Information and Software Technology 56 (2014) 321–334

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Model-based requirements verification method: Conclusions from two controlled experiments

Daniel Aceituna [a,1], Gursimran Walia [a,2], Hyunsook Do [a,3], Seok-Won Lee [b,*]

[a] Department of Computer Science, North Dakota State University, IACC 258, 2740, P.O. Box 6050, Fargo, ND 58108-6050, United States
[b] Department of Software Convergence Technology at Ajou University, San 5 Woncheon-dong, Youngtong-gu, Suwon-si, Gyeonggi-do 443-749, Republic of Korea

ARTICLE INFO

Article history:
Received 15 January 2013
Received in revised form 6 November 2013
Accepted 11 November 2013
Available online 22 November 2013

Keywords:
Requirements verification
Model-based verification
NLtoSTD
Fault checklist
Controlled experiments

ABSTRACT

Context: Requirements engineering is one of the most important and critical phases in the software development life cycle, and should be carefully performed to build high quality and reliable software. However, requirements are typically gathered through various sources and are represented in natural language (NL), making requirements engineering a difficult, fault prone, and a challenging task.
Objective: To ensure high-quality software, we need effective requirements verification methods that can clearly handle and address inherently ambiguous nature of NL specifications. The objective of this paper is to propose a method that can address the challenges with NL requirements verification and to evaluate our proposed method through controlled experiments.
Method: We propose a model-based requirements verification method, called NLtoSTD, which transforms NL requirements into a State Transition Diagram (STD) that can help to detect and to eliminate ambiguities and incompleteness. The paper describes the NLtoSTD method to detect requirement faults, thereby improving the quality of the requirements. To evaluate the NLtoSTD method, we conducted two controlled experiments at North Dakota State University in which the participants employed the NLtoSTD method and a traditional fault checklist during the inspection of requirement documents to identify the ambiguities and incompleteness of the requirements.
Results: Two experiment results show that the NLtoSTD method can be more effective in exposing the missing functionality and, in some cases, more ambiguous information than the fault-checklist method. Our experiments also revealed areas of improvement that benefit the method's applicability in the future.
Conclusion: We presented a new approach, NLtoSTD, to verify requirements documents and two controlled experiments assessing our approach. The results are promising and have motivated the refinement of the NLtoSTD method and future empirical evaluation.

1. Introduction

Requirements verification is a process of determining whether requirements specifications capture the desired features of the system being built correctly. This process can be difficult because, typically, requirements are gathered through various sources and are represented in natural language (NL) as a means of communication between different stakeholders (i.e., both technical and non-technical). Requirements written in NL are prone to errors due to the inherent imprecision, ambiguity, and vagueness of natural language. Evidence suggests that if left undetected, these requirement errors can cause major re-work during the later stages of software development (i.e., during the implementation and testing phases). Furthermore, finding and fixing problems earlier rather than later is cheaper, and less expensive [7,36]. To ensure high-quality software, successful organizations focus on identifying and correcting problems in the software artifacts developed during the early stages of the software-development lifecycle.

To ensure requirements quality, to date, researchers have developed various verification methods (Section 7 summarizes the related work.) for detecting and removing the early lifecycle faults (i.e., mistakes recorded in the requirements and design documents [6,8,10,16]) and have validated the methods through controlled experiments and case studies (e.g., [9,11,21,26]). In particular, *software inspections*, in which a team of skilled individuals review a software work-product (e.g., a requirements document or a design document) to identify faults, is an effective verification method. Software inspections have been widely used to help developers identify different types of early lifecycle faults. To aid developers

* Corresponding author. Tel.: +82 31 219 3548; fax: +82 31 219 1621.
E-mail addresses: daniel.aceituna@ndsu.edu (D. Aceituna), gursimran.walia@ndsu.edu (G. Walia), hyunsook.do@ndsu.edu (H. Do), leesw@ajou.ac.kr (S.-W. Lee).
[1] Tel.: +1 701 231 8562; fax: +1 701 231 8255.
[2] Tel.: +1 701 231 8185; fax: +1 701 231 8255.
[3] Tel.: +1 701 231 5856; fax: +1 701 231 8255.

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.11.004

with detecting a larger number of faults during an inspection, researchers have developed variants of Fagan's inspection [16] that range from an ad hoc inspection to a simple, fault-based checklist to a more detailed, step wise abstraction of the artifact.

However, even when faithfully applying these methods, it is estimated that 40–50% of development effort is still spent fixing problems that should have been corrected early in the lifecycle [7]. Much of this rework is the result of the fact that inspection methods rely on the inspectors' abilities to understand the requirements, and often, their interpretations are different from what the requirement developers intended. Because of the flexibility and inherently ambiguous nature of NL specifications, inspectors can have different interpretations of the same requirements without noticing the ambiguities and inconsistencies. Further, as the size of the requirement document grows, this tendency also increases.

Model-based approaches [2,19,20,24] can detect such faults more easily because, when the requirements are modeled or checked with formal methods, the properties, such as inconsistencies and ambiguities, are clearly addressed and handled. For this reason, many researchers have utilized model-based approaches for verifying NL specifications.

While model-based approaches provide a systematic way to identify inconsistent and incomplete requirements, building models often requires NL translation, and this translation process can be highly subjective because stakeholders can interpret NL requirements differently [5,17]. An erroneous translation of NL requirements can result in the wrong model and, thus, can eventually produce software that stakeholders do not want. To alleviate the problem with the erroneous translation, several researchers have proposed modeling techniques using an automated NL translation approach [4,13,15,22]. 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 with manual inspection methods and model-based methods, we propose a new method that translates NL requirements into a State Transition Diagram (STD) in an incremental manner (hereafter referred to as NLtoSTD) and allows requirement engineers and other stakeholders to participate in the translation process. This approach can correct and refine requirements during the translation process by identifying ambiguities and incompleteness in the NL requirements.

The NLtoSTD method provides a means of exposing faults in a set of NL requirements while transforming the requirements into an STD. While the requirement faults have also been explored by other approaches [25,30], our method differs from the existing techniques in that the direct mapping from NL to an STD model is preserved in the translation process. Each NL requirement becomes a segment of the STD, resulting in a direct mapping between the requirements and the STD. This means that any adjustments or changes made to the STD can be directly traced back to the requirements, and vice versa. (A detailed example of this process is shown in Section 2.2.)

To investigate the feasibility and applicability of our approach, we designed and conducted two controlled experiments which evaluated the user's fault-detection ability during the translation of NL requirements (contained in different requirement documents) into the building blocks (or segments) of the STD. These experiments validated the use of translating NL requirements into STD segments as an effective requirements verification method. Our results showed that the proposed method can be improved to make it more effective in exposing the missing functionality and ambiguous information in NL requirements when compared to a fault-checklist method.

1.1. Our contribution

This paper is substantially expanded from the previous published results [37], and the new contributions are highlighted as follows:

(1) *Using more extensive universe of data*: This paper reports the results from a second experiment that used the improved NLtoSTD-BB method (based on the lessons learned from the first experiment) and was designed to address the validity threats unaddressed in the first experiment. Further, we also included new data analysis from the first experiment that has not been reported elsewhere.
(2) *Stronger findings*: We integrated the results of the Data Sets (of different subjects and requirement artifacts) from both the experiments to draw more general conclusions and compared findings against previous findings.
(3) *Additional insights and Future research directions*: We also discuss the further improvement of the latest version of the NLtoSTD-BB (that was used in the second experiment) to highlight the strengths and limitations of the NLtoSTD-BB method during the inspection of NL requirement documents.
(4) Through the experimental results, we were able to highlight the promise of using NLtoSTD-BB as a requirements verification method that is more effective in detecting requirements faults than the traditional inspection method. Our method preserves the direct mapping from requirements to the model during the translation process, which supports traceability between two software artifacts (requirements and the model). Our method allows both technical and non-technical stakeholders participate in the requirement verification process.

The rest of the paper is organized as follows. Section 2 describes our NLtoSTD approach in detail, and Section 3 provides the experimental framework used to evaluate our research approach. Sections 4 and 5 present our two experiments, including the design, threats to validity, data and analysis, and result interpretation. Section 6 discusses the results we observed from the two experiments and the practical implications. Section 7 discusses related work. Finally, Section 8 presents conclusions and discusses possible future work.

2. NLtoSTD method

This section explains the NLtoSTD method and describes the application of the NLtoSTD-BB on a set of example NL requirements for detecting faults.

2.1. Basic concepts underlying the NLtoSTD method

The rationale behind developing the NLtoSTD method was to be able to translate NL requirements into a formalized form, which can facilitate exposing incompleteness and ambiguities in the original NL requirements. An STD is a formal description of the system-to-be behavior, and that 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 (that makes up the STD) 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 consists of two steps. The first step includes the translation of NL requirements into STD-BBs (thereby referred to as NLtoSTD-BB) and the second step includes the construction of an STD using the STD-BBs (referred as STD-BBtoSTD).

A high-level overview of the NLtoSTD method is shown in Fig. 1. During this translation, requirement engineers and stakeholders can readily observe the ambiguities and incompleteness of the NL requirements by examining the individual STD-BBs and the resulting STD. Once the faults are detected, the NL requirements and the STD model can be revised to correct the detected ambiguities and incompleteness. Fig. 1 also highlights the "*NLtoSTD-BB*" translation (the left side box with dotted line) and the "*STD-BBtoSTD*" construction (the right side box with solid line) steps.

Also shown in Fig. 1, the elements that make up an STD-BB (i.e., S1 – *initial state*, T1 – *transition*, and S2 – *final state*) are precisely extracted from each individual requirement during the NLtoSTD-BB translation. Our belief is that this translation can help detect the inherent requirement problems that are otherwise left undetected using traditional inspection methods.

The intent behind the development of the NLtoSTD-BB step was to allow a direct mapping of an individual NL requirement to an STD-BB. The selection of the three elements: (1) current state (*Sc*), (2) next state (*Sn*), and (3) transition (*T*), that make up the STD-BB were based on the examination that 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*) information can cause different interpretations for the same requirement by different stakeholders or inspectors.

Therefore, the NLtoSTD-BB method would force the inspectors to look for these three elements in a NL requirement, thereby ensuring that the requirements are as concise and clear as possible. As shown in Fig. 1, the complete NLtoSTD method also includes the construction of the STD-BBs into a STD as a means to expose inconsistencies and other faults left undetected during the earlier step. The STD's behavior can be simulated by computer in order to expose faults that may not be evident unless the STD is enacted. There is also the potential to examine the STD by automatically producing path traversals and examining those traversals, for desired behavior.

The scope of this research is limited to validating the NLtoSTD-BB step and is discussed as follows.

2.2. Application of the NLtoSTD-BB method

As explained in Section 2.1, the NLtoSTD method converts a set of NL requirements into an STD model by transforming each requirement into an individual STD building block [1]. 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 missing information in the requirements more easily, as compared to inspecting a set of NL requirements.

To facilitate direct traceability between a set of NL requirements and the resulting STD, we transform each NL requirement into an STD-BB by extracting three elements {*Sc*, *T*, *Sn*} from each requirement (as shown in Fig. 1). The resulting building blocks (one per requirement) are then used to construct an STD. The fundamental idea for this NLtoSTD-BB transformation is that a functional requirement should typically describe an entity transitioning from one state to another. For example, the requirement "*The cell phone shall go into a charge mode when the unit is plugged in, when in the warning mode*", would map to the three elements: {*Sc: Warning Mode, T: Unit is plugged in, Sn: Charge mode*}. The unit is described as transitioning (*T*) from Warning Mode (*Sc*) to Charge mode (*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 may ambiguously imply one or more values for *Sc*, *T*, and *Sn*, and thus identifying a value for each element would not be obvious. For instance, the prior requirement may have stated: "*The unit shall go into a charge mode when the unit is plugged in*" Note that *Sc* is not explicitly stated but, rather, implied, assuming the interpreter of the requirement has some idea of when the user may be prompt to plug the phone in. It may also be assumed that a charge mode is achieved whenever the phone is plugged, regardless of a prior state. In any case, the requirement to find *Sc* forces the interpreter of the requirement to question the intent of the requirement. Without having this mechanism, the requirements could be interpreted in a wrong way, thus the engineers could produce an undesired software product. In the resulting STD-BB, questions marks (*???*) are used to denote an element that is not documented. Thus, in this example, we would define the three elements as {*Sc: ???*, *T: Unit is plugged in*, *Sn: Charge mode*}. It may be safe to assume that no particular current state is required, but it still results in an assumption, and should be verified. 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 "warning mode," "Anytime," or both. It is important to document what may not 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, that result in ambiguities.

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

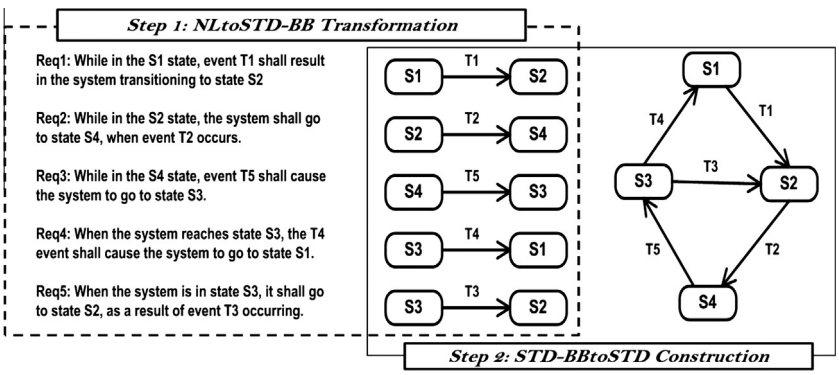

Fig. 1. High-level overview of the NLtoSTD method.

we use the following three questions: (1) What is currently happening? – This question identifies the current state (*Sc*); (2) What will happen next? – This question identifies the next state (*Sn*).; and (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. Fig. 2 also displays how the missing information and ambiguities can be corrected in the NL requirements after the STD-BB's have been constructed. For example, Req1 does not explicitly state *Sn*, resulting in a "*???*" in place of *Sn* in its STD-BB. This, is later corrected by adding the details of the next state.

The ambiguities and incompleteness may not be obvious in the NL requirements, but the process of extracting STD-BBs from these requirements highlight these problems. Requirement engineers and stakeholders can see what the STD-BBs lacks, and together, they can work towards its completion. To correct the STD-BBs and in turn the requirements, requirement engineers and stakeholders would define the implied (*???*) elements, add requirements details, remove requirements details, or do what it takes to remove the incompleteness and ambiguities in NL requirements.

In summary, for our NLtoSTD method, the formal representation (i.e., STD) exposes and, subsequently, corrects the ambiguities and incompleteness in the informal representation (i.e., NL). A key to our method is the manual (vs. automatic) translation of the natural language into a STD-BB, which achieves two important goals. First, the manual translation adds an extra level of user inspection to the process. Second, the automatic-translation processes that we have seen do not result in the same bi-directional traceability between the model and NL that our manual translation produces (e.g., the existing work [13,15,22] uses a multi-step process that diminishes traceability). This traceability is important for our method's ability to correct requirements.

Because this study was an initial feasibility investigation, our research focused on empirically evaluating the NLtoSTD-BB transformation process (the center part of Fig. 2) through a series of two quasi-experiments. The results from these experiments helped improve the process and will be used to evaluate the construction of an STD from the building blocks (the rightmost side of Fig. 1) in future studies.

3. Research evaluation

The major goal of our experiments was to validate the NLtoSTD-BB transformation process as an effective and efficient method of detecting ambiguities and incompleteness in software-requirement documents. We conducted a family of two experiments in order to assess the usefulness of the NLtoSTD-BB method by comparing it against a standard fault-checklist-based requirement inspection method. While these experiments focused on the same research questions, the result from the first experiment helped improved the NLtoSTD-BB translation to be able to improve the performance of the users when extracting the elements needed to build the STD-BBs in order to detect requirement faults. The revised NLtoSTD-BB method was evaluated in the second experiment, whose design was slightly modified based on the unaddressed threats and lessons learned during the first experiment.

Section 3.1 discusses the research questions as well as the independent and dependent variables common to both experiments. Section 3.2 provides a high-level overview of each study along with the major results. More detailed descriptions of each study are presented in Sections 4 and 5, respectively.

3.1. Research questions

Three research questions were investigated in our two experiments.

RQ1: Is NLtoSTD-BB more effective (i.e., the number of faults) at detecting incomplete and ambiguous requirements compared to the fault-checklist based method during inspection of an NL requirement document?
RQ2: Is NLtoSTD-BB more efficient (i.e., faults per hour) at detecting incomplete and ambiguous requirements compared to the fault-checklist based method during inspection of an NL requirement document?
RQ3: Is NLtoSTD-BB viewed to be useful for improving the software quality?

Table 1 provides a list of independent and dependent variables.

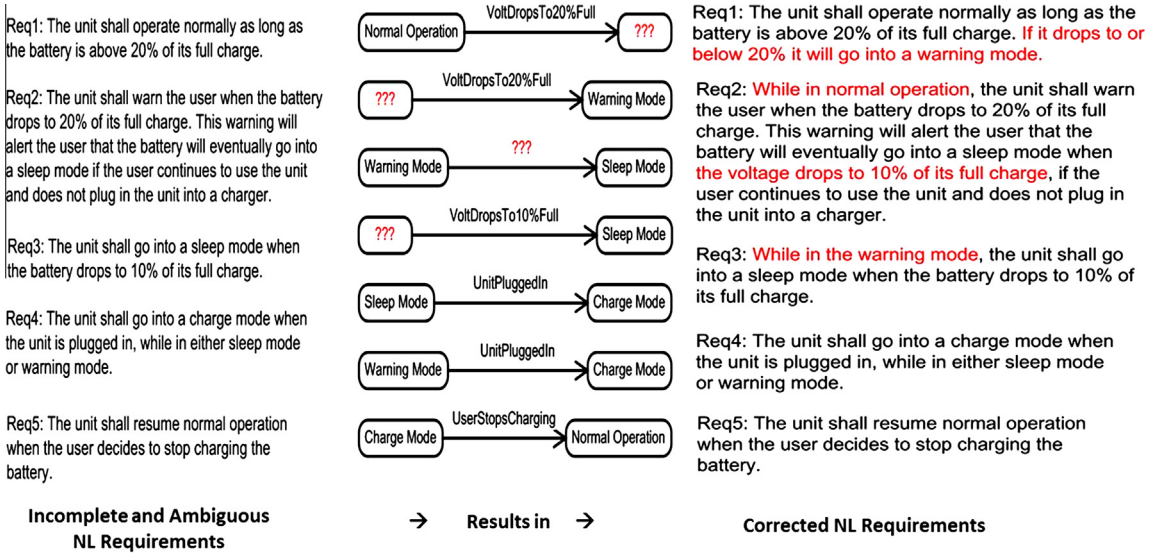

Fig. 2. Translation method reveals the five battery requirements are ambiguous and incomplete requirements.

Table 1
Study Variables.

	Definition
Independent variable	
Inspection technique	Subjects employed the NLtoSTD-BB translation method and the fault-checklist based method to inspect NL requirement documents
Training usefulness	Measures the perceived usefulness of the training procedure for each participant
Effort spent	Amount of time spent during the translation of NL into STD-BBs
Difficulty level	Measures the degree of difficulty the students perceived while performing the experimental tasks
Dependent variable	
Effectiveness	The number of faults found
Efficiency	The number of faults found per hour

3.2. Organization for discussing the two experiments

During the first experiment, the subjects worked in teams to develop the requirement documents for different software systems. Next, each subject individually inspected two different requirement documents (that they did not develop) using the standard fault checklist based inspection method and then using our NLtoSTD-BB method in a repeated-measures fashion. These inspections resulted in a list of faults (incompleteness and ambiguity) for each subject using both methods for all the documents. Next, the recorded faults were evaluated for false positives, and the true faults were analyzed to compare the fault-detection abilities of both methods and gathered feedback to improve our NLtoSTD-BB translation process. The results showed that, during the translation of NL requirements into STD-BBs, the subjects were able to find a larger number of incomplete requirements as compared to the fault checklist based inspection. Also, the results from the experiment helped us revise the NLtoSTD-BB translation process to help improve its effectiveness when applied on NL [37].

A major threat in the first experiment was that it is possible that some of the performance increase when using the NLtoSTD-BB method could have been because the subjects became better at inspecting the NL requirements (even though they were using the different treatment method and inspecting a different document). To address this threat, the second experiment replicated the first experiment by dividing subjects into two groups and adding a complete counterbalancing of the order of treatments. One group inspected an external requirement document using the fault-checklist based inspection method and reinspected it using the NLtoSTD-BB method (revised from the first experiment). The other group of subjects inspected the same document using the NLtoSTD-BB method and reinspected it using the fault-checklist method. The results from this experiment confirmed the NLtoSTD's potential as an effective fault-detection method compared to the fault-checklist based inspection method.

Section 4 briefly discusses the experiment 1 design and summarizes the major results that have been reported earlier [37]. To reduce duplication, some of the experiment design details are presented in Section 4 that are common to both experiments. Section 5 then provides details on experiment 2 design (which is a classic control group experiment), results, and validity threats.

4. Experiment 1: a feasibility study

The goal of first experiment, a repeated measures quasi-experiment [29], was to understand whether the original version (V1.0) of NLtoSTD-BB can be effectively used to detect faults in NL requirement documents. To accomplish that goal, we compared the NLtoSTD-BB V1.0 against the traditional fault-checklist based method in the context of their ability to find faults in the software requirement documents developed by student teams.

4.1. Experiment design

The participating subjects included sixteen (16) students enrolled in the Requirement Definition and Analysis course at North Dakota State University in Fall 2010. During the semester, the students worked in teams (of three or four subjects) to elicit and document the requirements for a different software system (that was selected by the students and agreed upon by the instructor). A total of five different requirement documents (referred as Document A to E) were developed by student teams and the details about them can be found in [37], along with the size (in terms of the number of pages) of the requirement documents.

After the requirements development, each subject individually inspected two different requirement documents (developed by other student teams) using two different inspection methods: the first inspection using the fault-checklist method followed by the second inspection (of a different document) using the NLtoSTD-BB method. The experiment steps, training and output produced during the experiment steps are provided in Table 2.

4.2. Data collection

This section provides a brief description of the *qualitative* and *quantitative* data collected during the experiment run. The quantitative data includes the faults found by participants using the fault checklist technique (i.e., Step 1) and the faults found by participants using the NLtoSTD-BB method (i.e., Step 2). The participants reported the start and end times of the inspection, the time they found each fault, and any breaks they took which were used to calculate the efficiency measures. The qualitative data includes the subjective feedback and ratings (on a 5-point likert scale) on different characteristics related to the usability of the NLtoSTD-BB method.

We validated that the faults reported by each participant were true-positives. We read through the faults reported by each participant to remove any false positives before analyzing the data. While evaluating the fault data from the subjects who inspected

Table 2
Experiment 1 steps, trainings, and outputs produced.

	Description of experiment steps and outputs produced
Training 1 – Fault Checklist	Trained on how to use fault-checklist to find faults and record faults
Step 1 – Inspection using Fault-Checklist	Each participant inspected a requirements document (that they had not developed) to identify faults; ***Output – Individual Fault Lists***
Training 2 – NLtoSTD-BB	Trained on how to use NLtoSTD-BB to document the BB elements, find "Ambiguities" and "Incompleteness" in requirements
Step 2 – Inspection using NLtoSTD-BB	Each subjects inspected a new requirement document (that they had not developed and different from the document inspected in Step 1) to identify faults; ***Output – Individual Fault Lists***
Post-Study Survey	Gathered feedback about NLtoSTD-BB method

Documents D and E using NLtoSTD-BB method, it was found that the subjects were not able to (or did not correctly) differentiate and identify the BBs that resulted in data that could not be relied upon. Therefore, fault data from Documents D and E was not included in the analysis. However, the underlying cause of this unreliable data was linked to a limitation in the NLtoSTD-BB method discussed later in Section 4.1 and how it was overcome.

4.3. Summary of major results

The results are organized around the comparison between the NLtoSTD-BB method and the traditional fault-checklist method in detecting requirement faults.

The results showed that the subjects who were able to correctly extract the STD-BBs were able to find larger number of "incompleteness" in NL requirements when compared to the fault-checklist method. In one case, the subjects using the NLtoSTD-BB method found three times as many missing functionality faults in Document C as compared to the fault-checklist inspection of the same document (an average of 15 faults vs. 5 faults).

Using a 5-point likert scale (ranging from "very low" to "very high", the participants rated the NLtoSTD-BB and the fault-checklist method on eight different attributes related to its usability during the inspection. These attributes include: *simplicity (sim)*, *understandability (und)*, *comprehensiveness (comp)*, *intuitiveness (int)*, *ease of classifying faults (eocf)*, *usability (usa)*, *uniformity across products (uap)*, and *adequacy of faults found (aff)*. Table 3 shows the median value of the ratings for each characteristic for both methods. For each characteristic, we conducted a non-parametric one-sample Wilcoxon Signed Rank test to determine whether the median response was significantly greater than the mid-point of the scale (i.e., 3). The shaded cells in Table 3 represent significantly positive characteristics (i.e., $p < 0.05$). The results showed that the *simplicity, understandability, and usability* attributes were rated significantly positive in NLtoSTD-BB. Regarding the fault-checklist method, only the *comprehensiveness* attribute is rated significantly positive. This result was expected since the fault-checklist method lists all different types of faults (e.g., extraneous information, missing environment, miscellaneous, etc.) whereas the NLtoSTD-BB in this experiment was designed to detect only the missing functionality (MF) and ambiguous information (AI) fault types. None of the attributes were rated significantly negative (i.e., the median value was significantly less than the mid-point of the scale).

Additionally, the traditional checklist method generally has limitations due to the inability of the inspectors to correctly classify the faults found during the inspection. We had hoped to improve this attribute (eocf) through NLtoSTD-BB method, but the results in Table 3 do not show any improvement. To understand this result better, the subjects response to post-study questionnaire revealed that, it was, sometimes, difficult to choose *Sc*, *T*, and *Sn* values when translating NL into STD-BBs for the requirements. Furthermore, analyzing the fault record forms provides insights that the NLtoSTD-BB method has limitations where there are multiple requirements specified for a single functionality, that causes the users to select from multiple candidate values of STD-BB elements (i.e., S_c, T, S_n) causing an incorrect transformation of NL requirements. Based on the analysis of the reported data, the NLtoSTD-BB translation process was improved to be able to make it more effective (discussed in Section 4.3.4).

4.4. Revised NLtoSTD-BB method

From the results of experiment 1, we learned that the NLtoSTD-BB method could provide benefits for developers when they use it correctly. Nevertheless, variations in subject performance prompted us to re-evaluate the way that the three elements, *Sc*, *T*, and *Sn*, are determined from the NL requirements. Further, we questioned whether using only three elements is sufficient to capture and model the description found in a typical NL requirement.

Fig. 3 illustrates the revised NLtoSTD-BB method using an example requirement (i.e., Req2 for the battery charger in Fig. 2). In the revised NLtoSTD-BB method, the following changes were made and are discussed along with their reasoning.

(1) The first change is that we explicitly added 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. We 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. Fig. 3 shows an example of a non-atomic requirement: essentially, it describes three requirements. Based on our new NLtoSTD-BB method, an inspector can identify three entities: unit, battery, and user. Each entity has its own current and next states as shown in Fig. 3. For example, the battery has "20%Level" as the current state and "sleepMode" as the next state.

(2) The second change in the revised NLtoSTD-BB method is allowing users to make an assumption. As shown in Fig. 3, "unit (normalOp?)" has a question mark. This indicates that the "normOp" state is not explicitly stated, but the user can assume that it is the intended state and label it with a question mark for future follow up. This 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 in the future. 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.

Table 3
Experiment 1: Median value of the ratings of the NLtoSTD-BB on different characteristics.

Inspection method	Middle point	1. simp	2. und	3. comp	4. int	5. eocf	6. usa	7. uap	8. aff
NLtoSTD-BB	"Medium" – 3	4	4	3	3	3	4	3	3
Fault-checklist		3	3	4	3	3	3	3	3

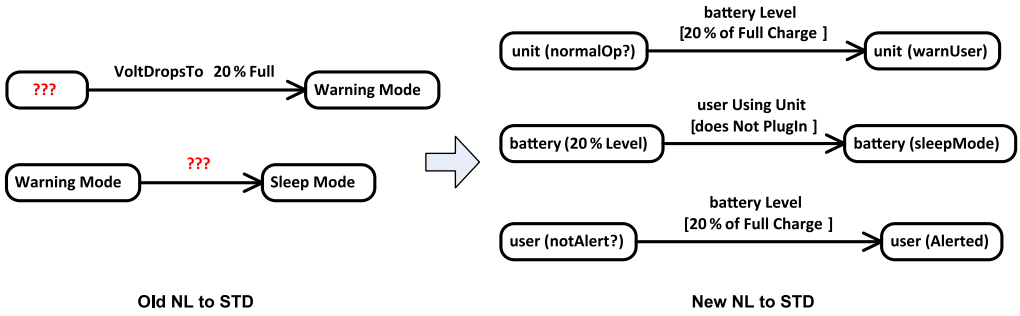

Fig. 3. Example for the revised STD-BB method.

4.5. Threats to validity

We faced the following threats to validity in this experiment.

4.5.1. Conclusion validity

The threat due to the heterogeneity of participants was controlled because all participants were drawn from the same course and had same level of education. While we tried to address the conclusion validity threat by using a 5-point scale (as opposed to a 3-point scale); there remains a threat that we treated these scales as an interval scale rather than ordinal scale, following the standard practice in the social sciences.

4.5.2. External validity

The artifacts inspected in this study contained naturally occurring faults that were inserted while developing the artifacts rather than artificially seeded. However, the artifacts in this study were developed by student teams, and these artifacts may not be representative of an industrial-strength document. Also, the nature of faults made by students can differ from the faults made by professionals.

4.5.3. Internal validity

We reduced the threat due to the learning and maturation effects, by asking the subjects to perform a second inspection using NLtoSTD-BB (that was a new method) on a document that they were reading for the first time. An important internal validity threat was due to the lack of a control group: i.e., we cannot determine the portion of faults found during the second inspection that was due to using the NLtoSTD method and the portion that was found because subjects became experienced at inspecting the requirement document.

5. Experiment 2: a replicated study

Using the revised NLtoSTD-BB method (referred to hereafter as NLtoSTD-BB V2.0), we conducted the second experiment by replicating the first experiment with a different experimental design. This study utilized a repeated-measure design (with a complete counterbalancing of the treatment order) in which the subjects inspected the same NL requirement document (developed externally) using both the NLtoSTD-BB V2.0 and the fault-checklist methods. The experiment occurred over a two-week period, and the subjects were divided into two groups. During week one, a group of subjects inspected the document using the fault-checklist method while the other group of subjects inspected the same document using NLtoSTD-BB V2.0. During the second week, each subject inspected the same requirement document using the other inspection technique and reported new faults that were not detected during the first inspection. 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 study details are provided in the following subsections.

5.1. Study design

Sixteen Computer Science graduate students enrolled in the Software Design course at North Dakota State University during Spring 2011 participated in this experiment. These individuals were predominantly master's and Ph.D. students and had taken a requirement engineering course prior to this study. The Software Design course focused on analyzing design decisions and implementing software designs. During this two-week experiment, the participants inspected a generic NL requirement document describing the requirements for the Loan Arranger Financial System (LAFS) that was created by professional developers at the Microsoft organization. The Loan Arranger system is responsible for bundling loans for sale based on user-specified characteristics. For use in previous studies [35], researchers seeded the artifact with realistic defects.

The subjects used the NLtoSTD-BB V2.0 and fault-checklist methods to find faults in the LAFS requirement document in a similar fashion as they had in the first experiment except for the following changes:

(a) Unlike the first experiment (where the subjects were asked to list all categories for faults found using the fault checklist), the subjects in this experiment were asked to only detect the *incompleteness* and *ambiguities* in the requirements using the fault-checklist method. This was done to objectively compare the efficiency between the NLtoSTD-BB method (that is specifically designed to find only incompleteness and ambiguities) and the fault-checklist method. We believed that this would make comparison fair because more time would have been spent detecting other faults (e.g., inconsistencies and extra functionalities in the requirements).
(b) To evaluate the fault-detection performance of NLtoSTD-BB V2.0 against the fault-checklist method, the repeated-measures experiment was designed with a complete counterbalancing of the treatment order.

As shown in Fig. 4, sixteen participating subjects were "*randomly*" divided into two groups of eight subjects each (Groups A and B). Also, each subject performed two inspections of the same requirement document (LAFS) using different inspection methods (fault checklist and NLtoSTD-BB V2.0). Finally, subjects within Group A and Group B differed in their treatment order for inspection methods during the first and the second inspection cycles.

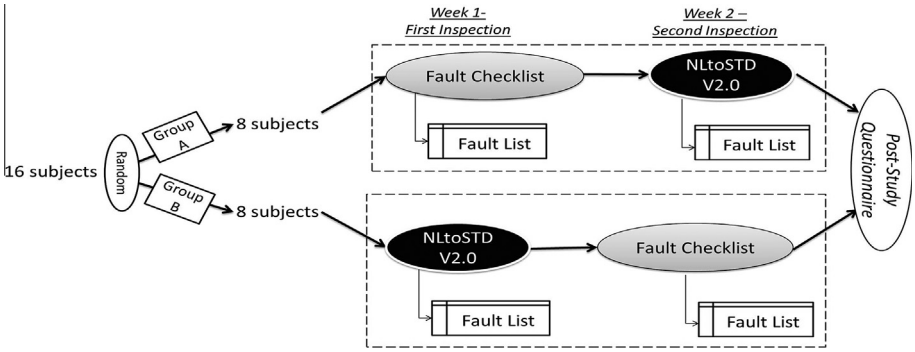

Fig. 4. Procedure for experiment 2.

5.1.1. First inspection cycle

The procedure followed by Group A during the first inspection was as follows:

(1) Training 1: Fault-checklist method: Eight subjects were trained how to use the fault checklist. The fault-checklist training used in this study was same as in Experiment 1, except that the subjects only focused on locating the incompleteness and ambiguities in the SRS (Software Requirements Specification) document.
(2) Step 1: Inspecting SRS for faults using the fault checklist: Using information from Training 1, each subject inspected the requirement document using a fault checklist. This step produced eight individual fault lists (one per subject).

The procedure followed by Group B during the first inspection was as follows:

(1) Training 1′: NLtoSTD V2.0: A second group of eight subjects was trained to use the NLtoSTD-BB V2.0 method. The training was same as in Experiment 1 except that the subjects were trained how to use the revised NLtoSTD-BB V2.0 method to locate faults using examples and a practice problem, and how to record faults.
(2) Step 1′: Inspecting SRS for faults using NLtoSTD-BB V2.0: Using the information from Training 1′, each subject inspected the requirement document using NLtoSTD-BB V2.0. This step produced eight individual fault lists (one per subject).

At the conclusion of the first inspection cycle, the subjects submitted their fault lists. Then, students received training for re-inspection of the LAFS requirement document.

5.1.2. Second inspection cycle

During the second inspection cycle, Group A subjects re-inspected the LAFS using the NLtoSTD-BB V2.0 method, whereas Group B subjects used the fault-checklist method to perform a re-inspection. The procedure followed by Group A during the second inspection was as follows:

(1) Training 2: NLtoSTD-BB V2.0: This training was identical to the Group B subjects during the first inspection (i.e., Training 1′).
(2) Step 2: Inspecting SRS for faults using NLtoSTD-BB V2.0: Using the information from Training 2, each subject inspected the requirement document using NLtoSTD-BB V2.0. This step produced eight individual fault lists (one per subject).

The procedure followed by Group B during the second inspection was as follows:

(1) Training 2′: Fault checklist method: This training was identical to the Group A subjects during the first inspection (i.e., Training 1).
(2) Step 2′: Inspecting SRS for faults using the fault checklist: Using the information from Training 2′, each subject inspected the requirement document using a fault checklist. This step produced eight individual fault lists (one per subject).

At the conclusion of the second inspection cycle, the subjects submitted their fault lists. Participants were given an opportunity to provide feedback using the post-study questionnaire discussed below.

5.1.2.1. Post-study questionnaire. The participating subjects provided feedback regarding the usefulness of the NLtoSTD-BB V2.0 method and the fault-checklist method. Based on their experience, the subjects rated these two inspection techniques on different characteristics and answered some other survey questions.

5.2. Data collection

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

5.2.1. Quantitative data

The quantitative data included the "Incompleteness" and "Ambiguity" faults found by participants using the fault checklist (i.e., during the first inspection by Group A subjects and during the second inspection by Group B subjects). The quantitative data included the same fault types found by participants when using NLtoSTD-BB V2.0 (i.e., during the second inspection by Group A subjects and during the first inspection by Group B subjects). The fault-reporting forms used during the inspections provided the participants with space to indicate timing information, including the start and end times for the inspection, the time they found each fault, and the time they took breaks. One of the paper's researchers validated whether the faults reported by each participant were true positives. The researcher read through the faults each participant reported to remove any false positives before analyzing the data. If any faults were unclear, the researcher clarified them with the participant to accurately determine the fault validity.

5.2.2. Qualitative data

For the qualitative data, using a 5-point Likert scale (ranging from "very low" to "very high"), the participants were asked to rate the NLtoSTD-BB V2.0 and the fault-checklist method on the same eight characteristics used in the first experiment. The participants also rated their ability to find the STD-BBs to help understand the results better using a 5-point scale.

5.3. Data analysis and results

This section describes the results from analyzing the data collected during the experimental run (as described in Section 5.2). An alpha value of 0.05 was used to judge the results' statistical significance. The results are organized in a similar fashion (i.e., around the three research questions listed in Section 3.1).

5.3.1. Fault detection effectiveness (RQ1)

The effectiveness of the inspection methods performed by the subjects in Group A and Group B was compared during the first and second inspections. The effectiveness was calculated by analyzing the total number of faults (i.e., *Incompleteness* and *Ambiguity*) found by each subject.

During the first inspection, the average effectiveness of NLtoSTD-BB V2.0 was higher than the fault-checklist method, finding an average of 11 faults compared to an average of 7 faults. The results from an independent samples t-test did not show a significant improvement in the effectiveness of NLtoSTD-BB V2.0 over the fault-checklist method ($p = 0.03$). The methods used by both groups were equally effective during the second inspection. That is, both groups using the fault checklist and NLtoSTD-BB V2.0 found an average of 6 faults. This result is shown in Fig. 5.

We also compared the effectiveness of NLtoSTD-BB V2.0 and the fault-checklist method separately for the "Missing Functionality (MF)" and "Ambiguous Information (AI)" fault types at each inspection cycle. The results are shown in Fig. 6.

The major observations from Fig. 6 are as follows:

1. Considering the average number of faults found for each fault type, during the first inspection:
   a. NLtoSTD-BB V2.0 was more effective at detecting MF and AI fault types than the fault checklist.
   b. Further, the effectiveness of NLtoSTD-BB V2.0 for detecting the AI faults was better as compared to the MF faults in a requirement document.
   c. However, the result from an independent samples *t*-test showed that NLtoSTD-BB V2.0 was not significantly more effective than the fault checklist for detecting the AI and MF faults in the NL requirement document ($p = 0.33$ for MF fault types, $p = 0.1$ for AI fault types).
2. During the second inspection:
   a. NLtoSTD-BB V2.0 was more effective for detecting AI faults than the fault-checklist method (an average of 4 faults using NLtoSTD-BB V2.0 vs. an average of 3 faults using the checklist).
   b. Both NLtoSTD-BB V2.0 and the fault checklist were equally effective for detecting MF faults.

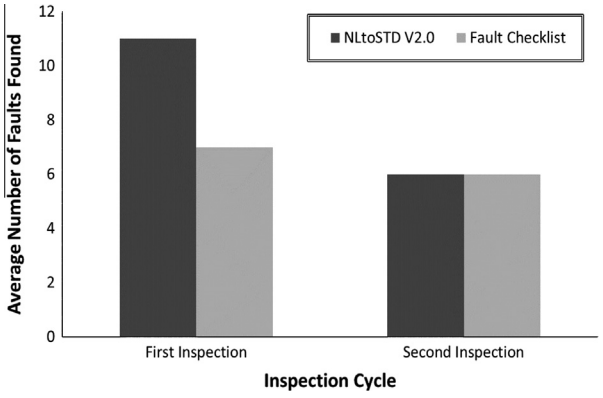

Fig. 5. Effectiveness during the first and second inspections.

Therefore, based on the results from Figs. 5 and 6, NLtoSTD-BB V2.0 outperformed the fault-checklist method in terms of detecting AI faults, MF faults, and total faults (i.e., MF + AI), but the improvement was not statistically significant. Furthermore, the results also showed that NLtoSTD-BB V2.0 was able to detect faults that were not found by subjects using the fault-checklist method. This was concluded after comparing all the faults found by subjects using NLtoSTD-BB V2.0. vs. the fault-checklist during the experimental run. Faults that were found by both inspection methods were not counted in this analysis.

The result also showed that subjects who inspected the LAFS document using the fault checklist found more "unique" faults during re-inspection of the same document using the NLtoSTD-BB V2.0 method (as compared to the group who performed an inspection using NLtoSTD-BB V2.0 followed by a re-inspection using the fault checklist). We also analyzed the false positives (faults which the subjects reported which did not represent a true fault). False positives associated with the fault-checklist method were due to the inspectors' inability to remember all the information they had read. The inspectors often reported false faults that included information they had assumed was either missing (MF) or not clearly defined (AI) in the functional requirements, but was sometimes found in the preceding sections, such as the glossary or purpose. False positives associated with NLtoSTD-BB V2.0 were mainly attributed to a lack of fully understanding how to apply the method, difficulty deciding which entity to use (especially in the case of a lengthy, non-cohesive requirement), or mistaking an operator for state. However, none of the false positives with NLtoSTD-BB V2.0 were due to the incorrect assumptions or being unable to properly comprehend the complete requirement list.

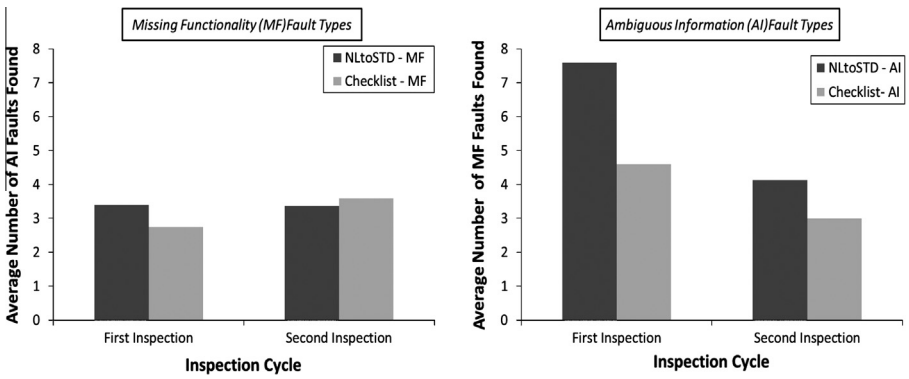

Fig. 6. Average effectiveness during the first and second inspections for MF and AI fault types.

These results support our belief that the nature of the NLtoSTD-BB translation process mitigates the human influence on the inspection results and helps detect problems that can otherwise be left undetected during the fault checklist based inspection process, reducing the false positive overhead associated with inspection results.

5.3.2. Fault detection efficiency (RQ2)

The efficiency (fault/time) was calculated for each inspection cycle (i.e., during the first inspection and the second inspection) and for each inspection method (i.e., NLtoSTD-BB V2.0 and fault checklist) combination. The efficiency comparison is shown in Fig. 7.

The result shows that NLtoSTD-BB V2.0 was more efficient (i.e., found faults faster) than the fault-checklist method during the first and second inspections. During the first inspection, subjects using NLtoSTD-BB V2.0 found an average of 19 faults per hour compared to subjects using the fault-checklist method who found an average of 16 faults per hour. However, the difference in the efficiency values was not statistically significant ($p = 0.49$). During the second inspection, NLtoSTD-BB V2.0 (an average of 10 faults per hour) was slightly more efficient than the fault-checklist method (an average of 9 faults per hour). Therefore, even though the results were not statistically significant, NLtoSTD-BB V2.0 improved the efficiency of the participating subjects compared to the fault checklist for both inspection cycles.

5.3.3. Usefulness of the NLtoSTD-BB V2.0 method (RQ3)

As in the first experiment, the participants used the same 5-point scale to evaluate NLtoSTD-BB V2.0 and the fault-checklist method on the same eight characteristics. Table 4 shows the median value of the responses for each characteristic. Similar to the first experiment, we used a one-sample Wilcoxon Signed Ranks test to see if the ratings were significantly greater than the mid-point of the scale.

The shaded cells in Table 4 highlight those results that were significant for each inspection method. Based on these results, the results showed that the subjects rated the fault-checklist and the NLtoSTD-BB V2.0 significantly positive on all the attributes.

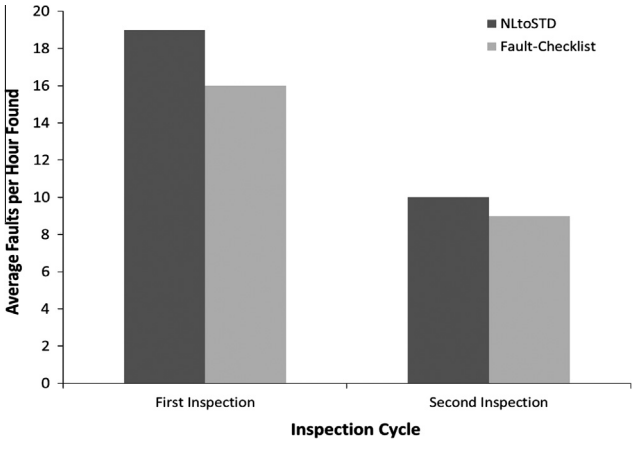

Fig. 7. Efficiency values during the first and the second inspections.

An explanation of the improved result for the fault-checklist method (which was not modified unlike the NLtoSTD method) could be that, the subjects using the fault-checklist were only asked to focus on the MF and AI fault types (which are same fault types for the NLtoSTD-BB method) to able to perform a fair comparison of the efficiency of the methods to find the MF and AI fault types. This could have reflected in the ratings where the subjects found more simple and understandable (as compared to the first experiment results).

Furthermore, a Paired-sample Wilcoxon Signed-Rank test on each pair of rating values revealed significant different between two methods for each characteristic (i.e., $p < 0.05$). Therefore, based on these results, the subjects rated the NLtoSTD-BB V2.0 more favorably than the NLtoSTD-BB v1.0 (Table 2) and reflect the positive changes made to the NLtoSTD-BB method based on results from experiment 1. While the fault-checklist method performance also improved (from the experiment 1 results), the NLtoSTD-BB V2.0 includes more steps (and is more detailed than the fault-checklist) and yet it performed equally well on all the relevant attributes.

Also, we wanted to evaluate the impact of these changes to gain more insight about the usability of the NLtoSTD-BB V2.0 method as an effective fault-detection method. To accomplish that goal, the post-study questionnaire asked the participating subjects to rate the difficulty level for finding the "current state," "transition," "condition," and "next state" of an entity during the translation of NL requirements into STD building blocks. Using a 5-point Likert scale (ranging from 1-very difficult to 5-very easy), the participants rated their difficulty level for each of the four STD-BBs.

Table 5 shows the median value of the ratings for each building block (BB). The result shows that the "*next state*" was easier to find (as compared to the current state) when translating NL requirements to STD-BBs. The numbers in Table 5 substantiate our prior assumptions, namely that it would be easier to derive the next state than the current state from typical requirement verbiage, since a requirement typically describes an action that results in a new system state. For example, for a requirement such as *"The system shall allow a customer to make a deposit,"* the resulting *next state* is explicitly stated and, thus, easier to find.

On the other hand, the "*current state*" was rated as hardest to find when translating NL requirements to STD-BBs. This was due to the fact that the *current state* is typically not explicitly stated in a requirement document. In our prior example, *"The system shall allow a user to make a deposit,"* one must ask the following question: "What state is the system supposed to be in prior to the deposit taking place?" One must assume the *current state* because the requirement does not explicitly state it, leaving it up to a person's interpretation (which can result in ambiguity).

Table 5
Evaluation of the BBs of the revised method: NLtoSTD-BB V2.0.

Building block	Median value
Current state	3
Transition	3
Condition	4
Next state	4

Table 4
Experiment 2: Median value of the ratings of the NLtoSTD-BB on different characteristics.

Inspection method	Middle point	1. simp	2. und	3. comp	4. int	5. eocf	6. usa	7. uap	8. aff
NLtoSTD-BB	"Medium" – 3	4	4	4	4	4	4	4	4
Fault-Checklist		4	4	4	4	4	4	4	4

One of the motivations behind the NLtoSTD's conception was the fact the preconditions are often not explicitly stated in a set of requirements. We feel that the lack of defining preconditions can contribute to the level of ambiguity in a given set of requirements. This is because when preconditions are not explicitly stated, the user is forced to assume the intended precondition. By requiring that a user find the current state of a STD-BB, we are forcing the user to consider whether or not a precondition is being specified. From Table 5, the second most difficult element to find is the transition between the current and next state. This is likely due to a similar reason to that of the current state; the transition is sometimes not obvious from the requirement's wording; it is not always explicitly stated. The user may also be confronted with having to choose from more than one possible candidate for a transition. There is also the problem of how to phrase the transition. All this factored into the decisions made to redefine how the transition is derived, and is explained later in the paper.

5.4. Threats to validity

This section discusses the validity threats that we were able to address and those threats that were unaddressed.

5.4.1. Threats addressed

The threat due to the heterogeneity of participants was addressed because all participants were drawn from the same course and had a similar education level. To address external validity, the inspection artifact was an industrial-strength requirement document that was developed at the Microsoft organization. Also, realistic faults were seeded (rather than naturally occurring) in the LAFS document without any author involvement.

5.4.2. Threats unaddressed

There remains a threat because participants were students in an educational setting and did not represent professional developers. A major threat to the generalizability of the results stems from a relatively small sample size. We plan to address this threat in the future. Also, we have treated 5-point scales as interval data rather than ordinal data. To reduce construct validity problems, we used nonparametric tests, where appropriate, to minimize data assumptions. However, there still remains a threat that we treated the 5-point scale as an interval scale, following the standard practice in the social sciences. This practice means that the p-value needs to be treated with care when interpreting the results.

6. Discussion of results

This section combines the results from the first and the second experiments to answer the original research questions listed in Section 3. The results that are common across both the experiments are used to draw general conclusions.

6.1. Discussion of research questions

Research Question 1: Is NLtoSTD-BB more effective (i.e., the number of faults) at detecting incomplete and ambiguous requirements compared to the fault-checklist based method during an inspection of an NL requirement document?

Through the two controlled experiments, the results showed that the NLtoSTD-BB method can help detect a larger number of incomplete and ambiguous requirements when compared to the fault-checklist method. This increase was even larger for subjects who clearly understood the application of the NLtoSTD-BB method. This was especially evident from the inspection results. Also, the results from the second experiment showed that NLtoSTD-BB helped locate precondition related problems that were otherwise undetected during the fault-checklist based inspection process. These results supported our belief that the nature of the NLtoSTD-BB translation process exposes the human tendency to not explicitly state the precondition (current state) associated with a given requirement. We believe that exposing the lack of stating preconditions will help toward reducing ambiguities that normally find their way into later development phases.

Based on these results, the NLtoSTD-BB method can be *effective* at exposing the incompleteness and ambiguities for NL requirements. In particular, the first experiment demonstrated that it was feasible to expose faults using an approach where a set of NL requirements is formalized into a state transition diagram. The results of the second experiment showed further improvement over the results from the first experiment. The results showed more consistent (across all subjects) improvement for effectiveness when using the NLtoSTD-BB V2.0 method. In particular, NLtoSTD-BB V2.0 was significantly more effective for AI fault type at $p = 0.1$ level than a traditional fault-checklist based inspection method. Some of the effectiveness results are extremely positive, for example, subjects inspecting Document C using the NLtoSTD-BB method during the first experiment found three times the number of MF faults in comparison to the fault-checklist inspection. This motivates us to replicate the findings with larger data sets to make more definite conclusions.

The analysis and the results presented in this paper also contribute to the research community by highlighting the problems faced in the NL translation into the models and how they can be overcome. For example, through the experiment, we showed that the original NLtoSTD-BB method favored the highly cohesive requirements. Accordingly, we proposed a new set of heuristics (in NLtoSTD-BB V2.0) and improved the usability of the method for different types of NL requirements. We identified some further improvements in the NLtoSTD-BB V2.0 and would interest other researchers as we continue this research.

Research Question 2: Is NLtoSTD-BB more efficient (i.e., faults per hour) at detecting incomplete and ambiguous requirements compared to the fault-checklist based method during an inspection of an NL requirement document?

The result from the first experiment showed that the fault checklist was more efficient at finding faults. This was mainly because of the two reasons: (1) the subjects using the fault checklist looked for ten types of faults, whereas the subjects using the NLtoSTD-BB method focused on detecting two fault types (MF and AI); and (2) four (of eleven) subjects using the NLtoSTD method found no true faults due to their misunderstanding of the translation process. However, the results from the first experiment provided us insight to improve the translation process. That knowledge improved the reviewer's efficiency during the second inspection. The results showed that the NLtoSTD-BB V2.0 method was more efficient than the fault-checklist method during the first and the second inspections.

Although the results from the second experiment was promising, they furthered the need to improve NLtoSTD-BB V2.0 in those areas that we felt were hindering the ability to use the method, as it was revealed from the third research question (RQ3).

Research Question 3: Is NLtoSTD-BB viewed to be useful for improving the software quality?

The subjects' responses to the post-study survey show that, in general, the NLtoSTD-BB method is viewed favorably for most attributes. For the first experiment, some subjects reported problems that they faced while choosing the values for *Sc*, *T*, and *Sn*

when translating NL into STD-BBs. Based on the students' responses and feedback, we revised the NLtoSTD method to make it easier to understand and apply to NL requirements. Comparing the results in Tables 3 and 4, the subjects rated the NLtoSTD-BB V2.0 significantly positive on all the attributes in comparison the NLtoSTD-BB V1.0. So, the revisions were justified by the improvement in the responses of NLtoSTD-BB V2.0.

The questionnaire, which addressed RQ3, showed that, while the method used in the second experiment was easier to apply, there was still room for improvement. These changes are described in the next section.

One difference we observed in the results between the first and second experiment was the variation in the number of incompletenesses and ambiguities exposed. The first experiment showed that the method was substantially better at exposing incompletenesses than ambiguities. The second experiment showed that the improvement made to the method enabled it to find a greater amount of ambiguities relative to incompleteness. This was an encouragement because the second experiment suggested that the method's improvement resulted in the approach that can more equally expose incompleteness and ambiguities. Further improvements made to the NLtoSTD-BB V2.0 method are discussed in the following subsection.

6.2. Further improvement to the NLtoSTD-BB V2.0

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. We, therefore, began to ask about the least amount of information we can derive and still have enough to construct building blocks and to model the system's behavior.

To minimize the amount of derived information, we focus 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.

Thus, 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.

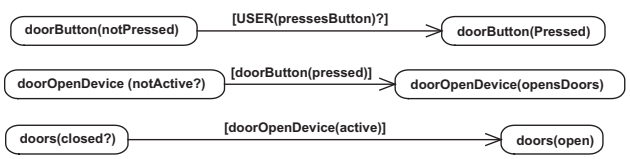

Fig. 8. An example building block from the modifications of experiment 2.

Fig. 8 shows an example of the concept that we just described. The requirement listed at the top of Fig. 8 was converted into three building blocks by simply finding six entity states: *doorButton(notPressed)*, *doorButton(Pressed)*, *doorOpenDevice(notActive)*, *doorOpenDevice(active)*, *doors(closed)*, and *doors(open)*. Note that the entities *doorbutton*, *dooropenDevice*, and *doors* were found by reading the requirement. The final states of *pressed*, *opensDoors*, and *open* were also plainly listed in the requirement. The initial states were deduced as the opposite of the final states. As we can see from this example, operators and constraints were embedded in the entity states. For example, we were able to use entity states as an operator by identifying the causal relationship between the entity states listed in the requirement. In Fig. 8, the entity state *doorbutton(pressed)* causes the entity *doorOpenDevice* to change from the assumed *notActive?*state to the *openDoor* state. (Note that the *USER(pressesButton?)* is not considered part of the system but that it is considered an external event.)

7. Related work

In this section, we mainly discuss two areas of related work: inspection-based verification and model-based verification.

7.1. Inspection-based verification methods

Inspection is one of the widely accepted and used verification techniques because it is relatively easy to apply and because it does not require technical knowledge for stakeholders to use it, especially when using a checklist as a guide. Inspections are also a form of manual static testing that some consider more cost effective than automated testing [16]. Other inspection variations include walkthroughs which involve the key stakeholders getting together to review a requirement document on a line-by-line basis [8,25,32].

Despite its popularity and merits, inspection can be error prone due to human limitations for finding faults in documents describing concurrent, reactive systems. Inspection lists tend to focus the user's attention on one requirement at a time. This can make the verification of a reactive system difficult because a reactive system can involve many state transitions that often occur concurrently. Thus, it can be problematic when trying to verify all the possible functional interactions when inspecting one requirement at a time, which is the way inspectors typically review requirements. With large, manual inspection, methods can fall under three categories: Ad Hoc (reviewing a document with no well-defined criteria), Checklist (using a well-defined criteria), and scenario-based [8] (compiling system-to-be scenarios and verifying that those scenarios can be enacted by the requirements under inspection). Experiments have been conducted to determine the relative effectiveness of Ad Hoc, Checklist, and Scenario-based inspection approaches [30]. Such experiments point to the scenario-based approach as being the most effective of the three [31], however, even with

scenarios, there is still the challenge of verifying a requirement set that specifies concurrent behavior.

Typically, there are two areas of concern that can be addressed while inspecting a requirement document, assuming that an inspector is consciously looking for them and that these two areas of concern have been addressed by various researchers [33,34]: operational context and pre/post conditions. The operational context addresses the fact that a system cannot be accurately specified without considering the context in which it will be used [33]. Reactive systems, such as embedded systems, must be able to function correctly within the environment in which they are meant to operate. This operating environment forms the operational context of the system to be. To correctly address this, operational context must be included with the requirement document, and during the inspection process, context should be considered. The pre/post conditions are also an important aspect to consider because reactive systems act deterministically [34], given a user-input set or external asynchronous events (such as unpredictable interrupts). Their deterministic response is also affected by preconditions. Post conditions are important when trying to assess how the system responds to a negative situation.

While the two aforementioned areas of concern can be addressed during manual inspection, they do add some elements that an inspector must consider while inspecting a requirement document. This means that there are more things for an inspector to examine, which can be more easily addressed if one takes the approach of converting the NL requirements into a model.

7.2. Model-based verification methods

Modeling has been accepted as a fundamental activity throughout the requirement engineering processes [13,22,23]. Typically, the models are "virtual" in the sense that they exist in the computer as mathematical or logical representations of the requirements. Some of the "virtual" models that have been used with embedded systems include Petri Nets, which can model concurrent systems [12], and model checking, which can model temporal characteristics of a system [3]. Another method involves analyzing NL requirements (in the form of scenarios) with computational linguistics and generating Message Sequence Charts (MSC) [19]. The MSC charts are then used for verification purposes [19]. Similarly, Sutcliffe et al. [27,28] present a method that converts Use Case Diagrams into scenarios semi-automatically and validates scenarios using rule-based frames that detect incomplete/incorrect event patterns.

MSCs are interactive diagrams that belong to the Specification and Description Language (SDL). They are similar to the sequence diagrams found in UML, which others have used to model requirements [40]. They have become a popular means of specifying scenarios, which describe interactions between objects in a system. They are regarded as useful early in the development stage [1]. An approach by Damas et al. [13,14] addresses the problem about how to automate a modeling process using scenarios collected from end users. Another target model that has been used is the Object Oriented Analysis Model (OOAM). One approach uses a tool that automatically creates OOAMs for NL requirements that have been rewritten in a constraint language that facilitates the conversion process [24]. Other approaches use a conceptual model (an ontology) [20,39], and goals in an extended version of a Label Transition System (LTS) [38].

Building models often requires NL translation, and this translation process can be problematic due to the inherent incompleteness and ambiguities of NL [5,17]. To address this problem, researchers have proposed various NL to model translation approaches. These methods include approaches based on translating goals to state machines [13], scenarios to state machines [22], and NL to UML [15]. Automation can certainly reduce human errors and improve the translation process, but complete automation of this process is not possible because, often, NL requirements can be interpreted in multiple ways, thus human judgment is inevitable to lead to correct/sensible interpretations.

7.3. Other verification methods

The determination of incompleteness and ambiguities can also be achieved by various techniques beyond the standard review checklist [8]. Linguistic analysis tries to address the domain-knowledge communication problem between domain experts and software engineers [6,10]. Consistency checking [18] is an automatic means of fault detection used with the Software Cost Reduction (SCR) tabular notation. It detects faults such as type errors, missing cases, and circular definitions [18].

8. Conclusions

We have presented a new approach, NLtoSTD, to verify requirements documents and two controlled experiments assessing our approach. Our results show that the NLtoSTD-BB method can be more effective in exposing the missing functionality and, in some cases, more ambiguous information than the fault-checklist method. Our experiments also revealed areas of improvement that benefit the method's applicability in the future as we described in Section 6.

Our future work includes experimentation using the improved method suggested by the results from the second experiment. We also plan to perform experiments that include the construction of an STD from the transformation process's building blocks so that we can evaluate all the benefits that the NLtoSTD method offers. In the future, we wish to automate as much of the heuristics as possible, including the NLtoSTD building block portion: automatically determining the entities and their states. 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.

Ultimately, we would like to achieve a method that provides the advantages associated with having people involved in the verification/validation process while retaining the advantages of automated reasoning with the subsequent requirement model. The human interaction would allow the involvement of non-technical stakeholders whose contributions are more within the context of the domain knowledge.

Acknowledgments

This work was supported, in part, by NSF Awards CNS-0855106 and CCF-1050343 as well as NSF CAREER Award CCF-1149389 to North Dakota State University. This work was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2013M3C4A7056233) to Ajou University.

References

[1] D. Aceituna, H. Do, S. Lee, A human interactive approach to building requirements models, in: International Symposium on Software Reliability Engineering, fast abstract, 2010.

[2] R. Alur, A. Chandrashekharapuram, Dispatch sequences for embedded control models, in: 11th IEEE Real-Time and Embedded Technology and Applications Symposium, vol. 11, 2005, pp. 508–518.

[3] R. Alur, C. Courcoubetis, D. Dill, Model checking for real-time systems, in: Proceedings of 5th Symposium on Logic in Computer Science, 1990, pp. 414–425.

[4] R. Alur, K. Etessami, M. Yannakakis, Inference of message sequence charts, in: Software Concepts and Tools, 2003, pp. 304–313.
[5] D. Barry, Ambiguity in natural language requirements documents, Lecture Notes in Computer Science, LNCS 5320 (2008) 1–7.
[6] D. Berry, E. Kamsties, Perspectives on Software Requirements, Kluwer Academic Publishers, 2004.
[7] B. Boehm, V. Basili, Software defect reduction top 10 list, IEEE Computer (January) (2001) 135–137.
[8] B. Brykczynski, A survey of software inspection checklists, ACM Software Engineering Notes 24 (1) (1999) 82–89.
[9] J. Chaar, M. Halliday, I. Bhandari, R. Chillarege, In-process evaluation for software inspection and test, IEEE Transactions on Software Engineering 19 (11) (1993) 1055–1070.
[10] F. Chantree, B. Nuseibeh, A. de Roeck, A. Willis, Identifying nocuous ambiguities in natural language requirements, Requirements Engineering (2006) 59–68.
[11] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Wong, Orthogonal defect classification – a concept for inprocess measurements, IEEE Transactions on Software Engineering 18 (11) (1992) 943–956.
[12] L.A. Cortes, L. Alej, R. Corts, P. Eles, Z. Peng, Verification of embedded systems using a petri net based representation, in: Proceedings of 13th International Symposium on System, Synthesis, 2000, pp. 149–155.
[13] C. Damas, B. Lambeau, A. van Lamsweerde, Scenarios, goals, and state machines: A win-win partnership for model synthesis, in: Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, November 2006, pp. 197–207.
[14] C. Damas, B. Lambeau, P. Dupont, A. Lamsweerde, Generating annotated behavior models from end-user scenarios, IEEE Transactions on Software Engineering 31 (2005) 1056–1073.
[15] D. Deeptimahanti, M. Babar, An automated tool for generating UML models from natural language requirements, Automated Software Engineering (2009) 680–682.
[16] M. Fagan, Advances in software inspections, IEEE Transactions on Software Engineering 12 (7) (1986) 744–751.
[17] D. Gause, User DRIVEN design – the luxury that has become a necessity, in: Proceedings of 4th International Conference on, Requirements Engineering, 2000.
[18] C. Heitmeyer, R. Jeffords, B. Labaw, Automated consistency checking of requirements specifications, ACM Transactions on Software Engineering and Method 5 (3) (1996) 231–261.
[19] L. Kof, Scenarios: identifying missing objects and actions by means of computational linguistics, in: Proceedings of 15th International Requirements, Engineering Conference, 2007, pp. 121–130.
[20] L. Kof, R. Gacitua, M. Rouncefield, P. Sawyer, Ontology and model alignment as a means for requirements validation, in: International Conference on Software Engineering, 2010, pp. 46–51.
[21] M. Leszak, D.E. Perry, D. Stoll, A case study in root cause defect analysis, in: Proceedings of the 22nd International Conference on, Software Engineering, 2000, pp. 428–437.
[22] E. Letier, J. Kramer, J. Magee, S. Uchitel, Monitoring and control in scenario-based requirements analysis, in: Proceedings of the 27th International Conference on, Software Engineering, 2005, pp. 382–391.
[23] B. Nuseibeh, S. Easterbrook, Requirements engineering: a roadmap, in: Proceedings of the Conference on The Future of Software Engineering, 2000, pp. 35–46.
[24] D. Popescu, S. Rugaber, N. Medvidovic, D. Berry, Reducing ambiguities in requirements specifications via automatically created object-oriented models, in: Monterey Workshop on Computer Packaging, 2007, pp. 103–124.
[25] A. Porter, L. Votta, Comparing detection methods for software requirements inspections: a replication using professional subjects, IEEE Transactions on Software Engineering 21 (1995) 563–575.
[26] S. Sakthivel, Survey of requirements verification techniques, ACM Transactions on Software Engineering and Method (1991) 68–79.
[27] A. Sutcliffe, N. Maiden, S. Minocha, D. Manuel, Supporting scenario-based requirements engineering, IEEE Transactions on Software Engineering 24 (12) (1998) 1072–1088.
[28] A. Sutcliffe, M. Ryan, Experience with SCRAM, a scenario requirements analysis method, Requirements Engineering (1998) 164–171.
[29] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.
[30] A.A. Porter, L.G. Votta, An experiment to assess different defect detection methods for software requirements inspections, in: International Conference on Software Engineering (ICSE), 1994, pp. 103–112.
[31] A.A. Porter, L.G. Votta, V.R. Basili, Comparing detection methods for software requirements inspections: a replicated experiment, Transactions on Software Engineering (TSE) 21 (6) (1995) 563–575.
[32] G.M. Schneider, J. Martin, W.T. Tsai, An experimental study of fault detection in user requirements documents, ACM Transactions on Software Engineering and Method 1 (2) (1992) 563–575.
[33] X. Franch, A. Maté, J. Trujillo, C. Cares, On the joint use of i* with other modelling frameworks: a vision paper, in: IEEE 19th International Requirements, Engineering Conference, August 2011, pp. 133–142.
[34] S. Uchitel, R. Chatley, J. Kramer, J. Magee, Goal and scenario validation: a fluent combination, Requirements Engineering Journal (Springer) 11 (2) (2005) 23–137.
[35] J. Carver, N. Nagappan, A. Page, The impact of educational background on the effectiveness of requirements inspections: an empirical study, IEEE Transactions on Software Engineering 34 (6) (2006) 800–812.
[36] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.
[37] D. Aceituna, H. Do, G. Walia, S. Lee, Evaluating the use of model-based requirements verification method: a feasibility study, in: International Workshop on Empirical Requirements Engineering (EmpiRE), August 2011, pp. 13–20.
[38] E. Letier, W. Heaven, Requirements modeling by synthesis of deontic input–output automata, in: International Conference on Software Engineering, May 2013, pp. 592–601.
[39] N. Innab, A. Kayed, A.S.M. Sajeev, An ontology for software requirements modeling, in: International Conference on Information Science and Technology, 2012, pp. 485–490.
[40] L. Liu, X. Zhu, Y. Wang, Software maintainability requirements modeling based on UML Profile, in: International Conference on Prognostics and Health Management, 2012, pp. 1–4.