Published in final edited form as:

Proc IEEE Int Requir Eng Conf. 2004 September ; 2004: 262–271. doi:10.1109/ICRE.2004.1335683.

RETNA: From Requirements to Testing in a Natural Way

Ravishankar Boddu, Lan Guo, Supratik Mukhopadhyay, and Bojan Cukic

West Virginia University Morgantown, WV 26506

Ravishankar Boddu: boddu@csee.wvu.edu

Abstract

Most problems in building and refining a system can be traced back to errors in requirements. Poorly organized requirements, most often in natural language are among the major causes of failures of software projects. In this paper, we present a requirements analysis tool called RETNA and the technology behind it. RETNA accepts natural language requirements, classifies them, interacts with the user to refine them, automatically translates natural language requirements to a logical format so that they can be validated and finally generates test cases from the requirements.

1. Introduction

Requirements engineering [23] is the systems engineering activity primarily concerned with organizing, documenting and tracking requirements of software systems. Requirements are the basis for every software project, defining what the stake-holders – users, customers, suppliers, developers, businesses, – in a potential new system need from it and also how the system must behave in order to satisfy that need. Most problems in building and refining a system can be traced back to errors in requirements. Poorly organized requirements weakly related to the users and changing too rapidly are among the major causes of failures of software projects [27].

Even for medium sized software projects, eliciting, analyzing and managing requirements is a daunting task involving, among other tasks, relating users and different documents, retrieving information from them, classifying such information, accommodating for changes, formalizing the information obtained and checking for its consistency and completeness. In order to meet the demands of these tasks, specialized requirements management tools like DOORS [1] have been developed. While such tools allow capturing, linking, tracing, analyzing, and managing a wide range of information to ensure a products compliance to specified requirements and standards, and are widely used in practice, a sharp divide exists in the current state-of-practice in requirements engineering [14]. On the lower end of this divide lies the vast bulk of industrial practice in which requirements are written in a natural language and managed statically in the form of documents and document-templates. The construction of such documents by itself is a highly labor-intensive and skill-intensive task prone to errors. Advocates of formal methods have suggested the use of model-based formal notations such as Z [37] to replace natural language with all its ambiguities. However, as Finkelstein and Emmerich observe in their position paper [14] “... The reality belies the story. Most specifications in formal schemes such as Z are accompanied by large bodies of natural language text and are unusable without it...”. The

use of natural language in requirements documentation seems to persist with increasingly important roles and is unlikely to be replaced. Requirements that the acquirer (or the user) expects the developer to contractually satisfy must be understood by both parties and natural language seems to be the only medium of communication understood by both parties. Even engineers and developers not trained in formal logical notation seem to be more comfortable with natural language descriptions. Natural language seems to be language of description of design patterns [17]. Even in formal software design methodologies like design by contract [29], informal specification of the contract (the table entries for the obligations and benefits) are written in a natural language before being formalized.

The use of natural language to describe the requirements of complex software systems suffers from at least three severe problems [38]: (1) ambiguity and impreciseness, (2) inaccuracy, and (3) inconsistency and incompleteness. Critics of natural language requirements have argued that such requirements are vestiges of outmoded practice as a result of poor training and technology transfer or inadequacies of the current analysis methods [14]. In order to rectify this situation, they propose to develop new logical notations to which such natural language requirements need to be translated (of course manually) before they can be analyzed. For developers ill-trained in formal methods, such a task is daunting and error-prone. The experience of the last author of the current paper has been that even senior PhD students tend to confuse with the quantifiers while translating natural language to formal logic (e.g., while translating “a cow has four legs”, whether it is there exists a cow or for all cows). While, it is undeniable that formalization of requirements is a necessary step before any worthwhile analysis can start taking place, manual formalization is difficult except for the smallest and simplest software projects. Critics have argued [35] against the automation of the formalization step using natural language processing the primary reason cited being that at its current state of the art natural language processing is inadequate for the purpose. However, during the past decade, natural language processing [24] and text mining technology have taken a big step forward as is evident in their use in diverse commercial and government applications ranging from web search to aircraft control [2, 25].

In this paper, we make a modest attempt to solve some of the requirements engineering problems arising out of imprecise and inaccurate natural language requirements as described above using natural language processing (NLP) and text mining (TM) technology for analyzing requirements written in natural language. More precisely, our contribution in this paper is: we report a requirements analysis tool called RETNA (REquirements to Testing in a NATural way) that builds on NLP and TM technologies. RETNA accepts requirements as a discourse in natural language (i.e., a set of requirements instead of one requirement at a time), classifies them according to certain complexity measures, refines them based on user interaction, checks for the consistency of the requirements and finally generates test cases, thus proceeding all the way from requirements to testing. Several proof of concept case studies have been conducted using RETNA that have produced encouraging results.

The rest of the paper is organized as follows. Section 2 introduces the technology behind the tool RETNA. Section 3 describes the building blocks of RETNA. Section 4.1 describes some case studies in requirements analysis using RETNA. Section 4.2 describes some case

studies in automatic testing using RETNA. Related research is described in Section 5. Finally, Section 6 concludes the paper.

2. The Technology

In this section, we provide an overview of the capabilities of RETNA and the technology behind it. The requirements (which form a discourse) are first classified according to their type and complexity. Currently, the number of nouns and verbs is taken to be the measure of complexity while the type of a requirement statement is taken to be either conditional or non-conditional (our notions of type and complexity have been guided by the ARM [42] tool; also these notions of type and complexity have been preferred by our contractor partners who are currently using RETNA for analyzing the requirements of the CM1 project within NASA's Metric Data Program (MDP)). The requirements are then translated to an intermediate predicate argument structure [28] and finally to a discourse representation structure (DRS) [7]. The anaphoric pronouns are then resolved. The system then finds out the undefined predicates and searches for their definition in a library. In case, a definition of a predicate is found in the library, the user is asked whether the definition can be used to interpret that predicate. The user might agree with the definition or she might chose to enter her own definition in natural language. In either case, the definition (or the translation of the natural language definition) is used to reify the requirements. In case, the user specifies her own definition in natural language, it along with its predicate calculus translation is stored in the predicate library. Once a stage is reached when no more predicates are found to be undefined, a translator converts the discourse representation structures to the FMONA [8] syntax. FMONA is a high level language for describing WS1S (Weak monadic second order logic over one successor). An FMONA compiler translates the FMONA requirements to low level MONA [21] syntax (MONA is a low level language for describing WS1S/M2LStr (Monadic Second Order Logic over strings) formulas). The MONA tool converts WS1S/M2LStr formulas into equivalent state machines that accepts models of those formulas. The problem of checking consistency of the requirements then reduces to that of checking emptiness of the generated state machine (automaton). Each path from an initial state to a rejecting state of the state machine constitutes a test case. To demonstrate the effectiveness of RETNA, we have applied it to several case-studies like the Bay-Area Rapid Transit BART requirements [40], the requirements of a purchase tracking system [3], a California hotel reservation system [41] and analyzing some of the ACIS software requirements specification (<http://acis.mit.edu/acis/sreq/sreq-boot.html>).

Apart from use as a requirement analyzer, RETNA can be used as an automatic testing tool not only for black box testing but also for structural white box testing. The input and the output specifications can either be written in English or chose from a library of English patterns (that includes the temporal logic patterns [12]). As indicated above, the specifications can be refined and then converted to MONA input format. State machines that accept models of the specification are generated by MONA. A set of selection strategies can be used to select relevant test cases by traversing the state machines. While there is a possibility of explosion in the number of test cases generated as well as the number of states for arbitrary requirement specifications, such explosion can be dealt with by letting the user chose the most critical requirements which in most cases are much smaller compared to the

original requirements specification. From the output test cases, a test oracle is generated using JTrek [5]. Automatic testing/debugging of Java implementations is done using a combination of this oracle with JDB. We have conducted several case studies using this technology: a faulty implementation of a mutual exclusion protocol and a Java implementation of the XtangoAnimator [39] where we detected deadlocks.

Testing of software, in particular test generation, is a labor-intensive process. While it is generally accepted that automation of the testing process can significantly reduce the cost of software development and maintenance, most of the current tools for automatic (specification-based) testing require a formal specification to start with [11, 9, 34]. Usually, the specification is written in some flavor of temporal logic [11, 34]. For engineers lacking training in formal methods, writing specifications in some arcane logical formalism is a daunting task. This is cited as one of the reasons for the lack of acceptance of formal methods in industry. Introduction of specification patterns such as temporal logic patterns [12] relieves the engineer from writing formal specifications. But even such patterns are written in a logical formalism which makes them difficult to read. Also in order to use them one has to understand them for customizing them to one's own needs. For example, it is difficult to parse a pattern like $TRAssert(PExistsBetween(Q,R)) \Rightarrow \$$
 $\$TRProcessLastResult(base ++);\$$ and understand its meaning without a formal training. Graphical specification languages like timing diagrams [15] and graphical interval temporal logic [26] have been designed to alleviate these problems. We believe that a combination of natural language along with graphical notations will go a long way in making formal software engineering methodology a standard industrial practice.

The high-level architecture of RETNA is shown in Figure 1. The bi-directional arrows denote bothway interaction. In the next section, we describe the different blocks of RETNA in details.

3. RETNA: From Requirements to Testing in a Natural Way

3.1. Overview

The RETNA architecture mainly consists of two stages: the natural language stage and the logical stage. The natural language stage comprises of: parsing the natural language input, classifying the requirements according to complexity and type, performing a morphological analysis of nouns and verbs, identifying the ambiguities, performing a semantic analysis resulting in a predicate argument structure, translating a discourse into a discourse representation structure and finally resolving the anaphoric pronouns (we have not currently implemented cataphoric pronoun resolution). The current tool can not check for coherence of the discourse nor can it deal with ellipsis. The logical part consists in discovering the predicates that are undefined, interpreting the predicate with the aid of the user, resolving the ambiguities with the aid of the user, translating the DRS's to the MONA input and finally checking for consistency of the refined requirements and generating the state machine corresponding to the requirements. When used as an automatic test generation tool, RETNA can separate the logical representation of the specification (obtained by translation) into input and output specification using a simple heuristic. The test cases derived from the input specification can be used to derive the program (currently only Java is supported)

while corresponding test cases from the output specification can be used to generate a test oracle. The following subsections describe the different components of the RETNA architecture in details.

3.2. Text Acquisition

A graphical user interface is provided for entering the text or a text file. Currently only ASCII text is supported. The entire discourse has to be provided as input since the meaning of a sentence depends on the context in which it occurs. A script converts the input discourse to a collection of sentences delimited by `< s > ... < /s >`, a format accepted by the natural language parser (the whole set of statements is enclosed within one `< s > ... < /s >`; as mentioned earlier, RETNA analyzes a full set of requirements rather than one requirement at a time). Currently, due to the limitations of the parser (to be described in the next subsection) the maximum sentence length is limited to 399 words and punctuation. While in most practical case studies, the text describing the requirements will be more than 400 words, human expertise can be used to eliminate redundant/unimportant chunks of text thereby significantly reducing the size of the resulting requirements document.

Alternately, for use as an automatic testing tool, a range of specification patterns in English are provided for the user to choose from. The patterns include the natural language versions of all the temporal logic patterns [11] along with certain patterns for counting like evenness (a property holds true at all even states on the trace).

3.3. Parsing Natural Language

For parsing natural language (English) we use the popular Charniak parser [10]. The parser parses a discourse presented as a collection of sentences delimited by `< s > ... < /s >` and produces a Penn treebank style parse tree [24]. A parse of an English sentence can produce several parse trees. The Charniak parser is based on a probabilistic generative model. For all sentences s and all parses p of s , the model assigns a probability $P(s,p)$. For any sentence s the parser returns the parse that maximizes this probability. A probability to a parse is assigned by a top-down process of considering each constituent in a parse, first guessing its pre-terminal, then its lexical head and then its expansion into further constituents. A maximum-entropy inspired approach is used to condition the probabilities required in the model. The details are beyond the scope of this paper.

The Charniak parser produces a parse tree annotated with tags from the Penn Treebank tagset [28]. The Penn treebank tagset includes 45 tags like NNP (proper noun singular), JJR (adjective comparative) etc. For details consult [28]. The output of the parse tree for the sentence “If a train is on a track, then its speed should be less than the specified speed of the track” (this sentence is taken from the BART requirement specification) is given below.

```
(S1 (S ( ' ' )
( ' ' )
(SBAR (IN If)
(S (NP (DT a) (NN train))
```

```

(VP (AUX is) (PP (IN on) (NP (DT a)
(NN track))))))
(ADVP (RB then))
(NP (PRP $ its) (NN speed))
(VP (MD should)
(VP (AUX be)
(ADJP (ADJP (JJR less))
(PP (IN than)
NP (NP (DT the) (VBN specified) (NN speed))
(PP (IN of) (NP (DT the) (NN track))))))))
( . . )
( ' ' )
( ' ' ' ) ) )

```

Here *SBAR* indicates a clause (“train is on a track”) introduced by a subordinating conjunction (“If”). This indicates that the sentence is possibly a complex conditional one. *IN* denotes that “If” is a subordinating conjunction, “a train” is the noun phrase (subject) in the antecedent clause where *DT* “a” is the determiner, *NN* indicating that “train” is a singular common noun, “is on a” is the verb phrase in the antecedent clause. In the consequent clause, the tag *PRP*\$ indicates that “its” is a possessive pronoun. Explanations for the other tags can be similarly obtained by referring to the Penn treebank tagset.

3.4. From Parse Tree to Meaning

The Penn tree bank style parse tree obtained from the parser does not provide a clear concise distinction between verb arguments and adjuncts where such distinctions are clear, with an easy-to-use notational device to indicate where such distinction is somewhat murky. The goal of a predicate argument scheme is to label each argument of a predicate with an appropriate semantic label, as well as distinguish the arguments of the predicate as well as adjuncts of the predication.

Before proceeding with a syntax-directed semantic analysis to obtain the predicate-argument structure, we first determine the complexity of the requirement specification as well as the type of each statement in the specification. In our case, as discussed previously, the complexity measure is the number of nouns and verbs present in the requirements specification. The number of nouns is a measure of how many roles are involved in the requirements while the number of verbs is a measure of the number of tasks that may need to be accomplished in order to meet the requirements as well as a measure of the number of relations existing between the different roles. The ARM tool uses similar measures for measuring the quality of a requirements specification. The complexity of a requirements specification can be obtained by simple traversal of the parse trees corresponding to the individual sentences. In our current implementation of RETNA, we support classification of requirement statements based on the conditional/non-conditional criterion. The current detection procedure for conditional statements consists in detecting the *SBAR* tag indicating the presence of a clause introduced by a subordinating clause. It then searches for the *IN* tag for detecting the presence of a subordinating conjunction. Finally, it detects presence of

phrases like “if...then...”, “...because...”, “...so that...”, “...as...” etc. Based on feedback from the user the type of a sentence may be modified.

The next step is a morphological analysis of the *inflected words*. An *inflection* [24] is a combination of a word stem with a grammatical morpheme (smaller meaning-bearing unit), usually resulting in a word of the same class, and usually filling some syntactic function like agreement. For example, English has the inflectional morpheme *-s* for marking the plural on nouns, and the inflectional morpheme *-ed* for marking the past tense on verbs. For each noun or verb, we obtain the stem using a Perl implementation of the Porter’s stemming algorithm [24]. While stemming algorithms such as Porter’s are not perfect, we avoid the use of a more accurate finite state transducer based method due to the large on-line lexicon required.

Finally, we use a syntax-directed semantic analysis to extract the predicate-argument structure. The Penn tree bank style parse tree is augmented with semantic attachments. Overall, our basic translation scheme largely follows that of [24]. Special semantic attachments are used for complex sentences like conditional ones (obtained by detecting keywords and user feedback). For example, for sentences of the form “If S1 then S2” (grammar rule: $S \rightarrow \text{If } S1 \text{ then } S2$) or of the form “If S1, S2”, the semantic attachment is $S1.sem \Rightarrow S2.sem$ where for each $i \in \{1,2\}$, $S_i.sem$ is the meaning representation associated with S_i . Similarly, semantic attachments are given to sentences linked by “so that”. For conditional statements with subordinating conjunctions “because”, “since”, “as” like “S1 because S2”, we use the semantic attachment $S2.sem \Rightarrow S1.sem$. Following [24], we use a reified representation for common nouns, adjectives and verbs (e.g., $isa(x, trian)$ for the common noun “train”). For verbs, we use a λ -notation to describe the semantic attachment. Thus the semantic attachment corresponding to the verb “eats” is $\lambda x \lambda y \exists e \text{ isa}(e, \text{eats}) \wedge \text{eater}(e, y) \wedge \text{eated}(e, x)$ where stands for the event “eats” while x and y are place holders for the two roles of the verb “eats”, viz., the subject and the object. A meta-rule transforms the λ notation to predicate argument structure. Adjective phrases are dealt with by *intersection semantics* [24] as usual. For example the semantic attachment corresponding to “top view” is $\lambda x \text{ isa}(x, \text{view}) \wedge \text{isa}(x, \text{top})$. Quantifier scoping for universal quantifiers is done using an adhoc set of heuristic preference rules. Imperative sentences and genitive noun phrases are dealt with in the same way as [24]. For lack of space, we will not describe the full grammar for translation.

The next step combines the predicate-argument structures obtained from the individual sentences to form the discourse representation structure. A discourse representation structure is a semantic structure involving a set of variables called discourse referents and a set of conditions. The conditions can either be atomic formulas or boolean combination of other discourse representation structures. For a precise definition of a DRS, we refer the reader to [7].

The algorithm followed in obtaining a DRS is similar to the recursive procedure described in [7]. Each predicate-argument structure is first converted into an atomic discourse representation structure (i.e., a DRS that does not include a nested DRS). For each variable, a discourse referent is created. The predicates form the conditions. If a simple sentence follows another sentence, then its information is added to the DRS of the previous sentence.

For conditional statements, implications (like “all men are mortal”) and compound statements, we follow the threading strategy using the λ notation of [7]. Finally, anaphoric pronoun resolution is done by computing accessibility [7]. Accessibility is a simple geometric concept based on the way DRS’s are nested one inside another. Roughly, a DRS D_1 is accessible from another DRS D_2 if D_1 equals D_2 or D_1 subordinates D_2 where a set of rules define when a DRS subordinates another [7]. The translator from the Penn tree bank style parse tree to the DRS is written using 700 lines of Awk script. The DRS corresponding to the sentence “If a train is on a track then its speed should be less than the specified of the track” whose parse tree was given in Section 3.3 is shown below.

```

EX X1
EX X2
end referent
isa ( X1 , train )
isa ( X2 , track )
ison ( X1 , X2 )
end discourse
= >
EX X3
EX X4
EX X5
end referent
isa ( X3 , speed )
of ( X1 , X3 )
isa ( X4 , speed )
of ( X4 , X2 )
isa( X4,specified)
shouldbelessthan ( X3 , X4 )
end discourse
end discourse

```

The above DRS is of the form $D_1 \Rightarrow D_2$ where D_1 and D_2 are atomic DRS’s. The discourse referents for D_1 are X_1 and X_2 while the conditions are the atomic formulas $isa(X_1,train), isa(X_2,track)$ and $ison(X_1,X_2)$. Similarly for the DRS D_2 . Notice that the possessive pronoun “its” has been resolved to “train” in $of(X_1,X_3)$. The first DRS D_1 describes the antecedent, i.e., X_1 is a train, X_2 is a track and X_1 is on X_2 , while the second DRS D_2 describes the consequent, i.e., X_3 is the speed of X_1 , X_4 is the specified speed of X_2 and X_3 should be less than X_4 .)

3.5. Requirement Refinement

The requirements obtained from the user may be too abstract and imprecise. RETNA now tries to refine the requirements based on user interaction. The end result of this step is an *ontology* involving the roles in the requirements and the relations that exist between them.

To refine the requirements, RETNA first tries to discover in the requirements collections of words that are synonyms of one another. For example, a train's speed may be referred to as "speed" in one sentence and as "velocity" in another. To this end, RETNA consults Wordnet [13] for the set of synonyms of a word. Wordnet is a well-developed and widely used database of lexical relations for English. The database can be accessed programmatically through the use of a set of C library functions. If a word is found in the list of synonyms of another, the user is asked whether the same meaning can be attached to both, i.e., whether the second word can be "replaced" by the first word. Thus depending upon the feedback from the user, a condition $isa(X, velocity)$ in a DRS may get replaced by $isa(X, speed)$.

The next step consists of determining which atomic formulas are undefined in the DRS. Thus in the DRS presented in Section 3.4, the atomic formula $shouldbelessthen(X3, X4)$ is undefined (uninterpreted). To interpret this atomic formula, RETNA first searches in its library for the definition for the atomic formula $shouldbelessthen(X, Y)$. If a definition is found, it consults the user about whether to interpret the atomic formula with a definition from the library (it presents the user with all the definitions of the atomic formula found in the library). If either no definition is found in the library, or the user does not agree with definitions in the library, she will be asked to specify what it means by "should be less than" in the sentence "If a train is on a track then its speed should be less than the specified speed of the track". The user might specify "X should be less than Y if $X < Y$ ". This input will be used to interpret the atomic formula $shouldbelessthen(X3 < X4)$ (after another round of parsing and translation) as $shouldbelessthen(X3, X4) \Leftarrow (X3 < X4)$ (the user might be less precise in specifying the meaning of "should be less than"; in this case more refinement is needed and the user will be prompted to refine her specification). Thus for the sentence "Managers can access the database", the user will be asked the meaning of the word "Managers". If the user specifies "Tom and Jim are managers", the interpretation for $isa(X, Manager)$ will be $isa(X, Manager) \Leftarrow (X = Tom \text{ or } X = Jim)$. Finally, we use a "closed world assumption" to interpret the atomic formulas. Hence, lack of extra information would mean that in a closed world $isa(X, Managers) = (X = Tom \text{ or } X = Jim)$ and $shouldbelessthen(X3, X4) = X3 < X4$. The definitions of $shouldbelessthen(X, Y)$ and $isa(X, Managers)$ are then stored in RETNA's library along with their English interpretations "X should be less than Y if $X < Y$ " and "Tom and Jim are managers" respectively for use in future sessions. Thus a refinement of the requirements results in a model-theoretic interpretation of the atomic formulas. The requirement refinement engine is written in Perl involving 100 lines of code. It communicates with the user through a user interface written in shell script. A screen shot of the interaction of RETNA with the user for refining the requirements is given in Figure 2.

3.6. From DRS to FMONA

FMONA [8] is a high level language that acts as a frontend to the MONA tool. It is possible to define enumerated and record with update types and quantify over them. Furthermore, it allows the definition of higher order macros parameterized by types and predicates. The translation of DRS to FMONA follows the algorithm used in [7] for converting DRS's to first order logic. For each predicate, we create a record type whose fields have the the same type as the predicate. For example, for the predicate $isa(X, trian)$, we define X as having a

generic type. We create an enumerated type *role* whose elements are the set of all words occurring in the second argument of the *isa* predicate. We can then define a record *isa* having two fields one of generic type and the other of type *role*. The rest of the translation procedure is straightforward. Below we show the FMONA translation of the DRS presented in Section 3.4 after refining the definition of the *shouldbelessthan* predicate.

```

var nat N;
type generic=...N;
type time =...N;
type roles={Train,Track,Speed,Specified};
type isa=record{X: generic; r: roles;};
type ison=record{X1:generic;
X2: generic;u: time;};
type ofx=record{X1:generic;
X2:generic;u:time;};
pred shouldbelessthan(var generic X,
var generic Y)=(X<Y));
(ex generic X1:(ex generic X2:(ex isa isa1
:(ex isa isa2:(ex ison ison1:(isa1.X=X1
& isa1.r=Train & ison1.X1=X1 & ison1.X2=X2
& isa2.X=X2 & isa2.r=Track))))))=>
(ex generic X3:
(ex generic X4:(ex isa isa3: (ex isa isa4:
(ex isa isa5:(ex ofx ofx1:(ex ofx ofx2:
(isa3.X=X3 & isa3.r=Speed & ofx1.X1=X1 &
ofx1.X2=X3 & isa4.X=X4 & isa4.r=Speed &
ofx1.X1=X4 & ofx1.X2=X2 & isa5.X=X4 &
isa4.r=Specified &
shouldbelessthan(X3,X4))))))))));

```

3.7. From FMONA to MONA

MONA [21] is a logic-based programming language and a tool that translates programs, that is, formulas, to finite-state machines that accept models of the formulas. The MONA tool implements decision procedures for weak monadic second order theory of one successor (WS1S). While the complexity of the decision procedure is nonelementary recursive, MONA is known to perform well in practice. Although it is known that natural language is not regular [24], we believe that finite state machines can be a good approximate representation of requirements specification. The translation from FMONA to the MONA input language is done by the FMONA [8] compiler.

Checking consistency of the requirement specification now reduces to the problem of checking (non)emptiness of the finite state machine (automaton) generated by MONA. MONA has procedures that can check emptiness of the generated state machine. In order to

deal with the state explosion problem, MONA uses BDD's to represent the state space symbolically. For the example FMONA code, MONA generate an automaton 5 states.

Tests are generated by traversing the state machine to find a rejecting path.

4. Case Studies and Experience

4.1. Requirements Analysis

We have conducted several case studies in requirements analysis using RETNA. Our first case study was done using the popular BART requirements specifications [40]. The requirements contained statements like "A train should not be close to another train ahead so that if the train ahead stops, it will hit it.". For these sentences RETNA asked us (we started with an empty library, except for a mathematical library) to explain what it means for a train to be ahead. We replied back with "A train is ahead of another train if its location is greater than the location of the other train". It then asked us what it means by "greater than" in the previous sentence. Our reply: "A is greater than B if $A < B$ ". For the adverb close, RETNA asked us to explain its meaning. The answer was "A train is close to another train if the difference between its location and the location of the other train is less than mindistance". After refining "difference between" and "less than" RETNA asked us to specify mindistance. We replied with " $20 < mindistance < 30$ ". After a few refinements, we decided to check the consistency of the requirements. It turned out that the requirements were consistent but MONA came back with a state machine with more than a million states. We continued experimenting to find out the effect of refinements on the size of the generated state machine. It was found that as the requirements were getting refined the size of the generated state machine was getting reduced. Finally, we could generate a state machine with 13000 states.

Our other case studies in requirements analysis were with a California hotel reservation system [41], a purchase tracking system [3] and fragment of the ACIS software requirements specification from MIT (<http://acis.mit.edu/acis/sreq/sreq-boot.html>). We had similar experience in all these case-studies. Currently, in a joint project with SAIC, we are analyzing the requirements of the CM1 project within the NASA's Metric Data Program (<http://mdp.ivv.nasa.gov/index.html>).

4.2. Automatic Testing

We have conducted case studies using RETNA as a tool for automatic testing. For this, the specification is first input to RETNA in a natural language (or chosen from the set of patterns in natural language provided by RETNA). With the aid of the user and a few heuristics, RETNA can split up the specification into input and output specification. Both specifications then get refined following the procedure described above ultimately getting converted to MONA input language. MONA converts both specifications into state machines. By exploring the state machine corresponding to the input specification, test data for running the program under test is obtained (a test case is a path from the initial state to a rejecting state in the state machine). To avoid state explosion as well as explosion in the number of test cases generated, the user can be asked to provide only the most critical requirements specification to RETNA. Such a strategy can eliminate most of the unwanted

and irrelevant requirements and the resulting will generate a much smaller state space. From the state machine corresponding to the output specification, a test oracle for Java programs is generated in JTrek. We modified JTrek library [5] from Compact Inc. to monitor the runtime values of variables. When the Java implementation is executed, the runtime values of the monitored variables will be output into a log file with the aid of Jtrek.

We used this testing technology to test several Java programs. The first program is a Java implementation of a scheduler for five processors (this program was known to deadlock). We chose one of the natural language pattern for specifying deadlock (this pattern originates from Dwyer's temporal logic patterns [12]). After a few refinements, RETNA came up with a test oracle. While executing the Java implementation, the deadlock was found; it was due to a faulty initialization of a turn variable. We then tried our technique on the XtangoAnimator Program (<http://www.mcs.drexel.edu/shart-ley/ConcProgJava/lib/XtangoAnimation/XtangoAnimator.java>) We used the same deadlock pattern as our specification. A previously known [39] deadlock was discovered.

Our experience during these case studies have been that the state machines generated by MONA can have thousands of states (for the experiments that we have conducted so far, we haven't experienced larger state machines). Hence there can be potentially millions of test cases. At the current stage of our implementation we just chose a few of them in an adhoc way. Currently we are in the process of integrating sophisticated test selection strategies into RETNA.

5. Related Research and Discussion

According to the IEEE guide to Software Requirements Specification [4] four of the properties that characterize a good software requirements specification are: (1) completeness, (2) verifiability and testability, (3) consistency, and (4) traceability. Typically initial requirements are under-specified, necessary and incomplete requiring refinement. Refinement is done manually by asking "how" and "why" questions in a goal-directed way [40]. Testing is closely related to requirements at every level. Testing should begin as early as possible, since waiting until the design is almost complete before carrying out any kind of testing can lead to very expensive design changes and rebuilds. Further, modern lightweight software engineering methodologies like extreme programming require that unit tests be written before any coding can proceed. Consistency ensures that it is possible to find a design that conforms to the requirements. Traceability entails understanding how the high-level requirements – objectives, goals, etc.– are transformed into low level requirements.

For any reasonably sized software project, ensuring completeness, generating test cases, checking consistency and tracing the requirements manually is difficult. On the other hand, it is not possible to carry out these steps automatically with requirements in a raw natural language form. Hence, we need to extract out of the natural language requirements (automatically) a formal logical description that allows automatic classification of the requirements based on its type (e.g., conditional/non-conditional) and complexity (e.g., number of nouns and verbs), automatic identification of the goals at which refinement/reification or disambiguation is necessary and helps the engineer formulate the "how" and

“why” questions to elicit the refinement/disambiguation information from the user, automatically checking consistency once the requirements are sufficiently reified and constructing a model from which test cases can be extracted automatically.

Recently, there has been a surge of activity in using natural language processing for software requirements analysis. This is witnessed by the attendance at a recent tutorial on). natural language processing at the Requirements Engineering Conference 2003. Previous works have ranged from using natural language processing for requirements analysis to converting natural language specifications to those accepted by a model checker. The works that come closest in spirit to ours are [6, 31, 18, 32, 16, 22, 20, 19, 36]. In [6], Circe, a web-based environment for gathering natural language requirements is described. A tool called Cico acting as a front end for other components of Circe perform natural language recognition. Cico works on the text on a requirement by requirement basis. In contrast RETNA can process a whole discourse. Instead of using on-the-fly user-assisted refinement like RETNA, Cico depends on a glossary accompanying the requirements. Finally, unlike RETNA, Circe cannot proceed all the way through the software design cycle from requirements to test generation. In [31], a tool that processes requirements in controlled English is presented. Writing requirements in controlled English can be as painful as writing them in a formal notation. Besides, most of the requirements that we find in practice do not restrict themselves to any controlled version of English. In [18], the work started by [6] is continued. They apply the technique for analyzing the NASA specification of the node control software on the International Space Station. Unlike Circe, or the tool described in [31], RETNA can also be used as an automatic testing tool. In [16], a controlled fragment of English called Attempto Controlled English is converted to first order logic. Formal verification with natural language specifications is considered in [22]. A methodology and a prototype tool called LIDA for linguistic assistance in the development of models from natural language requirements is presented in [32]. LIDA allows conceptual modeling through linguistic analysis. It does not provide facilities for classifying or validating requirements or generating test cases. In [20], a methodology for analyzing the domain knowledge communication problem from the perspectives of current cognitive linguistic theory is presented. Gunter [19] presents a case study translating natural language specifications into linear temporal logic but no implemented tool has been reported. A method, similar to ours, for translating natural language specifications to the universal fragment of Computation Tree Logic is presented in [30]. They use this translation for the purpose of formal verification which is orthogonal and much limited in scope compared to this paper. The tool PROPEL [36] uses natural language patterns for specification. The tool ARM [42] uses natural language processing techniques to automatically analyze the “complexity” and the quality of natural language requirements, but, as our partner contractors working on NASA projects found out, it is very limited in scope. PROPEL and ARM [42] can be seen as a step in the right direction. RETNA can be considered as the next step forward in the same direction.

On the automatic specification-based testing side, we have already mentioned that the works of [34, 9, 11, 33]. Each of these require formal specifications to start with. Besides, we can write properties that are not expressible in the formalisms used by these systems. For

example consider the property that on any run of the program, X is 0 at every even step. Such a property is neither expressible in temporal logic (the input specification language for [11, 34, 33]) nor in Java predicates (the input specification language for [9]). This property (with a little refinement) can be easily translated into MONA input language.

6. Conclusion and Future Work

RETNA can, by no means, be considered to be a panacea for all requirements engineering problems. RETNA depends on the user: the more precisely the user interacts with it, the better RETNA performs. Of course RETNA is not a complete tool. RETNA's performance depends on the quality of the natural language requirements. But certainly, RETNA is a proof of the concept that natural language processing tools can become part of the requirements engineer's life.

In terms of future work, we are concentrating on translating natural language requirements to other notations like SCR tables where they can be easily simulated and visualized.

References

1. <http://www.telelogic.com/products/doorsers/index.cfm>
2. <http://www.teoma.com>
3. http://members.aol.com/acockburn/papers/prts_req.htm
4. Recommended practice for software requirements specifications. IEEE Std. 1993:830–1993.
5. 2003. <http://h18012.www1.hp.com/java/download/jtrel/>
6. Ambriola, V.; Gervasi, V. Processing natural language requirements; Proceedings of the IEEE International Conference on Automated Software Engineering; 1997.
7. Blackburn P, Bos J. Working with Discourse Representation Theory : An advanced Course in Computational Semantics. 1994
8. Bodeveix, JP.; Filali, M. Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer; 2000. Fmona: A tool for expressing validation techniques over infinite state systems.
9. Boyapati C, Khurshid S, Marinov D. Korat: Automated testing based on java predicates. Proc. International Symposium on SOftware Testing and Analysis. 2002 May.
10. Charniak E. A maximum-entropy-inspired parser. Technical Report CS-99-12. 1999
11. Drusinsky, D. <http://www.time-rover.com/TRindex.html>
12. Dwyer, MB.; Avrunin, GS.; Corbett, J. Patterns in property specifications for finite state verification; Proceedings of the International Conference on Software Engineering; 1999.
13. Fellbaum C. Wordnet: An electronic lexical database. 1998
14. Finkelstein A, Emmerich W. The future of requirements management tools. Information Systems in Public Administration and Law. 2000
15. Fisler K. Diagrams and computational efficiency. Words, Proofs, and Diagrams. 2002
16. Fuchs NE, Schwertel U, Schwitter R. Attempto Controlled English — not just another logic specification language. Lecture Notes in Computer Science. 1999; 1559:1–20.
17. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. 1995
18. Gervasi V, Nuseibeh B. Lightweight validation of natural language requirements. Software Practice and Experience. 2002; 32:113–133.
19. Gunter E. From natural language to linear temporal logic: Aspects of specifying embedded systems in ltl. Proceedings of the Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation. 2003

20. Hanks KS, Knight JC, Strunk EA. A linguistic analysis of requirements errors and its application. 2002
21. Henriksen JG, Jensen J, Jorgensen M, Klarlund N, Paige R, Rauhe T, Sandholm A. Mona: Monadic second-order logic in practice. TACAS'95. 1995
22. Holt A. Formal verification with natural language specifications: guidelines, experiments and lessons so far. South African Computer Journal. 1999 Jan.24:253–257.
23. Hull, MEC.; Jackson, K.; Dick, AJJ. Requirements Engineering. Springer; 2002.
24. Jurafsky, D.; Martin, JH. An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition. Prentice Hall: 2000.
25. Kenneth W. Eucalyptus: Integrating natural language input with a graphical user interface. 1994
26. Kutty G, Ramakrishna YS, Moser LE, Dillon LK, Melliar-Smith PM. A graphical interval logic toolset for verifying concurrent systems. Computer Aided Verification. 1993:138–153.
27. Lutz, R. Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press; 1993. Analyzing software requirements errors in safety critical embedded systems; p. 126-133.
28. Marcus M, Kim G, Marcinkiewicz M, MacIntyre R, Bies A, Ferguson M, Katz K, Schasberger B. The penn treebank: Annotating predicate argument structure. In ARPA Human Language Technology Workshop. 1994
29. Meyer B. Object Oriented Software Construction. Prentice Hall. 1997
30. Nelken, R.; Francez, N. Automatic translation of natural language system specifications into temporal logic; Lecture Notes in Computer Science (LNCS 1102), Proc. CAV96, the 8th International Conference on Computer Aided Verification; 1996.
31. Osbourne, M.; MacNish, C. Processing natural language software requirements specifications; Proceedings of the International Conference on Requirements Engineering; 1996.
32. Overmyer, SP.; Lavoie, B.; Rambow, O. Proceedings of the 23rd international conference on Software engineering. IEEE Computer Society; 2001. Conceptual modeling through linguistic analysis using lida; p. 401-410.
33. Rayadurgam, S.; Heimdahl, MP. Coverage based test-case generation using model checkers; 8th International Conference and Workshop on the Eng Of Computer Based Systems (ECBS'01); 2001.
34. Richardson, DJ.; LeifAha, S.; O'Malley, TO. Specification-based oracles for reactive systems; Proceedings of the International Conference on Software Engineering; 1992.
35. Ryan K. The role of natural language in requirements in requirements engineering. Proceedings of the International Symposium on Requirements Engineering. 1993:240–242.
36. Smith RL, Avrunin G, Clarke LA, Osterweil L. Propel: an approach supporting property elucidation. Proceedings of ICSE 2002. 2002
37. Spivey, JM. The Z Notation: A Reference Manual. Prentice Hall: 1992.
38. Stokes DA. Requirements analysis. Computer Weekly Software Engineers Reference Book. 1991
39. Stoller, S. Testing concurrent java programs using randomized scheduling. ENTCS. 2002.
40. van Lamswerde, A. Goal-oriented requirements engineering: Aguided tour; Proc RE'01 - International Joint Conference on Requirements Engineering; 2001.
41. Wills, A. Patterns for specification and refinement. <http://www.trieme.com>
42. Wilson WM, Rosenberg LH, Hyatt LE. Automated quality analysis of natural language requirement specifications.

Figure 1.
Architecture of RETNA

There are following explanations in the database:

- 1 Component is part of the flight
- 2 Component is a function of the program

Please enter your choice
(Enter 0 to indicate no satisfying choice): 0
Please make your own definition,
each sentence per line,
or xxx to skip or finish.
An attitude control system is a component

Figure 2.
Screenshot from RETNA requirements refinement