the system. An important part of the analysis is conceptual modeling [1], the development of domain models which capture the shared knowledge driving collaboration among stakeholders.

Use cases [2] provide a popular way to describe functional requirements of a software system. Compared to the classical "The System shall..." type of requirements lists or user stories [3] use cases provide a process-like view of the requirements. Following the terminology in [4], use cases are utilized both in plan-driven methods [5] and agile methods [6]. Notice that the so-called derivation technique as given in [7] determines (domain) classes through use cases and conceptual models. Because use cases in their main flow describe the interaction between users and the system, as well as include possible relations to other use cases, they provide both contextual and structural information for problem solving communication. From our point of view, it is precisely this dialogue that provides the starting point for semiautomatic generation of a conceptual model.

We present here a prototype of a system which is designed to automatically analyze use cases and create a conceptual model based on the analysis. The idea of such an automatic transformation is not new [8, 9, 10], but to best of our knowledge, use cases have not been used as an input in the earlier work. As in [9], the so-called Abbott's heuristic [11] is used as the central ingredient in the modeling process. The basic idea of our approach is that no design-level decisions (e.g. object-orientation [12]) are included in the model. Instead, we try to represent the relations and entities within the application domain as the use cases describe them. The system is not trying to guess whether a term refers to class or its instance, rather all of them are considered as just entities of the current problem domain. Subsequently, the model can be used to proceed, for example, with object oriented analysis [13].

Contents of the work are as follows: Section 2 contains a short depiction of relevant related research. Section 3 describes the main functionality and design of the UCOT (*from Use Cases to Original enTities*) system. Samples illustrating the system behavior and its limitations are given and analyzed in Sections 4 and 5. Finally, conclusions and some directions to enlarge the system are provided.

2 Preliminaries

This section provides a short introduction of relevant fields of related research. Basically our SuD (System-under-Description) uses natural language processing/parsing (Section 2.1) techniques to instrument the use case for the subsequent analysis (Section 2.2) which creates the model that can be further utilized with object oriented analysis and as part of domain engineering efforts (Section 2.3).

2.1 Natural Language Parsing

Natural language parsing is a complicated task. Historically there have been two main approaches to address the problem. Structural method uses different kinds of grammars to produce a hierarchical parse tree [14]. The other

approach focuses on dependencies between the words and parts of the phrases, providing a more meaning-oriented approach [15].

Currently statistical parsers dominate the research field [16]. Many of them build a context-free grammar from annotated text. This text is called a corpus or, if tagged in the form of a parse tree, treebank. While parsing the parser tries to find the most preferable parse tree for the given sentence using probabilistic and statistical methods.

In our system the implementation of the parser is hidden behind a modular architecture. The system is developed considering the output of structural parsers. Basically we are interested in extracting information that the heuristic needs. Mostly this means finding subject, object, and predicate within the textual description of requirements.

2.2 Heuristic Rules for Model Generation

The oldest approach for identifying classes from natural language phrases is the noun analysis, introduced by Abbott in 1983 [11]. This approach is also known as Abbott's heuristic, where objects correspond to nouns and methods to verbs. This approach was inspired by Booch's object oriented design method of analyzing data flow diagrams (see [12]). Abbott's approach can be viewed as a simple way of underlining nouns as possible objects found in the analyzed text. Several other approaches to identify classes from natural language exist today, such as Taxonomic Class Modeling methodology [8]. What all these approaches have in common is that they all require natural language processing of the material that is being analyzed.

In our system the heuristic module applies some of the heuristic rules presented by Abbott [11]. Not all of the rules were implemented because they would need knowledge of the language of the use cases, and this would have made the changing of the language by just switching the parser impossible. Only the simpler rules (nouns to entities, and verbs to relations between entities) were implemented.

2.3 Domain-centric approaches

The goal of domain engineering is to create reusable domain components [17]. It is based on the idea of being able to recreate similar software systems with less overlapping components by identifying domains, discovering commonalities, and bounding them together. This information is represented in domain models and explicated with domain analysis [18].

Domain models include and represent a set of requirements that are common for a set of applications - a product line [19]. With our approach it could be possible to (semi)automate the analysis of shared features, thus helping the creation of a product line. More precisely, creating a conceptual model can help locating recurring patterns of domain entities, helping to find common attributes among different systems or within a single system. These findings can then be formalized using ontology engineering, by modeling general-purpose, shared domain knowledge, independent of a particular system or application [20].

Utilization of domain specific models and languages (e.g. resulting from a metalevel analysis) can yield to automatic generation of code from domain models [21]. However, doing model-based code generation means that the model itself becomes part of the solution. Here we restrict ourselves to the field of problem analysis, i.e. concentrate on what is to be developed without committing on how to do it. Especially, our models are on higher abstraction level compared to OOA (even if domain entities can be considered as and transformed to objects), whose utilization fixes the further development along object-oriented track [22]. Moreover, automatic OOA is not straightforward because requirements specifications often produce only sparse information on domain objects [23].

To conclude, the strong coupling between the problem and its solution is characteristic for the domain-specific and OOA approaches. In particular, with these approaches the focus in the analysis phase easily shifts into the solution (design) structures instead of the main purpose: to analyze the main concepts in the domain, their unambiguous meaning, and preliminary structural relations.

3 The System

UCOT system is structured according to the classical *pipes and filters* architectural style [24]. The system was designed to make it possible to add or replace components (i.e. filters, modules) later. Because of the modular design it is easy to switch to different natural language parser or use new import or export data types by realizing the corresponding component (see Figure 2). Key component of the system is `Core`, whose responsibility is to control other modules, loading them on startup, and direct the data flow.

Figure 1 shows how the use case is processed in the system. The use case is first loaded (`InputAdapter`) and then parsed in an internal data structure (`ParserAdapter`). After this the conceptual model is formed from parsed information by applying heuristic rules into it (`HeuristicModule`). The achieved model can be modified before saving or exporting it (`OutputAdapter`).

3.1 Loading and Parsing Use Cases

The system can read different kinds of input files using different combinations of the input and parsing modules. Here use case means a textual structure that has title and sequence of steps which are considered one sentence long each. Additionally use case steps can refer to some other use case. After the structure is read from the file, parser component takes care of the parsing the use case steps.

Two input formats were implemented in the system: structured text file (see Section 4) and ProcML [25]. Structured text contains the use case information separated by line feeds and tags written in square brackets. ProcML is

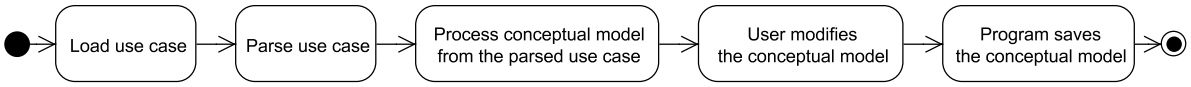

Figure 1. Flow of processing the use case into conceptual model

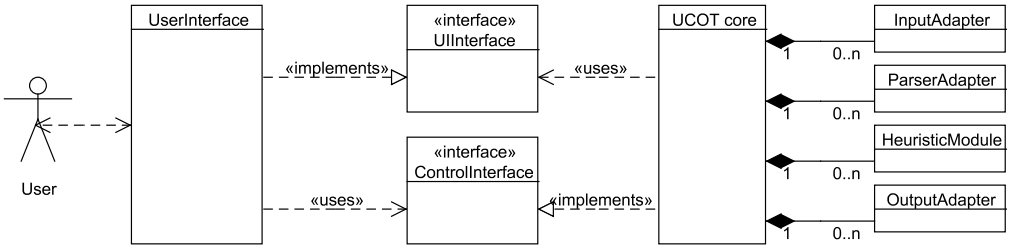

Figure 2. The architecture of the system

an XML-based language that describes business processes and their interactions. The system is able to extract information from process steps and use them as use case steps.

After the input file is read and use case steps are extracted the system parses the sentences using parser module, producing the so-called parsed use case. In a parsed use case steps have been annotated with the grammatical information obtained from the parser.

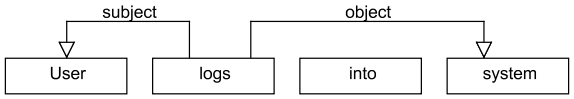

Figure 3. The phrase model used in the system

Figure 3 is an example of representation that is used in the parsed use case. This structure is used to do the analysis to produce the conceptual model. The phrase model contains words and their relations in the sentence. This data structure is used to pass information from the parser to `HeuristicModule`, independently from the language and implementation of the used parser. In practice, Stanford Parser[1] is used to parse use cases written in English. The parser combines probabilistic context-free grammar and probabilistic dependencies of word and tag pairs [26].

3.2 Conceptual Model Processing

The conceptual metamodel contains three kinds of relations between domain entities: influence, inheritance, and attribute (see Figure 4). The inheritance relation describes plain inheritance but it can also be used to describe that some entity is an instance of some other. The attribute relation can be seen as an aggregation or a composition in UML. The influence relation is used for other types of relationships and it is further described with a name. The influence arrow is drawn to point at the entity which is the target of action. An entity may also have a role (i.e. a stereotype) which can be written under the name of the entity.

When use cases are loaded into the system, they are added into the list of use cases. The user then selects which use case is to be processed and added into the conceptual model. If the use case is complex and contains many sub use cases the model becomes easily scattered, making it hard to analyze. Hence, it is also possible to add the use cases into the model one by one. The system highlights the entities that are derived from the lastly selected use case. User is able to modify the conceptual model by adding or removing relations, changing the cardinality of attributes, and renaming influence relations and entities. User interface of the system is illustrated in Figure 5.

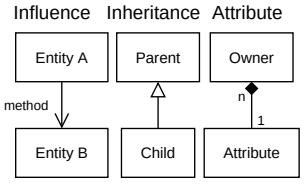

Figure 4. Relations used in the conceptual model

3.3 Model Export

The system is able to produce different kinds of output formats depending on the needs of the user. The model view is generated using visualization tool `dot` from Graphviz[2] package, capable of drawing directed graphs. Hence, the model can be exported as a plain image or as `dot`'s graph description file. These can be used when the user is more interested in visualization than further analysis of the

[1] Available at http://nlp.stanford.edu/downloads/lex-parser.shtml

[2] Available at http://www.graphviz.org/

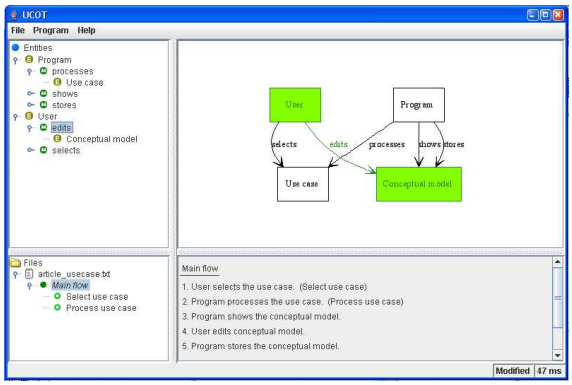

Figure 5. Screenshot from UCOT user interface

model. Adding new output formats is easy by realizing a new `OutputAdapter`. Additionally, GXL[3] output is implemented, which enables importing the produced model to another editor or analysis tool that supports the format.

4 Using the System

In this section, the usage of UCOT system is demonstrated by analyzing the use cases that were used to specify the system itself before its development. The use cases read as follows in structured text format (a number in parenthesis after a step provides a reference to another use case, describing the step in more detailed level):

```
[name] Main flow [id] 1
[steps]
  User selects the use case. (2)
  Program processes the use case. (3)
  Program shows the conceptual model.
  User edits the conceptual model.
  Program stores the conceptual model.
[end]

[name] Select use case [id] 2
[steps]
  User selects the source of the use cases.
  Program presents the list of the use cases
    contained in the source.
  User selects use case from the list of use cases.
[end]

[name] Process use case [id] 3
[steps]
  Program passes the use case to the parser.
  Parser returns the parsed use case.
  Program passes the parsed use case
    to the heuristic.
  Heuristic returns the conceptual model.
[end]
```

The descriptions are next used to illustrate the system's behavior. Each use case is first analyzed separately and the results are compared to manually determined models. Finally, all the three use cases are processed at once and the resulted model is edited to show the editing capabilities of the system. Notice that often there is no absolutely correct model and the correctness depends on what aspects are considered important and how much they are emphasized.

[3] Graph eXchange Language, see http://www.gupro.de/GXL/

4.1 Automatic model generation

Automatically produced conceptual model for the `Main flow` use case represents the concepts quite well (Figure 6). However, when the steps are more complex the created model can shift away from an ideal representation, as in `Select use case`. Its conceptual model (Figure 7) should contain some aggregations but the program does not create these automatically. "Source" should contain "Use cases" (Figure 9) instead of the plain influence relation as stated in the second step. "List of use cases" is removed from the edited picture because the aggregation relationship contains roughly the same information.

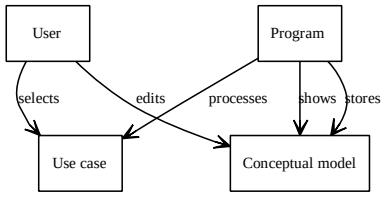

Figure 6. The conceptual model of the `Main flow` use case

While many problems occur due to limitations in the natural language parsing, there are some shortcomings in the conceptual metamodel as well. There should be a way of representing n-ary relations (i.e. relations made of three or more participants). Figure 8 is hard to read because of this. Basically the model is correct, but excessive use of "passes" is not desirable. "Passes" between "Program" and "Parser" marks the target of the passing, but with "Use case" it marks the object being passed. The created model lacks information about the roles these entities are playing while passing. Figure 10 shows a hand drawn idealized representation of the model for `Process use case`. This problem is present mainly due to the way program represents the parsed data and elaborated in Section 5.

4.2 Editing

Figure 11 shows a model that contains all the use cases with no editing. Same problems can be seen as with single use cases. The excessive use of "passes" is corrected by removing them and adding new links "uses" between "Program", "Parser", and "Heuristic". The "Heuristic" should have some connection to "Parsed use case" so "evaluates" relation is created. Same goes to "Parser" and "Use case" so the "parses" link is added between them. "Use case" is also added as an attribute to the entities "Source" which is merged from the original entities "List of use cases", and "Source of use cases". Finally, role information is added to the concepts. Figure 12 shows the final conceptual model.

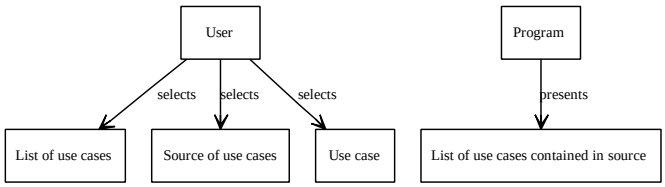

Figure 7. The conceptual model of the `Select` use case

The quality of the representation can be evaluated by comparing it to design decisions actually made. From Figure 12 one can see that user interacts with the system by editing the ”Conceptual model” and by selecting ”Use case”. All the methods implemented in `ControlInterface` (Figure 2) fall in to these two categories. Our choice for modularization can be further backed up by the limited links between ”Heuristic” and other entities. Same goes for ”Parser” entity. Because they have no excessive linkage between other items they are good candidates for modules.

The data flow in the system is harder to see, but some parts of it are visible (Figure 1). We can see that ”Parser” parses the ”Use case”; however, this is likely to happen after ”User” has selected it. Likewise, ”Parsed use case” can not be returned before ”Parser” has parsed ”Use case”. Thus, there are some implicit pointers for the order of the operations, although such dynamic information is not meant to be depicted in the static entity model.

5 Evaluation

It seems that UCOT system provides appropriate support to speed up domain understanding by focusing the domain analysis efforts on the most essential domain entities, their relations, and roles as part of the problem to be solved. More thorough evaluation of the system can be divided as follows: natural language parsing, internal representations and methods, and the quality of the produced model.

The biggest internal challenge with the system is the representation of the parsed data (Figure 3). The initial design for the structural information assumed that mostly subject and object relations (i.e. syntactic information) would be needed for the heuristic [11] which is true in the simple

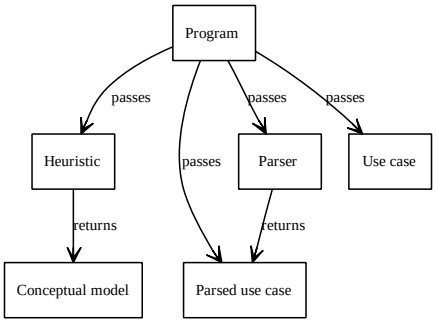

Figure 8. Corrected model of the `Process` use case

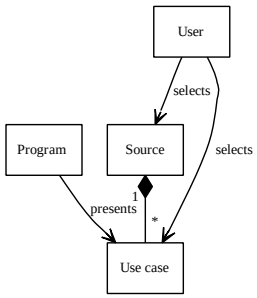

Figure 9. Corrected model of the `Select` use case

cases. Main problem is that a verb between subject and object can mean a simple interaction (”User shutdowns system”) or a more specific relation like aggregation (”User owns secure id”). Additionally, other parts of the sentence can be as important as the object. For example, in sentence ”User shutdowns the system with power switch”, ”Shutdowns” is predicate, ”User” is subject, and ”System” is object, and thus ”User” ”shutdowns” ”System” relationship is extracted. If the sentence is written differently, e.g. ”User uses power switch to shutdown the system” would produce a different result despite the fact that the meaning is approximately the same. Currently, the only place for such meaning extraction is within the heuristic module, but adding such functionality into this module would destroy its language independence. Therefore, in future development additional semantic annotations should be obtained in the parsing phase to extract more specific entity roles.

The quality of output depends essentially on the quality of the work done in earlier phases necessary to produce it. For example, Li’s method for use case normalization [10] improves the quality of the models. Use cases should be written using simple sentences, preferably with subject-predicate-object structure. Same verb should be used to describe the same action and subject or object should be referred with its full name. This is because the program does not understand the text, but is bound to handle synonyms as different words rather than combining them to one entity. This problem could be minimized by adding thesaurus check or metadata support and extract it from the input source at the parsing phase.

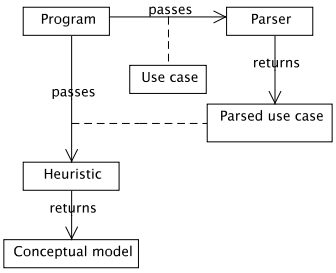

Figure 10. Idealized correction of the `Process` use case

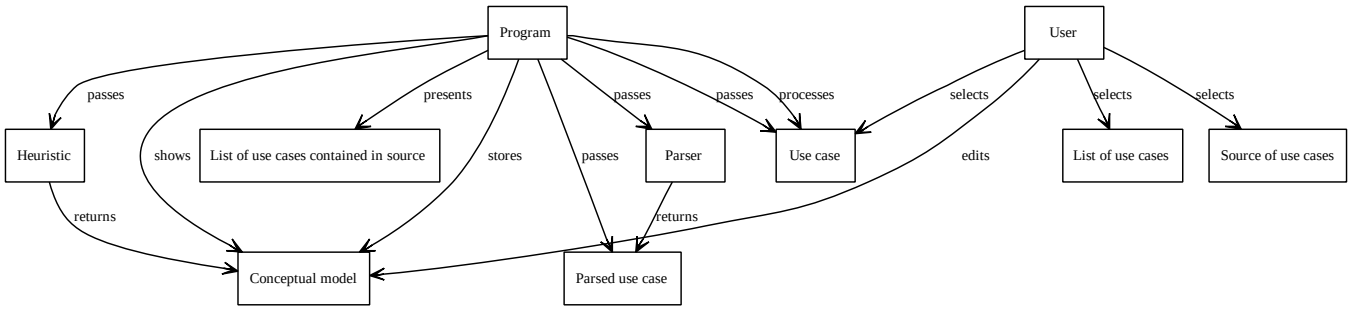

Figure 11. The combined model

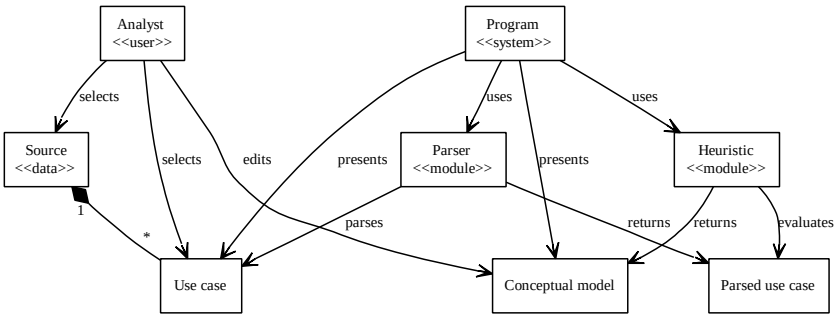

Figure 12. Modified version of the combined model

The usability of even the most perfect model can be diminished by using confusing graph layout to represent it. For example, crossing of lines representing relations are to be avoided [27], but here we are bound to `dot`. The user has little means to alter the layout, but the problem can be bypassed by exporting the model to a more sophisticated graph editor. Moreover, increase of the amount of use cases to describe system requirements also increases the need for creating more abstract views of them, although the amount of domain information in the overall model can exceed the limits of human cognitive capacity [28]. However, such challenge is evident for all models of practical size.

6 Conclusion and Future Work

We have developed and described a proof-of-concept prototype for automatic analysis of use cases. According to the general idea of pipes and filters [24], the effectiveness of our system is determined by the individual filters. However, the quality of generated conceptual models depends essentially on the writing conventions of the use cases. Ambiguities can be avoided by following certain guidelines [6, 10], such as writing use cases using subject-predicate-object structure and minimizing usage of synonyms. Although Abbott's heuristic could be augmented, it allows to produce the basic entity model, to be modified concerning the unique entities and actual form of their relations. As e.g. Figure 12 shows the model as static, but it could be presented in a more behavioral oriented (cf. sequence diagram in UML [29]). To this end, also the semantics (i.e. types of relations) of the entity model could be enlarged, but one should keep in mind that all the relevant domain stakeholders should understand all possibilities occurring in the conceptual model (cf. association types in UML [29]).

The system can be enhanced with new types of inputs and outputs. As long as the inner model used in the system contains enough information, different views of the use cases can be exported. This would ease many nowadays manual domain engineering related efforts by using the system to translate the use cases to a metamodel describing the domain components, or (provided that sufficient metadata is available) even a domain ontology, a formal specification of the concepts.

Even with the current shortcomings the system can be useful for creating representations from the use cases, although its comprehensive evaluation remains as a future work. User gets a holistic view of all the use cases included in the model. This helps keeping track of the linkages between actors and data processing, addressing possible shortcomings in the domain analysis and helps to assess the quality of the use cases [30]. Overall, semiautomatically generated conceptual models seem to provide appropriate support to requirements analysis and domain engineering.

Acknowledgements

This work was supported by National Technology Agency of Finland under project ProductionPro (Dnro 2832/31/06).

References

[1] I. Sommerville. *Software Engineering (7th Edition)*. Addison Wesley, 2004.

[2] I. Jacobson. *Object-oriented software engineering*. ACM Press, 1992.

[3] M. Cohn. *User Stories Applied For Agile Software Development*. Addison Wesley, 2004.

[4] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley, 2003.

[5] IBM. Rational unified process: Best practices for software development teams. Technical report, Rational Software White Paper, 2001.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[7] B. Anda and D. I. K. Sjøberg. Investigating the role of use cases in the construction of class diagrams. *Empirical Software Engineering*, 10(3):285–309, 2005.

[8] I.-Y. Song, K. Yano, J. Trujillo, and S. Luján-Mora. A taxonomic class modeling methodology for object-oriented analysis. In J. Krostige, T. Halpin, and K. Siau, editors, *Information Modeling Methods and Methodologies*. Idea Group Publishing, 2004.

[9] H. G. Pérez-González, J. K. Kalita, A. S. N. Varela, and R. S. Wiener. GOOAL: an educational object oriented analysis laboratory. In *Companion to the 20th ACM SIGPLAN conference OOPSLA'05*, 2005.

[10] L. Li. Translating use cases to sequence diagrams. In *ASE '00: Proc. of the 15th IEEE int. conf. on Automated software engineering*, 293–296, 2000.

[11] R. J. Abbott. Program design by informal English descriptions. *Commun. ACM*, 26(11):882–894, 1983.

[12] G. Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings, 1994.

[13] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition*. Prentice Hall, 2003.

[14] J. Carroll. Chapter 12. parsing. In R. Mitkov, editor, *The Oxford Handbook of Computational Linguistics*, 233–248. Oxford University Press, 2003.

[15] I. A. Bolshakov and A. Gelbukh. *Computational Linguistics: Models, Resources, Applications*. IPN-UNAM-FCE, 2004.

[16] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[17] J. Foreman. Product line based software development - significant results, future challenges. In *Software Technology Conference*, 1996.

[18] R. Prieto-Díaz. Domain analysis for reusability. In *Proc. of the 11th annual int. computer software and applications conf. (COMSPAC 87)*. IEEE, 1987.

[19] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[20] P. Spyns, R. Meersman, and M. Jarrar. Data modelling versus ontology engineering. *SIGMOD Rec.*, 31(4):12–17, December 2002.

[21] J.-P. Tolvanen and S. Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *Proc. of the 9th Int. Conf. on Software Product Lines, SPLC 2005*, 2005.

[22] D. W. Embley, R. B. Jackson, and S. N. Woodfield. OO systems analysis: Is it or isn't it? *IEEE Software*, 12(4):19–33, 1995.

[23] D. Svetinovic, D. M. Berry, and M. W. Godfrey. Increasing quality of conceptual models: is object-oriented analysis that simple? In *ROA '06: Proc. of the 2006 int. workshop on Role of abstraction in software engineering*, 19–22. ACM Press, 2006.

[24] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison Wesley, 2005.

[25] M. Nurminen, A. Honkaranta, and T. Kärkkäinen. ProcMiner: Advancing process analysis and management. In *IEEE 23rd Int. Conf. on Data Engineering Workshop (TDMM)*, 760–769, 2007.

[26] D. Klein and C. D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*. MIT Press, December 2002.

[27] K. Wong and D. Sun. On evaluating the layout of UML diagrams for program comprehension. *Software Quality Journal*, 14(3):233–259, September 2006.

[28] G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[29] OMG. Unified modeling language: Superstructure, version 2.1.1. Technical Report 2007-02-03, 2007.

[30] K. T. Phalp, J. Vincent, and K. Cox. Assessing the quality of use case descriptions. *Software Quality Control*, 15(1):69–97, 2007.