Evaluating the Use of Model-based Requirements Verification Method: A Feasibility Study

Daniel Aceituna, Hyunsook Do, Gursimran Singh Walia
Computer Science Department
North Dakota State University
Fargo, ND, USA
{daniel.aceituna, hyunsook.do, gursimran.walia}@ndsu.edu

Seok-Won Lee
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA
slee@cse.unl.edu

Abstract - Requirements engineering is one of the most important and critical phases in the software development life cycle, and should be carefully performed to build high quality and reliable software. However, requirements are typically gathered through various sources and represented in natural language (NL), making requirements engineering a difficult, fault prone, and a challenging task. To address this challenge, we propose a model-based requirements verification method called *NLtoSTD*, which transforms NL requirements into a state transition diagram (STD) that can be verified through automated reasoning. This paper analyzes the effect of *NLtoSTD* method in improving the quality of requirements. To do so, we conducted an empirical study at North Dakota State University in which the participants employed the *NLtoSTD* method during the inspection of requirement documents to identify the amibiguities and incompleteness of requirements. The experiment results show that the proposed method is capable of finding ambiguities and missing functionalities in a set of NL requirements, and provided us with insights and feedback to improve the method. The results are promising and have motivated the refinement of NLtoSTD method and future empirical evaluation.

Keywords - Requirements verification, inspection, model-based verification, STD model

I. INTRODUCTION

While all activities in the software development life cycle should be carefully performed to build high quality and reliable software, it is well recognized that requirements engineering is the most important and critical phase to such success [3, 4]. Typically, requirements are gathered through various sources and represented in natural language (NL), so NL requirements can be interpreted differently by various stakeholders; this makes requirements engineering a difficult, fault prone, and a challenging task.

To improve the quality of requirements specifications written in natural language, many approaches have been developed and validated through controlled and case studies (e.g., [24-27]). Researchers have devoted a considerable effort to developing methods for detecting and removing the early lifecycle faults i.e., mistakes recorded in the requirements and design documents [19-22].

Researchers have developed and empirically evaluated fault checklist based inspection methods to help developers identify different types of early lifecycle faults [19-22]. However, despite the reported success of fault-based inspection techniques, they do not lead developers to find all type of problems. Furthermore, previous researchers have utilized methods beyond standard fault checklist based inspection to detect the ambiguities and incompleteness in NL requirements. Most notable amongst these methods include Walkthroughs [19], Linguistic Analysis [21, 22], Consistency checking [23]. However, even when faithfully applying these methods, it is estimated that the majority of software development effort is still spent on fixing problems that should have fixed early in the lifecycle [3].

Much of this rework is the result of the fact that the fault-based inspection methods rely on the reader's ability to understand the things the same way as the writer of the requirements document. Because of the flexibility and inherently ambiguous nature of NL specifications, different people can have different interpretations of the requirements without noticing the ambiguity. Similarly, due to the requirements amalgamation, it may be difficult to find all the required requirements and discover the related requirements.

Model-based approaches [10, 12, 13] can detect such types of defects more easily because when the requirements are formally modeled or checked by formal methods, the properties, such as inconsistency and ambiguity, are clearly addressed and handled. For this reason, to date, many researchers have utilized the model-based approaches for verifying the natural language specifications. For example, Kof [10] proposes a method that analyzes NL requirements with computational linguistics and generates Message Sequence Charts to verify NL requirements. Similarly, Sutcliffe et al. [16, 18] present a method that converts use cases into scenarios semi-automatically and validates scenarios using rule-based frames that detect incomplete/incorrect event patterns. Other researchers have focused on automating the modeling process using scenarios collected from end-users [12, 13, 15, 17].

While model-based approaches provide a systematic way to identify aforementioned requirements problems, building models often requires NL translation and this translation process can be problematic due to the inherent incompleteness and ambiguities of NL [1, 5]. An erroneous translation of NL requirements can result in a wrong model, and thus eventually can produce software that stakeholders do not want. To address this problem, previous researchers have proposed modeling techniques using an automated NL translation approach [6-9]. Automation can certainly reduce human errors and improve the translation process, but complete automation of this process is not possible because often NL requirements can be interpreted in multiple ways and thus human judgment is inevitable to lead correct/sensible interpretations.

To address this problem, we propose a new method that translates NL requirements into a State Transition Diagram (STD) in an incremental manner (hereafter refer to as NLtoSTD) and allows requirement engineers and other stakeholders to participate in the translation process. This approach can correct and refine requirements during the translation process by identifying ambiguities and incompleteness in the NL requirements. We define incompleteness, as a missing requirement or any missing element that results in a disconnected STD, whereas an ambiguity results when an element is not explicitly stated, but its phraseology is such that its value is implied, resulting in the user's interpretation and an STD that is partially defined by the user.

The NLtoSTD method we propose provides a means of exposing incompleteness and ambiguities in a set of natural language requirements, while transforming the requirements into a STD. While both defects have been explored by others, our method differs from the aforementioned approaches in that the direct mapping from NL to model is preserved in the translation process. Each NL requirement becomes a segment of the STD. This means that any adjustments made to the model can be directly made to the requirements, and visa-versa.

To initially investigate the feasibility of our approach, we conducted a controlled experiment to see whether the NLtoSTD method can help detect the missing functionalities and ambiguities in the natural language requirement specifications. A controlled experiment with university students was performed to determine if students using the NLtoSTD method were able to find lager number of ambiguous and incompleteness faults than using the fault checklist inspection method. Our results show that the NLtoSTD can be more effective in exposing missing functionality and in some cases more ambiguous information than a fault checklist method. More importantly, the results provided insights into how the proposed method can be improved with respect to its effectiveness and efficiency.

The rest of the paper is organized as follows. Section II describes our NLtoSTD approach in detail. Section III describes the study design, and Section IV presents data analysis and results. Section V discusses the threats to validity. Section VI discusses our results, the lessons learned from this study, and the suggested improvements for the proposed method. Finally, Section VII presents conclusions.

II. Methodolgy

The basis of our NLtoSTD method is to turn a set of nature language requirements directly into a STD model, by transforming each requirement into a STD building block [11]. Requirements engineers and stakeholders can readily observe where the conceptual gaps and ambiguities lay in the NL requirements by examining the resulting STD. Once the faults are corrected in the STD model, the corrections can be mapped back to the NL requirements, due to the direct transformation that occurred from the NL to the STD. Subsequently, the NL requirements can be revised to correct for the detected ambiguities and incompleteness.

Furthermore, once a STD is obtained, it can be analyzed automatically to expose other potential faults, such as inconsistencies by looking for path traversals that are inconsistent with one another. Whereas, exposing inconsistencies in the NL representation, by inspection, involves thorough reviewing of all the requirements and looking for terms that are semantically contradictory.

To achieve direct traceability between the STD and the NL, for a given NL requirement, we transform it into the three elements {***Sc***, ***T***, ***Sn***} that make up a STD Building Block (STD-BB) (Figure 1). The building blocks (one per NL requirement) are then used to construct a STD. The basis for this NL to STD-BB transformation is that a functional requirement typically describes an entity transitioning from one state to another. For example, a requirement: "While the car is moving forward, the driver shall be able to stop it, by applying the brake", would map to the three elements: {***Sc***: Moving, ***T***: Applying Brake, ***Sn***: Stop}. The entity (Car) is described as transitioning (***T***) from moving (***Sc***) to stopping (***Sn***), which is then represented by a STD-BB (Figure 1). In this requirement, the three elements are explicitly stated, yielding definable values for ***Sc***, ***T***, and ***Sn***.

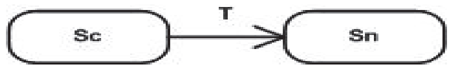


Figure 1. The STD Building Block (STD-BB)
(Sc: moving forward, T: ApplyingBrakes, and Sn: Stop)

In practice, however, often requirements ambiguously imply one or more values for ***Sc***, ***T***, and ***Sn***, thus identifying a value for each element would not be obvious. For instance, the prior requirement may have stated as: "The driver shall stop the car, by applying the brakes." Note that ***Sc*** is not explicitly stated as "Moving", but rather implied. In our STD-BB, we use questions marks (???) to denote an element that is not explicitly specified. Thus, in this example, we would define the three elements as {***Sc***: ???, ***T***: Applying Brake, ***Sn***: Stop}. It may be safe to assume that the car is moving prior to stopping, but this requires an assumption, and assumptions can be erroneous. In this example, it is not clear whether we assume "moving forward", "moving backward", or both. It is better to explicitly state what may seem obvious than to allow the possibility of an erroneous (and costly) assumption, therefore a key goal of the NLtoSTD method is to expose assumptions.

To illustrate the steps of our methodology, we will use a set of five NL requirements of a simple battery control system in a cell phone. The left side of Figure. 2 shows these five requirements. To systematically identify the three elements, for each of the five requirements, we use the following three questions:

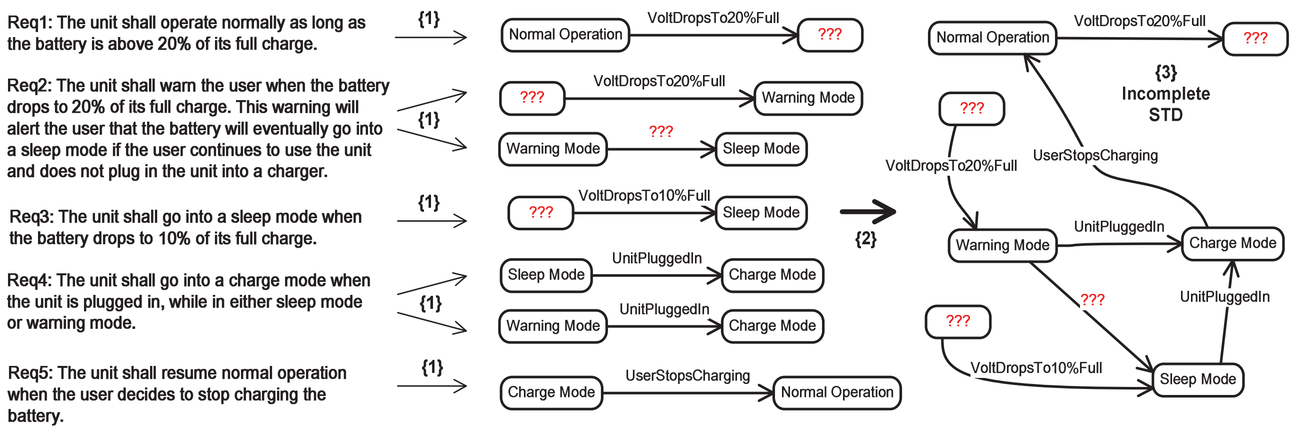

Req1: The unit shall operate normally as long as the battery is above 20% of its full charge.
Req2: The unit shall warn the user when the battery drops to 20% of its full charge. This warning will alert the user that the battery will eventually go into a sleep mode if the user continues to use the unit and does not plug in the unit into a charger.
Req3: The unit shall go into a sleep mode when the battery drops to 10% of its full charge.
Req4: The unit shall go into a charge mode when the unit is plugged in, while in either sleep mode or warning mode.
Req5: The unit shall resume normal operation when the user decides to stop charging the battery.
{1}
{2}
{3} Incomplete STD
Normal Operation
Warning Mode
Sleep Mode
Charge Mode
???
VoltDropsTo20%Full
VoltDropsTo10%Full
UnitPluggedIn
UserStopsCharging

Figure 2. Translation method reveals that the five battery requirements are ambiguous and incomplete because they produce an incomplete STD.

(1) What is currently happening?: This question identifies the current state (***Sc***).
(2) What will happen next?: This question identifies next state (***Sn***).
(3) What causes the next state to happen?: This question identifies the transition (***T***).

Asking these three questions identifies explicit/implied values for {***Sc***, ***T***, ***Sn***}, resulting in a STD-BB (this transformation from NL to STD-BB is denoted as step {1} of Figure. 2). Figure. 2 also displays a STD (denoted as step {3}) that is gradually being constructed piece-wise as each requirement is transformed into a STD-BB step {2}).

However, the STD is incomplete, reflecting the ambiguity and incompleteness present in the five NL requirements. For example, Req1 does not explicitly state ***Sn***, resulting in a "???" in place of ***Sn*** in its STD-BB. This in turn results in the state on the upper right side of the STD to be disconnected from the rest of the STD.

The ambiguities and incompleteness may not be obvious in NL requirements, but they have now been made obvious in a STD. Requirements engineers and stakeholders can see what the STD lacks, and together they can work towards its completion. To produce the complete STD, requirements engineers and stakeholders would define the implied (???) elements, add requirements, remove requirements, or do what it takes to complete the diagram.

In summary, in our NLtoSTD method, the formal representation (i.e., STD) exposes and subsequently corrects the ambiguities and incompleteness in the informal representation (i.e., NL). The result is a complete and well defined set of NL requirements, and a corresponding STD that can be used for automated verification. A key to our method is the manual (versus automatic) translation of the natural language into a STD, which achieves two important goals. First of all, the manual translation adds an extra level of user inspection to the process. Secondly, the automatic translation processes that we have seen, does not result in the same bi-directional traceability between the model and NL that our manual translation produces. This traceability is important to our method's ability to correct requirements.

Since this is an initial feasibility investigation, the scope of this experiment is limited to only evaluating the NL to STD-BB transformation process (the center part of Figure 2). This experiment did not include the construction of an STD from the building blocks from the transformation process (the rightmost side of Figure 2).

III. Empirical Study

The major goal of this study is to evaluate the usefulness of the NLtoSTD transformation process as a defect detection method as it compares to fault checklist inspection. This experiment is a repeated-measure design [28] in which each team of three or four participating students developed a requirement document for a different system. Next, each subject evaluated two different set of requirement documents (both of whom were developed by other students). To evaluate the first document, the subjects used the fault checklist inspection method and then used NLtoSTD to inspect the second document. These inspections resulted in a list of faults for each subject using both methods. The details of the study are provided in the following subsections.

A. Research Questions

Three research questions were investigated in this study.
RQ1: Is NLtoSTD more *effective* (i.e., the number of faults) at detecting incompleteness and ambiguities in requirement specifications as compared to the fault checklist inspection?
RQ2: Is NLtoSTD more *efficient* (i.e., faults per hour) at detecting faults in the requirement documents as compared to the fault checklist inspection?
RQ3: Is NLtoSTD viewed useful for improving the software quality?

B. Variable and Measuress

We manipulated one independent variable.
***Inspection Method**:* Each Subject used the fault checkist method and the NLtoSTD method to inspect two different sets of requirement documents.
We also measured the following dependent variables.
Effectiveness: the number of faults found by each subject.

978-1-4577-1076-6/11/$26.00 ©2011 IEEE

Efficiency: the number of faults found by each subject per hour.

C. *Participating Subjects*

Sixteen Computer Science graduate students enrolled in *Requirement Definition and Analysis* course at North Dakota State University participated in this study.

TABLE I. REQUIREMENTS DOCUMENTS USED IN THE EXPERIMENT

Doc	Subjects	System Description	Number of Pages
A	4	A web-based tool for managing student elections	42
B	3	A campus event calendar	21
C	3	A help desk management system	28
D	4	An intelligent rating system for electronic entertainment media	17
E	2	An event registration system	33

D. *Artifacts*

There were two phases to this study (development and inspection). First, during the development phase, each team of three to four participants developed a requirement document for a different system. Table I provides a list of these systems. One subject dropped the course leaving two subjects that developed Document E. Second, for the inspection phase, each subject was assigned two different requirement documents (out of five documents listed Table I) to inspect using the fault checklist and NLtoSTD methods.

E. *Experiment Procedure*

The experiment design includes several steps. Figure 3 shows the details of the experiment steps. The details are provided in the following subsections:

1) Phase I – Developing Requirement Documents: Sixteen subjects working in five different teams developed the requirements documents for their identified system as shown in Table I. This phase resulted in five different requirement documents (namely A, B, C, D, and E).

2) Phase II – Inspecting Requirement Documents: This phase involved using the fault checklist and the NLtoSTD methods to inspect the requirement documents. This phase included the following steps:

a) Training 1– Fault Checklist Method: During this 30 minute training session, the subjects were given description of the fault checklist and a list of the fault classes. Subjects were instructed on how to use the fault checklist to locate faults present in the requirement documents and how to record faults using fault form. The fault checklist used in this experiment has been used in empirical studies for comparing defect detection methods for inspections [2].

b) Step 1 - Inspecting Requirement Documents Using Fault Cheklist Method: Using the information from Training 1, each subject was randomly assigned a requirement document (with the constraint that the document was developed by other subjects in class) to inspect it using a fault checklist. This step resulted in a list of 16 individual *Fault Lists* (one per subject).

c) Training 2 - NLtoSTD Method: During this 30 minute training session, the subjects learned about the NLtoSTD method. The subjects were first trained on how to map the natural language (NL) requirements to STD building blocks (STD-BB). Next, the subjects were taught how to document the buidling block elements (i.e., the precondition(s), transition, and postcondition(s)) using an example system using an excel spreadsheet. Next, the participants were taught how to record the "*Ambiguities*" and "*Incompleteness*" in the requirements found during the inspection (using NLtoSTD).

d) Step 2 - Inspecting Requirement Documents Using NLtoSTD Method: Each subject was randomly assigned a requirement document to inspect that was different from the document inspected by that subject in Step 1 and with an additional constraint that the document was developed by other subjects. Using the knowledge from Training 2, each subject inspected the assigned requirements document to detect faults using the NLtoSTD method. The output of this

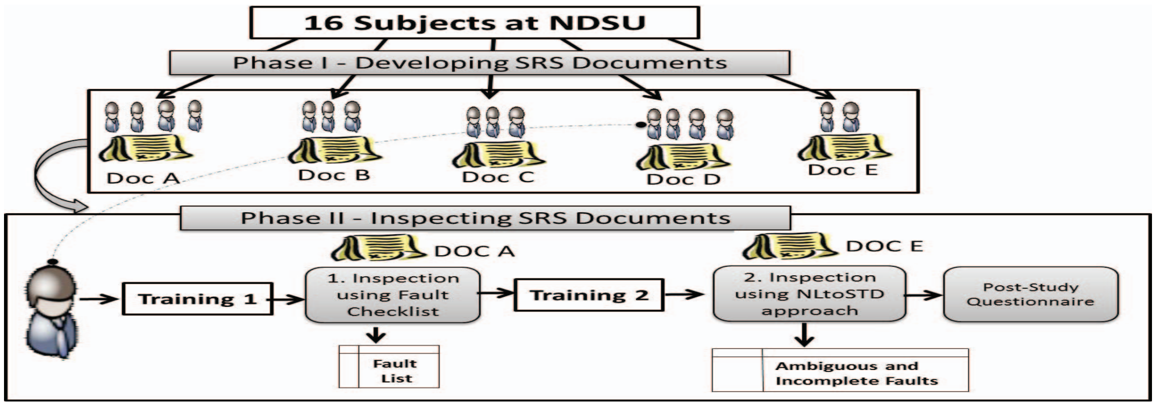

16 Subjects at NDSU
Phase I - Developing SRS Documents
Doc A
Doc B
Doc C
Doc D
Doc E
Phase II - Inspecting SRS Documents
DOC A
DOC E
Training 1
1. Inspection using Fault Checklist
Training 2
2. Inspection using NLtoSTD approach
Post-Study Questionnaire
Fault List
Ambiguous and Incomplete Faults

Figure 3. Experiment Procedure: Experiment Steps, Training Steps and Output Produced

TABLE II. ASSIGNMENT OF DOCUMENTS FOR INSPECTION

	Doc A	DocB	Doc C	Doc D	Doc E
Fault Checklist	4	3	3	3	3
NLtoSTD	4	3	3	3	3

step was 16 individual *Fault Lists* (one per participant). This scenario is illustrated in Figure 3, where a subject involved in the development of Document D was assigned Document A to inspect it using fault checklist during Step 1, and was then assigned Document E during Step 2 to inspect it using the NLtoSTD method. The result of these assignments in terms of the number of subjects that inspected each document using both methods is shown in Table II.

e) Step 3 - Post-Study Questionnaire: The subjects were provided an opportunity to provide feedback about the fault checklist and NLtoSTD inspection methods to help researchers better understand the results.

F. Data Collection

This section provides a brief description of data collected during the experiment run. The data included the faults found by subjects using the fault checklist and NLtoSTD methods. The fault checklist technique helps reviewers to focus on different fault types, namely, *Missing Functionality (MF)*, *Missing Environment (ME)*, *Ambiguous Information (AI)*, *Inconsistent Information (II)*, *Incorrect Fact (IF)*, *Extraneous (E)*, *and Miscellaneous (M*); whereas the faults focused during the NLtoSTD inspection were "*Ambiguous Information* (AI)" and "*Missing functionalities* (MF)".

The fault lists required to students to self-classify the faults found during each inspection into these fault classes. The fault lists also required the students to indicate the time they had found each fault (along with the breaks they took) during the inspection process. The subjects also rated the fault checklist and the NLtoSTD method on different characteristics and answered other survey questions that were based on a 5 point Likert-scale.

While evaluating the fault data from the individual fault lists for data analysis, the fault data recorded by students inspecting Documents D and E using the NLtoSTD method was excluded from the analysis due to following reasons:

- In Document D, the requirements were written in a high level of abstraction, thus it was harder for the subjects to find explicitly stated values for ***Sc***, ***T***, and ***Sn***. In this situation the subjects should have given "???" for the vast majority of the values. However, all three subjects entered values based on their assumptions of what those values should be, and the majority of their assumptions were not correct. This resulted in data that did not represent what the requirements actually express.
- Document E was written using a use case format, with each use-case's, multi-step, sequence described in its entirety. The format of the document made it hard for subjects to differentiate between the states and transitions, making the translation process hard to understand. This resulted in data that is unreliable.

Therefore, Section IV provides the analysis of the fault data from the documents A, B, and C (as shown in Table I).

IV. DATA ANALYSIS AND RESULTS

This section provides an analysis of the fault data. This section is organized around the research questions presented in Section III.A. An alpha value of 0.05 was used for all statistical tests.

A. Fault Detection Effectiveness (RQ1)

This section compares the number of "*Missing Functionality (MF)*" and "*Ambiguous Information (AI)*" faults found by subjects during the fault checklist inspection (Step 1) and the NLtoSTD inspection (Step 2) for each document.

Because each document was inspected individually by three or four subjects (depending on the document as shown in Table II), the individual data from the Step 1 and Step 2 inspections was combined into a team score. The fault detection effectiveness of a team during fault checklist inspection (Step 1) was calculated by combining the list of faults each subject found in a particular document. Similarly, the fault detection effectiveness of a team during the NLtoSTD inspection (Step 2) combined the individual scores for each document. This analysis was performed separately for Documents A, B, and C. The reason for this analysis is because we were only interested in the investigating the coverage of the fault space by individual team member's knowledge (as opposed to the list of unique faults found by the team) when using our NLtoSTD method to detect the incompleteness and ambiguities in natural language requirement specifications.

Figure 4 compares the total number of MF and AI faults found by an inspection team using the fault checklist and NLtoSTD methods for each document.

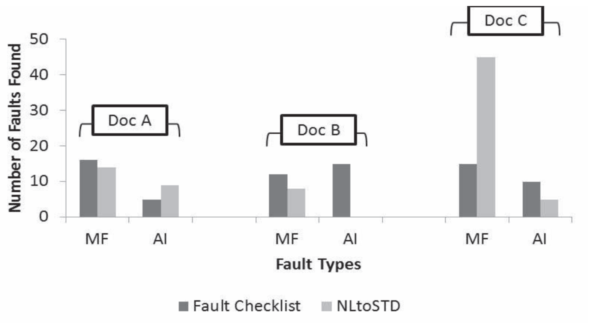
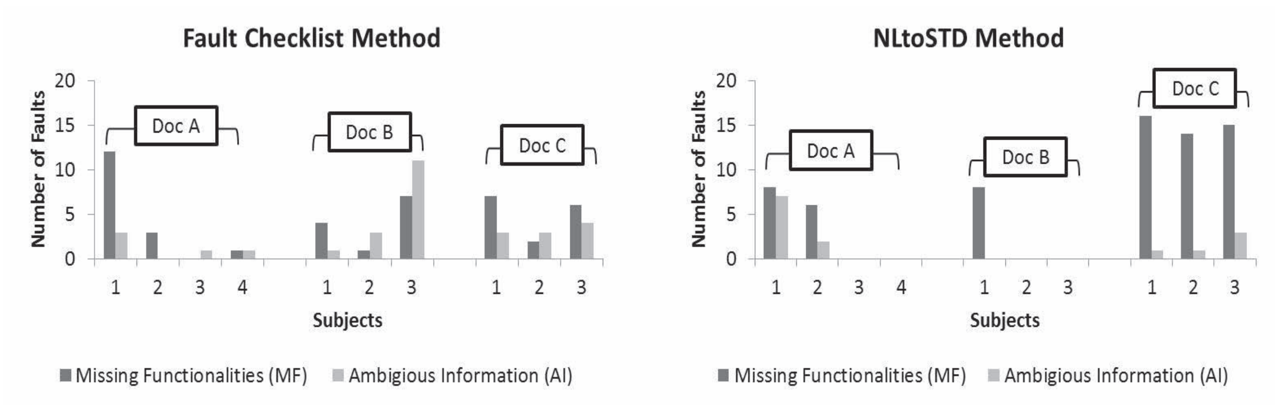


Figure 4. Comparison of the Number of Faults found by Subjects Using Fault Checklist and NLtoSTD method

Some observations from Figure 4 are as follows:

- For Document C, the team using NLtoSTD was visibly more effective at detecting the incomplete-ness faults (MF) than the fault checklist method, finding an average of 15 faults compared to an average of 5 faults (per document, the average number of faults was calculated by dividing the total number of faults by the number of subjects who inspected the document). On the contrary, for Documents A and B, teams using the fault checklist method found more MF type faults (an average of 4 faults for both documents) as compared to teams using the NLtoSTD method (an average of 3.5 faults for Doc A and 2.5 faults for Doc B).

- Regarding the AI faults, NLtoSTD exposed larger number of AI faults when applied on Document A, but the fault checklist method was more effective at finding the ambiguities in Documents B and C.
- A surprising result was that the team using NLtoSTD to inspect Document B, did not even find a single AI fault. While we expected that the subjects would find more MF faults than the AI faults due to the nature of the experiment design (i.e., the subjects were not asked to construct a requirement model using the STD-BBs which can highlight hidden ambiguities in individual requirements), this result was unexpected.



To further investigate the research results, the performance of individual subjects was analyzed to determine if the result was consistent throughout the sample. Figure 5 shows the number of MF and AI faults found by each subject using both inspection methods for all three documents. Some interesting observations are as follows:

- As the figure shows, the fault data seems to be more evenly distributed for the fault checklist method. While each subject using the fault checklist method found some faults, four subjects (two subjects each in Documents A and B) did not find any fault using NLtoSTD.
- Further, we examined the data for these subjects (i.e., subjects 3 and 4 inspecting Document A, and subjects 2 and 3 inspecting Document B) that found no faults using NLtoSTD. Upon examining the data, we found that these subjects did not have a clear understanding on how to apply NLtoSTD which in turn, resulted in choosing incorrect values for ***Sc***, ***T***, and ***Sn*** when transforming NL to STD-BBs.
- As expected, the subjects using NLtoSTD consistently found larger number of MF faults as compared to the AI faults. This was especially true in the case of Document C.

These results showed that the subjects, who had a clear understanding on how to apply NLtoSTD, were effective at finding the incompleteness in NL requirements.

B. Fault Detection Efficiency (RQ2)

The efficiency values (i.e., the number of faults found per hour) for each inspector using the fault checklist and the NLtoSTD methods were computed. For the fault checklist inspection, *faults* include all type of faults (e.g., MF, ME, IF, AI, II, E, M) found by the subjects. For NLtoSTD, *faults* include the MF and AI faults.

The result in Figure 6 shows that NLtoSTD did not improved participants efficiency in any of the documents except for two subjects in Document C. This was not surprising considering the fact that effort spent during the fault checklist inspection led to a large number of other types of faults (in addition to MF and AI fault types). In particular, four subjects inspecting the Documents A and B (two subjects each in Documents A and B) reported all false faults and had zero efficiency.

Based on these results, we see rooms for improvement in the application of NLtoSTD by making it easier to find ***Sc***, ***T***, and ***Sn*** values correctly.

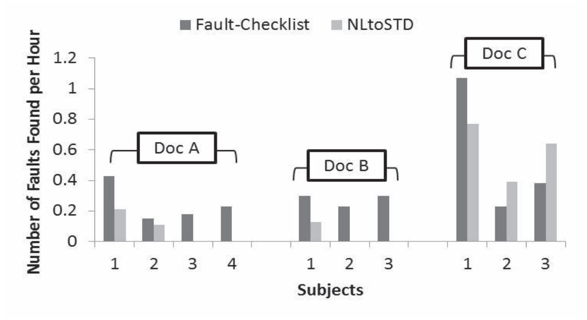


Figure 6. Comparison of the Efficiency Values of Subjects Using Fault Checklist and NLtoSTD method

C. Usefulness of the NLtoSTD Method (RQ3)

NLtoSTD was evaluated using feedback from the subjects on the following eight attributes: simplicity, understandability, comprehensiveness, intuitiveness, ease of classifying faults, usability, uniformity across products, and adequacy of faults found.. Each subject rated the attributes on a 5-point Likert scale (1-very low, 2-low, 3-medium, 4-

high, or 5-very high). A non-parametric binomial test was conducted to determine whether the mean response was significantly greater than medium (the midpoint of the 5-point scale). The result was only significant (p= 0.021) for the *simplicity*, and *easy to understand* attributes. All other attributes (except the "*adequacy of faults found*" and "*ease of classifying faults*"), were rated positively and the mean rating was greater than 3 (i.e., the mid-point of scale).

We especially wanted to analyze the feedback from the subjects who did not clearly understand how to apply NLtoSTD and reported incorrect values of ***Sc***, ***T***, and ***Sn*** (as discussed in Section IV.A). Since the survey questions were filled out anonymously, we could not match the responses to the survey questions with their fault data. However, the comments and responses to survey questions helped us improve the process of translating the NL into STD-BBs and are discussed in Section VI.

V. THREATS TO VALIDITY

In this study, there were some threats to validity that were addressed. First, the artifacts inspected in this study contained naturally occurring defects that were inserted while developing the artifacts rather than artificially seeded. Secondly, to reduce the threat due to the learning and maturation effects, the subjects performed second inspection using NLtoSTD (that was a new method) on a document that they were reading for the first time.

However, there were some validity threats that were not addressed. The artifacts in this study were developed by student teams and it may not be representative of industrial strength document. Also, the nature of faults made by students can differ from the faults made by professionals. An important internal validity threat was due to the lack of a control group: i.e.,we cannot determine the portion of faults found during the second inspection were due to the use of NLtoSTD and the portion that were because the subjects were became experienced at inspecting the requirements document. We plan to address this threat in future.

VI. DISCUSSION OF RESULTS

This section discusses the results and their implication in light of the original research questions. The findings and the lessons learned from this study are then discussed.

A. Discussion of Major Findings

Effectiveness: For team effectiveness, the results show that NLtoSTD is beneficial at finding the incompleteness in the requirements when the subjects clearly understand the process of translating the NL to STD-BBs. This is evident from the inspection results for Document C where the number of MF faults detected during the NLtoSTD inspection is threefold that detected during the fault checklist inspection. However, this is an extremely small data set to make any definite conclusions.

Also, we find that the characteristics of the NL requirement specifications affected the effectiveness results. For example, the requirements in Document C are written in a short concise manner, with one functionality per requirement (i..e., the requirements exhibited high cohesiveness). NLtoSTD favors high cohesiveness because its goal is to derive one building block per requirement. By contrast, in Document B (which exhibited the worse performance), each requirement describes a sequence of actions pertaining to one use case. The subjects reported that the sequence of actions creates multiple candidates for ***Sc***, ***T***, and ***Sn***, making it difficult to obtain a building block.

Efficiency: The results clearly show that the fault checklist method is more efficient at finding faults. This is mainly because: 1) the fault checklist looks for ten types of faults whereas NLtoSTD focuses on detecting only two types of faults (MF and AI); and 2) four (out of 11) subjects using NLtoSTD found no true faults due to their misunderstanding of the translation process.

The results from this initial investigation have provided us with insights to improve the translation process that can in turn improve the reviewer's performance. Also, we want to extend our research method to help detect other important types of faults (e.g., inconsistencies, and incorrectness).

Student's Feedback on the Implementation of NLtoSTD: The subjects' responses to post-study survey show that, in general, NLtoSTD is viewed favorably for most of the attributes. However, some subjects reported the problems that they faced while choosing the values for ***Sc***, ***T***, and ***Sn*** when translating NL into STD-BBs. Based on the students' responses and feedback, we plan to revise NLtoSTD method to make it easier to understand and apply on NL. The potential improvements are discussed below.

B. Lessons Learned

From the results of this study, we learned that NLtoSTD could provide benefits to developers who use it correctly. However, document format and variations in subject performance have prompted us to reevaluate the way that the three elements ***Sc***, ***T***, and ***Sn*** are determined. Figure 7 shows an example of the potential revision of the STD-BB, using Req2 of the battery charger example used in Section II (Figure 2). In the revised STD-BB, we will explicitly add an entity to a state to represent ***Sc*** and ***Sn*** as follows: ***entity (state)***. Allowing for multiple entities should alleviate the problem encountered with the requirements that are not ideally written in an atomic manner. Figure 7 shows an example of a non-atomic requirement: essentially it describes three requirements. Based on our new format, we identify three entities: unit, battery, and user. Each entity has its own current and next states. For example, the battery entity has "20%Level" as a current state and "sleepMode"

Req2: The unit shall warn the user when the battery drops to 20% of its full charge. This warning will alert the user that the battery will eventually go into a sleep mode if the user continues to use the unit and does not plug in the unit into a charger.

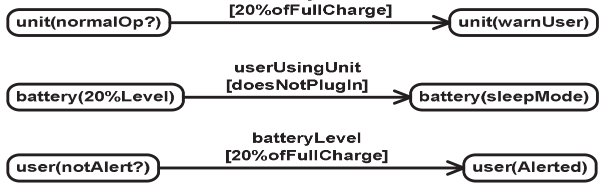


Figure 7. The revised STD-BB

as a next state.

Another potential change in the NLtoSTD method is allowing users to make an assumption. As shown in Figure 7, "unit (normalOp?)" has a question mark. This indicates that the "normOp" state is not explicitly stated, but the user can assume that it is the intended state and label it with a question mark for future follow up. This should improve the method's ability to expose ambiguities, since an assumption on the user's part means that something was left up to the user's interpretation, and needs to be clarified. We also consider allowing users to add conditions when they describe the transition (***T***). This alleviates the problem that arises when a requirement seems to state more than one transition.

Finally, the original version of NLtoSTD strictly handled functional requirements. The proposed improvements has the potential to model sub-problems, thus, handling non-functional requirements is a possibility in the future.

VII. Conclusion And Future Work

Our empirical study revealed two important things. First it confirmed NLtoSTD's potential to expose incompleteness and ambiguous. Second, it revealed areas of improvement that benefits the method's applicability in future.

Our future work includes further experimentation using the improved method suggested by the results from this experiment, as well as applying the improved method to various types of requirements documents and applications. A major motivation in developing this method is to increase the participation of the non-technical stakeholders in NL to STD translation and gradually reducing the gap between the informal and formal requirements space by making the method more user-friendly to those stakeholders. We also plan to implement a tool that can automatically determine which NL requirements need to be addressed, and in what manner, by assessing the STD's various properties in its graph representation.

Acknowledgment

This work was supported in part by NSF under Awards CNS-0855106 and CCF-1050343 to North Dakota State University.

References

[1] D. Berry, Ambiguity in natural language requirements documents, LNCS, 2008.

[2] Porter, A.A., Votta, L.G., Basili, V.R., Comparing detection methods for software requirements inspections: A replicated experiment, TSE, 21(6): 563-575, 1995.

[3] Boehm, B., and Basili, V.R. Software defect reduction top 10 list, IEEE Computer, 34 (1):135-137, 2001.

[4] B. Nuseibeh and S. Easterbrook, Requirements engineering: A roadmap, ICSE, p.p35-41, 2000.

[5] D. Gause, User DRIVEN design - The luxury that has become a necessity, ICRE, Tutorial T7, 2000.

[6] C. Damas, B. Lambeau, and A. Lamsweerde, Scenarios, goals, and state machines: A win-win partnership for model synthesis, FSE , pp. 197-207, 2006.

[7] E. Letier. J. Kramer, J. Magee, and S. Uchitel, Monitoring and control in scenario-based requirements Analysis, ICSE, pp. 382 - 391, 2005.

[8] R. Alur, K. Etessami, and M. Yannakakis, Inference of message sequence charts, ICSE, pp. 304-313, 2000.

[9] D. Deeptimahanti and M. Babar, An automated tool for generationg UML models from natural language requirements, ASE, pp. 680-682, 2009.

[10] L. Kof, Scenarios: Identifying missing objects and actions by means of computational linguistics, RE, pp. 121-130, 2007.

[11] D. Aceituna, H. Do, and S. Lee, A human interactive approach to building requirements models, ISSRE, fast abstract, 2010.

[12] D. Popescu, S. Rugaber, N. Medvidovic, and D. Berry, Reducing ambiguities in requirements specifications via automatically created object-oriented models, Monterey Workshop, pp. 103-124, 2007.

[13] L. Kof, R. Gacitua, M. Rouncefield, and P. Sawyer, Ontology and model alignment as a means for requirements validation, ICSC, pp. 46-51,2010.

[14] L. Kof, From Requirements documents to system models: A tool for interactive semi-automatic translation, RE, pp. 391 – 392, 2010.

[15] C. Damas, B. Lambeau, and A. Lamsweerde, Scenarios, goals, and state machines: A win-winpartnership for model synthesis, FSE, pp. 197-207, 2006.

[16] A. Sutcliffe and M. Ryan, Experience with SCRAM, a scenario requirements analysis method, RE, pp. 164-171, 1998.

[17] C. Damas, B. Lambeau, P. Dupont, and A. Lamsweerde, Generating annotated behavior models from end-user scenarios," TSE, 31(12):1056-1073, 2005.

[18] A. Sutcliffe, N. Maiden, S. Minocha, and D. Manuel, Supporting scenario-based requirements engineering, TSE, 24(12): 1072-1088, 1998.

[19] B. Brykczynski, A survey of software inspection checklists, ACM SE Notes, 24(1):82,1999.

[20] M. Fagan. Advances in software inspections. TSE., 12(7):744–751, 1986.

[21] D. Berry and E. Kamsties. Perspectives on Software Requirements, Kluwer Academic Publishers, 2004.

[22] F. Chantree, B. Nuseibeh, A. de Roeck, and A. Willis. Identifying nocuous ambiguities in natural language requirements. RE, pp 59–68, 2006.

[23] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications, TOSEM, 5(3):231-261, 1996.

[24] S. Sakthivel. Survey of requirements verification techniques. Journal of Information Technology, pp. 68-79, 1991.

[25] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, Orthogonal defect classification - A concept for in-process measurements. TSE, 18(11): 943-956, 1992.

[26] J. Chaar, M. Halliday, I. Bhandari, and R. Chillarege, In-process evaluation for software inspection and test, TSE, 19(11): 1055-1070, 1993.

[27] M. Lezak, D. Perry, and D. Stoll, A case study in root cause defect analysis, ICSE, pp. 428-437, 2000.

[28] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering An Introduction, Kluwer Academic Publishers, 2000.