

Automatic Transformation of Natural Language Requirements into Formal Specifications *

Beum-Seuk Lee

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

Abstract

In software 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 used Two-Level Grammar (TLG) to construct a bridge between a NL requirements specification and a formal specification, and the contextual NL processing to overcome the challenges of the ambiguity in NL.

Research Area: natural language requirements analysis, formal specification languages, execution of requirements

1. Problem Statement

The main goal of the research is to realize an automated conversion of a requirements document written in natural language (NL) into a formal specification language. Even though there are many formal specification languages, it isn't easy for a domain expert to learn and use them in practice. So usually the software engineer who has mastered these languages takes the requirements documentation written in NL by the

domain experts and then manually translates the requirements into the formal specification to build a system based on the requirements. When the system to be built is large and complex this manual translation is very error prone and even impractical in the worst case. There are some reasons for these errors such as miscommunication between the domain experts and the engineer and different formalism levels of NL and the specification language. Although there may be some ways to overcome these issues, the ambiguity in the requirements documentation can be the most important issue, yet challenging to resolve. This is one of the reasons that there are very few attempts to convert requirements documents into formal specifications automatically. First the ambiguity has to be detected and then, if possible, resolved to realize this automated conversion.

2. Prior Research

One research project with similar goals as ours is the use of "controlled natural language" for requirements specification [5]. Controlled NL limits the syntax and semantics of the language that a user can use. Even though this controlled language is easier to learn than other formal languages, still the user has to learn and remember the restrictions on the syntax. Moreover instead of converting this NL subset into another formal specification language, the target language in this project is a logic programming language like Prolog which is not suitable for a general formal specification or implementation.

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 [11]. Alternatively, Wilson [10] argues that re-

*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.

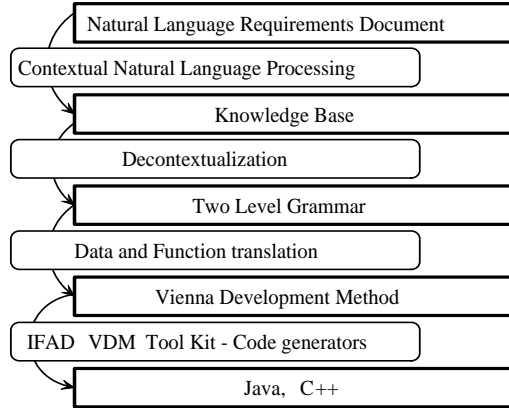


Figure 1: System Structure.

quirements documents should be written in a particular way. Neither of these approaches have been extended into a phase where the requirements are converted to any formal specification or executable code.

3. Proposed Research

Our hypothesis is that Contextual Natural Language Processing (CNLP) [8] and Two Level Grammar (TLG) [9] can overcome the challenge of an automated conversion from a requirements document written in NL into a formal specification to promote a detection method of ambiguity or errors in the requirements documentation, rapid prototyping of the requirements, as well as system implementation.

In CNLP a structured and formalized context is developed to resolve the ambiguity problem in the translation from the requirements documentation. TLG is a formal specification that can construct a bridge between an NL requirements document and a formal specification language (in our case the target language is the Vienna Development Method - VDM [1]) with the system structure shown in Figure 1.

This research has three phases. The first phase is the automated translation from a requirements documentation written in NL into a knowledge base. The second phase is to convert this knowledge base into TLG. And in the last phase the translation from TLG into VDM is made.

The unstructured document with requirements is converted into a structured knowledge base by decomposing and abstracting the information. Given a requirements document, first each sentence is parsed into a list of words (decomposition). Syntactic and semantic information is retrieved from these parsed sentences to build up the knowledge base (abstraction) [6]. The

knowledge base has to be constructed in a way to store the syntactic and semantic information from the requirements documentation while capturing the corresponding structure of TLG to which this knowledge base is mapped into later. Also there should be no data redundancy and the representation should facilitate manipulating the data [4]. Here the data to store are concepts (vocabularies), sentences, and contexts. A conceptual data structure is used to store the concepts whereas a contextual data structure to store the sentences and the contexts. We will show how the knowledge base of the example Automatic Teller Machine (ATM) requirements specification below is constructed using these data structures.

```

The bank keeps the list of accounts.
Each account has three integer data fields;
  ID, PIN, and balance.
The ATM machine has 3 service types;
  withdraw, deposit, and balance check.
For each service first it verifies ID and PIN
  from the bank.
  
```

```

Withdraw service withdraws an amount from
the account of ID with PIN in the bank in
the following sequence:
  First it gets the balance of the account
  of ID from the bank,
  if the amount is less than or equal to
  the balance then
    it decreases the balance by Amount,
    updates the balance of the account of ID
    in the bank,
    and then outputs the new balance.
  
```

```

Deposit service deposits an amount to the
account of ID with PIN in the
bank in the following sequence:
  First it gets the balance of the account
  of ID from the bank,
  it increases the balance by Amount,
  updates the balance of the account of ID
  in the bank,
  and then outputs the new balance.
  
```

```

Balance check service check the balance of
the account of ID with PIN in Bank
in the following order:
  It gets the balance of the account of ID
  from the bank,
  and then outputs the balance.
  
```

A conceptual data structure contains vocabularies, the syntactic information, the conceptual relationships with other words, and the definition of each word. The topological relational structure enables grouping the conceptually related words together and helps in determining the properties of a newly added word that belongs to a certain concept group. Part of the conceptual data structure of the ATM is shown in Figure 2.

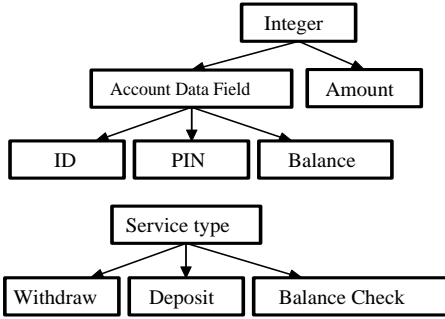


Figure 2: ATM conceptual representation.

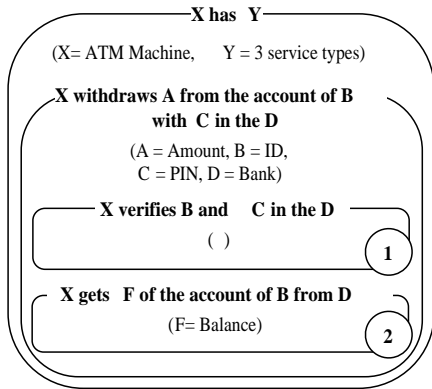


Figure 3: ATM contextual representation.

Notice that the structure captures not only the type of each data but also the inheritance hierarchy and aggregation. The type of Account Data Field is Integer, as are ID, PIN, and Balance, and ID, PIN, and Balance are grouped together as components of an account type. Also three service types of the ATM are grouped together to be used as the functions for the class ATM.

A context is a sentence with some variables or a set with sentences as its elements. This representation is also hierarchical in the sense that the meaning of the sentences in a set can be refined by referring to the definitions of its supersets (outer contexts). The contextual data structure stores the sentences and the relationship between them after taking the semantics and discourse level information of the document into account. Part of the contextual data structure of the ATM example is shown in Figure 3.

This contextual representation shows the functions that belong to the class ATM. The actual body of each function is expressed as subcontexts in the diagram.

Two-Level Grammar (TLG) may be used to achieve translation from an informal NL specification into a formal specification [2]. Even though TLG has NL-like syntax its notation is formal enough to allow for-

mal specifications to be constructed. TLG captures the abstraction of the requirements and also preserves 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. We have extended the basic functional/logic programming model of TLG to include object-oriented programming features suitable for modern software specification. 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 curly brackets denote repetition. Class inheritance and polymorphism are also provided for in a natural way.

For example we can construct the following quick sort class using the TLG notation.

```

class Quick_Sort.
  Pivot :: Integer.
  List_of_Integers,
  Integers_Less_Than_Pivot,
  Integers_Greater_Than_Pivot,
  Sorted_Integers_Less_Than_Pivot,
  Sorted_Integers_Greater_Than_Pivot :: Integer*.

  quick sort Empty_List into Empty_List.

  quick sort Pivot List_of_Integers
  into Sorted_Integers_Less_Than_Pivot Pivot
  Sorted_Integers_Greater_Than_Pivot :
  split List_of_Integers with Pivot
  into lists Integers_Less_Than_Pivot and
  Integers_Greater_Than_Pivot,
  quick sort Integers_Less_Than_Pivot
  into Sorted_Integers_Less_Than_Pivot,
  quick sort Integers_Greater_Than_Pivot
  into Sorted_Integers_Greater_Than_Pivot.

  split Empty_List with Pivot
  into lists Empty_List and Empty_List.

  split Integer List_of_Integers with Pivot
  into lists Integer Integers_Less_Than_Pivot
  and Integers_Greater_Than_Pivot :
  where Integer <= Pivot,
  split List_of_Integers with Pivot
  into lists Integers_Less_Than_Pivot and
  Integers_Greater_Than_Pivot.

  split Integer List_of_Integers with Pivot
  into lists Integers_Less_Than_Pivot and
  Integer Integers_Greater_Than_Pivot :
  where Integer > Pivot,
  split List_of_Integers with Pivot
  into lists Integers_Less_Than_Pivot and
  Integers_Greater_Than_Pivot.
end class Quick_Sort.
  
```

We can see the natural language-like notation as well as the object-oriented characteristic of TLG from the above example.

Translation from the knowledge base into a TLG specification is required to minimize any kind of context dependency from the knowledge base. Since NL itself is very object-oriented the translation from the knowledge base into TLG can be accomplished smoothly. First identifying each class and then collecting its data members and functions into the class from the knowledge base is the major task of the decontextualization. The conceptual data structure is used to identify the classes and the data of the classes whereas the sentential or contextual data structure can be used to find the functions of the classes in the document. The knowledge base of the ATM example would be translated into the following TLG specification.

```
class Bank.
Account::{ID Integer PIN Integer Balance Integer}*.
end class Bank.

class ATM.
ID, PIN, Balance :: Integer.
Amount :: Integer.

withdraw Amount from account of ID with PIN in Bank
giving Balance :
verify ID and PIN from Bank,
get Balance of the account of ID from Bank,
if Amount <= to Balance then
Balance := Balance - Amount endstmt
update Balance of the account of ID in Bank
endif.

deposit Amount to account of ID with PIN in Bank
giving Balance :
verify ID and PIN from Bank,
get Balance of the account of ID from Bank,
Balance := Balance + Amount,
update Balance of the account of ID in Bank.

check balance of account of ID with PIN in Bank
giving Balance :
verify ID and PIN from Bank,
get Balance of the account of ID from Bank.

end class ATM.
```

Because of the NL-like syntax of TLG the translation from the knowledge base is very achievable.

The object-oriented extension of the Vienna Development Method metalanguage, VDM++ [3], 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
types
DATATYPE :: ID : int PIN : int Balance : int;
end Bank

class ATM_Machine
operations
withdraw_account : int*int*int*Bank ==> int
withdraw_account (Amount, ID, PIN, Bank) ==
(dcl Balance : int;
verify(ID, PIN, Bank);
get_account(Balance, ID, Bank);
if Amount <= Balance then
(Balance := ( Balance - Amount ) ;
update_account(Balance, ID, Bank) );
return Balance);

deposit_account : int*int*int*Bank ==> int
deposit_account (Amount, ID, PIN, Bank) ==
(dcl Balance : int;
verify(ID, PIN, Bank);
get_account(Balance, ID, Bank);
Balance := ( Balance + Amount ) ;
update_account(Balance, ID, Bank);
return Balance);

check_balance : int*int*Bank ==> int
check_balance (ID, PIN, Bank) ==
(verify(ID, PIN, Bank);
get_account(Balance, ID, Bank);
return Balance)
end ATM_Machine
```

We have defined translation schemes from various TLG constructions into corresponding VDM++ constructions. 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 [7] provides, to execute the requirements. 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.

4. Current Progress and Expected Contributions

Due to its NL-like syntax and flexibility without losing its formalism, TLG is chosen to construct a bridge between informal NL and a formal specification language. The knowledge base is built up from a NL re-

quirements document in order to capture the contextual information from the documentation 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. Therefore using the formalized context in NL processing and TLG as a bridge between the requirements document and a formal specification language, we can achieve an executable natural language specification for a rapid prototyping of requirements, as well as development of a final implementation.

Currently parsing some real-world requirements documents and determining grammatical information such as part of speeches and part of sentences have been successful. From this a knowledge base is built, yet without much contextual information at present. Extending the capabilities of this front end is still a significant task. The translation from TLG to VDM is working relatively well, so once the NL processing part has been completed, we will have a prototype system for executing NL requirements.

References

- [1] D. Bjørner and C. B. Jones. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1985.
- [2] B. R. Bryant. Object-Oriented Natural Language Requirements Specification. *Proc. ACSC 2000, 23rd Australasian Comp. Sci. Conf.*, pages 24–30, 2000.
- [3] 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.
- [4] 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.
- [5] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). *CLAW 96, First International Workshop on Controlled Language Applications, University of Leuven, Belgium*, 1996.
- [6] M. R. Girardi. *Classification and Retrieval of Software through their Description in Natural Language*. PhD thesis, Computer Science Department University of Geneva, Switzerland, 1996.
- [7] IFAD. *VDMTools - Java/C++ code generator*. IFAD, 2000.
- [8] J. McCarthy. Notes On Formalizing Context. Technical report, Computer Science Department. Stanford University. Stanford, CA, 1993.
- [9] A. van Wijngaarden. *Orthogonal Design and Description of a Formal Language*. Mathematisch Centrum, Amsterdam, 1965.
- [10] W. M. Wilson. Writing Effective Natural Language Requirements Specifications. Technical report, Naval Research Laboratory, 1999.
- [11] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated Quality Analysis Of Natural Language Requirement Specifications. Technical report, Naval Research Laboratory, 1996.