# INVESTIGATING THE USE OF MODEL-BASED METHOD FOR IMPROVING THE QUALITY OF NATURAL LANGUAGE REQUIREMENTS: A CONTROLLED EMPIRICAL STUDY

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Munmun Gupta

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

February 2014

Fargo, North Dakota

**North Dakota State University**
**Graduate School**

**Title**

Investigating the Use of Model-Based Method for Improving the Quality of Natural Language Requirements: A Controlled Empirical Study

**By**
Munmun Gupta

The Supervisory Committee certifies that this *disquisition* complies with North Dakota State University's regulations and meets the accepted standards for the degree of

**MASTER OF SCIENCE**

SUPERVISORY COMMITTEE:

Dr. Gursimran Walia
Chair

Dr. Brian Slator

Dr. Limin Zhang

Approved by Department Chair:

| 02/21/14 | Dr. Brian M. Slator |
|---|---|
| Date | Signature |

## ABSTRACT

Requirement engineering is a critical phase in software development that describes the customer needs and specifications for the software. Requirements are gathered through various sources and documented for a software product to be developed, written in Natural Language (NL). NL requirements are fault prone because they can be interpreted in different ways due its inherent imprecision, ambiguity, and vagueness. To address these problems, model-based requirements verification method called NLtoSTD (State Transition Diagram) is proposed. This paper evaluates the ability of NLtoSTD method in detecting faults when used on NL requirements and to improve its cognitive friendliness to the stakeholders. Motivated by the earlier study, we revised our proposed method and performed an empirical study. The participants employed the NLtoSTD method to inspect documents to identify ambiguities, incompleteness and inconsistencies. The experiment result shows an improvement over the previous results that the NLtoSTD is a method for verification of NL requirements.

## ACKNOWLEDGEMENTS

I'm grateful to my adviser Dr. Gursimran Walia for his continuous help, support, patience and guidance in the development and completion of this paper. He has been a great adviser who has motivated me at every step and guided me through my master's project. He was always available to answer my questions with detailed guidance instead of just replying with facts, and helped me refine the topic for my paper by patiently taking out significant amount of time over my semesters in the Master's program for discussion's.

I'm grateful to Dr. Hyunsook Do for help and guidance throughout my master's project and contributing to my publication.

I'm grateful to Dr. Brian Slator for taking out the time to be a part of my supervisory committee despite my not having had any courses or seminars with him in my Master's or Bachelor's program in the Computer Science department.

I'm grateful to Dr. Limin Zhang for taking time to be a part of my supervisory committee.

I especially thank, Dr. Kendall Nygard for always being helpful with various stages in the progress of my Bachelor's and Master's program and providing useful guidance in program related issues from first day until now.

I'm grateful to the Computer Science department faculty and staff in all the ways I could use their help in the progress and completion of my Master's program.

Finally, I'm grateful to my parents for their constant and unrelenting support towards my education and for their unconditional love for me. I would also like to thank my one special friend for being supportive all the way.

.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

Figure Page

## LIST OF APPENDIX TABLES

## LIST OF APPENDIX FIGURES

# CHAPTER 1. INTRODUCTION

In software development, all the activities throughout the life cycle of the software should be performed diligently to produce high quality software product. The first phase of the software life cycle, is to gather requirements through various sources with the help of different requirement elicitation methods like interviews, use case analysis, questionnaires, prototyping, etc. Software requirement analysis and determination is a tedious process that involves both technical (e.g., requirement engineers, programmers) and non-technical (e.g., end users, sponsors) stakeholders. During this phase, requirements are gathered and documented in the form of Natural Language from all the stakeholders that describe the customer needs and specifications. The output from this phase is a Software Requirement Specification (SRS) that includes requirements for a software product to be developed, and is written in NL which is usually prone to imprecision, ambiguity, and vagueness. It is important to detect and prevent faults, misinterpretation of requirements, etc. as early as possible, because finding and fixing a software problem in the later stage of the lifecycle (i.e., testing or after delivery) is 100 percent more expensive than during the requirement and design phase [1-3].

To ensure high-quality NL requirements, numerous fault-based verification approaches have been developed such as Walkthrough, Fault Checklist, Inspection, and, Consistency Checking [3, 8-11]. These methods have been validated for fault-detection effectiveness through experiments and case studies in classroom and industrial settings [3, 6].

Among various requirements verification approaches, software inspection has been empirically validated [3, 10] to find the largest number of faults during the inspection of software artifacts. However, even when faithfully applying software inspections, it is estimated that 40-50% of software development effort is still spent on fixing problems that should have been fixed

early in the lifecycle [1, 2]. Despite its success, the research has shown that the effectiveness of an inspection process is highly dependent on the inspectors' ability to understand the requirements in the same way as the requirement developers had intended. However, due to the inherent nature of the English language, inspectors can have different interpretations of the same requirements without noticing the ambiguities and inconsistencies. An alternative to the software inspection is the Model-based approach can detect faults easily because, when the requirements are modeled or checked with formal methods, the properties, such as ambiguities and inconsistencies, are clearly addressed [6]. To address this, the software engineering researchers use the model-based approach to verify NL specifications. For example, Kof [14] presented a method that analyzes NL requirements and generates Message Sequence Charts with computational linguistics. Similarly, Sutcliffe et al. [15, 16] presented a method that converts use cases into scenarios and validated those using rule-based frames that detect incomplete/incorrect event patterns.

Even though the model-based approach provides a systematic way to identify inconsistent and incomplete requirements, it requires the translation of NL requirements. This can be subjective since translation of requirements can have different interpretations from different stakeholders which can result in a wrong model, i.e. incompleteness and ambiguities of NL [12, 17] and, thus, can eventually produce software that stakeholders do not want or may not fulfill the requirements of stakeholders. To improve the problem with the erroneous translation, the researchers have proposed different modeling techniques by using automating NL translation approach [4, 7, 11, and 13] which helps to reduce human errors and thereby improve the NL translation process. These methods include approaches based on translating goals to state machines [4], and scenarios to state machines [13]. Automation can certainly reduce human errors and improve the translation process, but complete and error-free automation of this process is not

possible because, often, NL requirements can be interpreted in multiple ways; therefore, human judgment can inevitably lead to various correct and sensible interpretations.

To address these problems, Aceituna et al., proposed a method that translates Natural Language requirements into State Transition Diagram (STD). This is accomplished by translating each NLtoSTD in an incremental manner which would help in exposing ambiguities, incompleteness and inconsistencies of requirements documented in NL. This transition is carried out in a two-step process, where in the first step each NL requirement is translated into a STD building block thereby inspecting the requirements document and documenting the faults found during the translation on the spreadsheet. During the second step, the STD building blocks are used to create complete STD using an automated software tool which will be explained in Chapter 4. The stakeholders involved during the translation process can detect faults during each step (NLtoSTD-BB and STD-BBtoSTD) and direct mapping from NLtoSTD model is preserved in the translation and any changes to the STD model can be made directly to the requirements and vice-versa.

To investigate the feasibility and applicability of NLtoSTD method and the results from those two studies [5, 6] has validated the use of translating NL requirements into STD segments (i.e., NLtoSTD-BB) as an effective verification method [5, 6]. These two experiments were done to validate the process used during translation NLtoSTD-BB method before we could evaluate the STD-BBtoSTD method. The results from study1, i.e. NLtoSTD-BB method (V1.0) [5], showed that the subjects who correctly translated NLtoSTD building blocks were able to find significantly larger number of "incompleteness" in the NL requirements when compared to fault-checklist method. Based on the results from the previous experiments, the NLtoSTD-BB method (V1.0) [5] was improved to be able to more easily extract the elements of STD-BBs from NL

requirements which further aided us in exposing incomplete and ambiguous requirements and in some cases, more ambiguous information than the fault-checklist method. The results from the second study NLtoSTD-BB (V2.0) [6] demonstrated the effectiveness of NLtoSTD-BB V2.0 [6] in exposing the missing functionality and in some cases, more ambiguous information than the fault-checklist method. The major improvement is seen in the fault-detection effectiveness and efficiency of the NLtoSTD-BB method. So, these results from the second study helped further refine the NLtoSTD-BB method (V3.0) used in the current study.

Motivated by the results above, this research performs: a) a replication of study 1 to evaluate the fault-detection ability of NLtoSTD-BB and b) extend the research by evaluating the fault-detection ability of the STD-BBtoSTD method. The complete NLtoSTD method ((i.e., NLtoSTD-BB + STD-BBtoSTD) is evaluated for its ability as an effective requirements verification method in this paper.

# CHAPTER 2. BACKGROUND AND RELATED WORK

The motivation of this study comes from the results of an earlier two experiments that evaluated the NLtoSTD-BB translation step but did not evaluate the STD-BBtoSTD step. The design of the earlier studies, the results that were reported and the validity threat that were not addressed (in earlier studies) motivated this investigation. This section describes the innovative NLtoSTD (V1.0) [5] and the revised NLtoSTD-BB (V2.0) [6] methods and the tool used to perform the STD-BB to STD creation step (that was used the first time in the current study).

## 2.1. NLtoSTD: Basic Concepts

The main idea behind the proposed method NLtoSTD is to translate the NL requirements into an STD, so that the ambiguous, inconsistent, and incomplete requirements can be identified very easily. An STD is a formal description of the system-to-be behavior, and it can be decomposed into building blocks (BBs) that make up the STD. Assume that a complete and unambiguous STD exists, that means each BB (is also complete and unambiguous). This indicates that complete and unambiguous STD-BBs represent a set of NL requirements (that were translated into BBs) that do not contain faults.

The NLtoSTD method comprises of two steps. The first step includes the translation of NL requirements into STD-BBs (i.e., NLtoSTD-BB) and the second step includes the building of an STD using the STD-BBs (i.e., STD-BB to STD). Figure 1 shows, a high-level overview of the NLtoSTD method (Step 1 and Step 2 are highlighted). The NLtoSTD method supports the discovery of the problems in the NL requirements by investigating the individual STD-BBs and the resulting STD. As soon as all the faults are identified; the requirements author and other stakeholders can revise the NL requirements and the STD model to correct those problems.

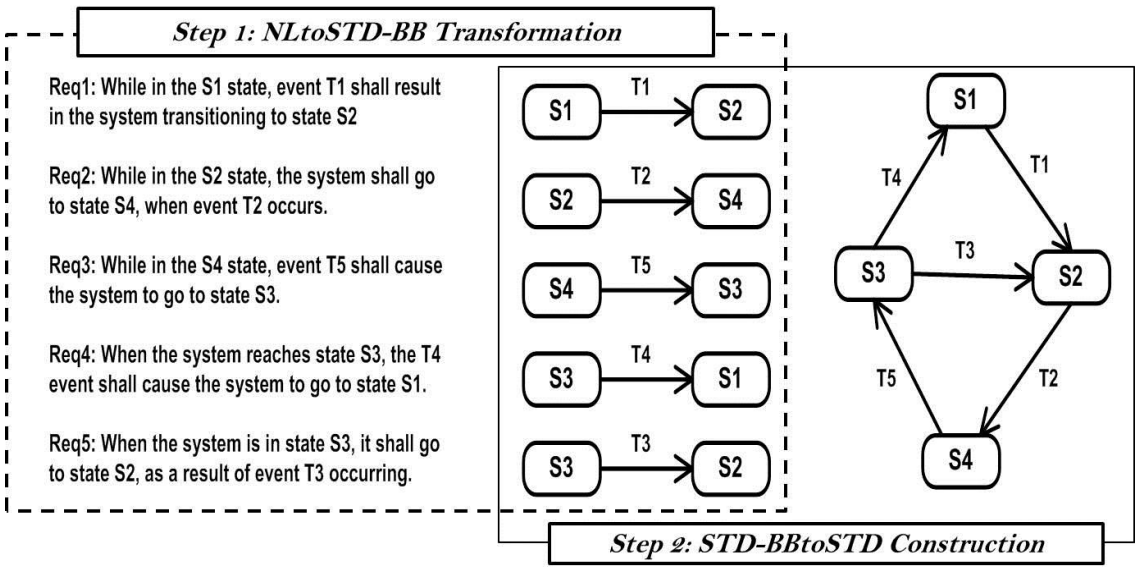



Figure 1. NLtoSTD Method Overview

Figure 1 highlights the "*NLtoSTD-BB*" translation (left side box with dotted line) and the "*STD-BB to STD*" construction (right side box with solid line).

As shown in middle part of Figure 1, the three elements that make up an STD-BB derived from a NL requirement includes: (1) current state (*Sc*), (2) next state (*Sn*), and (3) transition (*T*). The selection of these elements was based upon the characteristics of an ideal requirement and an inspection scheme that looks for those characteristics. Looking closely at the five requirements in Figure 1, the following characteristics can be readily observed.

1. Each requirement is stated so that it directly maps to an STD-BB.
2. Also, each requirement explicitly states its precondition in the form of the current state (*Sc*) and its post condition in the form of the next state (*Sn*).

However, typical NL requirements do not explicitly state current and next states, thus a requirement's preconditions and post conditions are often inferred and not explicitly stated. In the ideally stated requirement, preconditions and post conditions should be explicitly expressed to

minimize ambiguities and incompleteness. Similarly, the absence of the explicit transition (*T*) can cause difference in the interpretations of a requirement by different stakeholders. The NLtoSTD method forces the stakeholders to identify the aforementioned three elements so that they can detect the requirement faults while building the STD and ensures that the requirements are as consistent and concise as possible. We discuss more details on the NLtoSTD method and its revision history in the next subsections

### 2.2. NLtoSTD: Application

As mentioned in Section 2.1, the NLtoSTD method turns a set of NL requirements into an STD model by transforming each requirement into an individual STD-BB. The STD-BBs act as a formalized version of the NL requirements and can lead to the detection of faults for two reasons:

1. A formalized version of the NL requirements has only one specific interpretation, exposing ambiguities in the NL requirements, and
2. A formalized version exposes gaps in the requirements more readily, as compared to inspecting a set of NL requirements.

These STD-BBs can be used to inspect the STD model and its properties. Also, once an STD is obtained, it can be analyzed automatically to expose faults, by looking for inconsistent path traversals in the STD, whereas, exposing inconsistencies during the inspection involves reviewing all the requirements and looking for contradictory semantics.

The following subsections describe the original NLtoSTD-BB method (V1.0 used in Study 1) [5], followed by a description of Study 1 that help developed the first revised NLtoSTD-BB (V2.0 used in Study 2) [6]. Finally, we describe the latest NLtoSTD-BB (V3.0 used in Study 3 – the current study).

### 2.2.1. Original NLtoSTD-BB method [5]

The basis for the NLtoSTD-BB transformation is that a functional requirement should typically describe an entity transitioning from one state to another. For example, the 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 Car is described as transitioning (T) from moving (Sc) to stopping (Sn), which is then represented by an STD-BB.

In the above example requirement, the three elements are explicitly stated, yielding definable values for *Sc*, *T*, and *Sn*. In practice, however, requirements often 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: "*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 documented. 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 it requires an assumption. Undocumented assumptions can be erroneous and are often a cause of serious system failure (especially when the developers lack appropriate knowledge of the application domain). In this example, it is not clear whether we assume "moving forward," "moving backward," or both. It is important to document what may seem obvious, instead of allowing the possibility of an erroneous (and costly) assumption. Therefore, one goal of the NLtoSTD method is to expose undocumented assumptions.

To systematically identify the three elements for each of the requirements, the following three questions are used:

1. What is currently happening? – This question identifies the current state (*Sc*).

2. What will happen next? – This question identifies the next state (*Sn*).

3. What causes the next state to happen? – This question identifies the transition (*T*).

Asking these three questions identifies explicit or undocumented values for {*Sc, T, Sn*}, resulting in an STD-BB. While the ambiguities and incompleteness may not be obvious in the requirements, they are made obvious in an STD-BB. Requirement engineers and stakeholders can work towards its completion. To investigate the feasibility and applicability the NLtoSTD V 1.0, a controlled experiment was designed and conducted in order to evaluate the user's ability to translate NL requirements into the building blocks (or segments) of the STD. The experiment validated the underlying process used to translate NL requirements into STD segments before the STD can be constructed and analyzed.

The first study (Study 1), evaluated the NLtoSTD-BB V1.0 (as described in Section 2.1) using a repeated-measures experimental design. The experiment consisted of 16 graduate Computer Science and Software Engineering students enrolled in *Requirement Definition and Analysis* course in 2009 at North Dakota State University. The participating subjects were divided into teams of 3 or 4 people and were asked to develop a requirements document. There were 5 teams and each team developed a requirements document for a different system. Next, each subject inspected two different requirement documents (that they did not develop) using the fault checklist and using the NLtoSTD-BB translation method. During these inspections, each subject was first assigned a requirement document to inspect using a fault checklist followed by an inspection of a different requirements document using NLtoSTD-BB method. The data contained

the fault found by subject using the fault checklist and NLtoSTD-BB method for each requirements document.

The data was analyzed by comparing fault detection effectiveness i.e. number of "Missing Functionality (MF)" and "Ambiguous Information (AI)" faults found by subjects during the fault checklist inspection and the NLtoSTD-BB translation for each document. All analysis was done by combining the individual score from fault checklist and NLtoSTD-BB method into a team scores.

The result showed that NLtoSTD-BB is beneficial at finding the incompleteness in the requirements when the subjects assuredly understand the process of translating the NLtoSTD-BBs. The results (analysis of the subjective feedback from the participating subjects and quantitative data) from the first study provided the researchers with insights to help improve the translation process that can in turn improve the reviewer's performance as discussed in the following subsection.

### 2.2.2. NLtoSTD-BB V2.0 [6]: First revision

Based on the analysis of the fault record forms (that included the documentation of the NLtoSTD-BB translation process for each of the participating subject), the NLtoSTD-BB V1.0 method is limited in its effectiveness (as a verification method) when there are multiple requirements specified for a single functionality. In such case, the NLtoSTD V1.0 causes the users to select from multiple candidate values of STD-BB elements (i.e., Sc, T, Sn) causing an incorrect transformation of NL requirements. Because of the variations in user performance during Study 1 [5], the researchers were prompted to re-evaluate the way that the three elements (Sc, T, and Sn) were determined from the NL requirements. Also, it was questioned whether

using only three elements is sufficient to capture and model the description found in a typical NL requirement. The remainder of this section illustrates the revised (NLtoSTD-BB V2.0) method.

In the revised NLtoSTD-BB method, the following changes were made and are discussed along with their reasoning.

1) The first change the addition of an entity to a state to represent *Sc* and *Sn* as follows: *entity (state)*. Allowing for multiple entities should alleviate the problem encountered with requirements that are not ideally written in an atomic manner. It was also felt that separating the concepts of an entity and its given states, would make it easier to derive Sc and Sn, since the user could first decide which entity is being affected and then determine the entity states before and after the effect. This was done to allow the user to define Sc and Sn in a piecewise fashion. Based on the revised NLtoSTD-BB method, an inspector can identify three entities: unit, battery, and user. Each entity has its own current and next states.
2) The second change in the revised NLtoSTD-BB method is allowing users to make an assumption. This was done to improve the method's ability to expose ambiguities; an assumption on the user's part means that something was left up to the user's interpretation and needs to be clarified.
3) The third change is to allow 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. For example, if an *entity(state)* transition can occur only when transition T1 AND transition T2 occur, then the translator can now indicate it by labeling the NLtoSTD-BB with T1 ^ T2. The "^" represents the logical AND. Other allowable conditions are "v" for OR and "~" for NOT.

Based on these changes, the new NLtoSTD-BB method has the potential to model part of the system-to-be's operational context. Thus, handling nonfunctional requirements is a possibility Based on the above changes, the new NLtoSTD-BB method contains five elements (entity, entity's current state, entity's next state, transition, and condition for transition) instead of having three elements (current state, transition, and next state) as in the previous NLtoSTD-BB method. Also, the new NLtoSTD method allows users to make assumptions when they identify those five elements to fill in missing information.

The NLtoSTD-BB V2.0 [6] was evaluated using another controlled experiment (Study 2). The second experiment replicating the first experiment with a different experimental design.. In this experiment, 16 participating subjects were divided into two groups (Groups A and B) of eight subject each and they inspected a requirement document describing the requirements for the Loan Arranger Financial System (LAFS) that was created by professional developers at the Microsoft organization. The inspectors evaluated the requirements document using two different methods (fault checklist and NLtoSTD-BB V2.0)..

The participating subjects in Group A and Group B differed in their treatment order for inspection methods employed during the first and the second inspection cycles. In first inspection cycle, Group A inspected the document using the fault-checklist method while the Group B subjects inspected the same document using NLtoSTD-BB V2.0. During the second inspection cycle, Group A used NLtoSTD-BB V2.0 and Group B used fault checklist. Therefore, these two inspections resulted in a list of faults for each subject using NLtoSTD-BB V2.0 and a fault checklist during the first and second inspection cycles. The results showed from the second experiment showed a major improvement in the fault-detection effectiveness and efficiency of the NLtoSTD-BB method.

The results of the post questionnaire in the second experiment suggested that the NLtoSTD-BB V2.0 method was still not as user-friendly as it could be. There were some perceived difficulties with how NLtoSTD-BB V2.0 is applied. These difficulties could stem from the way in which a given requirement document is written or the type of software system being specified (i.e., reactive vs. interactive systems). These two factors tend to affect the degree to which an NL description can be easily represented by a state transition diagram. In light of these factors, deriving an entity, its initial and final states, a transition (operator), and guard (constraint) might have been too challenging for users who are not familiar with the STD concept. In particular, users had more difficulty applying the STD concept to the non-reactive systems, and this indicates that our method is more suitable for reactive systems. Furthermore, it was likely that the ease of use is inversely proportional to the number of modeling details. Therefore it was decided that the users should be asked to derive the least amount of information and still have enough to construct building blocks and to model the system's behavior.

To minimize the amount of derived information, the improvements were focused on an entity and its state change caused by other entities. For example, a requirement document for an elevator system (which is reactive in nature) would have various easily identifiable entities, such as doors, call buttons, send buttons, lift motors, and door sensors. People can easily surmise that these entities can change states: doors can open and close; buttons are pressed and depressed; and motors turn on and off as well as running in different directions and at different RPMs. The next subsection details the improved NLtoSTD method (V3.0).

### 2.2.3. NLtoSTD-BB V3.0: Second revision

The further improved NLtoSTD method determines a given entity and then asks what change (in state) is occurring to the entity. When answering this question, the user first finds the entity's final state and deduces the initial state as that which is the opposite of the final state. For example, if the entity is a door that has been closed by the requirement, then the initial state is assumed to be the opposite, that of being open. The result is to derive what we call an entity state using the following notation: Entity(State) (e.g., Door(Closed) and Door(Open)). Further, we think that this entity-state concept can be applied to identify operators and constraints. For instance, we can find the operator by simply asking whether the change in a given entity's state results in a change in another entity's state. In this way, we can find all the information we need to construct a building block by simply finding entity states and how they are related to one another.

The NLtoSTD V3.0 kept the concept of an entity (state), since NLtoSTD V2.0 showed improvements over V1.0 [5].

1. Limiting the choice of Sc and Sn to the same entity is the first change in V3.0. In this case, user is the decision maker on the entity's changing states. To solve the problem of describing more than one entity by a given requirement, multiple entities (state) per requirement was allowed.
2. Switching to the concept of transition from entity (state) was the second modification in V3.0. In contrast to V2.0 where transition was seen as action being constrained under certain conditions, in V3.0 entity it will produce a given action. In V3.0 Sc is labeled as Entity in its current state, Sn in its next states and T becomes an entity. Instead of having multiple entries, one entry would make the user's work easy

In Study 3, the NLtoSTD V 3.0 was evaluated by using a design similar to the Study 1, where in student teams developed their SRS documents that they later inspected using NLtoSTD-BB V3.0. Also, the Study 3 introduced the STD-BB to STD construction step. To aid the students during this step (i.e., STD-BB to STD), an automated software tool was developed that would load the STD-BB's (based on the students work in Step 1) and then use that information to construct the STD. The students then analyzed the resulting STD to find additional faults that were not found during the NLtoSTD translation. The detail on the tool used to perform the STD construction and analysis is discussed below.

### 2.3. STD-BB to STD: Application and Tool Support

The final step in the NLtoSTD method is the construction of STD based on the STD-BBs. The idea is to be able to both simulate the behaviors described in the requirements and to analyze these behaviors through the production of path traversals through the STD. This would potentially expose inconsistent and/or incorrect behaviors that can be readily traced back to the requirements. An incomplete STD, with its one-to-one traceability to the NL requirements, would expose incompleteness in the requirements verbage. Thus, the STD-BB to STD phase exposes potential faults by analyzing the STD's static and dynamic properties. The STD's construction was implemented through a software tool (shown in Figure 2) using a Graphviz API. The user enters the STD-BB data in an Excel spreadsheet (in Figure 2. Step1 NLtoSTD-BB Output), which is then read directly into the tool (by the tool's use of COM automation) (in Figure 2 Step2A: Loading NLtoSTD-BB). The STD is then displayed in a separate window (in Figure 2 Step 2B. Generation Of STD), and can be kept opened as more STD-BB data is being entered. The tool updates the STD, as changes are made to the STD-BB data. This allows the user to view the STD,

make changes to the STD-BBs that would correct any STD structural faults, and see the results of those changes in real time.

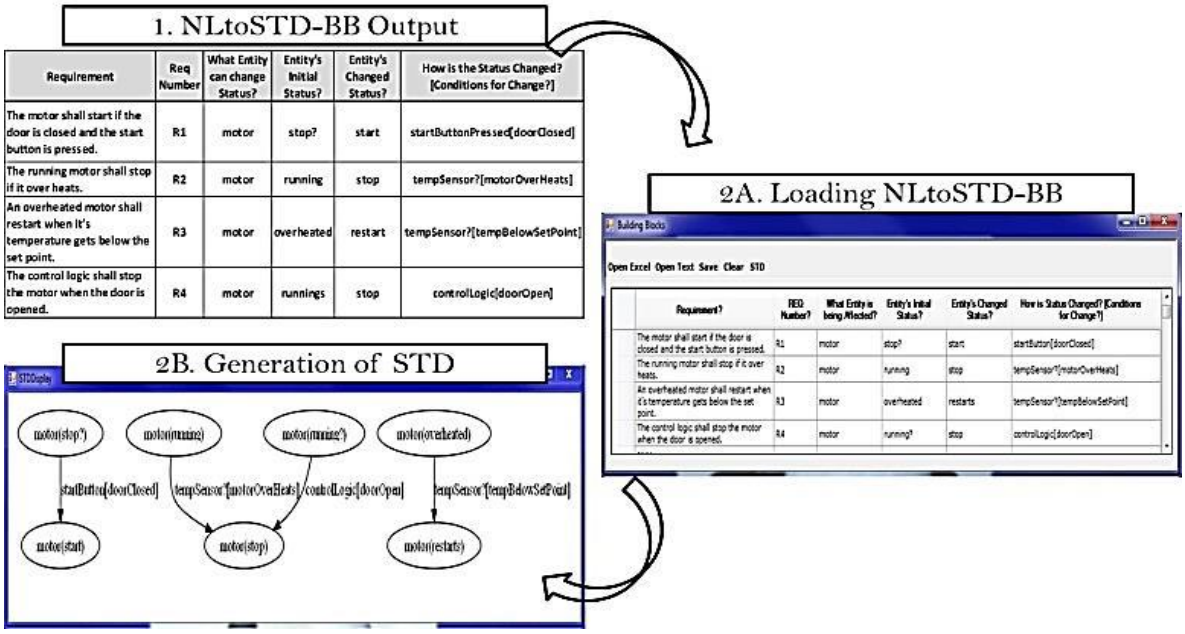



Figure 2. STD-BBtoSTD using Software Tool

# CHAPTER 3. EXPERIMENT

In the previous two experiments [5-6], the effectiveness and efficiency of the NLtoSTD-BB (V1.0 and V2.0) at finding incompleteness and ambiguities were compared to the fault checklist method during an inspection of requirements document. After analyzing the result, it is indicated that the NLtoSTD-BB translation method is very effective in detecting the missing functionalities and ambiguities. Also, these results helped improve the NLtoSTD-BB translation process by making it easier for users so that they can extract the elements needed to build the STD-BBs. This progress and the positive results from the previous study motivated us to replicate the study using the NLtoSTD-BB V3.0 and to extend this research assessment in order to analyze if additional more new faults (that were not exposed during the NLtoSTD-BB translation) can be found and later the construction of the STD. This experiment evaluates the complete NLtoSTD method as is major contribution of this master's paper.

This experiment is yet again a repeated-measure design (selected in order to increase the data points) in which different student teams (with varying number of members) developed a requirement document for a different system. Next, each participant individually inspected the requirement document (that was developed by them) using the NLtoSTD-BB method and kept a log of ambiguous, and missing requirements for respective documents. During the next step, the student individually worked to create STD from the STD-BBs (using an automated tool) and then analyze the resulting STD to log new faults that were not found previously. The details of the study are provided in the following subsections.

### 3.1. Research Questions and Hypotheses

The following research questions were investigated in this study. RQ 1 evaluates fault detection ability of the NLtoSTD-BB V3.0 (a revised method); RQ 2 and RQ 3 are focused on evaluating the STD-BB to STD method.

**Research Question 1 (RQ1):** Is the NLtoSTD-BB V3.0 effective at detecting incomplete and ambiguous requirements during inspection of NL requirements?

**Research Question 2 (RQ2):** Does creating the STD model result in detection of the faults in addition to those found during the NLtoSTD-BB transformation?

**Research Question 3 (RQ3):** What are the problems faced by the subjects when using the complete NLtoSTD method? How can the NLtoSTD method be improved?

### 3.2. Variables and Measures

Each subject used the NLtoSTD method to perform an individual inspection of their requirements document. We also conducted an in-class exercise as a pre-test to measure the performance of subjects using the NLtoSTD-BB translation and to understand their actual performance can be predicted based on their pre-test results.

We also measured following Dependent Variables:

- *Effectiveness* is the number of faults found by each subject
- *Efficiency* is the number of faults found by each subject per hour.

### 3.3. Participating Subjects

Sixteen computer science graduate students in Requirement Definition and Analysis course at North Dakota State University participated in this study. The student worked in teams of

two, three or four to develop a requirement document for different projects. Some students dropped the course which resulted in this irregular size of student teams and was outside the control of the researchers.

### 3.4. Software Requirement Artifacts

There were two phases to this study (development and inspection). First, during the development phase, each team developed a requirement document for a particular software system as shown in Table I. Second, during the inspection phase, each subject used the same requirement document developed by their team to inspect it using the NLtoSTD method (starting with NLtoSTD-BB followed by STD-BBtoSTD). Table I list the teams and their software systems.

Table 1. Artifacts Developed by Student Teams

| **Doc** | **Team #** | **No. of Subjects** | **System Description** | **Size (pages)** |
|---|---|---|---|---|
| A | 1 | 4 | Parking lot availability system | 25 |
| B | 2 | 4 | Web portal for student residence life | 22 |
| C | 3 | 3 | Virtual story board system | 28 |
| D | 4 | 2 | Matbus application for android | 25 |
| E | 5 | 3 | Professional development management system | 32 |

### 3.5. Experiment Procedure

The experiment design includes several steps. Figure 3 illustrates the experiment steps and the details are provided in the following subsections.

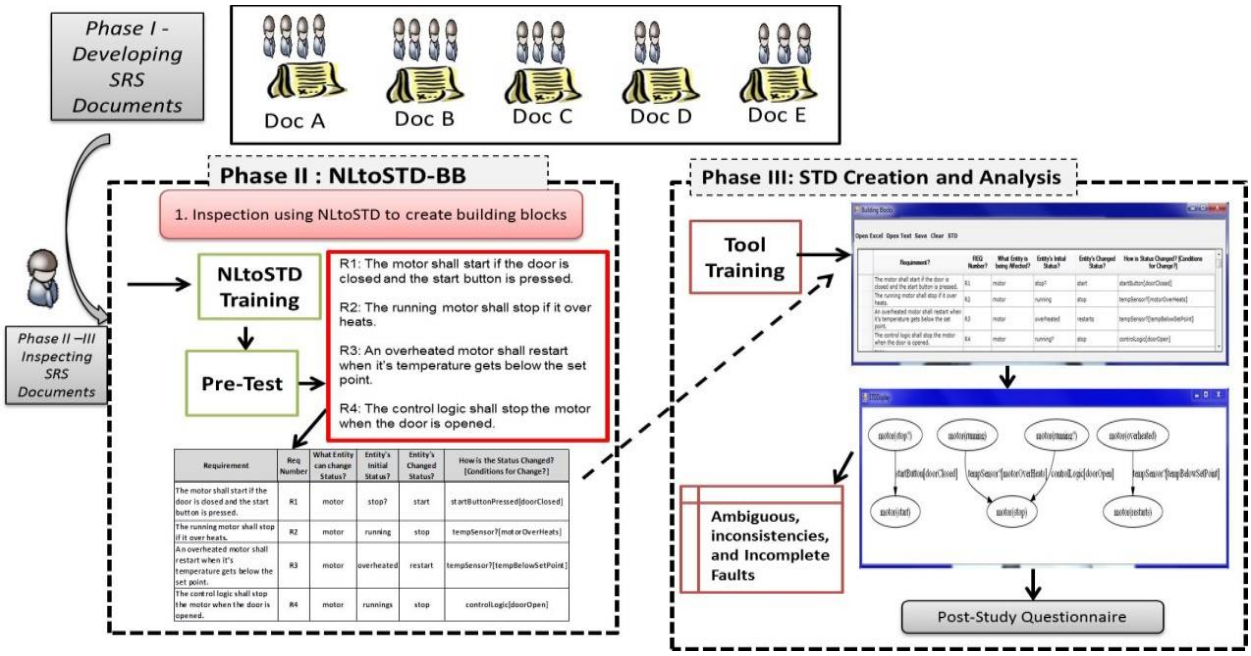



Figure 3. Experiment Procedure: Experiment Steps, Trainings, and Output

### 3.5.1. Phase I – Development of SRS documents

In the semester beginning, the participants were divided into 5 different teams of three or four participants each. The participants were allowed to select their own team members. Some subjects dropped the course, leaving 16 participants that developed five different requirement documents. Each team developed the requirements documents for their identified software system. The number of members in each team, the system for which they developed the requirements, and the size of their documents (in pages) are listed in Table I.

### 3.5.2. Phase II – Inspection of SRS documents using NLtoSTD-BB

During this step, the students in each team individually inspected their own SRS document using the NLtoSTD-BB V3.0 method (explained in II.B.3).

*a. Training 1* -- *NLtoSTD-BB and pre-tes:t* During this 30 minute training session, the subjects learned about the NLtoSTD method. The participants were first trained on how to map the NL requirements to STD-BBs. Next, the participants were instructed how to document the

building block elements (entity, initial status, changed status, conditions for change) on a spreadsheet using few examples. Next, the participants were instructed how to record the "ambiguities" and "incompleteness" or any other requirement faults that are found during the application of NLtoSTD-BB. They were given the description of the example system and few requirements. an and document the faults found during the translation of requirement into STD-BB using the same spreadsheet. The students' work on translating requirements into BBs and faults recorded serve as pretest to provide an idea of how well they understood the NLtoSTD-BB. To mitigate a potential validity threat of learning, two requirement lists were prepared that contained same requirements (but in different orders)

*b. Step 1-- Inspecting requirement document using NLtoSTD-BB:* During this 50 minute session, the participant used the information from Training 1 and individually inspected their own requirement document using the NLtoSTD-BB method. This step resulted in a list of 16 individual spreadsheets that contained the STD-BB elements and the faults found (one per participant).

### 3.5.3. Phase III – Tool practice and NLtoSTD

During this step, the subjects were trained on how to use the tool to construct the STD (after loading the STD-BBs from Phase II). Next, the subjects were instructed on analyzing their STD's to document the new faults that were not found earlier.

*a. Training 2 and tool support:* During this 15 minute training session, the participants learned about the STD tool. The subjects were instructed how to load the BBs from Phase II into the tool and then, how to construct an STD from the BBs. The subjects were instructed on how to navigate to the BB spreadsheet and to verify that the BBs have been loaded correctly. The subjects were then instructed to examine the constructed STD for faults on the basis of: 1) nodes

that have *no incoming paths or no outgoing paths*, 2) nodes that are *not connected together*, 3) faults caused *due to the question marks*, and 4) *any other faults* that come to mind by examining the STD. Finally, the participants were taught how to record the fault type i.e. incompleteness, ambiguous, or inconsistency in the fault spreadsheet. To ensure that subjects understood this process, the subjects practiced these steps through an example system during an in-class session.

*b. Step 2 – STD-BBtoSTD and inspecting STD for additional faults:* During this 30 minute, the subjects used the information from Training 2 and constructed the STD diagram from STD-BB. The output of this step was 16 individual STD diagrams (one per participant). The resulting STD diagram were analyze

d for potential incompleteness, inconsistencies in the behaviors, or any other requirement faults. A fundamental idea behind NLtoSTD is that missing building blocks in a STD are much easier to identify than missing statements in a set NL requirements. So, the participants were asked to analyze and record the faults during and after the creation of STD. The students were also asked to document the reason for that fault and classify that fault (into *incompleteness*, *ambiguous*, *inconsistency* or *other*) in the fault spreadsheet. This step resulted in a list of 16 individual fault lists (one per participant).

*c. Step 3 – Post Study Questionnaire:* The subjects were provided an opportunity to provide feedback about the NLtoSTD-BB and the STD-BB to STD creation. Also, at the conclusion of the study, an in-class discussion was held with subjects to help researchers better understand the results.

### 3.6. Data Collection

This section provides a description of the qualitative and quantitative data collected during the experiment run.

*Quantitative Data*: The quantitative data included the *ambiguous*, *missing*, and *inconsistent* faults found by each subject in their SRS document during: a) translation of NL requirements to the STD-BBs, and b) and after the construction of STD using the BBs. During a), the spreadsheets recorded the data on the different elements that make up STD-BBs for each NL requirement and a brief description of the related fault(s). During b), the spreadsheets recorded the data on the faults (and its categorization) and observation on what exposed the faults. Since, each subject was provided 50 minutes during NLtoSTD-BB step and 30 minutes during STD-BB to STD step, the timing data was used for analyzing the efficiency values.

*Qualitative Data*: The subjects rated the NLtoSTD method on characteristic of simplicity and ease of use by answered multi-question questionnaire that were based on a 5 point likert-scale. We also collected qualitative feedback from the post-study interview with participating subjects.

# CHAPTER 4. DATA ANALYSIS AND RESULTS

This section provides an analysis of the quantitative data collected during Phase II and III and the qualitative feedback collected during the post-study questionnaire and interviews. An alpha value of 0.05 was used for all statistical tests.

## 4.1. RQ 1: Fault Detection Effectiveness and Efficiency during the NLtoSTD-BB Translation

This section analyzes the *effectiveness* and *efficiency* of the NLtoSTD-BB V3.0 during the requirements inspection. One of the researcher determined the validity of the faults for each subject by reading during the fault spreadsheet reported by each participant to eliminate any false-positives (or if any faults were unclear) before analyzing the data. Next, the number of "*Missing Functionality* (MF)" and "*Ambiguous Information* (AI)" faults reported by each subject during the application of NLtoSTD-BB for their respective documents were counted.

As, the document created during the course was individually checked by each subject. The total number of AI and MF fault types is described by Figure 4 that were captured by the members of each team. Some observations from Figure 4 were as follow:

- Fifteen out of sixteen subjects found some faults (AI or MF) during the NLtoSTD-BB based inspection of their requirements document. The subjects (numbered 6) reported a lot of faults but none of them represented real problems. For subject 6, analyzing the fault reports and the mapping of NL requirements into STD-BBs revealed that this subject extracted completely different STD-BBs than the ones required in our method;
- AI vs MF faults did not have any consistent differences in the total number of faults captured by the subjects within each teams. A large number of MF faults were captured during the results from previous experiments when the subjects were using the NLtoSTD-

BB method consistently. This is a positive result and is observed due to the fact that the improved heuristics in V3.0 were able to focus the reviewer's attention on both types of faults whereas the previous mapping heuristics focused more heavily on identifying missing details when constructing the STD-BBs.

- For each document, the average number of faults for each document was calculated for each team by dividing the total number of unique faults by the number of subjects who inspected the document. an average of 15, 12, 18, 9, and 22 faults respectively were found through results of teams 1,2,3,4 and 5. This is comparable to the best performance of student teams from earlier studies [5] where, the NLtoSTD-BB method found an average of 15 faults vs. an average of 5 faults found using the fault-checklist based method.

Therefore, based on these results, fault detection effectiveness of the NLtoSTD-BB method has been improved from its first evaluation. This is shown by an increase in the number of AI and MF fault types found during this study, when compared to the performance on the previous two studies [5, 6]. Additionally, the distribution of faults is more consistent across both fault types and the fact that the revised NLtoSTD-BB mapping heuristics were able to highlight hidden ambiguities in individual requirements which were otherwise not detected during this step in the previous studies.

Regarding the efficiency (faults/hour) of the NLtoSTD-BB method, the student teams 1, 2, 3, 4, and 5 found an average of 19, 7, 10, 10, and 26 faults per hour respectively. In comparison to the results from the first experiment [5] this is a huge improvement and more similar to the results from the second experiment [6] where the subjects found an average of 19 faults per hour when using the NLtoSTD-BB method. Therefore, the effectiveness and efficiency results of this

replicated study with NLtoSTD are at least as good (or better, for Team 5) than the results from the previous studies.

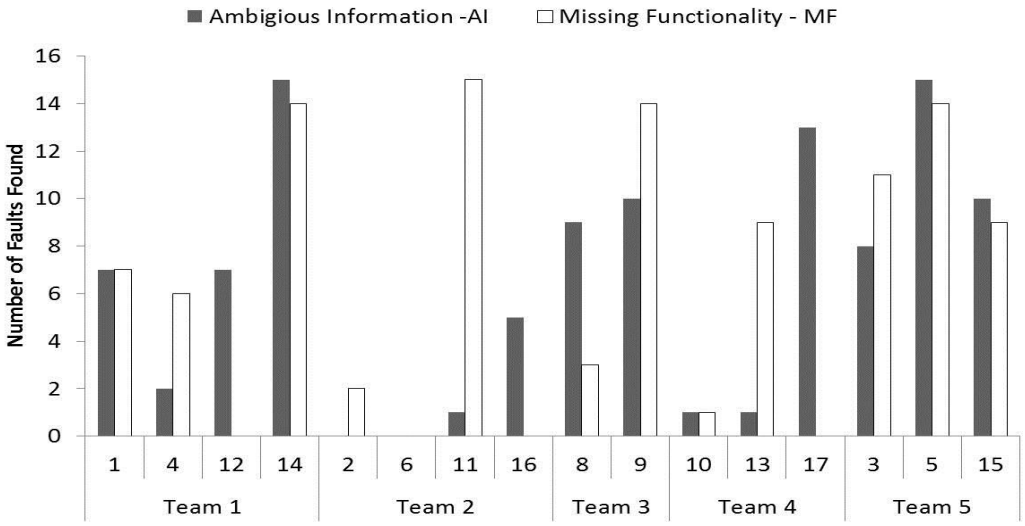



Figure 4. Number of AI and MF type Faults found by Subjects using NLtoSTD-BB V3.0 (Phase II)

### 4.2. RQ 2: Fault Detection during the STD Creation

Though, the previous studies and the results provided in Section IV.A validate the effectiveness and efficiency of the NLtoSTD-BB, we have not evaluated every benefit that the NLtoSTD method offers. The next step, translation of BBs into STD, can readily find the ambiguities and incompleteness in the NL requirements by examining the gaps (or disconnections) and inconsistent path traversals in the STD.

We had hypothesize that additional MF and AI faults will be exposed during the creation of STD that are otherwise left invisible during the transformation of NL into STD-BBs. Furthermore, we believe that the formation of STD aids the inspectors to identify the inconsistencies (Inc) in the requirements that were not reported during the NLtoSTD-BB step or during the checklist based inspection.

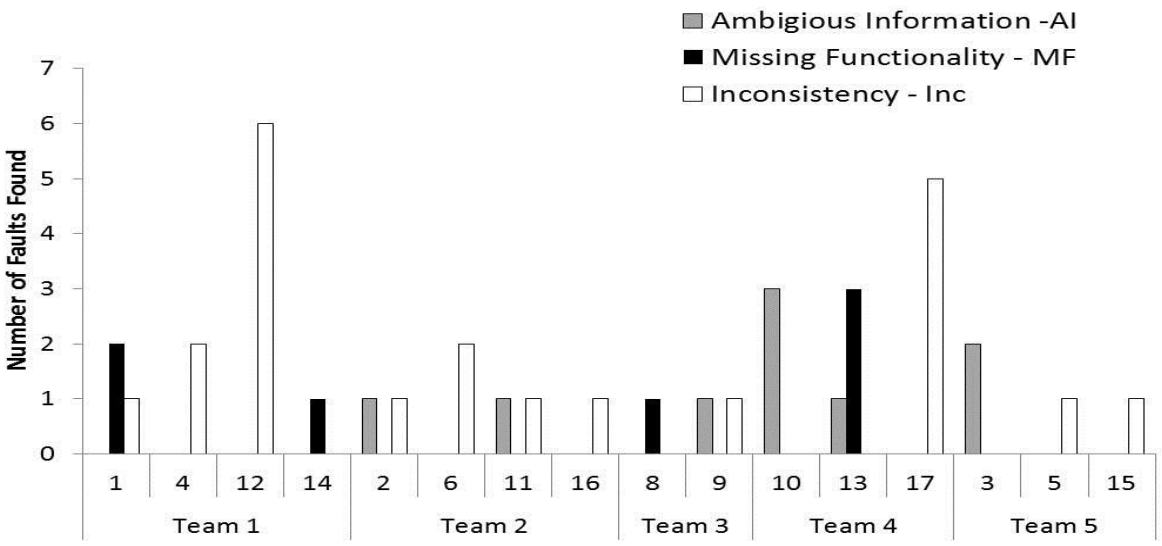



Figure 5. Number of AI and MF type Faults found by Team after the Creation of STD-BB (Phase III)

To investigate the validity of this step, we counted the number of new MF, AI and INC faults reported by each subject after the formation and analysis of STD for their individual documents as shown in Figure 5. The results shown in Figure 5 are organized by the teams. As we all know that each subject used their work from the previous step (i.e., NLtoSTD-BBs) and performed this step individually and reported the faults that they had not discovered during the previous step.

Some interesting observations from Figure 5 follow:

- Each subjects found at least one new fault (AI or MF or Inc) after creating the STD.
- A larger number of Inc faults are found during this step as compared to the AI and MF faults as expected. This is also consistent across all the teams. This is because; it is easy to observe the inconsistencies when looking at a complete STD as opposed to translating individual NL requirements (one at a time) to STD-BBs.

Since, the loading of BBs to create the STD is an automated process (using a tool); it is not amazing that subjects were able to discover extra faults by focusing their attention on

investigative the STD and recording faults. This is based on the students' feedback and is discussed in the next section.

To better understand the *effectiveness* results of the STD-BB to STD comparative to the overall effectiveness of the NLtoSTD method, Figure 6 shows the comparison of the total number of unique faults found by the subjects across each team during the phases II+III (i.e., NLtoSTD-BB + STD-BB to STD) vs. during the Phase III (i.e., STD-BB to STD). Figure 6 normalize the results by the percentage contribution of the Phase III inspection relative to the overall inspection effectiveness for each team (calculated by dividing the number of unique faults found during Phase III by sum of total number of unique faults found during Phases II and III combined).

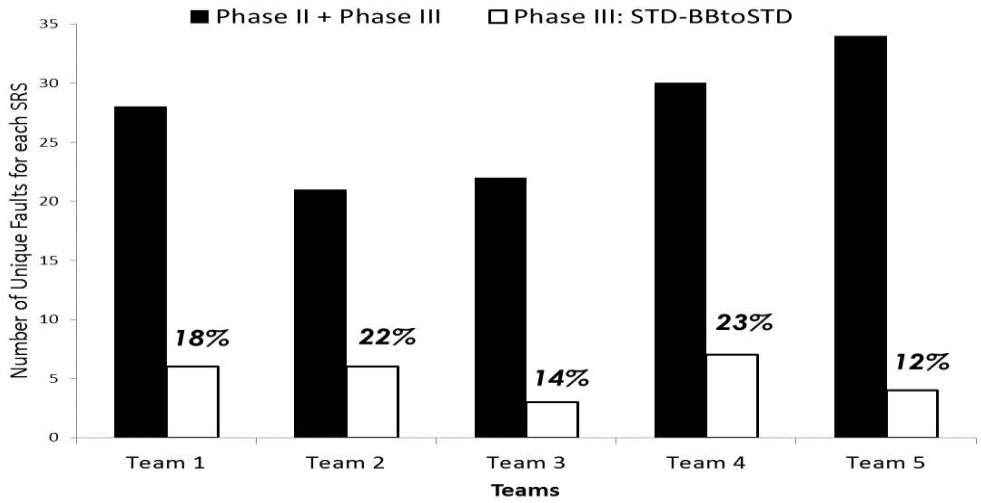



Figure 6. Comparison of the Total Number of faults found by Teams in Phase II and Phase III

As shown in Figure 6, the student teams, after the STD creation, found anywhere ranging from 14% to 23% of total faults. Also, we compared the average number of faults detected during Phase II vs. Phase III to get more insights into the worth of the complete NLtoSTD process during the inspection of requirements. After that all the data from the subjects were combined and the results exposed that the subjects found an average of 14 faults during Phase II vs. an average of 3 faults during Phase III. It is an obvious that the Phase II is more effective at finding faults in NL

requirements as compared to the Phase III results. But, we were investigating whether the Phase III would be able to discover any new faults that were not found during the NLtoSTD-BB and would have otherwise left undiscovered. On that end, we evaluated the contribution of the STD-BB to STD to the overall effectiveness of the NLtoSTD method.

To verify the usefulness of Phase III (creation and analysis of STD), a one-sample t-test was run separately to determine whether the number of faults found during re-inspection using STD was significantly greater than zero (0). The result was found to be statistically significant ($p < 0.001$).

These results specify a benefit of creating STD using the BBs, for finding missing, ambiguous, and inconsistent requirements. Moreover, as this was an initial investigation, the subjects did not get to actually collaborate when creating the STD model or during the analysis of the STD model. We believe, that allowing the stakeholders (that may have different interpretation for the same requirements) to collaborate during the STD creation will force them to make sure that the requirements are as consistent and complete as possible. Requirement engineers and stakeholders can see what the STD lacks, and together, they can work towards its completion.

### 4.3. Difficulties Faced by Subjects Using the NLtoSTD Method

The qualitative data (post-study questionnaire and interviews) collected during the previous studies [5-6] has been helpful in improving the heuristics of the NLtoSTD-BB method so that it is easier to transform a set of NL requirements into STD-BBs and find faults during this process. As a result, this experiment used the latest version of the NLtoSTD-BB method that also needed to be evaluated to add more insights about the usability of the method. To complete that, the participants were asked to rate the difficulty level for finding the "*Entity*", "*Initial Status*", "*Changed Status*", "*How is Status Changed*", and "*Conditions for Change*" when using the

NLtoSTD-BB method. Using a 5-point likert scale (ranging from 1-very difficult to 5-very easy), the participants rated the difficulty level for each of the five elements of an STD-BB. We conducted One-sample Wilcoxon Signed-Rank test to determine whether the medians ratings were significantly greater than 3 (midpoint of the scale). The results showed that the NLtoSTD-BB method received positive ratings (i.e., median value greater than or equal to three), but the results were not statistically significant.

The subjects also rated (on a 5-point scale) the difficulty level during the construction of STD (loading BBs using tool) and reporting (analyzing the constructed STD for faults). The results from One-sample Wilcoxon Signed-Rank test showed that the NLtoSTD method received significantly positive ratings (median value of 4; $p < 0.05$).

The complete NLtoSTD method was also evaluated using the feedback from subjects on the following seven attributes: *Simplicity*, *Ease of Understanding*, *Ease of Use*, *Intuitiveness*, *Comprehensiveness*, *Usefulness*, and *Ease of finding faults*. Each subject rated the attributes on a 5-point scale. Again, for each characteristic, we conducted One-sample Wilcoxon Signed-Rank test to determine whether the medians ratings were significantly greater than 3 (midpoint of the scale). Based on the results, the NLtoSTD received significantly positive ratings on *Ease of Understanding*, *Ease of Use*, and *Ease of finding faults* (i.e., $p<0.05$). The result was positive for all other attributes except for the *Intuitiveness*.

Ever since this was the first time we used the complete NLtoSTD method, we wanted to identify potential improvements. The survey question used to help with this was: *What problems did you encounter when using "NLtoSTD" method? What can be done to overcome these problems?*

After reviewing the subject responses and talking to the subjects, we found that the subjects favored the NLtoSTD method, and the subjects felt that constructing STD using BBs could be used to validate correct behavior, and not just to find faults. On the other hand, there were several concerns regarding the process used for analyzing the constructed STD for detecting faults as discussed:

- Specially, subjects reported that d more detailed checklist based method would like for analyzing the STD for faults (as opposed to use their intuition and looking for gaps and disconnected BBs). We plan to provide more detailed checklist to guide the inspection of STD for fault detection in future;
- The subjects' responses gave an impression that they did not think there was anything wrong with the requirements as they examined the STD, although the subjects BBs were displaying total disconnect with other BBs. Throughout the interviews, the subjects responded that this was the result of requirements not being a part of a reactive system or the subjects are pretentious that everything is correct because they did not find any more faults during the translation of NL into STD-BBs. We did emphasize that the subjects need to analyze the STD model carefully and report faults that they had missed during the earlier steps. Additional practice during the training session in future studies will help reinforce this point.

The subjects also provided insights into the usability of tool that was used to load STD-BBs and construct STD.

- Some subjects felt that the tool was easy to use, whereas few felt more automation was needed especially during the translation of NL requirements into STD BBs (which was performed manually). The subjects also felt more automation was needed to highlight the

gaps and inconsistencies in the constructed STD. As already mentioned, the tool could be incorporated with the ability to analyze the STD according to set of predefined properties. As to automating more of the NLtoSTD process, that could potentially include the automatic production of the STD-BBs using some degree of natural language processing (NLP). However, we suspect that a substantial amount of human intervention in the translation process will always be necessary.

- Two subjects felt their STD diagram was incomplete because they didn't have all the requirements transformed into BBs. However, they were still able to find major faults, in their requirements. Overall, the subjects felt that the success of the NLtoSTD depends on Phase II (NLtoSTD-BBs) and needs to be performed more carefully.
- Some subjects mentioned that they were able to analyze the STD diagram effectively, because they could see all the building blocks at once.
- One subject mentioned that the NLtoSTD method could be better suited for "hardware" systems. One felt the way (GraphViz) displays the STD is not good, and suggested that we use VISIO instead. This last suggestion was taken as an indication that the way the STD was displayed may play a role in determining the method's effectiveness in exposing faults. We feel however that a better addition to the tool would be to provide some built-in analysis of the STD, so as to point deficiencies in correct STD properties that may not be obvious to user. Another potentially useful addition is the means to automatically group the diagram into hierarchies in order to address a state explosion problem.

Also, the subjects reported that the NLtoSTD process helped them understand the major problems in requirements, and that the effort spent during the NLtoSTD inspection process was

worthwhile. The potential improvements regarding the tool and the guidance to help analyze the STD diagram will be implemented in future evaluations.

# CHAPTER 5. THREATS TO VALIDITY

Like any empirical study, this study has some limitations. The documents that we have used in this experiment contained naturally occurring faults (as opposed to faults those are artificially seeded), but the documents were developed by students and they may not represent the industrial strength document. Also, the study focused on students in a classroom setting and is likely to have diverse experience and time pressures than professionals in a real environment. Further, we applied the proposed approach considering a traditional development model, thus the findings from this study cannot be applied to other development models (e.g., agile approach)., But our approach can be easily applied to modern development models by incrementally building a requirement model starting with core (or prioritized) components. An important internal validity threat was a lack of a control group. We have conducted previous studies with a control group, but since this was the first application of the complete NLtoSTD method and because of the limited size of data (subjects), we did not employ a control group. We plan to address these threats by performing additional studies with industrial software products, different development models, and a control group.

# CHAPTER 6. DISCUSSION OF RESULTS

This Chapter would include the Conclusion reached after creating the current version of the Software in regards to the Objectives of the System. This section summarizes the results of our study and discusses their implication in light of the original research questions.

***Research Question 1****:* Is the NLtoSTD-BB V3.0 effective at detecting incomplete and ambiguous requirements during inspection of NL requirements?

The results showed that the revised NLtoSTD-BB method helped inspectors find natural ambiguities and incompleteness in requirements. The effectiveness and efficiency results were compared against the previous research results [5-6], and the major findings are summarized as follows:

- In comparison to the first evaluation, the fault detection *effectiveness* of the NLtoSTD-BB has been improved. The subjects were able to find larger number of total faults (on average), and the distribution of faults across fault types (MF and AI) was consistent.
- In our previous experiment [5], we had found that the characteristics of the NL requirement specifications affected the effectiveness results e.g., NLtoSTD V1.0 goal is to derive one building block per requirement. But The NLtoSTD V3.0 was not affected by the requirement characteristics and is applicable to different types of requirements.
- Examining the spreadsheets that recorded The mapping of the NL requirements into BB elements was recorded in the spreadsheet and those spreadsheets are examined which exposed that subjects had a much better understanding on how to apply the NLtoSTD method which in turn resulted in choosing correct values for e*ntity*, *Status (Initial and*

*Changed)*, and *Conditions for Change* . As a result, the faults were true and represented real problems.

- Also, the results showed that the NLtoSTD-BB method helped find the faults faster (i.e., higher efficiency) when compared to the previous results. The subjects also rated favorably the process used to translate NL requirements into STD-BBs.

Overall, the results were consistent with the previous findings and in some cases, the improved NLtoSTD-BB method is reflected in a higher increase in the effectiveness and efficiency values.

***Research Question 2****:* Does creating the STD model result in detection of the faults in addition to those found during the NLtoSTD-BB transformation*?*

As this was a new component that had not been previously evaluated, the answer to this question was even more encouraging to our future evaluations. A summary of major findings based on the results in Section IV.B are:

- During the examination of STD constructed from the BBs, additional MF and AI fault types are discovered
- The construction of STD aids inspectors at detecting a large number of inconsistencies (Inc) that are otherwise not apparent when looking at individual requirements;
- Each subject found some faults during Phase III that they had not discovered in Phase II. Also, the student teams found anywhere between 14% to 23% out of total faults during this inspection cycle;

Based on the statistical results, construction of STD and its subsequent inspection for faults is useful for overall inspection effectiveness using the complete NLtoSTD method.

***Research Question 3:*** *What are the problems faced by subjects when using the complete NLtoSTD method? How can it be improved?*

The results from the post-study questionnaire and follow-up discussion with participating subjects provided insights in to the use of the NLtoSTD method and improvements that can help improve the performance in future studies as discussed:

- As the subjects mentioned the promise of the method, the subjects mentioned a need of more detailed instructions to guide the examination of STD (one it is constructed using the tool) for fault detection. We plan to contain more comprehensive training instruction and also plan to hand them out during the future study run;
- Subjects also mentioned that the overall success of the NLtoSTD method is heavily dependent on the NLtoSTD-BB translation. We agree with that. However, the subjects assumed that if there were no more faults left to be found during the translation process then they didn't analyzed the STD for more faults. We need to reinforce the analysis of STD for detecting faults that are otherwise not found during the construction of BBs;
- The subjects provided suggestions on the aspects of the NLtoSTD method that can be automated. Specifically, the subjects mention that the tool should guide the NLtoSTD-BBs translation and should at least highlight parts of STD that are completely disconnected. We plan to make this process as much automated as possible without losing the promise of inspections;

Based on the results, we were also able to identify additional improvements to revise the NLtoSTD V3.0 method and are discussed in the following paragraphs.

The results of experiment three encouraged us to continue improving the use of entity states. Thus, in V4.0 we now ask the user to find all the entity states in the set of the requirements, combine similar entities into their own STD, and then use the entity(states) of a given STD to "trigger" the transitions in the other STDs. We feel this approach would be best suited for a synchronous system, such as a reactive system. As a side note, we have observed that the easier the method is to apply, the further we away we move from the one-to-one relation between NL requirements and STD-BB. We are, however, progressing at resolving this issue, and arriving at a suitable compromise between ease of use of one-to-one traceability.

# CHAPTER 7. CONCLUSION AND FUTURE WORK

Based on these results, the NLtoSTD method is effective method to detect AI, MF, and Inconsistency fault types during an inspection of NL requirements. We also identified the areas of improvement that would benefit the performance of subjects using the method in future evaluations. Our future work would include more replications, but with a classic control group design so that we can understand how many faults found during the Phase III are solely due to the STD creation and not just due to the re-inspection. We also wish to automate as much of the heuristics as possible, including the NLtoSTD building block portion. The STD analysis could be automated as well, using a reasoning engine written in a logic language such as Prolog, and this has already been achieved to a certain degree. Also new addition can be made as per subject suggestion like training section, help provided for tool , survey checklist, etc.

## REFERENCES


[1] B. Boehm and V. Basili. Software fault reduction top 10 list. IEEE Computer, pages 135–137, January 2001.

[2] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

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

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

[5] D. Aceituna, H. Do, G. Walia, and S. Lee. Evaluating the use of model-based requirements verification method: A feasibility study. *EmpiRE*, 2011, pages 13-20, August 30, 2011.

[6] D. Aceituna, G.Walia, H. Do, and S. Lee. Model-based requirements verification method: Conclusion from Two Controlled Experiments. 2013, North Dakota State University, NDSU-CS-TR-13-002. http://cs.ndsu.edu/~hdo

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

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

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

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

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

[12] D. Barry. Ambiguity in natural language requirements documents. *Lecture Notes in Computer Science*, *LNCS*, volume 5320, pages 1-7, 2008.

[13] E. Letier, J. Kramer, J. Magee, and S. Uchitel, Monitoring and control in scenario-based requirements analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 382–391, 2005.

[14] L. Kof, Scenarios: Identifying missing objects and actions by means of computational linguistics. In 15th IEEE International Requirements Engineering Conference, pages 121-130, 2007.

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

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

[17] D. Gause. User DRIVEN design - The luxury that has become a necessity. In Proceedings of 4th International Conference on Requirements Engineering, 2000.

# APPENDIX A. TOOL IMPLEMENTATION

This chapter includes the detailed design used to build the Virtual Requirement Prototype Tool. The system's design is used to create the functions and operations of the requirements in detail, including screen layouts, product function, user characteristics, functional and nonfunctional requirements, constraints, assumptions, dependencies, business rules, process diagrams, and other documentation. The output of this chapter describes the new system which is defined as a collection of modules and subsystems. For each requirement, there is a set of one or more design elements that are produced using the different prototypes. These design elements describe the desired software features, in detail, screen layouts, and class diagrams. The intention of these diagrams is to describe the software in detail so that the system can develop the application with less additional design input. The system’s mock screen shots are shown later in this chapter.

## A.1. Product Perspective

This section describes the general description of the Software used for generating the Building Blocks. The Virtual Requirement Prototype Tool is a desktop-based system.

## A.2. User Interface

This section describes the user interface found in the Virtual Requirement Prototype Tool is as follows.

Users are able to view the home page of the virtual requirement application, browse the different categories, like open excel sheet browse for the selected excel sheet, open text file, save the file for later viewing, clear the items in the building block, STD which can help to create

STD diagram from the STD-BB loaded from excel sheet, training presentations for creating NLtoSTD-BB and STD-BB to STD and help for tool is present in the user interface.

### A.3. Hardware Interface

The virtual requirement application shall provide minimum hardware requirements. The following hardware configurations are required for a PC using the virtual requirement application:

- Pentium processor
- 32 MB of free hard-drive space
- 128 MB of RAM

### A.4. Software Interface

This section lists the requirements that are needed to run the system efficiently. The operating system needed for the system to run effectively, the interface to run the application, the driver for running win form applications, the integrated development environment to develop the application, and the third-party tool used for editing purposes are as follows:

1. Operating System: Windows (Vista/Windows 7/ Window 8) or MAC OS
2. Integrated Development Environment: Microsoft Visual Studio 2010
3. Third-Party Tool: Graphviz API

### A.5. Product Function

This section describes the different functions of the virtual requirement application. The tool would have the following basic functions:

1. The system allows the user to open excel sheet of his choice.
2. The system allows the user to open text file of his choice.

3. The system allows the user to clear the data grid of building block screen.

4. The system allows the user to save file.

5. The system allows the user to form STD.

## A.6. Constraints

This section describes the hardware limitations and the Assumption and dependencies of the tool.

1. Hardware Limitations: The minimum hardware requirement for the system is 128 MB of Ram and a 32-MB hard-disc drive.

2. Assumptions and Dependencies

- User should have Microsoft office 2007 and onward installed.
- User should be familiar with the Microsoft office
- We assume that users of the system adhere to the system's minimum software and hardware requirements.

## A.7. High-Level Use Case Diagram

The System Use Case shows the user a detailed view of the system and how the actors would interact with each other and with the system. The explanation for each use case is then provided below the System Use Case (Figure A1.) and helps the user understand who the actors areas well as giving the description of each use case along with its pre and post conditions that should be satisfied once the use case is implemented in the software. Figure A1 demonstrates the use case of for a User where he or she has access to the application. The user can access the open excel, open file, save, clear and display STD.

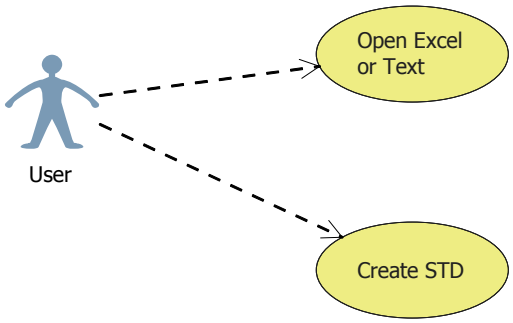



Figure A1. Virtual Requirement Application System Use Case Diagram

Below is the explanation of the different Use Cases in the System use case and the Actors associated with each use case. The Description is used for a novice user to better understand the working of the System and the pre-conditions that should be satisfied before invoking each use case

1. **Use-Case Number:** US-001

   **Application:** Virtual Requirement application

   **Use-Case Name:** Home Interface

   **Use-Case Description:** This use case lets the user view the home page which has different categories when they first run the application.

   **Primary Actor:** User

   **Precondition:** Run the application.

   **Post-condition:** The user successfully runs the application and is able to view the home interface with the different categories.

   **Basic Flow:**

   - Run the application
   - View the home interface/page

2. **Use-Case Number:** US-002

   **Application:** Virtual Requirement application

   **Use-Case Name:** Open Excel or Text File

   **Use-Case Description:** This use case will help the user to browse documents in the system and open excel sheet or text file.

   **Primary Actor:** User

   **Precondition:** Run the application and home interface appears.

   **Post-condition:** The user able to view the data from excels or text file loaded in the main home interface/page in the building block region.

   **Basic Flow:**

   - Run the application
   - View the home page
   - Browse the categories either open excel or text file
   - Click open and excel file open and load data in building block region.

3. **Use-Case Number:** US-001

   **Application:** Virtual Requirement application

   **Use-Case Name:** STD

   **Use-Case Description:** This use case lets the user to create the STD from STD-BB which is already load from excel or text file in tool.

   **Primary Actor:** User

   **Precondition:** Run the application.

**Post-condition:** The user successfully runs the application and is able to view the home interface and STD-BB is already loaded in tool.

**Basic Flow:**

- Run the application
- View the home page
- Browse the categories either open excel or text file
- Click open and excel file open and load data in building block region.
- Click STD and STD diagram will be created.

### A.8. Architectural Diagram

As shown in the Figure A2 , subsystems of the Virtual Requirement application consist of 4 adaptive service (1)Building block (2)Excel Automation (3)Graphviz (4) STD Display. The interface for the 4 adaptive consists of the following operation:

**Building block**: ClearTheGrid, DisplaySTD, OpenExcel, OpenText, LoadTheRequirements, LoadReqNumber, LoadTheEntities, LoadOperators, LoadInitialStates, LoadFinalStates etc.

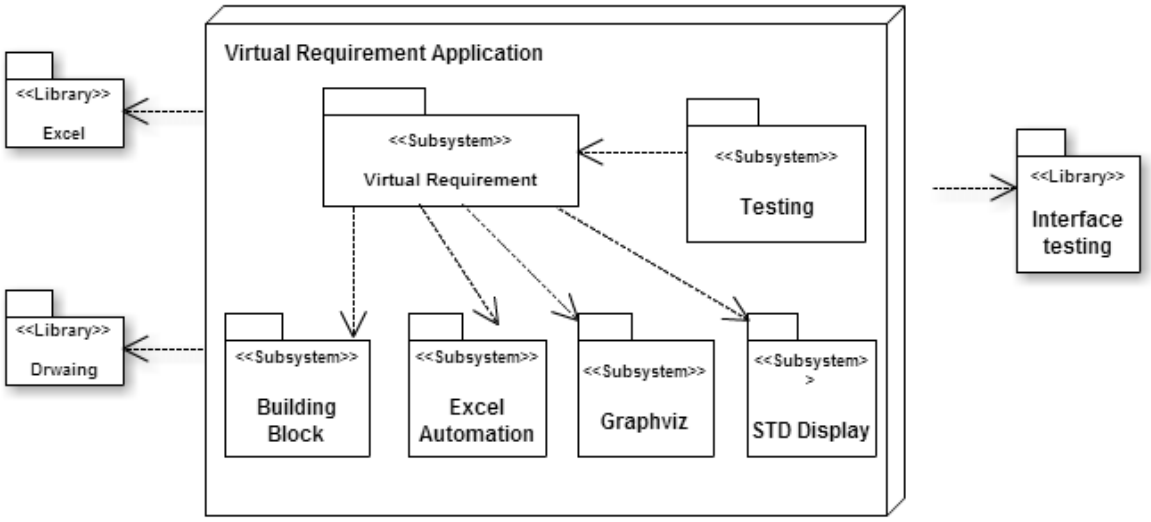



Figure A2. Architecture of Virtual Requirement Application

**ExcelAutomation:** CloseExcel, CloseExcelWorkBook, GetColumn A to F, LoadColumn A to G, OpenExcelWorkbook, SetWorkSheet etc.

**STD Dispplay:** CreateDigraphScript, DisplayTestImage, ResizeTheLabelLenght, SetTheImage and STDDisplay

**GraphViz:** All API of GraphViz

Upon receiving a request for a service these services class orchestrate a series to call to the other classes and the classes with in the each subsystem were made adaptable with the help of Visual Studio.

## A.9. Class Diagram

This class diagram show the inner working of the Virtual Requirement tool and below are explanation of each class.

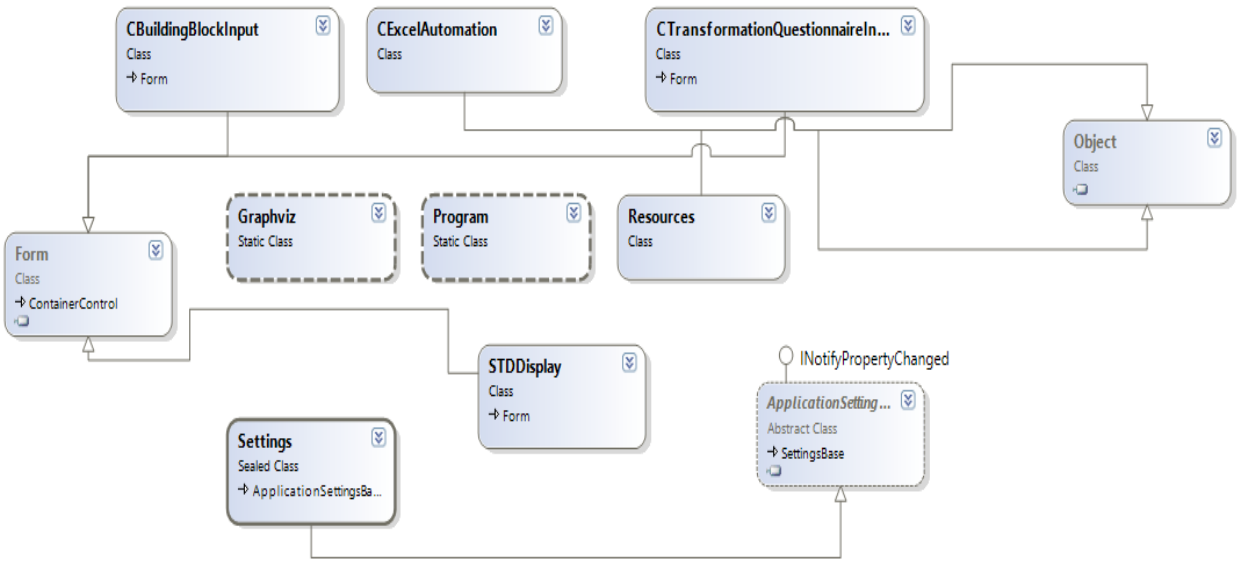



Figure A3. Virtual Requirement Application Class Diagram

1. **CBuildingBlockInput:** This class is used to process all information of the tool Figure A4 shows the methods that are used in this class.

- **LoadFinalStates :** For loading the requirement number from the excel column E, .LoadColumnE is used to load the items to data GridColumns3 by using a for loop which is useful for iterating over array [4, rowIndex]
- **LoadOperators :** For loading the requirement number from the excel column F, .LoadColumnF is used to load the items to data GridColumns4 by using a for loop which is useful for iterating over array [5, rowIndex]
- **Display STD**: In this method, create array and then with the help of array STDDisplay as an image and created as a script.
- **ClearTheDataGrid:** This method is used to clear the data grid of building block.
- **CBuildingBlockInput:** This method initializes the user interface objects with the values provided by you in the Property Grid of the form designer.
- **CBuildingBlockInput_Resize:** This method resize the QuestionnaireDataGrid width and height of data grid of main interface when data loaded from the excel or text file
- **Open Excel:** This method is used to open excel worksheet. OpenFileDialog instance is used to create an instance of the open file dialog box and then ShowDialog is used to Call the ShowDialog method to show the dialog box. Then, Open the selected file if user clicked OK and excel worksheet will open and all the information loaded into the data grid view otherwise if the file not selected the Message will appear "No file was selected".
- **Open Text:** This method is used to open text file. OpenFileDialog instance is used to create an instance of the open file dialog box and then ShowDialog is used to Call the ShowDialog method to show the dialog box. Then, Open the stream and read it back if

user clicked OK and text file loaded into the data grid view otherwise if the file is empty then “NULL” is show in data grid view.

- **LoadReqNumber:** For loading the requirement number from the excel column B, .LoadColumnB is used to load the items to data GridColumns1 by using a for loop which is useful for iterating over array [1, rowIndex]
- **LoadTheEntities**: For loading the requirement number from the excel column C, .LoadColumnC is used to load the items to data GridColumns2 by using a for loop which is useful for iterating over array [2, rowIndex]
- **LoadInitialStates :** For loading the requirement number from the excel column D, .LoadColumnD is used to load the items to data GridColumns3 by using a for loop which is useful for iterating over array [3, rowIndex]
- **LoadFinalStates :** For loading the requirement number from the excel column E, .LoadColumnE is used to load the items to data GridColumns3 by using a for loop which is useful for iterating over array [4, rowIndex]
- **LoadOperators :** For loading the requirement number from the excel column F, .LoadColumnF is used to load the items to data GridColumns4 by using a for loop which is useful for iterating over array [5, rowIndex]
- **Display STD**: In this method, create array and then with the help of array STDDisplay as an image and created as a script.
- **ClearTheDataGrid:** This method is used to clear the data grid of building block.

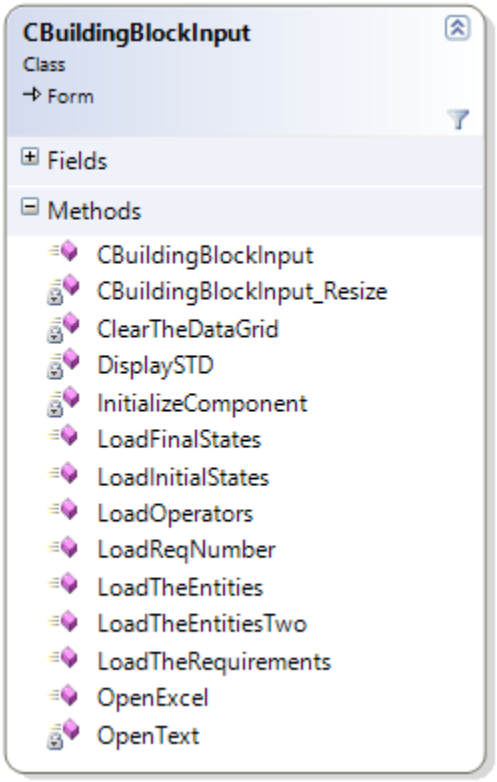



Figure A4. Building Block Input Class Diagram

**2. CExcelAutomation:** This class is used to process information of excel and load information in column of the data grid Figure A5 shows the methods that are used in this class.

- **CloseExcel:** In this method, simple quit command is used to quit the excel sheet.
- **CloseExcelWorkBook:** In this method, call the Close method to close the active workbook
- **GetColumnA to G:** In this**,** gets or fetch the value of the each Column from A to G and return in datagrid column as an array.
- **OpenExcelWorkbook:** This method help to open excel worksheet
- **SetWorkSheet:** This method makes excel sheet visible.
- **LoadColumn:** This method used to load the information from the Column A to G into the data grid column of building block by using for loop which is helpful for iterating an

array and Returns the contents of the specified cell in data grid column of the building block screen available in the system.

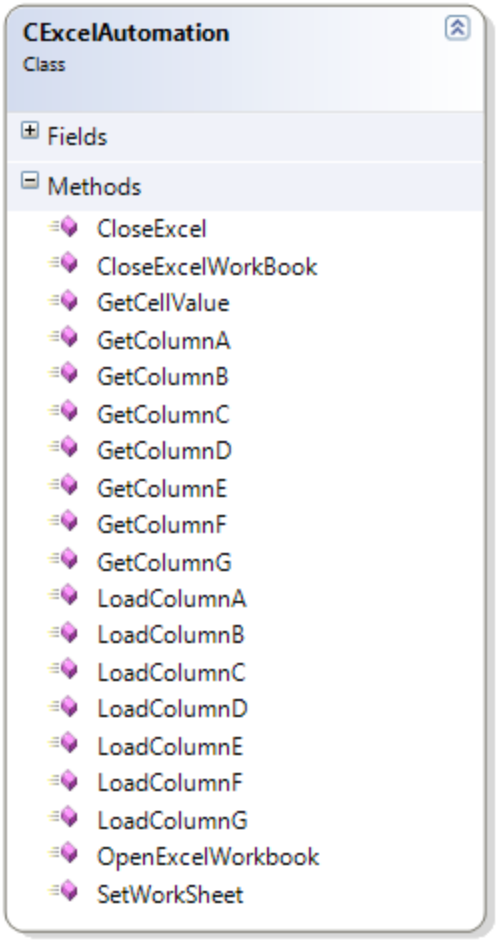



Figure A5. Excel Automation Class Diagram

**3. STDDisplay:** This class is used to process information for STD display from building block data grid to display STD Figure A6 shows the methods that are used in this class.

- **STDDisplay:** This method initializes the Display image.
- **SetTheImage:** This method adds an Image in the Bitmap format.
- **ResizeTheLabelLenght:** In this method, label length is resized means adjust according to label. First, label Initialize as string then it is strip is partitioned in different length and resized value returned.

- **CreateDigraphScript**: Digraph is a graph whose pair is ordered. In this method, when STD is created then all the labels are properly labeled and closs with brackets like (,),{, and }

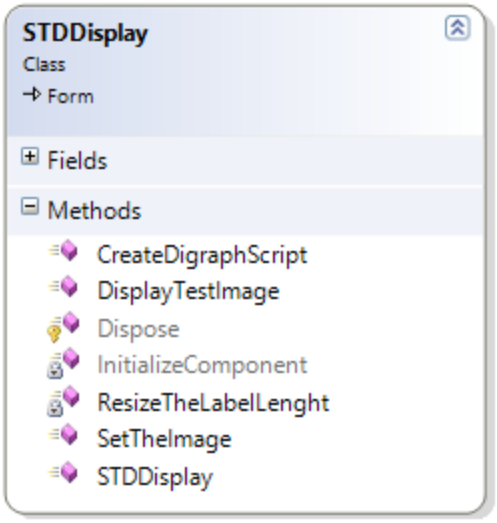



Figure A6. STD Class Diagram

**4. GraphvizApi**: This class consists of all the API of Graphviz which help us to display image of STD form from other classes information Figure A7 shows the methods that are used in this class.

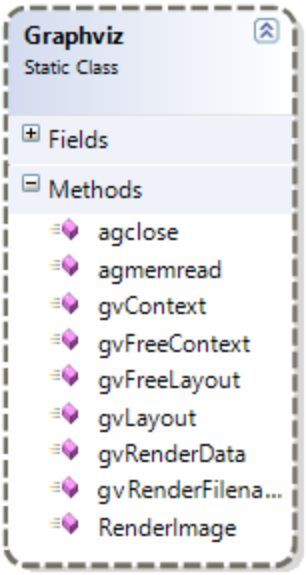



Figure A7. GraphViz Class Diagram

## A.10. Interface Description

This section describes the different interfaces of the System. It would contain a detailed description of each interface along with a screen shot of the Interface.

1. **Home Interface:** The home interface of the application (Figure A8) is common to all the users of the system. This interface shall be made available through the winform application.

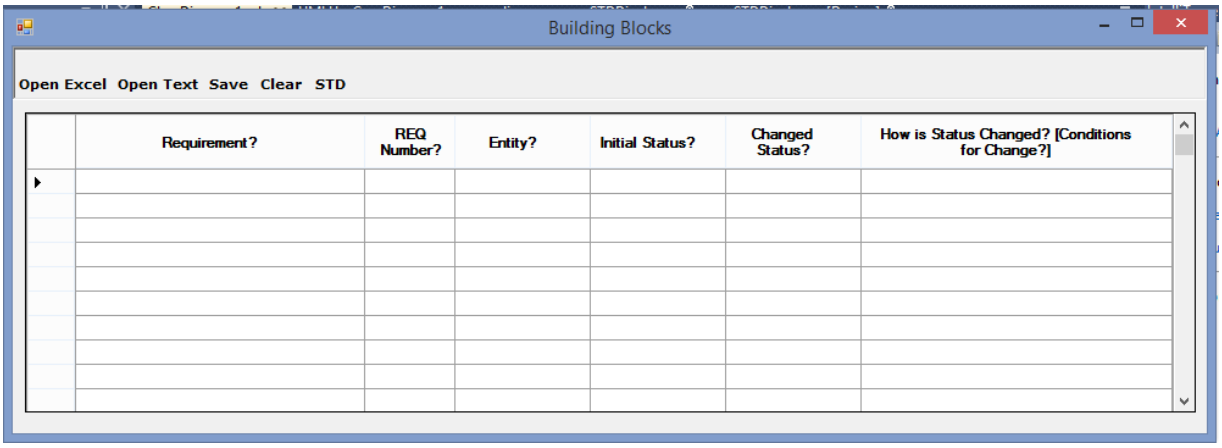



Figure A8. Screenshot of the Home Interface

2. **Open Excel:** Using this interface (Figure A9), the users can upload excel sheet from the initial phase where subjects have created STD-BB from NL. Here, when the subject click on the open excel button in home interface then Open interface popup (see Figure A10.) and then subject can select their document and excel sheet will pop up and information from excel sheet loaded in each column of data grid of Building block screen.

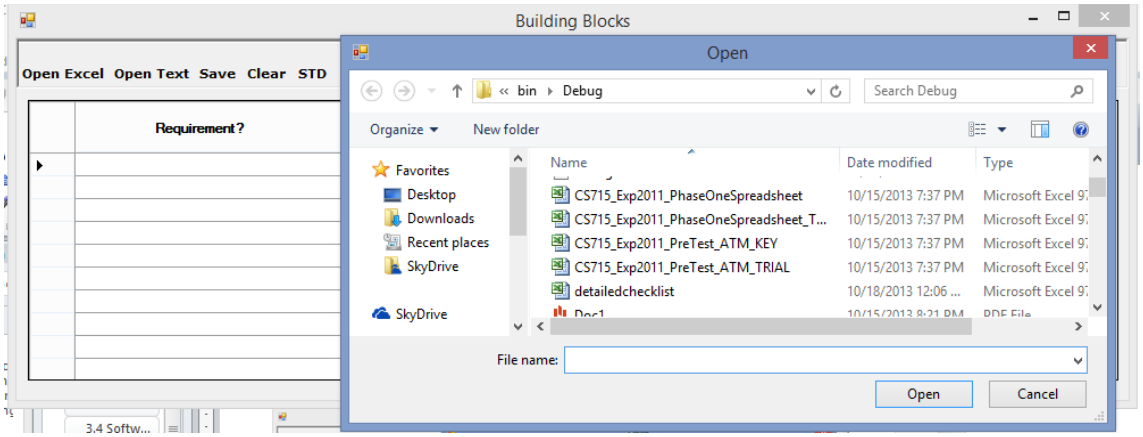


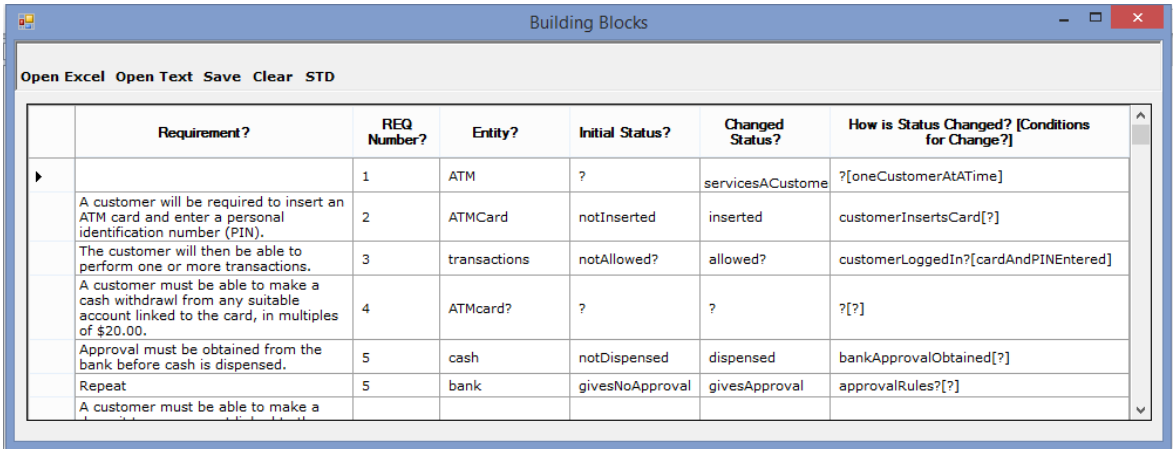

Figure A9. Screenshot of the Open Interface

| | Requirement? | REQ Number? | Entity? | Initial Status? | Changed Status? | How is Status Changed? [Conditions for Change?] |
|---|---|---|---|---|---|---|
| | Repeat | 1 | ATM | ? | servicesACustome | ?[oneCustomerAtATime] |
| | A customer will be required to insert an ATM card and enter a personal identification number (PIN). | 2 | customer | notInsertingcard | insertsCard | customerInsertsCard[?] |
| | Repeat | 2 | customer | notEnteringPIN | entersPIN | customerEntersPIN[?] |
| | Repeat | 2 | ATMCard | notInserted | inserted | customerInsertsCard[?] |
| | The customer will then be able to perform one or more transactions. | 3 | transactions | notAllowed? | allowed? | customerLoggedIn?[cardAndPINEntered] |
| | Repeat | 3 | customer | notAbleToTransact | ableToTransAct | customerLoggedIn?[cardAndPINEntered] |
| | A customer must be able to make a cash withdrawl from any suitable account linked to the card, in multiples of $20.00. | 4 | ATMcard? | ? | ? | ?[?] |
| | A17:NULL | 4 | customer | notAbleToWithdraw | ableToWithdrawl | ?[fromSuitableCardAccount AND inMultiplesOfTwenthy] |
| | Approval must be obtained from the bank before cash is dispensed. | 5 | cash | notDispensed | dispensed | bankApprovalObtained[?] |

Figure A10. Screenshot of the Open Excel

3. **Open Text:** Using this interface (Figure A11), the users can open or upload text file from the initial phase where subjects have created STD-BB from NL. Here, when the subject click on the Open Text button in home interface then Open Interface popup will appear same as Figure A9

and then the subject can select their text file and then the information from text file loaded in the data grid  column in Building block screen.




| | Requirement? | REQ Number? | Entity? | Initial Status? | Changed Status? | How is Status Changed? [Conditions for Change?] |
|---|---|---|---|---|---|---|
| ▶ | | 1 | ATM | ? | servicesACustome | ?[oneCustomerAtATime] |
| | A customer will be required to insert an ATM card and enter a personal identification number (PIN). | 2 | ATMCard | notInserted | inserted | customerInsertsCard[?] |
| | The customer will then be able to perform one or more transactions. | 3 | transactions | notAllowed? | allowed? | customerLoggedIn?[cardAndPINEntered] |
| | A customer must be able to make a cash withdrawl from any suitable account linked to the card, in multiples of $20.00. | 4 | ATMcard? | ? | ? | ?[?] |
| | Approval must be obtained from the bank before cash is dispensed. | 5 | cash | notDispensed | dispensed | bankApprovalObtained[?] |
| | Repeat | 5 | bank | givesNoApproval | givesApproval | approvalRules?[?] |
| | A customer must be able to make a | | | | | |

Figure A11. Screenshot of the Open Text

4. **Clear:** Using this interface, the users can clear information from building block screen by just clicking the Clear button so that new information can be loaded.

5. **Save:** Using this interface, the users can save information from building block screen as text file for future reference by just clicking the Save button.

6. **STD:** Using this interface (Figure A12), the users can create STD from STD-building block screen by just clicking the STD button.

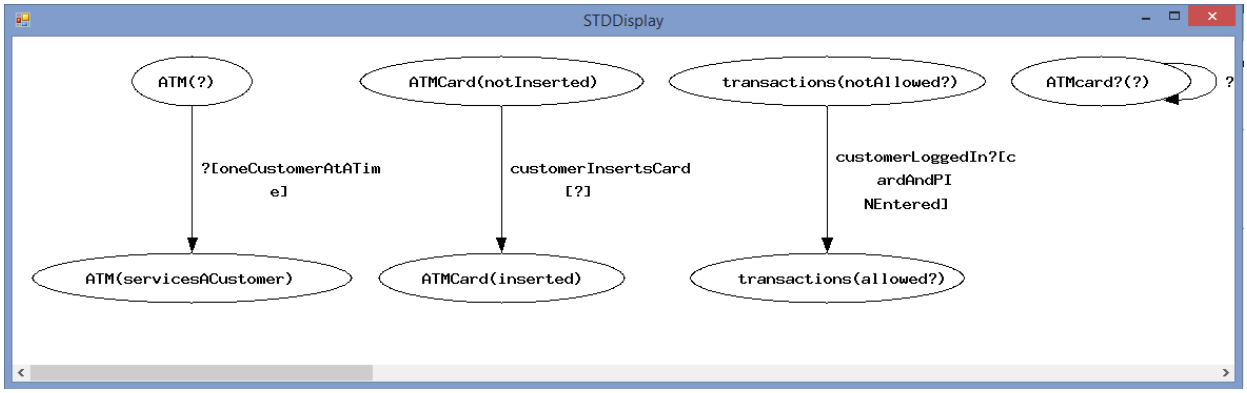



Figure A12. Screenshot of the STD created

7. **Training**: Using this interface (Figure A13), the users can open training presentation of Phase II- NL to STD-BB and Phase III- STD-BBtoSTD. Here, when the subject click on the Training button in home interface then Open interface popup (see Figure A14) and then subject can select presentation for Phase II or Phase III and click **Open** button.

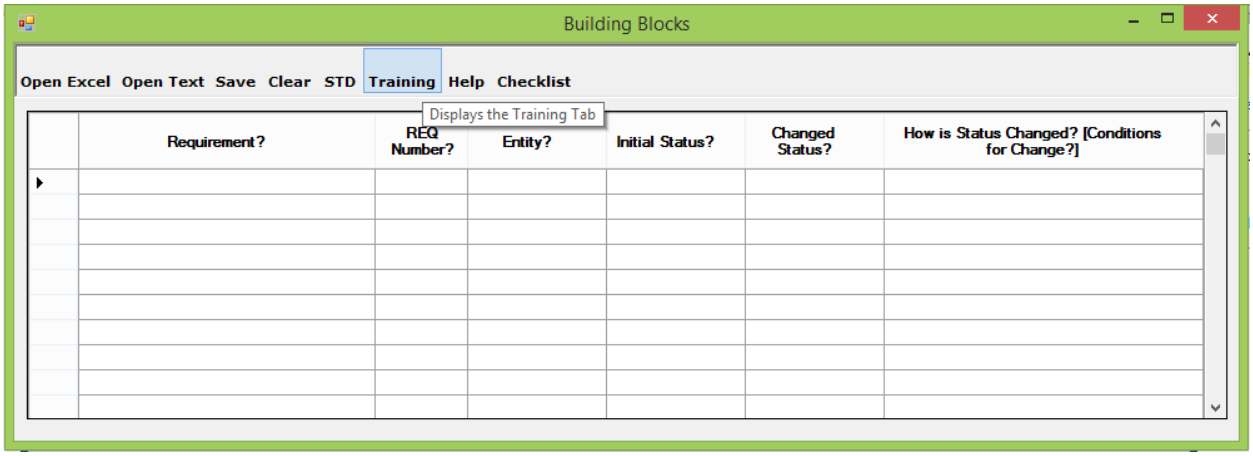



Figure A13. Screenshot of the Training

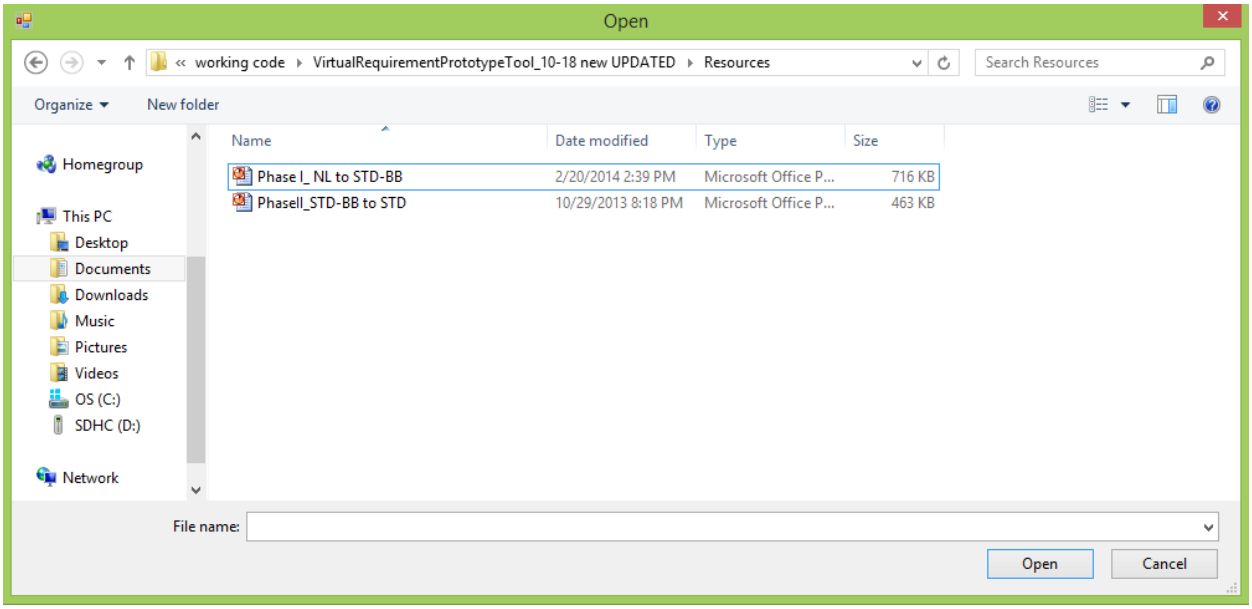



Figure A14. Screenshot of the Open Interface for Training

8. **Help:** Using this interface (Figure A15), the users can open tool help documentation for each button like Open Excel, Open text, Save, Clear, STD, and Training button. Here, when the

subject click on the Help button in home interface then Open interface popup (see Figure A16.) show all the pdf. For button and then subject can select their pdf. and pdf. will open.

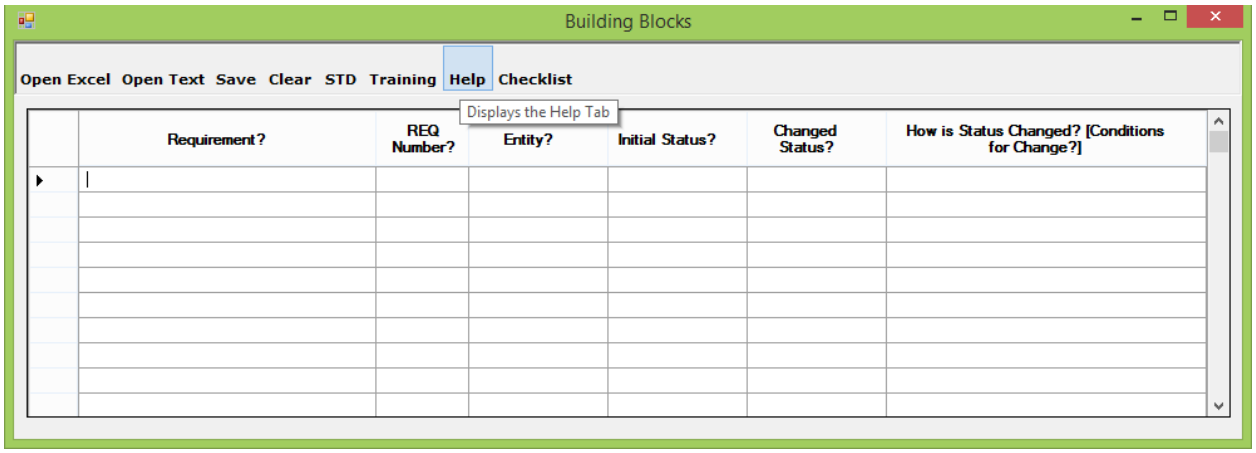



Figure A15. Screenshot of the Help

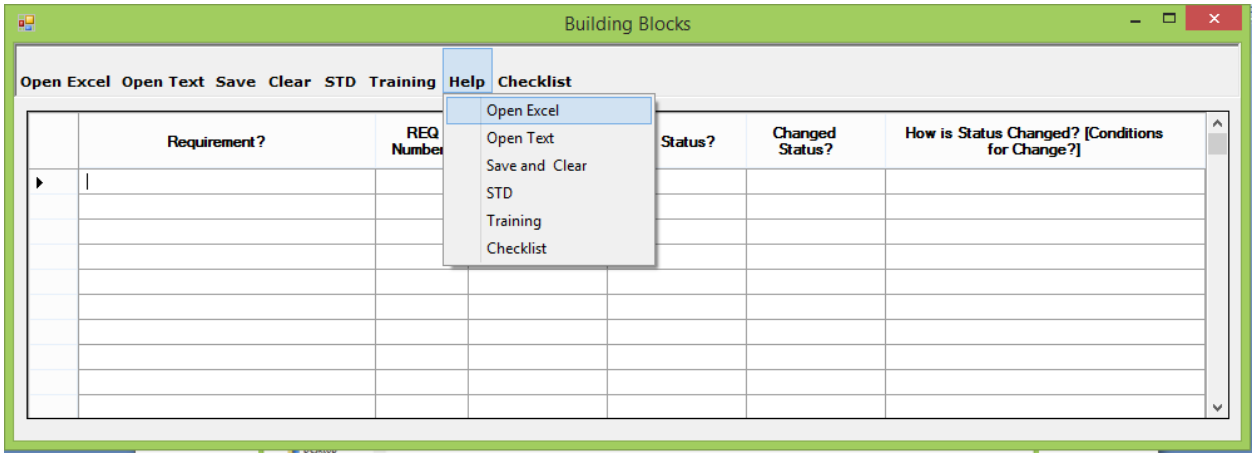



Figure A16. Screenshot of the Training

**9. Checklist:** Using this interface (Figure A17), the users can open Survey checklist after working with tool for Post study Questionnaire so that every subject can rate rated the NLtoSTD method on characteristic of simplicity and ease of use by answered multi-question questionnaire that were based on a 5 point likert-scale. Here, when the subject click on the Checklist button in home interface then Open interface popup show checklist and select open that.

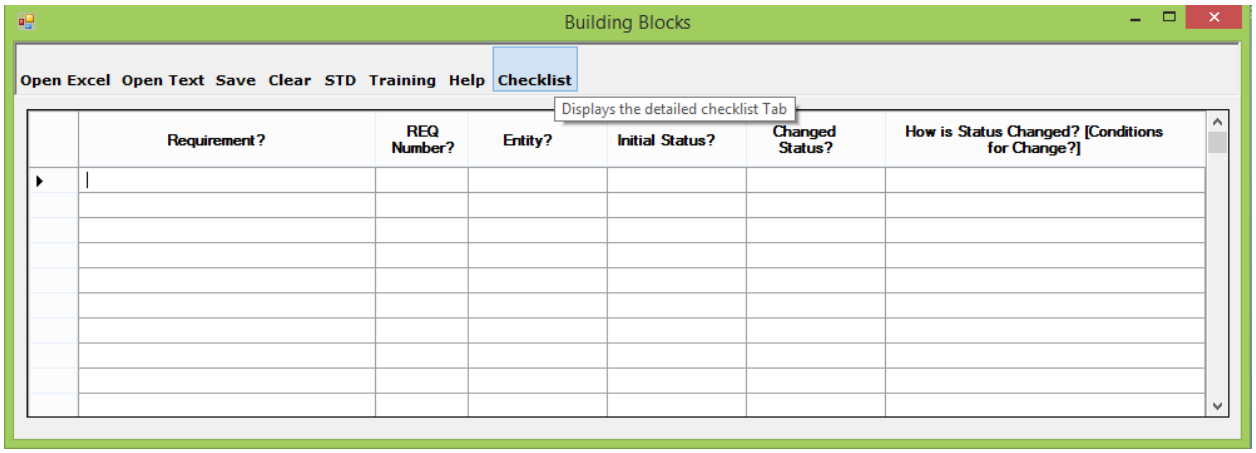



Figure A17. Screenshot of the Checklist

# APPENDIX B. TESTING

This chapter follows up with the testing methods that were used during the validation of the system. The Conclusion and the Future Work for the software are also given.

## B.1. Methodology

The testing method used for validating and verifying this software was different than the conventional testing route followed in the software industry (Binder, 1999) (Bochmann, G.V,. Petrenko, 1994) (Braind, Labiche, 2002) (Antonio Bertolino, 2005). This testing approach was easier and valuable for the software.

In this approach, as the specs were ready for a prototype to be shown, the tester started writing his or her code and saw if he or she could obtain the same results as the specs mentioned. This way, the specs were tested on each prototype, and continuous testing was applied. This also helped in minimizing the testing that would have to be implemented at the end of the software lifecycle. In the process, all aspects of the software were tested. Steps to follow while implementing the methodology:

1. Start with a base functionality that you want to implement.
2. Create a document with the detailed requirement definition, an activity diagram with a description of the flow, database tables that would be used and component diagram and description of each component with precondition and tables that would be affected by the component.
3. Give the document to the tester, and work with the tester while he or she writes the code to check if the steps in the document can be implemented and if the result of each use case can be achieved.

4. If the tester finds a step difficult to implement or thinks he or she is missing additional information to implement the functionality, then go to step 2; otherwise, go to step 3.
5. Ask the tester to log all the errors and difficulties he or she encountered while working on the prototype implementation.
6. Once the prototype is done and the results match between the developer's prototype and tester's prototype, work on the other requirement, and expand the prototype to final software.

When the testing approach was implemented the following pros and cons regarding the testing approach were realized.

1. **Pros of using the methodology:**
    - Helps give a better understanding about the requirements
    - Better design at the end of the cycle
    - Reduced testing to be performed at the end of the cycle
    - Documents produced would be of higher quality.
2. **Cons of using the methodology**
    - The person working on the document should be experienced.
    - There are increased time and money involved in testing.
    - Different viewpoints for the same problem can lead to different results.

## B.2. Interface Testing

This section lists the Product function used for creating the test-case table, the test cases that were used to verify the interface table, and the results for the test-cases table.

Table B1 lists the Product functional for the interface built for the Virtual Requirement Prototype application, along with a short description of each requirement.

Table B1. List of Product Function

| Product Function Number | Product Function Short Description |
|---|---|
| PF01 | The system allows the user to open excel sheet of his choice. |
| PF 02 | The system allows the user to open text file of his choice. |
| PF 03 | The system allows the user to clear the data grid view of building block screen. |
| PF 04 | The system allows the user to save file. |
| PF 05 | The system allows the user to form STD. |

### B.3. Test Cases

The Table B2 below shows the Functional requirements that were used to write the test cases along with the test case numbers for each test case and a short description of the test cases.

Table B2. List of Test Cases

| Product Function No. | Test Case No. | Test Case Short Description |
|---|---|---|
| PF01 | TC01 | To test user can click Open Excel button in home interface |
| | TC02 | To test Open interface as pop up will appear |
| | TC03 | To test user can select the excel document and press Open |
| | TC04 | To test user can select text file instead of excel document and press Open |
| | TC05 | To test data load and display in the data grid from excel sheet |
| PF02 | TC06 | To test user can click open text file |
| | TC07 | To test user can select the text file and press Open |
| | TC08 | To test data load and display in the data grid from text file |
| PF03 | TC09 | To test user can click clear and data from the data grid clear up |
| PF04 | TC10 | To test user can click Save and Save pop up appears |
| PF05 | TC11 | To test user can click STD and screen appear where STD created |
| | TC12 | To test data grid is empty and user can click STD then STD display interface open as popup or not |

The following list include the steps that should be taken by the user, the conditions that should be met for the successful execution of the test case and the end result that should be met for the test cases to pass.

Table B3. Test Case for Open Excel Button in Home interface

| **Test ID** | **TC01** |
|---|---|
| **Purpose** | To test user can click Open Excel button in home interface |
| **Pre-Conditions** | Run the application and home interface appears |
| **Inputs** | Click OPEN Excel button present in home interface tool bar |
| **Post-Conditions** | Open interface as popup will appear |
| **End Messages /Result** | i. If User → Click OpenExcel and popup to open interface then result is True |

Table B4. Test Case for Open Interface as Popup

| **Test ID** | **TC02** |
|---|---|
| **Purpose** | To test Open interface as pop up will appear |
| **Pre-Conditions** | i. Run the application and home interface appears<br>ii. Click Open Excel button in home interface tool bar |
| **Inputs** | Click OPEN Excel button present in tool bar and display Open Interface |
| **Post-Conditions** | In Open interface user able to select excel sheet |
| **End Messages /Result** | i. If openFileDialog open as pop up then result is True<br>ii. If openFileDialog doesn't open as pop up then result is False |

Table B5. Test Case for User can Select the Excel Document

| **Test ID** | **TC03** |
|---|---|
| **Purpose** | To test user can select the excel document and press Open |
| **Pre-Conditions** | i. Run the application and home interface appears.<br>ii. Click Open Excel button in home interface tool bar<br>iii. Open interface as a popup will appear |
| **Inputs** | Select the excel document of their choice and Press Open button to open |
| **Post-Conditions** | Open an excel sheet |
| **End Messages /Result** | i. If user selected the excel document and DialogResult.OK = true and then Press Open button then Excel sheet open then result is True<br>ii. If user not selected the excel document and press Cancel then display "No file was selected" |

Table B6. Test Case for User can Select the Text Document

| **Test ID** | **TC04** |
|---|---|
| **Purpose** | To test user can select text file instead of excel document and press OK |
| **Pre-Conditions** | i. Run the application and home interface appears.<br>ii. Click Open Excel button in home interface tool bar<br>iii. Open interface as a popup will appear. |
| **Inputs** | Select the text file and Press OK button to open |
| **Post-Conditions** | No sheet open |
| **End Messages /Result** | i. If .InitialDirectory path is select and select text file and DialogResult.OK = true then No sheet display then result is True |

Table B7. Test Case Data Display in the Datagrid

| **Test ID** | **TC05** |
|---|---|
| **Purpose** | To test data load and display in the data grid from excel sheet |
| **Pre-Conditions** | i. Run the application and home interface appears.<br>ii. Click Open Excel button in home interface tool bar<br>iii. Open interface as a popup will appear.<br>iv. Select excel sheet and Press OK button |
| **Inputs** | Excel document open up and data load in the data grid of home interface |
| **Post-Conditions** | Data loaded into the data grid of home interface |
| **End Messages /Result** | i. If Excel workbook open then first data from data grid clear up and then information from excel sheet load in the data grid like Requirement, Req Number, The Entities, Initial State, Final State and operators properly then result is True<br>ii. If Excel workbook open then first data from data grid clear up and then information from excel sheet load in the data grid like Requirement, Req Number, The Entities, Initial State, Final State and operators not properly then result is False<br>iii. If Excel workbook open then closed before data load in datagrid properly then result is False |

Table B8. Test Case for User can Click Open Text

| **Test ID** | **TC06** |
|---|---|
| **Purpose** | To test user can click Open Text button in home interface |
| **Pre-Conditions** | Run the application and home interface appears |
| **Inputs** | Click OPEN Text button present in home interface tool bar |
| **Post-Conditions** | Open interface as popup will appear |
| **End Messages /Result** | i. If User → Click OpenText and popup to open interface then result is True |

Table B9. Test Case for User can Select Text File

| **Test ID** | **TC07** |
|---|---|
| **Purpose** | To test user can select the text file and press Open |
| **Pre-Conditions** | i. Run the application and home interface appears.<br>ii. Click Open Text button in home interface tool bar<br>iii. Open interface as a popup will appear |
| **Inputs** | Select the Text document of their choice and Press Open button to open |
| **Post-Conditions** | Data load in datagrid of home interface |
| **End Messages /Result** | i. If user selected the Text document by selecting path of file and Press Open then result is True<br>ii. If user not selected the Text document and Press Open then nothing happen |

Table B10. Test Case for Display Data from Text File

| **Test ID** | **TC08** |
|---|---|
| **Purpose** | To test data load and display in the data grid from text file |
| **Pre-Conditions** | i. Run the application and home interface appears.<br>ii. Click Open Excel button in home interface tool bar<br>iii. Open interface as a popup will appear.<br>iv. Select the Text document of their choice and Press OK button to open |
| **Inputs** | Data load in the data grid of home interface |
| **Post-Conditions** | Data loaded into the datagrid display of home interface |
| **End Messages /Result** | i. If data from Text file read and then information from text file load in the data grid like Requirement, ReqNumber, TheEntities, Initial State, Final State and operators properly and in the end Text file close then result is True |

Table B11. Test Case for Clear Data

| **Test ID** | **TC09** |
|---|---|
| **Purpose** | To test user can click Clear button and data from the data grid clear up |
| **Pre-Conditions** | Data in datagrid are already loaded befor |
| **Inputs** | Click Clear button present in home interface tool bar |
| **Post-Conditions** | Datagrid is empty and ready to load new data |
| **End Messages /Result** | i. If User → ClickClear and all the row and column is empty from datagrid then result is True |

Table B12. Test Case for Save Data

| **Test ID** | **TC010** |
|---|---|
| **Purpose** | To test user can click Save and Save interface appears as pop up |
| **Pre-Conditions** | Data present in datagrid |
| **Inputs** | Click Save button present in home interface tool bar |
| **Post-Conditions** | Data from Datagrid saved for later use as text |
| **End Messages /Result** | i. If saveFileDialog Open as a popup and set the file path to be save and file save as.txt and then close in the end then result is True |

Table B13. Test Case for STD Display

| **Test ID** | **TC011** |
|---|---|
| **Purpose** | To test user can click STD and STD display interface open as popup where STD appears |
| **Pre-Conditions** | Data present in datagrid |
| **Inputs** | Click STD button present in home interface tool bar |
| **Post-Conditions** | STD diagram will appear |
| **End Messages /Result** | i. If STDDIsplay as image show up then result is true<br>ii. If STDDIsplay as image not show up then result is false |

Table B14. Test Case for STD Display when Datagrid is Empty

| **Test ID** | **TC012** |
|---|---|
| **Purpose** | To test datagrid is empty and user can click STD then STD display interface open as popup or not |
| **Pre-Conditions** | Datagrid should be empty |
| **Inputs** | Click STD button present in home interface tool bar |
| **Post-Conditions** | STD diagram will not appear |
| **End Messages /Result** | i. If STD button click then NULL exception is display then result is true |

## B.4. Results

This section lists the results that were produced by running the test cases. The Table B3 below lists the test cases that were used while testing the Interface along with the expected result and the actual results for each test case.

Table B15. List of Test Case Results

| **Test Case Number** | **Expected Result** | **Actual Result** |
|---|---|---|
| TC01 | Pass | Pass |
| TC02 | Pass | Pass |
| TC03 | Pass | Pass |
| TC04 | Pass | Pass |
| TC05 | Pass | Pass |
| TC06 | Pass | Pass |
| TC07 | Pass | Pass |
| TC08 | Pass | Pass |
| TC09 | Pass | Pass |
| TC10 | Pass | Pass |