

Automated Conversion from a Requirements Documentation to an Executable Formal Specification *

Beum-Seuk Lee
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, Alabama, U.S.A. 35294-1170
leebs@cis.uab.edu

Abstract

In software requirements engineering there have been very few attempts to automate the translation from a requirements document written in a natural language (NL) to one of the formal specification languages. One of the major reasons for this challenge comes from the ambiguity of the NL requirements documentation because NL depends heavily on context. To make a smooth transition from NL requirements to one of the formal specification languages we need a specification that can mediate these two domains of different formalism level. We propose Two-Level Grammar (TLG) to construct a bridge between a NL requirements specification and a formal specification, and the Contextual Natural Language processing to overcome the challenges of the ambiguity in NL to promote rapid prototyping and reusability of requirements documents.

Keywords: Automated Software Specification, Contextual Natural Language Processing, Two-Level Grammar, Software Requirements Engineering, Formal Specification

1. Problem Statement

Recently many formal specification languages have been developed to handle complex systems with heavy interactions between their components by decomposition and abstraction of the requirements of the system [1]. However still the natural language (NL) has remained as the practical choice for the domain experts to specify the system because those formal specification languages are not easy to master. Therefore the requirements documentation usually written in NL has to be reinterpreted by software engineers into a formal

specification language. When the system is very complicated, which is mostly the case when one chooses to use formal specification, this conversion is both non-trivial and error-prone, if not implausible. This challenge comes from many factors such as miscommunication between domain experts and engineers. However the major bottleneck of this conversion is from the in-born characteristic of ambiguity of NL and the different level of the formalism between the two domains of NL and the formal specification. This is why there have been very few attempts to automate the conversion from requirements documentation to a formal specification language.

2. Prior Research

There have been some attempts to solve, from various angles, the problem of conversion from informal description of the system into a formal specification. To handle this ambiguity problem, some have argued that the requirements document has to be written in a particular way to reduce as much ambiguity as possible in the document [33]. Even though this approach provides a better documentation to work with, it hasn't accomplished transition between different level of formalisms.

Even with some skepticism about relying too much on natural language processing in requirements engineering [30], the common approach of the research dealing with this problem is to use natural language processing and semi-formal representation to handle the linguistic structure and information in a requirements document. This is why most of the research is more focused on the detection of errors, contradictions, and ambiguities in the requirements description and its verification using the final semi-formal representations.

It is well known that as much as 60 percent of the er-

rors that appear during a system's lifecycle have their origin in the requirements phase [9]. It is also well known that the cost to correct an error found in the development and later stages of system development is orders of magnitude higher than to correct the same error found during the requirements stage [4]. Therefore ensuring the correctness of the requirements cannot be overemphasized.

There are two distinctive types of methods for this approach; lightweight methods and heavyweight methods. Lightweight formal methods perform partial analysis on a part of specifications only whereas heavyweight methods have a commitment to translate an entire informal requirements into a formal one. Even with some merits of the lightweight methods in terms of human training and of computational resources needed [15], in most of the cases in requirements engineering the entire documentations is necessary to be analyzed. The following prior research can be categorized as heavyweight methods.

Nelken and Francez proposed a model checking method [28] of a requirements document by translating the document into temporal logic expressed in PROLOG through an intermediary representation called Discourse Representation Theory (DRT) while insuring the correctness of the translation. The syntax of the input natural language for this system is very restricted and it is usually the case that the requirements of a system describe more than sets of temporal events.

A system called NL2ACTL (Natural Language to Action-based Temporal Logic) that translates a requirements document into action-based temporal logic is developed by Fantechi et al. [11]. Using the final logic representation the system finds the logical ambiguity in the requirements. Also NL2ACTL allows the user to specify the dictionary for each system domain. The natural language sentences this system supports look more like the sentences directly paraphrased from predicate logic than real natural language sentences. Moreover the pronouns are not supported and there seem to be rather heavy dependency on user interactions to resolve ambiguities.

Vadera and Meziane proposed predicate logic combined with the entity relationship model to detect ambiguities and then to produce VDM data types and some common operations to manipulate the data types [23]. The method used in this system identifies not only the explicit but also implicit quantifiers in the requirements. Yet again this project limits the natural language by not allowing pronouns and conjunctions which occur frequently in the real-world requirements documents.

As a CASE tool based on a linguistic approach

for Databases and Information Systems development (DB/IS), OICSI (French acronym for intelligent tool for information system design) was proposed by Rolland and Proix [29]. This tool generates a conceptual model of the system by extracting the classes, their properties and constraints from the requirements in Semantic Network representation and verifies the resulting specification by paraphrasing it using text generation. However it has very restricted natural language scope which might be acceptable in some of applications in DB/IS domain. But for the general domains of the software development, this restriction is not practical. In addition, even though paraphrasing is a nice way to verify the result of the translation, it isn't sufficient as a formal verification method of a requirements document.

Another project developed as a DB/IS application using linguistic technology is CLARE (a Combined Language And Reasoning Engine) [26]. First CLARE expresses sentences in QLF (Quasi-Logical Form) representation handling reference resolution, compound nominals, pronouns, and ellipsis. Then this QLF is converted into TRL (Target Reasoning Language) as an intermediate representation to obtain SQL query statements of the specifications. This system collects contextual information for the reference resolution and dialogue-based interaction. It would be interesting to extend the scope of the project to formalize text-based requirements documents in general system domains as well.

An automated natural language based CASE tool called CM-Builder was developed to build an integrated discourse model of a requirements document represented in Semantic Networks and then to automatically construct an initial UML Class Model representing the object classes and relationships among them [19]. This system finds the missing agents using linguistic information about each sentence, identifies implicit attributes using WordNet [24], and handles pronouns. Also the methods used to evaluate and compare the system formally with other systems is commendable. Yet it isn't clear how the system deals with inconsistency and ambiguity nor how it ensures the validity of the transition of the semantic network and UML representation from the requirements.

Burg and van de Riet proposed a system that uses natural language and a scenario based approach to build CEMS (Color-X Event Models) representation of the requirements document for requirements elicitation and validation (using natural language paraphrasing and logic) [8]. The use of WordNet to find the correct word sense is a nice approach. However this process is done manually. Also the sentences the scenario can

accept are too simple for complicated real world requirements documents.

Others have proposed controlled natural languages which limit the syntax and semantics of NL to avoid the ambiguity problem. Newspeak [25] is developed to detect ambiguities in the software requirements specification using a controlled language to construct a framework called Goal-Structured Analysis (GSA). GSA constructs the specification according to the system goal by classifying each entity in the specification such as goal, effect, fact, or condition. However the system requires too much efforts from the software analyst for the disambiguation process.

Another project using controlled natural language called Attempto Controlled English (ACE) has similar goals as ours to realize the automated conversion [13]. However restrictions on the syntax and semantics of the language results in losing the flexibility of NL. Also the user still has to remember the restrictions. Moreover the target language of this controlled language is PROLOG which is good for prototyping but lacking important properties such as strong typing to be used as a formal specification language.

Goldin and Berry suggested a tool called AbstFinder to find abstractions in natural language text following a repetition-based approach mainly to assist the requirements elicitation [17].

Another approach to natural language requirements analysis is to search each line of the requirements document for specific words and phrases for the purpose of quality analysis [34]. A similar project [16] focuses mainly on the automatic indexing and reuse of the software components in the requirements documents.

In summary, there have been a few attempts to formalize an informal requirements document through use of semi-formal representations such as logic, semantic networks, controlled language, PROLOG and so on. The various goals of these attempts were requirements validation, requirements elicitation, or requirements analysis. Prototyping, paraphrasing, and logic were used to ensure the correct conversion from the informal specification into a semi-formal representation and also to detect ambiguities and contradictions in the requirements document. However the linguistic descriptions in the requirements document as the input of the system were too restricted and controlled. Also no research has achieved a full conversion of the requirements specifications into a formal specification (only semi-formal representations) and implementation of the specifications.

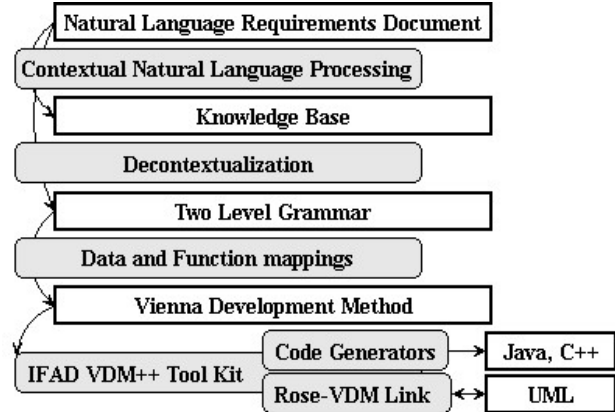


Figure 1. System Structure.

3. Project Approach

In our research project, Contextual Natural Language Processing (CNLP) [22] is used to handle the ambiguity problem in NL and Two Level Grammar (TLG) [32] is used to deal with the different formalism level between NL and formal specification languages to achieve the automated conversion from NL requirements documentation into a formal specification (in our case the Vienna Development Method - VDM [3]). First a knowledge base is built from the requirements documentation using the contextual natural language processing by parsing the documentation and storing the syntactical, semantic, and contextual information. In this phase, the ambiguity is detected and resolved, if possible. And then the knowledge base is converted into TLG by removing the contextual dependency in the knowledge base. Finally the TLG code is translated into VDM by data and function mappings. This research involves the disciplines such as NL processing, requirements analysis, formal specification, and programming languages. The entire system structure is shown in Figure 1.

The following simple specification of an Automatic Teller Machine (ATM) will be used as the running example throughout the paper to illustrate the system. The details of how the bank verifies ID and PIN and how it updates the balance are omitted for the sake of simplicity.

```

Bank keeps list of accounts. It verifies ID and PIN
giving the balance and updates the balance with ID.
An account has three data fields; ID, PIN, and
balance. ID and PIN are integers and balance is a
real number.
ATM has 3 service types; withdraw, deposit, and
balance check. For each service first it verifies ID
and PIN from the bank giving the balance.
  
```

ATM withdraws an amount with ID and PIN giving the balance in the following sequence. If the amount is less than or equal to the balance then it decreases the balance by the amount. And then it updates the balance in the bank with ID. ATM deposits an amount with ID and PIN giving the balance in the following order. It increases the balance by amount and then updates the balance in the bank with ID. ATM checks the balance with ID and PIN giving the balance.

3.1. Construction of Knowledge Base from Requirements Document

The knowledge base is built from a document using syntactic, semantic and contextual (discourse) information. The knowledge base has to be constructed in a way to store the information without any data redundancy while facilitating ease in manipulating the data [12]. Also the knowledge representation has to capture the corresponding structure of TLG for the later translation. When the document is read by the system, the system parses it into sentences and each sentence into words. At the syntactical level, the part of speech (e.g. noun, verb, adjective) of each word is determined by bottom-up parsing, whereas the part of sentence (e.g. subject, object, complement) of each word is determined by top-down parsing [2]. Separating the parsing process into these two different sub-processes, a unique approach compared with most other parsing techniques, makes the algorithm simpler because the latter process is very context-sensitive about the features like verb form and sub-categorization whereas the former one is context-sensitive about person and number features [14]. By using the predetermined part of speech for each word from the part of speech parsing, the number of the rules for the context free grammar for the part of sentence parsing is reduced substantially. The corpora of statistically ordered part of speeches (frequently used ones being listed first) of about 85000 words from [18] are used to resolve the syntactic ambiguity in this phase. Also elliptical compound phrases, comparative phrases, compound nouns, and relative phrases are handled in this phase as well. A part of the result of this process for the ATM example is shown as follows.

```
Bank keeps list of accounts
Part of speech : bank(noun) keeps(verb)
                accounts_list(noun)
Part of sentence : ( subject verb object )

It verifies ID and PIN giving the balance and
updates the balance with ID
Part of speech : it(pronoun) verifies(verb) ID and
                PIN(noun) giving(verb) the(article) balance(noun)
Part of sentence : ( subject verb object
                    helping:( verb adjective object ) )
Part of speech : it(pronoun) updates(verb)
```

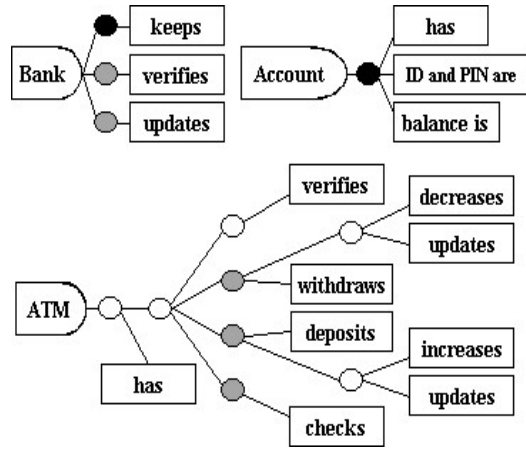


Figure 2. Knowledge base for ATM.

```
the(article) balance(noun) with(preposition)
ID(noun)
Part of sentence : ( subject verb adjective object
                    adverb preposition_object )
```

Using the semantic information and the syntactic information gathered from the previous phase, anaphoric references (pronouns) are identified. A pronoun can represent a word, a sentence, or even another context.

Once the references of pronouns are determined, each sentence is stored into the proper context in the knowledge base. This involves the syntactic, semantic, and most importantly the contextual information. This part of the project is the most challenging part because if a sentence is located in a long context, the meaning of the sentence can totally change than what is originally meant. A contextual knowledge base is formalized as a tree-like data structure not only to store each sentence in its correct context but also to make a smooth conversion from the knowledge base to TLG. Meta-level context (context for context) determines where to put each sentence in the tree according to the discourse level information. The current context is created or switched dynamically. For instance, in the ATM example the phrase “in the following sequence” indicates that the following sentences are likely to stay in the same context as the current context. Alternatively a sub-context to hold the following sentences has to be created under the current context. The contextual structure of the knowledge base is shown in the Figure 2.

In Figure 2, only the main verb for each sentence is shown in a rectangle due to the space limitation. The black ovals indicate the contexts that hold the data type information whereas the gray ovals indicate the contexts that contain the functional information. Note the location of the sentence “For each service first it

verifies ID and PIN from the bank giving the balance”. It is positioned under the same context with the sentences of the withdraw, deposit, and balance check service operations to be used for each service later. If it were located in any other context, the sentence couldn’t operate as originally intended. In summary, the syntactic, semantic, and contextual information is used to build up the contextual knowledge base from the requirements documentation.

3.2. Conversion from Knowledge Base to Two Level Grammar

Two-Level Grammar (TLG) may be used to achieve translation from an informal NL specification into a formal specification. Even though TLG has NL-like syntax its notation is formal enough to allow formal specifications to be constructed using the notation. It is able not only to capture the abstraction of the requirements but also to preserve the detailed information for implementation. The term “two level” comes from the fact that a set of domains may be defined using context-free grammar, which may then be used as arguments in predicate functions defined using another grammar. The combination of these two levels of grammar produces Turing equivalence [31] and so TLG may be used to model any type of software specification. We have extended the basic functional/logic programming model of TLG [7] to include object-oriented programming features suitable for modern software specification [5]. The syntax of the object-oriented TLG is:

```
class Class_Name.
  Data_Name {, Data_Name}
  :: Data_Type {, Data_Type}.
  Rule_Name : Rule_Body {, Rule_Body}.
end class.
```

where the term that is enclosed in the curly brackets is optional and can be repeated many times, as in Extended Backus-Naur Form (EBNF). The data types of TLG are fairly standard, including both scalar and structured types, as well as types defined by other class definitions. The rules are expressed in NL with the data types used as variables. The conversion from the knowledge base to TLG flows very nicely because the knowledge base is built with the structure taking this translation into consideration. The root of each context tree becomes a class. And then the body of each class is built up with the sentence information in the sub-contexts of the root. The knowledge base of the ATM example would be translated into the following TLG specification.

```
class Bank.
  Accounts_List :: AccountList.
  ID :: Integer.
  PIN :: Integer.
  Balance :: Float.
```

```
  verify ID and PIN giving Balance.
  update Balance with ID.
end class.
class Account.
  ID :: Integer.
  PIN :: Integer.
  Balance :: Float.
end class.
class ATM.
  Balance :: Float.
  Amount :: Float.
  ID :: Integer.
  PIN :: Integer.
  withdraw Amount with ID and PIN giving Balance :
    verify ID and PIN from Bank giving Balance,
    if Amount <= Balance then
      Balance := ( Balance - Amount ),
      update Balance in Bank with ID
    endif.
  deposit Amount with ID and PIN giving Balance :
    verify ID and PIN from Bank giving Balance,
    Balance := ( Balance + Amount ),
    update Balance in Bank with ID.
  check balance with ID and PIN giving Balance :
    verify ID and PIN from Bank giving Balance.
end class.
```

When the system proceeds with the ATM knowledge base, it detects the fact that the data type of the ‘Amount’ hasn’t been specified. So it asks for its manual input from the user. Also observe that the sentence that increases or decreases the balance is mapped into the TLG ‘assign’ statement. NL has a fairly large size of vocabularies whereas TLG uses specific words for the language-defined operations. Therefore there is a many-to-one mapping between a NL expression and a specific TLG operation just like the ‘assign’ operation example. Therefore the systematic structure of the knowledge base and the formal yet flexible syntax of TLG make very smooth translation from the former to the latter.

3.3. Translation from Two Level Grammar to Vienna Development Method

The object-oriented extension of the Vienna Development Method meta-language, VDM++ [10], has been selected as a target specification language for this project because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Although TLG and VDM are both formal specification languages, the translation from TLG into VDM is not simply a direct mapping between them. TLG is a procedural specification as well as a logical specification whereas VDM is only procedural. Therefore TLG supports non-deterministic rule firing through the substitution and resolution mechanisms just like other logical programming languages. There are other differences between

the two languages that provide challenges in the translation such as function overriding, non-determinism, and the like. The TLG specification of the ATM example would be translated into the following VDM code.

```

class Bank

instance variables
  private Accounts_List : seq of Account := []

operations
  public verify : int * int ==> real
  verify (ID, PIN) ==
    (dcl Balance : real := 0;
     return Balance);

  public update : real * int ==> ()
  update (Balance, ID) ==
    return

end Bank

class Account

instance variables
  private ID : int ;
  private PIN : int ;
  private Balance : real

end Account

class ATM

instance variables
  private CBank : Bank := new Bank()

operations
  public withdraw : real * int * int ==> real
  withdraw (Amount, ID, PIN) ==
    (dcl Balance : real;
     Balance := CBank.verify(ID, PIN);
     if Amount <= Balance then
       (Balance := ( Balance - Amount ) ;
        CBank.update(Balance, ID));
     return Balance);

  public deposit : real * int * int ==> real
  deposit (Amount, ID, PIN) ==
    (dcl Balance : real;
     Balance := CBank.verify(ID, PIN);
     Balance := ( Balance + Amount ) ;
     CBank.update(Balance, ID);
     return Balance);

  public checkBalance : int * int ==> real
  checkBalance (ID, PIN) ==
    (dcl Balance : real;
     Balance := CBank.verify(ID, PIN);
     return Balance)

end ATM

```

We have defined translation schemes from various TLG constructions into corresponding VDM++ constructions [6]. Once we have translated the TLG specification into a VDM++ specification we can convert

this into a high level language such as JavaTM or C++, using the code generator that the VDM ToolkitTM provides [20]. Not only is this code quite efficient, but it may be executed, thereby allowing a proxy execution of the requirements. This allows for a rapid prototyping of the original requirements so that these may be refined further in future iterations. Another advantage of this approach is that the VDM Toolkit also provides for a translation into a model in the Unified Modeling Language (UML) [27] using a link with Rational RoseTM [21].

4. Expected Contributions and Research Progress

The main contribution of this research stems from the automated conversion of informal requirements documents in natural language into formal specifications. When modification are made in the system specifications the changes can be made in the requirements document. Therefore by making automated translation of the requirements into a formal specification and then into the implementation code, the changes will be reflected in all the specifications still maintaining integrity and consistency. Also by rapid prototyping the system can be verified and validated at the requirements phase. Partial or entire requirements can be implemented to check if the system behaves according to the stakeholders' needs.

Because the users can express their needs in natural language and the software engineers in a formal specification at the same time, the negotiation and elicitation of requirements become smooth and fast.

Also the software requirements document can be easily reused because the current document is consistent with the system already developed and verified. Therefore the requirements document evolves as the system evolves throughout the software development. Also instead of throwing the requirements document away after completion of the system development, the requirements document is used again later to develop a similar system or revised one.

Currently parsing some real-world requirements documents and determining grammatical information such as part of speeches and part of sentences have been successful. Also the contextual knowledge base has been formalized and implemented. The translation from the knowledge base into TLG, even in the primitive stage for now, has been tested successively with the simple ATM example. The conversion from TLG to VDM is working relatively well after extensive testing.

The proposed system will be evaluated by thorough

testing with some real-world requirements documents. It is planned to collect a corpus of reasonable requirements documents and evaluate how well this system is able to process them, with respect to detection of ambiguity, translation into formal specification, and execution behavior. After thorough testing, we will have a prototype system for executable NL requirements.

5. Conclusion

This research project is developed as an application of formal specification and linguistic techniques to automate the conversion from a requirements document written in NL to a formal specification language. The knowledge base is built up from a NL requirements document in order to capture the contextual information from the document while handling the ambiguity problem and to optimize the process of its translation into a TLG specification. Well structured and formalized data representations especially for the context are used to make smooth translations from NL requirements into the knowledge base and then from the knowledge base into a TLG specification. Due to its NL-like syntax and flexibility without losing its formalism, TLG is chosen as a formal specification to fill the gap between the different level of formalisms of NL and formal specification language. Therefore by using the formalized context in natural language processing and TLG as a bridge between the requirements document and a formal specification language, we can achieve an executable NL specification for a rapid prototyping of requirements, as well as development of a final implementation.

References

- [1] V. S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer-Verlag, 1998.
- [2] J. Allen. *Natural Language Understanding*. Benjamin/Cummings, 2nd edition, 1995.
- [3] D. Bjørner and C. B. Jones. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1985.
- [4] B. W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, 10(1):4–21, January 1984.
- [5] B. R. Bryant. Object-Oriented Natural Language Requirements Specification. *Proc. ACSC 2000, 23rd Australasian Comp. Sci. Conf.*, pages 24–30, 2000.
- [6] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Hawaii Int. Conf. System Sciences (to appear)*, Jan. 2002.
- [7] et al. Bryant, B. R. Two-Level Grammar: Data Flow English for Functional and Logic Programming. *Proc. 1988 ACM Computer Science Conf.*, pages 469–474, 1988.
- [8] J. Burg and R. Riet. A Natural Language and Scenario based Approach to Requirements Engineering. In *Proc. Natuerlichsprachlicher Entwurf von Informationssystemen (NEI'96)*, pages 219 – 233, 1996.
- [9] A. Davis. *Software Requirements Analysis and Specification*. Prentice Hall, 1990.
- [10] E. H. Dürr and J. van Katwijk. VDM++ - A Formal Specification Language for Object-Oriented Designs. *Proc. TOOLS USA '92, 1992 Technology of Object-Oriented Languages and Systems USA Conf.*, pages 63–278, 1992.
- [11] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting Requirement Formalization by Means of Natural Language Translation. *Formal Methods in System Design*, 3:243–263, 1994.
- [12] D. Fensel, R. Groenboom, and G. R. Renardel. Modal Change Logic (MCL): Specifying the Reasoning of Knowledge-based Systems. *Data and Knowledge Engineering*, 26:243–269, 1998.
- [13] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). *Proc. CLAW 96, 1st Int. Workshop Controlled Language Applications*, 1996.
- [14] G. Gazdar, E. Klein, G. K. Pullum, and I. Sag. *Generalized Phrase Structure Grammar*. Brasil Blackwell, 1985.
- [15] V. Gervasi and B. Nuseibeh. Lightweight Validation of Natural Language Requirements: A Case Study. In *Proc. 4th IEEE Int. Conf. on Requirements Engineering*, 2000.
- [16] M. R. Girardi. *Classification and Retrieval of Software through their Description in Natural Language*. PhD thesis, Computer Science Department University of Geneva, Switzerland, 1996.
- [17] L. Goldin and D. Berry. AbstFinder, A Prototype Natural Language Text Abstraction Finder

- for Use in Requirements Elicitation. *Automated Software Engineering*, 4:375–412, 1997.
- [18] W. Grady. Moby Part-of-Speech II (data file), 1994.
- [19] H. M. Harmain and R. Gaizauskas. CM-Builder: An Automated NL-based CASE Tool. *In Proc. ASE 2002, 15th IEEE Int. Conf. Automated Software Engineering*, pages 45–54, 2000.
- [20] IFAD. VDMTools - Java/C++ Code Generator. Technical report, IFAD, 2000.
- [21] IFAD. VDMTools - The Rose-VDM++ Link. Technical report, IFAD, 2000.
- [22] J. McCarthy. Notes On Formalizing Context. Technical report, Computer Science Department, Stanford University, Stanford, CA, 1993.
- [23] F. Meziane and S. Vadera. From English to Formal Specifications. *Computer Journal*, 37(9):753–63, 1994.
- [24] G. Miller. Wordnet: An On-line Lexical Database. *International Journal of Lexicography*, 4(3), 1990.
- [25] M. Osborne and C. K. MacNish. Processing Natural Language Software Requirement Specifications. *Proc. ICRE '96, 2nd Int. Conf. on Requirements Engineering*, pages 229–236, 1995.
- [26] S. Pulman, H. Alshawi, D. Carter, R. Crouch, M. Rayner, and A. Smith. CLARE: A Combined Language and Reasoning Engine. Technical report, SRI International, 1993.
- [27] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 2000.
- [28] R. Nelken and N. Francez. Automatic Translation of Natural-Language System Specifications into Temporal Logic. *In In Proc. CAV '96, 8th Int. Conf. Computer Aided Verification*, volume 1102, pages 360–371. Springer Verlag, 1996.
- [29] C. Rolland and C. Proix. A Natural Language Approach For Requirements Engineering. *Proc. CAiSE'92, 4th Int. Conf. Advanced Information Systems Engineering*, pages 257–277, 1992.
- [30] K. Ryan. The Role of Natural Language in Requirements Engineering. *Proceedings of the IEEE Int. Symp. on Requirements Engineering*, pages 80–82, 1993.
- [31] M. Sintzoff. Existence of van Wijngaarden's Syntax for Every Recursively Enumerable Set. *Ann. Soc. Sci. Bruxelles 2*, pages 115–118, 1967.
- [32] A. van Wijngaarden. *Orthogonal Design and Description of a Formal Language*. Mathematisch Centrum, Amsterdam, 1965.
- [33] W. M. Wilson. Writing Effective Natural Language Requirements Specifications. Technical report, Naval Research Laboratory, 1999.
- [34] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated Quality Analysis Of Natural Language Requirement Specifications. Technical report, Naval Research Laboratory, 1996.