

Prototyping of Requirements Documents Written in Natural Language *

Beum-Seuk Lee Barrett R. Bryant

Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170 U. S. A.
{leebs, bryant}@cis.uab.edu

Abstract

In software engineering a system requirements document written in a natural language (NL) needs to be translated into one of the formal specification languages for system prototyping. When the system is large and complex this translation, if manually done, is error prone, if not impossible. There is the major bottleneck to overcome for making this translation automated : the ambiguity in NL and different formalism between requirement documents and the formal specification. We use Contextual Natural Language Processing to overcome the ambiguity, and Two-Level Grammar (TLG) to construct a bridge between different formalism. The result is a formal representation of the informal requirements in NL for prototyping and even for implementation.

1 Introduction

In software development still the natural language (NL) has remained as the practical choice for the domain experts to specify the system even with recent active development of many formal specification languages. This is due to the fact that formal specification languages are hard to master and inappropriate as a communication medium. However the syntax and semantics of NL, even with its flexibility and representation power, is not formal enough to be used directly for verification, prototyping, or implementation of the system. Therefore the requirements document in NL is translated, usually manually, into a formal specification. However, 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. In addition, the requirements document may be reused for another similar system (which is a very strong trend in recent software development) but the manual translation itself is usually hardly reusable. The major bottleneck of this conversion is from the inborn characteristic of ambiguity of NL and the different level of the formalism between the two domains of NL and the formal specification.

To handle this ambiguity problem, some have argued that the requirements document has to be written in a particular way to reduce ambiguity in the document [17]. Others have proposed controlled natural languages (e.g., Attempto Controlled English (ACE) [7]) which limit the syntax and semantics of NL to avoid the ambiguity problem. Another approach to NL requirements analysis is to search each line of the requirements document for specific words and phrases for the purpose of quality analysis [18]. A similar project [8] focuses mainly on the automatic indexing and reuse of the software components in the requirement documents. However there has been no attempt to automate the conversion from requirements documentation into a formal specification language for implementation.

In our research project, Contextual Natural Language Processing (CNLP) [13] is used to handle the ambiguity problem in NL and Two Level Grammar (TLG) [16] 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 VDM++ - an object-oriented extension of the Vienna Development Method [1]). First the requirements document and its domain knowledge are converted into Extensible Markup Language (XML) [2] format. Then a knowledge base is built from the XML requirements document [12] and domain knowledge using CNLP by parsing the documentation and storing the syntax, se-

* This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350 and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

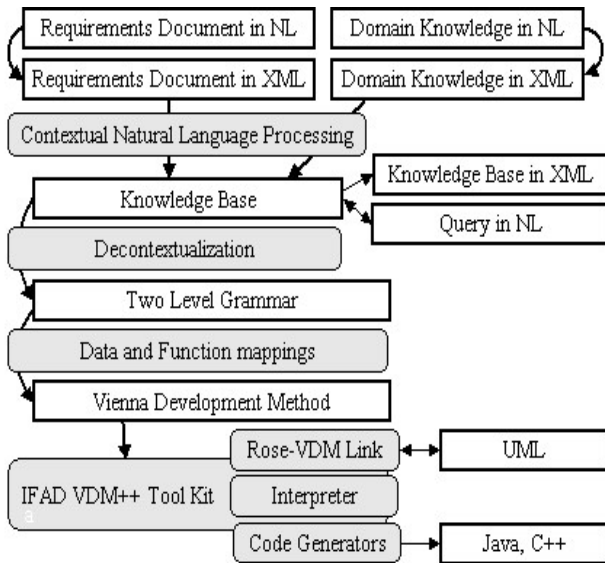


Figure 1: System Structure.

mantics, and pragmatics information. In this phase, the ambiguity is detected and resolved, if possible. Once the knowledge base is constructed, its content can be displayed in XML or queried in NL. Next 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. Once VDM++ representation of the specification is acquired we can do prototyping of the specification using the VDM++ interpreter. Also we can convert this into a high level language such as JavaTM or C++ or into a model in the Unified Modeling Language (UML) [14] using the VDM++ Toolkit [11]. The entire system structure is shown in Figure 1.

Our system also allows the user to configure and to fine-tune the automation. This is necessary because our system carries out a supervised automation iteratively reflecting the user’s intention and guidance for the automation. In the sections which follow, we will illustrate our approach and describe the various system components using the simple Automatic Teller Machine (ATM) example below.

```

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.

```

2 Construction of Knowledge Base from Requirements

The knowledge base is built from a requirements document and a domain knowledge document using syntactic, semantic and contextual (discourse) information. A requirements document usually contains specific information about how the system should work whereas a domain knowledge document describes the relationship between components and other constraints which are usually presumed in requirements documents. For example, the requirements says “the user inputs the 4 digit PIN number by pressing the buttons.” And the fact that the set of the buttons is a component of the ATM machine is implicitly assumed and therefore not explicitly mentioned in the requirements documents. So this kind of information is needed to be specified as the domain specific knowledge. The units of measurements, who passes what to whom, which synonyms of a word are used, what each acronym stands for, etc., are some of the examples of the domain specific knowledge that can supplement to the requirements documents.

First these documents are converted into Extensible Markup Language (XML) [12] by dividing the document into sections, subsections, paragraphs, and finally sentences. The requirements documents can vary in their formats and styles depending on the organization or even on each individual who writes it. Therefore by converting them into XML the documents can have one standard format. Our system supports a tool for the user to rearrange the structure of the document by allowing the user to drag, drop, edit, and search the sentences, sections, and chapters anywhere in the document with ease.

Once the document is formatted in XML, each sentence is read by the system and each sentence is parsed. This linguistic procedure is necessary not only to build the knowledge base with contextual information but also to be used later to identify the classes and their operations at the time of the conversion of the knowledge base into TLG. Usually the subjects are good candidates for the classes and the verbs usually indicate what kind of operations the classes carry out given the parameters (the objects of the sentence).

At the syntactical level, the part of speech (e.g. noun, verb, adjective) and the part of sentence (e.g. subject, object, complement) of each word are determined. In this phase the corpora of statistically ordered parts of

speech (frequently used ones being listed first) of about 85000 words from [9] are used to resolve the syntactic ambiguity. In other words, when there is more than one valid parsing tree for a sentence, this corpora is used to break the tie.

Elliptical compound phrases, comparative phrases, compound nouns, and relative phrases are handled as well. This is to ensure that the input documents in NL is as less restricted as possible. Also the system allows the user to guide the parsing instead of being forced to rephrase the sentence. For example if the system mistakes the part of speech of the word ‘keeps’ as a noun in the sentence “Bank keeps list of accounts” the user can make the system remember that the part of speech of ‘keeps’ is verb in this specific context instead of rephrasing the sentence into “Bank maintains list of accounts” for instance.

Using the semantic information and the syntactical information gathered from the previous phase, anaphoric references (pronouns) are identified. A pronoun can represent a word, a sentence, or even another context. This is done according to the recency constraints (the recent word has a higher priority than less recent ones) and the discourse focus (the co-referred one has a higher priority than ones that aren’t) [3] [10].

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 from what is originally meant. This process is sensitive to how the sentences are divided and arranged in the original document and heavily depends on the words (e. g. “the **following** procedure” and “in the **above** condition”) which are used to indicate the discourse relationships between sentences.

A contextual knowledge base is formalized as a tree-like data structure not only to store each sentence in its right 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 according to the discourse level information (sections, subsections, and paragraphs) in XML and semantics information in sentences. For instance, in the ATM example the phrase “in the following sequence” indicates that the following sentences are likely to stay in the current context. The contextual structure of the knowledge base is shown in Figure 2.

The black ovals indicate the contexts that hold the data type information whereas the gray ovals indicate the contexts that contain the functional information.

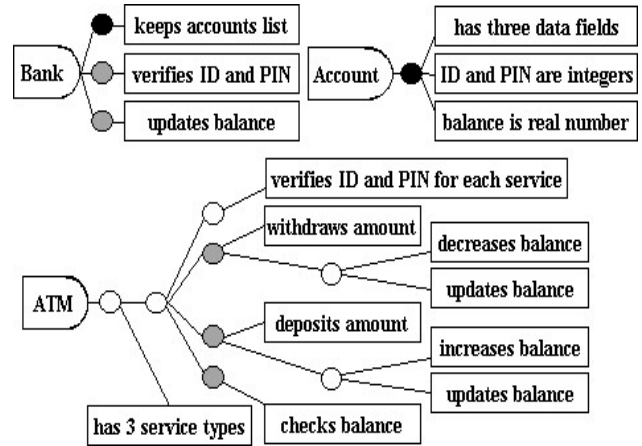


Figure 2: Knowledge base for ATM.

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 definition later. If it were located in any other context, the sentence couldn’t operate as originally intended. The system also allows the user to reconstruct the resulting knowledge base as well. When a sentence is misplaced by the automation because the contextual information for the sentence is too weak or too implicit.

Once the knowledge base is built into database tables from the document, its content can be displayed in XML. This enables the user to see the structured context information of the system. Because both the context in the knowledge base and XML data have the tree-like structure, the knowledge base in XML visualizes the same structure of the knowledge base shown in Figure 2.

Also the content of the knowledge base can be queried by the user in natural language. The following dialogue between the system and a user shows some example queries about ATM.

```
User : What does the bank keep?
System : Bank keeps list of accounts.
User : How does the ATM deposit the amount?
System : ATM deposits amount with ID and PIN
giving balance in following order,
ATM increases balance by Amount,
then ATM updates balance in bank with ID
```

Because the requirements are stored in a structural format according to the context, the relevant other information is also retrieved as shown in the answer for the second query.

In summary, the syntactic, semantic, and contextual information is used to build up the contextual knowledge base from the requirements documentation.

3 Conversion from Knowledge Base to TLG

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 [15] and so TLG may be used to model any type of software specification. The basic functional/logic programming model of TLG is extended to include object-oriented programming features suitable for modern software specification [4].

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 [Class_Name].
```

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.
```

Observe that the sentence that increases or decreases the balance is mapped into the TLG assign statement. NL has a fairly large size vocabulary 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. As seen in the above TLG specification of ATM, TLG has flexibility with its NL-like syntax as well as formality with its strong typing and formal semantics.

4 Translation from TLG to VDM++

The object-oriented extension of the Vienna Development Method meta-language, VDM++ [6], has been selected as a target specification language which is translated from TLG, 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. We have defined

translation schemes from various TLG constructions to corresponding VDM++ constructions. This translation is carried out in three levels : class level, data type declaration level, and finally operational level. For more details on this translation we refer the readers to [5]. The VDM++ specification of the ATM example is shown below.

```

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

```

Once we have translated the TLG specification into a VDM++ specification we can convert this into a high

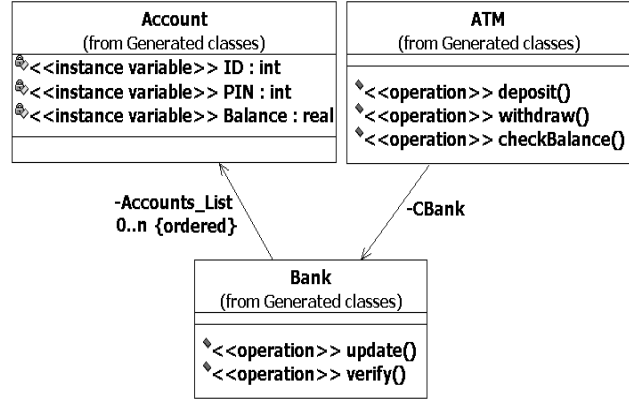


Figure 3: UML for ATM.

level language such as JavaTM or C++, using the code generator that the VDM++ ToolkitTM provides. 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. Namely the inconsistencies, contradictions, and ambiguities hidden in the informal description can be discovered in the formal representation using the VDM++ Toolkit. Another advantage of this approach is that the VDM++ Toolkit also provides for a translation into a model in the Unified Modeling Language (UML) using a link with Rational RoseTM (Figure 3).

5 Summary and 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 and a domain knowledge 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.

The system can currently handle the presented example completely, as described. We are performing additional evaluations of the system for other requirements documents, including some requirements documents de-

scribing actual U. S. Army systems. It is expected that the technology we are developing will be applicable to these requirements documents as well. If successful, this will provide a very useful tool to assist software engineers in moving from the requirements document to the formal specification. Our future work is to continue developing the system to improve system usability and robustness with respect to its coverage of requirements documents. When finalized, it is expected that by using the formalized context in CNLP 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.

Acknowledgements : The authors would like to thank IFAD for providing an academic license to the IFAD VDM++ Toolbox in order to conduct this research.

References

- [1] D. Bjørner and C. B. Jones. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1978.
- [2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C (<http://www.w3c.org/xml>), 2000.
- [3] S. Brennan, L. Friedman, and C. Pollard. A Centering Approach to Pronouns. *Proc. 25th ACL Annual Meeting*, pages 155–162, 1987.
- [4] B. R. Bryant. Object-Oriented Natural Language Requirements Specification. *Proc. ACSC 2000, 23rd Australasian Comp. Sci. Conf.*, pages 24–30, 2000.
- [5] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Hawaii Int. Conf. System Sciences*, Jan. 2002.
- [6] 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.
- [7] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). *Proc. CLAW 96, 1st Int. Workshop Controlled Language Applications*, 1996.
- [8] M. R. Girardi. *Classification and Retrieval of Software through their Description in Natural Language*. PhD thesis, Computer Science Department University of Geneva, Switzerland, 1996.
- [9] W. Grady. Moby Part-of-Speech II (data file), 1994.
- [10] B. J. Grosz, A. K. Joshi, and S. Weinstein. Providing a Unified Account of Definite Noun Phrases in Discourse. *Proc. 21st ACL Annual Meeting*, 155–162:44–50, 1983.
- [11] IFAD. The VDM++ Toolbox User Manual. Technical report, IFAD (<http://www.ifad.dk>), 2000.
- [12] B.-S. Lee and B. R. Bryant. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. *Proc. ACM 2002 Symposium on Applied Computing*, 2002.
- [13] J. McCarthy. Notes On Formalizing Context. Technical report, Computer Science Department. Stanford University. Stanford, CA, 1993.
- [14] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 2000.
- [15] M. Sintzoff. Existence of van Wijngaarden’s Syntax for Every Recursively Enumerable Set. *Ann. Soc. Sci. Bruxelles 2*, pages 115–118, 1967.
- [16] A. van Wijngaarden. *Orthogonal Design and Description of a Formal Language*. Mathematisch Centrum, Amsterdam, 1965.
- [17] W. M. Wilson. Writing Effective Natural Language Requirements Specifications. Technical report, Naval Research Laboratory, 1999.
- [18] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated Quality Analysis Of Natural Language Requirement Specifications. Technical report, Naval Research Laboratory, 1996.