

Automation of Software System Development Using Natural Language Processing and Two-Level Grammar

Beum-Seuk Lee and Barrett R. Bryant

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

Abstract. In software engineering, even with recent active research on formal methods and automated tools, users' involvement is inevitable and crucial throughout the software development lifecycle. Automation of these manual tasks would assist the developers throughout the development. Our project goal is to help the engineers to resolve ambiguity in natural language (NL) using Natural Language Processing and to overcome different levels of abstraction between requirements documents and formal specifications using Two-Level Grammar (TLG). The result is a system that assists developers to build a formal representation from the informal requirements for rapid prototyping and even implementation.

Keywords: Natural Language Processing, Formal Specification, Automated Software Engineering, Two-Level Grammar (TLG)

1 Problem Statement

Even the rigorous development of formal specifications and automated tool kits in recent years hasn't eliminated the practical importance of requirements documents written in natural language and the necessity of users' involvement throughout the software development life cycle.

Even though natural language is inherently object-oriented and descriptive with strong representation power, its syntax and semantics are not formal enough to be used directly as a programming language. Therefore the requirements documentation written in NL has to be reinterpreted into a formal specification language by software engineers. Pohl rightly stated regarding this process that improving an opaque system comprehension into a complete system specification and transforming informal knowledge into formal representations are the major tasks in the requirements engineering [1]. When the system is very complicated, which is mostly the case when one chooses to use formal specification, this conversion, if manually done, is both non-trivial and error-prone, if not implausible.

Many similar tasks of manual involvement occur and are repeated to translate the requirements documents into a formal specification or into final executable code regardless the type of the system under development. Some examples of these tasks are domain-specific knowledge collection, correct interpretation of requirements, specification update, and maintenance of consistency, to name a few.

It is well known that as much as 60 percent of the errors that appear during a system's life cycle have their origin in the requirements phase [2]. It is also well known that the closer 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 [3]. Therefore ensuring the correctness of the requirements as well as their interpretation and translation cannot be overemphasized.

The challenge of formalizing a natural language requirements document, which takes up major portion of human involvement in the system development, results 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 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 [4]. Others have proposed controlled natural languages (e.g., Attempto Controlled English (ACE) [5]) 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 [6]. A similar project [7] 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 prototyping as well as implementation.

In our research, Natural Language Processing (NLP) [8] is used to handle the ambiguity problem in NL and Two Level Grammar (TLG) [9] 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 [10]) and to reduce and reuse the developers involvement.

2 Introduction

To achieve the conversion from requirements documents to a formal specification several levels of conversions are required. First the original requirements written in natural language is to be refined as a preprocessing of the actual conversion. This refinement task involves checking spellings, grammatical errors, consistent use of vocabularies, organizing the sentences into the appropriate sections, etc.

Next the refined requirements document is expressed in XML format. By using XML to specify the requirements, XML attributes (meta-data) can be added to the requirements to interpret the role of each group of the sentences during the conversion. The information of the domain-specific knowledge is specified in XML. The domain-specific knowledge describes the relationship between components and other constraints that are presumed in requirements documents or too implicit to be extracted directly from the original documents.

Then a knowledge base is built from the requirements document in XML using NLP to parse the documentation and to store the syntax, semantics, and pragmatics information. In this phase, the ambiguity is detected and resolved, if possible. Once the knowledge base is constructed, its content can be queried in NL. Next the knowledge base is converted, with the information of the domain specific knowledge, into Two Level Grammar (TLG) by removing the contextual dependency in the knowledge base. TLG, the most NL-like specification language which is a unification of functional, logic, and object-oriented programming styles, is used as an intermediate representation to build a bridge between the informal knowledge base and the formal VDM++ representation.

Finally the TLG code is translated into VDM++ by data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the 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) [11] using the VDM++ Toolkit [12]. The entire system structure is shown in Figure 1.

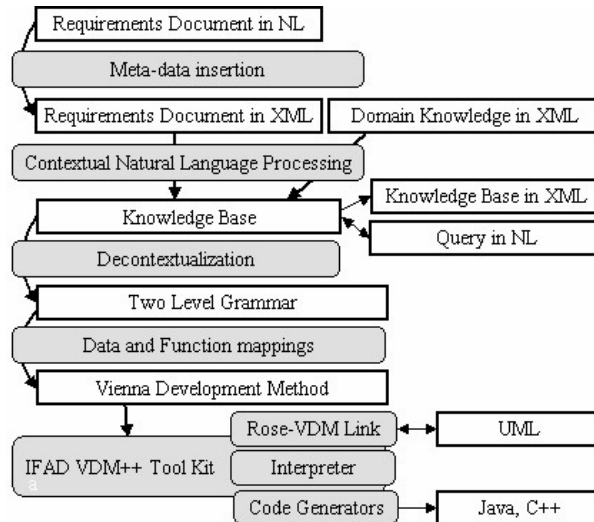


Fig. 1. System Structure.

The translation of our system is incremental and iterative reflecting the changes made throughout the system development. The user interaction is likely to happen any stage of the translation to supervise and assist the automation. By keeping track of user's preferences and configurations for each iteration and automating the translations accordingly, the user's involvement can be reasonably reduced.

In the sections which follow, we will present the following simplified (thus incomplete) Computer Assisted Resuscitation Algorithm (CARA) [13] Infusion Pump Control System to illustrate our approach and describe the various system components.

HOST is powered up and all software subsystems are available.
The pump software system is now in the wait operating state. Patient with IV/pump running is placed onto the HOST. Pump cable is connected to the HOST. HOST now provides power for pump. Pump software system detects pump connection and monitors occlusion and airlock logic levels. Pump subsystem display is automatically brought forward to the secondary display. Pump software subsystem detects back EMF and fluid impedance and begins to log infusion rate. Pump continues to operate on it's hardware setting. Pump software system is now in manual operating state. One of the blood pressure sensors is connected to the patient. Pump software system detects clean blood pressure signal and activates automatic servo-control start button. When the start button is pressed the MAC controls the pump and begins resuscitation to the prescribed blood pressure setpoint. The system is now in the automatic servo-control on operating state when the pump is infusing fluid into a patient using the hardware(HW) flow setting on the pump. If for any reason (change IV bags, change or fix blood pressure sensor, etc.) it becomes necessary to pause the MAC, the pause button on the display may be pressed. This causes the infusion pumping to cease. The system is now in the automatic servo-control paused operating state. The system maybe restarted at any time. When the patient is to be removed from the HOST, the pump software system should be returned to the manual operating state. The blood pressure sensor should be removed from the patient and then the pump cable can be removed from the HOST. This allows the pump to continue operating in standalone mode or the IV infusion to be discontinued.

3 Requirements in XML

Rearranging related information together in the requirements will ease the conversion. Specially because we are assuming that the requirements can contain different aspects of information (functional, non-functional or even mixture of both) about the system. Even requirements that are functionality-oriented can have different types of functionality. For example, they can be object-oriented, procedural, real time-based, event-based, etc. Rearranging related information together will ease the conversion. This can be achieved by specifying the role

of each paragraph using XML data structure and notations. This will help the knowledge base to maintain the correct structure.

The CARA specifications in XML is shown as follows.

```
<document>
<c title = "Mode" meta = "mode">
<c title = "wait state" meta = "submode">
  <p meta = "pre_cond">
    <s>HOST is powered up and all software subsystems are available</s>
  </p>
  <p meta = "pre_exec">
    <s>Patient with IV/pump running is placed onto the HOST</s>
    <s>Pump cable is connected to the HOST</s>
  </p>
  <p meta = "exec">
    <s>HOST now provides power for pump</s>
  </p>
  <p meta = "break_cond">
    <s>When the pump is infusing fluid into a patient using
the hardware (HW) flow setting on the pump the system is no longer in
the wait state</s>
  </p>
</c>
<c title = "manual state" meta = "submode">
  <p meta = "pre_exec">
    <s>Pump software system detects pump connection and monitors occlusion
and airlock logic levels </s>
    <s>Pump subsystem display is automatically brought forward to the
secondary display</s>
    <s>Pump software subsystem detects back EMF and fluid impedance and
begins to log infusion rate</s>
    <s>Pump continues to operate on it's hardware setting</s>
    <s>One of the blood pressure sensors is connected to the patient</s>
    <s>Pump software system detects clean blood pressure signal and
activates automatic servo-control start button</s>
  </p>
</c>
<c title = "autocontrol on state" meta = "submode">
  <p meta = "pre_exec">
    <s>When the start button is pressed the MAC controls the pump and
begins resuscitation to the prescribed blood pressure setpoint</s>
  </p>
</c>
<c title = "autocontrol paused state" meta = "submode">
  <p meta = "pre_exec">
    <s>If for any reason (change IV bags, change or fix blood pressure
sensor, etc.) it becomes necessary to pause the MAC, the pause
button on the display may be pressed</s>
    <s>This causes the infusion pumping to cease</s>
  </p>
</c>
</document>
```

```

<p meta = "break_cond">
  <s>The system maybe restarted at any time</s>
</p>
<p meta = "break_exec">
  <s>When the patient is to be removed from the HOST, the pump software
system should be returned to the manual operating state</s>
  <s>The blood pressure sensor should be removed from the patient and
then the pump cable can be removed from the HOST</s>
  <s>This allows the pump to continue operating in standalone mode or
the IV infusion to be discontinued</s>
</p>
</c>
</c>
</document>

```

The meta attribute in XML indicates the role of each paragraph. Namely it shows if the group of the sentences describes state types (mode), execution types (`_exec`), various conditions (`_cond`), etc. `submode` indicates the state. In CARA example, there are four distinctive states; wait state, manual state, autocontrol on state, and autocontrol paused state. In a state, preconditions (`pre_cond`) have to be satisfied to enter the state. Some statements (`pre_exec`) will be executed when the system enters into a state. Other statements (`exec`) will be executed while the system is in the state. If any break conditions (`break_cond`) are satisfied in the state, the system will leave the state. There may be some cases where break conditions will execute some statements (`break_exec`) before breaking out of the state. Also some default statements (`post_exec`) are executed before leaving the state. We have specified these meta attributes for various types of functionality in requirements to cover wide range of different requirements documents. Using tree-like structure in XML the specifications become more descriptive as the tree expands further. Organizing and representing the requirements document in XML according to the roles of the specifications of the system not only enhances understanding of specifications but also helps to standardize requirements composition.

4 Domain-specific knowledge in XML

A requirements document usually contains specific information about how the system should work whereas the domain knowledge describes how the system is composed by its components and the constraints imposed on the components or on the relations among them. The domain-specific knowledge is a world knowledge specific to a certain domain in which the system is defined. This is well tied into the concept of the family or the ontology of systems. Depending on the level of abstraction (or the details described) of the domain knowledge, the effort to construct it can vary. By limiting the level of abstraction, the body of the knowledge can be reduced into a reasonable size and so can the effort to build it. Usually the domain-specific knowledge is defined informally or only for

a specific project, not reusable or extensible for similar systems (the systems in the same family). By using XML to specify the domain knowledge with a minimum semantics, not only can the specification be formally defined but also it can be extensible gradually building up an ontology of systems.

In our research the domain knowledge specified in XML shares many similarities with DARPA Agent Markup Language (DAML) [14] which is a frame-based language with semantics to describe ontology. Because domain knowledge is more than just an ontology so DAML is not expressive enough to describe the whole aspect of the domain knowledge. However using the XML syntax a domain knowledge can be specified in various ways leaving the interpretation of its semantics totally up to the system that uses it [15]. Therefore when a specification for domain-specific knowledge in XML is to be developed, its formal semantics as well as its expressiveness has to be considered at the same time.

The following describes an example of the domain knowledge of Car to illustrate the use of domain-specific knowledge expressed in XML in our project.

```
<system name = "Car">
  <component name = "Engine">
    <amount type ="exactly" value = "1"/>
    <unit type = "volume" value = "liter"/>
    <subcomponent name="Cylinder" type = "integer">
      <amount type ="one_of" value = "4,6,8"/>
    </subcomponent>
    <relation with = "Starter" type ="pass_to" value ="signal"/>
  </component>
  <component name = "Wheel"/>
  <component name = "Body">
    <relation with = "Frame" type ="synonym"/>
  </component>
  <relation with = "Vehicle" type ="inheritance" value ="parent"/>
  <relation with = "Van" type ="inheritance" value ="child"/>
</system>
```

According to the above domain specification, Car is composed of Engine, Wheel, Control, and Body. Vehicle is a parent of car whereas Van is a type of Car. Car can have exactly one Engine and the unit of engine is volume expressed in liter. Engine has Cylinder as its subcomponent. The number of Cylinders, which is as an integer number and is representative part of the subcomponent, can be either 2, 6, or 8. Starter passes a signal to Engine (to turn the motor). Body of Car also can be called as Frame.

The following is Document Type Definition (DTD) for the domain knowledge in XML, which defines the formal semantics of the domain-specific knowledge while pertaining proper expressive power.

```
<!ELEMENT system (component|relation)*>
<!ELEMENT component (amount?, unit?, (subcomponent|relation)*)>
<!ELEMENT subcomponent amount?>
<!ELEMENT amount EMPTY>
```

```

<!ELEMENT unit EMPTY>
<!ELEMENT relation EMPTY>

<!ATTLIST system name CDATA #REQUIRED>
<!ATTLIST component name CDATA #REQUIRED>
<!ATTLIST subcomponent name CDATA #REQUIRED type CDATA #IMPLIED>
<!ATTLIST amount type CDATA "exactly" value CDATA #REQUIRED>
<!ATTLIST unit name CDATA #IMPLIED type CDATA #REQUIRED>
<!ATTLIST relation with CDATA #IMPLIED type CDATA #REQUIRED value
CDATA #IMPLIED>

```

Note that the domain-specific knowledge in XML for the translation doesn't have to describe the domain exhaustively. Namely most of the elements and attributes are optional and attribute values can be any character strings. For example, the relationship element can represent inheritance, acronyms, message passing, etc. The minimum information required to guide the translation would be sufficient with the possibility of adding on more information later when necessary.

The domain knowledge for our CARA example is shown as follows.

```

<system name = "CARA system">
  <component name = "Computer Assisted Resuscitation Algorithm">
    <subcomponent name = "Display"/>
    <subcomponent name = "Button"/>
    <subcomponent name = "Pump Software System"/>
    <subcomponent name = "MAC"/>
    <relation with = "System" type = "hypernym"/>
    <relation with = "Software" type = "hypernym"/>
    <relation with = "Algorithm" type = "hypernym"/>
    <relation with = "CARA" type = "acronym"/>
  </component>
  <component name = "Patient"/>
  <component name = "HOST">
    <subcomponent name = "Pump"/>
  </component>
</system>

```

The above specification describes that the whole system is composed of by Computer Assisted Resuscitation Algorithm, Patient, and HOST. Computer Assisted Resuscitation Algorithm is a type of Algorithm, Software, or System that can be abbreviated as CARA.

In the natural language documents one concept can be represented by many different ways causing the translation hard to cluster similar information together. These can be acronym, synonym, and hypernym. From the CARA example, the word 'Computer Assisted Resuscitation Algorithm' is interchangeable with 'Algorithm' or 'CARA'. By using a minimum set of representative words that describes the entire components in the domain-specific knowledge, one-to-many relations between words and their various representations can be obtained and thus provides a simpler source to translate. The full set of words in the

requirements documents are mapped into the minimum set of representative words by measuring similarity among words. The hypernym and the location of the common words are used for this estimation.

In summary, by specifying domain-specific knowledge in XML and limiting the scope of the knowledge the effort needed to build up the domain knowledge for the translation can be greatly reduced.

5 Conversion from XML to Knowledge Base

The raw information of the requirements document in natural language is not in the proper form to be used directly because of the ambiguity and implicit semantics in the document. Therefore an explicit and declarative representation (knowledge base) is needed to represent, maintain, and manipulate knowledge about a system domain [16]. Not only does the knowledge base have to be expressive enough to capture all the critical information but also it has to be precise enough to clarify the meaning of each knowledge entity (sentence). In addition, the knowledge base has to reflect the structure of TLG into which the knowledge base is translated later.

The knowledge base isn't a simple list of sentences in the requirements document. The linguistic information of each sentence such as lexical, syntactic, semantic, and most importantly discourse level information has to be stored with proper systematic structure.

Each sentence of the requirements documents has to be represented in a way that eases the interpretation of the sentence. In computational linguistics this is done by constructing a parse tree of the sentence, which contains the syntactic information of the sentence. By using this semantic information we can tell what type of operation a certain object executes on other objects.

To build a parse tree, each sentence in the requirements document is read by the system and tokenized into words. At the syntactical level, the part of speech (e.g. noun, verb, adjective) and the part of sentence (e.g. subject and object) of each word are determined by standard parsing technique [8]. The corpora of statistically ordered parts of speech (frequently used ones being listed first) of about 85,000 words from Moby Part-of-Speech II [17] are used to resolve the syntactic ambiguity when there is more than one valid parsing tree. The system is able to handle elliptical compound phrases, comparative phrases, compound nouns, and relative phrases to allow the natural language in the requirements documents to be less controlled thus more natural.

Also the anaphoric references (pronouns) in a sentence are identified according to the current context history. A pronoun can represent a word, sentence, or even context. It is worthwhile to mention here that the requirements documents are easier to process than other types of textual documents in the sense that usually requirements documents have well defined structures with less ambiguities and infrequent use or narrow reference scope of pronouns.

Once the references of pronouns are determined, each sentence is stored into the proper context in the knowledge base. The structure of knowledge base re-

ffects the structure of the requirements in XML. The meta attribute information from XML is also stored in the knowledge base to be used for the translation from knowledge base into TLG. If no meta attribute or data structure is specified in the requirements in XML, the system totally relies on the linguistic information in the document to build the knowledge base according to the context. For more information on this process, we refer the readers to [18]. A part of the CARA knowledge base is shown in the Figure 2. The knowledge base of CARA

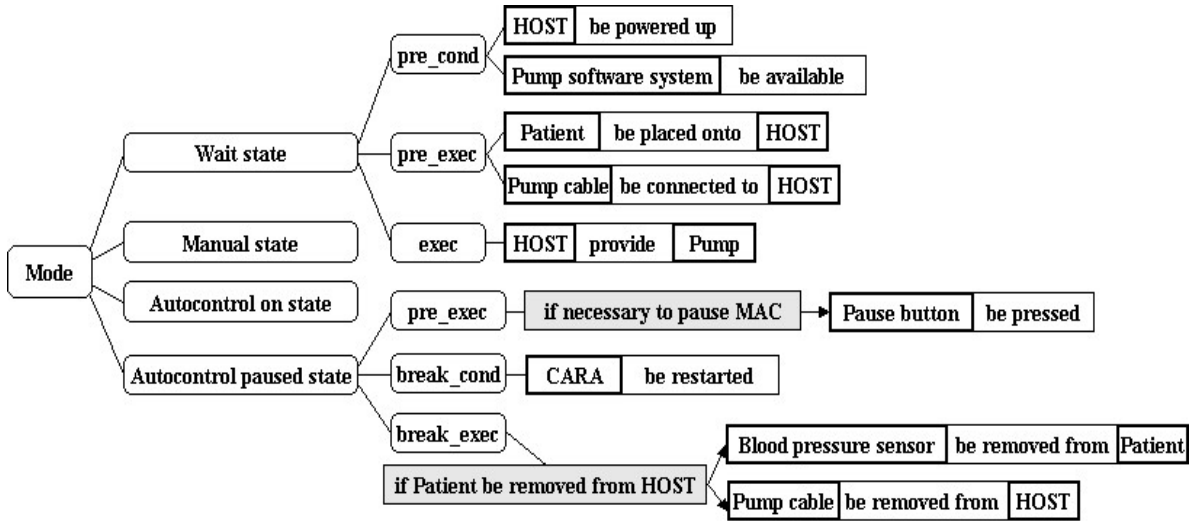


Fig. 2. Knowledge base for CARA.

system contains the meta information from the XML requirements in its tree-like structure as well as the linguistic knowledge.

In summary, the knowledge base stores not only the linguistic information of each sentence but also the data structure and meta information of related sentences as specified in the requirements in XML. Along with this process, linguistic ambiguity is detected and resolved in parsing and construction of the knowledge base.

6 Transition 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. 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 [19]. 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. Combined with the specification in the domain-specific knowledge, the knowledge base of the CARA example would be translated into the following TLG specification.

```
class Mode.  
  
  main :  
    wait state;  
    manual state;  
    autocontrol on state;  
    autocontrol paused state.  
  
  wait state:  
    HOST be powered up,  
    Pump_Software_System be available,  
    Patient be placed onto HOST,  
    Pump Cable be connected to HOST,  
    while true then  
      if Pump be infusing Fluid into Patient then  
        break,  
        HOST provide Power for Pump  
      endwhile.  
  
  manual state:  
    Pump_Software_System detect Pump_Connection,  
    Pump_Software_System monitor Occlusion and Airlock_Logic_Levels,  
    Pump Display be brought to Secondary_Display,  
    Pump_Software_System detect Back_EMF and Fluid_Impedance,  
    Pump_Software_System begin to log Infusion_Rate,
```

```

Pump continue to operate on Hardware_Setting,
Blood_Pressure_Sensor be connected to Patient,
Pump_Software_System detect Clean_Blood_Pressure_Signal,
Pump_Software_System activate Automatic_Servo-control_Start_Button.

autocontrol on state:
  if Start_Button be pressed then
    MAC control Pump,
    MAC begin Resuscitation to Prescribed_Blood_Pressure_Setpoint
  endif.

autocontrol paused state:
  if necessary to pause MAC then
    Pause_Button be pressed,
    cause Infusion_Pumping to cease
  endif,
  while true then
    if Patient be removed from HOST then
      Blood_Pressure_Sensor be removed from Patient,
      Pump Cable be removed from HOST,
      allow Pump to continue operating in Standalone_Mode,
      allow IV_Infusion to be discontinued,
      break
    endif
  endwhile.

end class.

```

The main function will execute all 4 state functions (wait state, manual state, autocontrol on state, autocontrol paused state) in parallel. However preconditions (pre_cond) in each state will be used as guarded statements to determine which state the system is currently in. For each state function, first the preconditions will be checked. If all the preconditions are met, the pre_exec statements are executed once. Then in the infinite while loop exec statements are executed. If tt break_exec and tt break_cond statements are used for the system to break out the loop. If there are any tt post_exec statements, they are executed before returning from the function.

The TLG code is translated into VDM++ by data and function mappings (for more details on this translation we refer the readers to [9]). 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. 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.

7 Contribution and Conclusion

This research project is developed as an application of formal specification and computational linguistic techniques to automate the conversion from a requirements document written in NL to a formal specification language while assisting the developers with repetitive tasks. The knowledge base is built up from a NL requirements document in XML 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 with aid of domain-specific knowledge in XML. Due to its NL-like flexible syntax 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 is working for some small examples such as the requirements for an Automatic Teller Machine (ATM), with associated banking system domain knowledge. We are performing evaluations of the system for various, more complex, requirements documents, such as the CARA Infusion Pump Controller. The system has been useful in identifying problems and ambiguities with such specifications and in identifying additional information necessary to complete the implementation. 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 NLP and TLG as a bridge between the requirements document and a formal specification language, we can achieve an executable and reusable NL specification for a rapid prototyping of requirements, as well as development of a final implementation assisting the developers throughout the software development life cycle.

Acknowledgements. 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. 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. Pohl, K.: The Three Dimensions of Requirements Engineering. Conference on Advanced Information Systems Engineering (1993) 275–292
2. Davis, A.: Software Requirements Analysis and Specification. Prentice-Hall (1990)

3. Boehm, B.W.: Software Engineering Economics. *IEEE Transactions on Software Engineering* **10** (1984) 4–21
4. Wilson, W.M.: Writing Effective Natural Language Requirements Specifications. Technical report, Naval Research Laboratory (1999)
5. Fuchs, N.E., Schwitter, R.: Attempto Controlled English (ACE). *Proc. CLAW 96, 1st Int. Workshop Controlled Language Applications* (1996)
6. Wilson, W.M., Rosenberg, L.H., Hyatt, L.E.: Automated Quality Analysis Of Natural Language Requirement Specifications. Technical report, Naval Research Laboratory (1996)
7. Girardi, M.R.: Classification and Retrieval of Software through their Description in Natural Language. PhD thesis, Computer Science Department University of Geneva, Switzerland (1996)
8. Jurafsky, D., Martin, J.: *Speech and Language Processing*. Prentice-Hall (2000)
9. Bryant, B.R., Lee, B.S.: Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Hawaii Int. Conf. System Sciences* (2002)
10. Bjørner, D., Jones, C.B.: *The Vienna Development Method: The Meta-Language*. Springer-Verlag (1978)
11. Quatrani, T.: *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley (2000)
12. IFAD: *The VDM++ Toolbox User Manual*. Technical report, IFAD (www.ifad.dk) (2000)
13. Walter Reed Army Institute for Research (WRAIR): *CARA Specification: Proprietary Document*. Technical report, WRAIR, Dept. of Resuscitative Medicine (2001)
14. Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M.C.A., Broekstra, J., Erdmann, M., Horrocks, I.: The semantic web: The roles of XML and RDF. *IEEE Internet Computing* **4** (2000) 63–74
15. Cleaveland, J.C.: *Program Generators with XML and Java*. Prentice-Hall (2001)
16. Lakemeyer, G., Nebel, B.: *Foundations of knowledge representation and reasoning*. Volume 810. Springer-Verlag Inc. (1994)
17. Grady, W.: *Moby Part-of-Speech II (data file)* (1994)
18. Lee, B.S., Bryant, B.R.: Contextual Knowledge Representation for Requirements Documents in Natural Language. *Proc. 15th International FLAIRS Conference* (2002) 370–374
19. Bryant, B.R.: Object-Oriented Natural Language Requirements Specification. *Proc. ACSC 2000, 23rd Australasian Comp. Sci. Conf.* (2000) 24–30