9-2014

Automatic early defects detection in use case documents

Shuang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Yue ZHANG

Bimlesh WADHWA

See next page for additional authors

Automatic Early Defects Detection in Use Case Documents

Shuang Liu[1], Jun Sun[2], Yang Liu[3], Yue Zhang[2],
Bimlesh Wadhwa[1], Jin Song Dong[1] and Xinyu Wang[4]

[1]National University of Singapore
[2]Singapore University of Technology and Design
[3]Nanyang Technological University
[4]ZheJiang University

ABSTRACT

Use cases, as the primary techniques in the user requirement analysis, have been widely adopted in the requirement engineering practice. As developed early, use cases also serve as the basis for function requirement development, system design and testing. Errors in the use cases could potentially lead to problems in the system design or implementation. It is thus highly desirable to detect errors in use cases. Automatically analyzing use case documents is challenging primarily because they are written in natural languages. In this work, we aim to achieve automatic defect detection in use case documents by leveraging on advanced parsing techniques. In our approach, we first parse the use case document using dependency parsing techniques. The parsing results of each use case are further processed to form an activity diagram. Lastly, we perform defect detection on the activity diagrams. To evaluate our approach, we have conducted experiments on 200+ real-world as well as academic use cases. The results show the effectiveness of our method.

Categories and Subject Descriptors:.
D.2.1 [SOFTWARE ENGINEERING]: Requirements/Specifications

Keywords:.
Natural language processing; Use cases

1. INTRODUCTION

Use cases are the main technique for understanding user requirements, which have been widely adopted in the modern software development life cycle over the last two decades. Each use case describes a sequence of interactions between a software system and an external actor such that the actor is able to achieve some goal. Collectively, use cases are used to define all the necessary system activities that have significance to the users. As use cases are developed during a very early stage of the software development life cycle, they also serve as the basis for developing detailed functional requirements, help in design development and validation, system testing and maintenance. High quality use case documents can improve the sustainability of software.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642969.

Use case documents are usually written in natural languages, which may inevitably introduce defects like inconsistency, redundancy and incompleteness. Moreover, those defects are hard to identify or verify due to their informal format. In the current practice, use case analysis is conducted manually, e.g., requirement analysts manually extract analysis models (e.g., state machine, activity diagram) from use cases, and search for defects in the models or validate them against test cases. Manual analysis is hardly ideal as it requires a lot of human efforts and is often error-prone. As a result, use cases are much less useful than they could or should be.

There are existing works on automatic analysis of use cases [7, 10, 6]. But still, we identify the following challenges which have not been addressed satisfactorily. Firstly, actual use case documents are often larger and more complex than those have been reported in existing works [6]. For large use case documents, the diversity of grammar rules and ambiguities presented in the document raise great technical challenges in automatic "understanding" them. Secondly, common problems in use cases are inconsistency and incomplete flows. Existing approaches have so far mainly focused on analyzing inconsistency problems [7, 6] and leave incomplete flows unconsidered. Lastly, some existing approaches (e.g., [10, 6]) rely on users to provide use-case-specific templates for parsing, which is ad-hoc and may require knowledge about shallow parsing techniques.

In this work, we are motivated to provide automatic techniques to identify defects in use case documents during the early stage of system development. We attempt to answer the following research questions. (1) How can we automate the process of extract useful information from use case documents as much as possible? (2) Can we formally define the common defects in use cases and develop systematic methods to find those defects? (3) How can we generalize the method to handle use case documents from different domains, even when the documents are written by different people? We contribute in the following three aspects. (1) We explore dependency parsing technique to help understand use case documents. We provide 8 rules based on general English grammar to analyze the dependency parsing results. (2) We formally define common consistency and integrity related defects in use cases and provide algorithms to automatically check those defects. (3) We conduct experiments with use case documents of 5 different systems from different application domains. The results show that our method can achieve good accuracy in analyzing sentences from different domains as well as in finding defects.

Related Work Xiao et.al. [10] extracted Access Control Policies (ACP) from software descriptions and checked the validity of the policies against the use case steps within the same document. Semantic patterns for ACP sentences are provided based on the manually defined verb phrases and POS tags obtained from shallow

Use Case 1: Receive the order with special group
Initiating Actor: Trader
Pre-Conditions
1. The order is legal.
Main Flow
41. GSYS accepts the symbol of order.
42. Check the order.
43. If the order is legal, record values of the group.
44. Find the constraint in the system according to the group name.
45. Save the order into database.
46. Price the order.
47. During the processing, it could create matches only when the constraints are permitted. For example, no match should be created if there is not enough cash in the group.
48. This ends the use case.
Alternative Flow
In step 3, if there is no such constraint in the system, the system will reject this order.
Post-Conditions
1. Order with special group is received by the system.

Figure 1: Example of Use Case Description

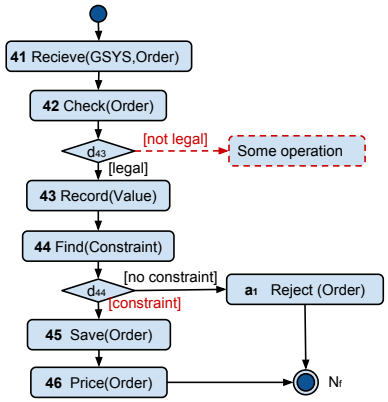

Figure 2: Example Activity Diagram

parsing. Gervasi and Zowghi [6] proposed to uncover inconsistencies in natural language use case descriptions with formal reasoning techniques. Propositional logic formulas are adopted to represent facts, hypotheses and constrains, which are extracted from natural language descriptions. Sinha et.al. proposed an analysis engine and a prototype tool Text2Test [7] for use cases. Shallow parsing techniques are used to analyze natural language sentences. Domain-specific knowledge is required for annotating concepts and context information. The works [10, 6, 7] depend on document-specific information, such as keywords, writing styles, etc. We explore to use dependency parsing technique, which provides richer syntactic information and enables adjusting rules without any domain specific information to be provided. Thus our approach is more adaptive. Tan et.al. [8] focus on detecting inconsistencies between program comments and the source code. Zhong et.al. [12] propose to infer resource specifications from API documentations to detect resource manipulation order problems. This kind of approaches handle natural language with limited patterns. Templates are usually used to extract information from those natural language sentences.

2. PRELIMINARY

This section introduces the use case document template adopted in our approach, and the UML activity diagram.

2.1 Use Case Document Template

There is no standard template for writing use case documents as concluded by Fowler [5]. The choice of use case styles may be highly project-dependent as affected by factors such as the criticality and the number of people involved. It is recommended that for small projects, a simple, casual use case template [4] can be chosen. For large, life-critical projects, it is more appropriate to use a hardened, fancier and fully-addressed template [4]. We focus on fully-addressed use cases since they are usually adopted by large, life-critical projects. There is no universally adopted fully-addressed use case template. However, it has been reported by Cockburn [4] that "the readers almost universally select the single-column, numbered, plain text, full sentence form". Therefore, in this work we focus on this most popular writing style in literature. Figure 1 is one use case in a stock trading system document[1] which follows roughly the Cockburn style [4]. *Note that our work does not aim at handling all the possible writing styles of use cases. We are rather interested in investigating advanced NLP techniques to aid defects detection in use case documents.* The issues caused by different writing styles can be tackled by providing more robust pre-processing steps. We formally define the concepts involved in use case descriptions below.

DEFINITION 1 (ACTION). *An action is defined as a tuple* $A \triangleq (vb, sub, obj)$*, where* vb, sub, obj *are natural language phrases representing the main verb, subject and object of the sentence.*

For example in Figure 1, the action tuple of the first sentence in main flow section is $(check, \_, order)$ (The subject is missing in an imperative sentence).

DEFINITION 2 (PREDICATE). *A* predicate *is defined as a tuple* $P \triangleq (ar, R, a_1, a_2)$*, where* $ar \in \{1, 2\}$ *is the arity of the predicate;* R *is the relation symbol of the predicate;* a_1 *and* a_2 *are the arguments of the relation symbol.*

The predicate can be monovalent or divalent, depending on the structure of the sentence. Predicates of higher arity are not used very frequently. Therefore we do not consider predicates with more than two arities in our work. To gain an intuitive view, a monovalent predicate $(1, is\_legal, order)$[2] can be generated from the sentence in the pre-conditions section in Figure 1.

DEFINITION 3 (SENTENCE). *A* sentence *is defined as a tuple* $S \triangleq (s\#, \alpha, c, n_s, n_j)$*, where* $s\#$ *is the sentence number in the corresponding section of the use case;* $\alpha \in A$ *is the action of the sentence;* $c \in P$ *is the guard condition for executing the sentence;* $n_s \in N$ *and* $n_j \in N$ *are the logical previous and succeeding sentence of the current sentence respectively.*

For example the alternative flow sentence in Figure 1 corresponds to the following sentence structure: $(a_1, (reject, system, order), (2, is\_no, there, constraint), 3, -1)$. The number 3 indicates that the current alternative flow step starts from main flow step 3. -1 indicates that there is no explicit assigned step after the current step, then the flow goes to the next neighboring step.

DEFINITION 4 (USE CASE). *A* use case *is defined as a tuple* $UC \triangleq (UCName, PreC, PostC, MF, AF)$*,* $UCName$ *is the name of the use case;* $PreC \subset P$ *and* $PostC \subset P$ *are the predicates extracted from sentences in the pre-condition and post-condition sections;* MF *and* AF *are the list of sentences* S *in the main flow and alternative flow sections of the use case.*

[1]This is a real system used for real-time stock trading in the amount of billions. The document is provided by our industry collaborator. We omit sensitive keywords due to the confidentiality.

[2]We use underline to replace spaces.

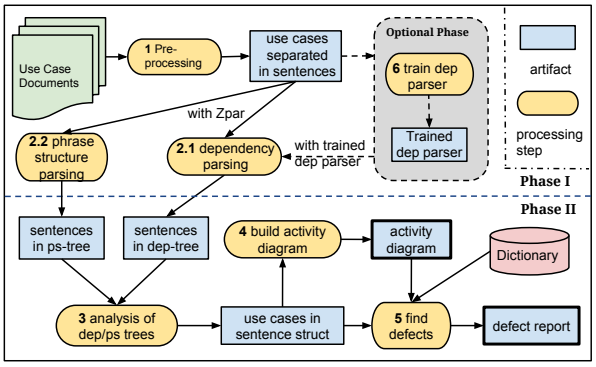

Figure 3: Overview of our approach

2.2 UML Activity Diagram

UML Activity Diagrams [2] are commonly adopted to describe the event flows in use case documents for the purpose of coordinating low-level behaviors.

DEFINITION 5 (ACTIVITY NODE). *An* activity node *is defined as* $N \triangleq N_a \cup N_c$ *where* $N_a \triangleq (Nm, \alpha)$ *is action node and* $N_c \triangleq (Nm, t)$ *is the control node.* Nm *is the name for each node.* $\alpha \in A$ *is the action associated with the action node.* $t \in \{decision, final, initial\}$ *is the type of the control node.*

In Figure 2, the rounded rectangles are action nodes. The diamond, enriched circle and solid circle represent the choice node, final node and initial node respectively. They are control nodes.

DEFINITION 6 (ACTIVITY EDGE). *An* activity edge *is defined as* $E \triangleq (sn, tn, g)$*, where* $sn \in N$*,* $tn \in N$ *and* $g \in P$ *are the source, target nodes and the guard condition of the activity edge.* The guard condition for an activity edge must be satisfied in order to fire the corresponding edge.

DEFINITION 7 (ACTIVITY DIAGRAM). *A UML activity diagram is defined as* $AD \triangleq (ADName, PreC, PostC, AN, AE)$*, where* $ADName$ *is the name of the activity diagram.* $AN \subset N$ *and* $AE \subset E$ *are the set of activity nodes and activity edges in the diagram.* $PreC \subset P$ *and* $PostC \subset P$ *are the pre-conditions and post-conditions of the activity diagram.*

In this work, we consider a subset of UML activity diagram features which are related to control flows. The features which capture object flows are not considered since our defects detection methods utilize only the control flow information.

3. OUR APPROACH

The overview of our approach is illustrated in Figure 3. The rectangles represent artifacts that are produced as (intermediate/final) processing results. The ellipses represent the processing steps. Our method consists of two phases. In the first phase, we take a use case document as input and parses each sentence in the document into parse trees (dependency tree and phrase structure tree). The second phase takes parse trees as input and generates a UML activity diagram for each use case. Afterwards, defects in the use cases are checked. The output of our method includes the UML activity diagrams and a defect report where all defects with horizontal links to the original document are listed. There is also an optional phase as enclosed in the dashed lined area. It provides a way to train a domain-adaptive dependency parser to improve the accuracy of dependency parsing. We discuss our approach in this section.

3.1 Pre-processing Use Case Documents

This step is conducted to filter noises from the input document so as to improve the accuracy of the dependency parser. We removes the irrelevant information and formatting symbols, such as

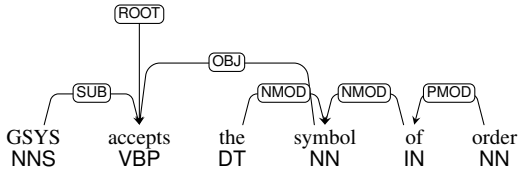

Figure 4: Example of a dependency tree

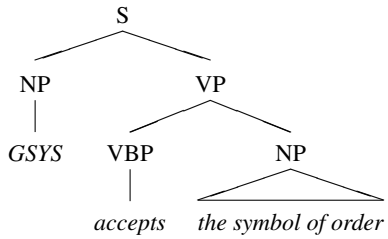

Figure 5: Example of a phrase structure tree

parenthesized comments and bullets, which may affect the parsing accuracy. The output text satisfies the following conditions.

(1) Each sentence is stored in a separate line.
(2) Each punctuation is preceded by a space.
(3) Step index number is stored in a separate line.
(4) Parenthesis are replaced by "–LBR–" or "–RBR–".
(5) There is no empty line in the document.

3.2 Free Text Parsing

In this work, we leverage on ZPar [11], a statistical dependency and phrase structure parser, for analyzing syntactic information.

Dependency Parsing The dependency parser (Step 2.1) is used to extract bootstrap information for action tuples. It conducts statistical analysis on POS tags based on a large data set, which guarantees that it provides more general results than directly analyzing POS tags based on the templates extracted from the sample document. The dependency parsing technique can also provide richer syntactic details, i.e., the dependency relation between pairs of words, which provide the subject, object and main verb information of a sentence directly. The output format of the dependency parser is a dependency tree. Figure 4 shows the dependency tree for the first sentence in the main flow section in Figure 1. The middle row is the original sentence (in tokenized words). The last row is the Part-Of-Speech (POS) tags of the corresponding words. The labeled links are dependency relations between two words. For example, the link from the word "GSYS" to the word "accepts" labeled with SUB represents that "GSYS" is the subject of "accepts". The word "accepts" is the ROOT, i.e., the main verb of the sentence.

Phrase-structure Parsing The phrase structure parser (Step 2.2) is used to identify the modified/supplement information. It provides a parse tree in which sentences are parsed into noun/verb phrases and sub-sentences based on the subordinating/modification relation, thus provides complete context information for an identified word. The parsing result is a phrase structure tree as shown in Figure 5. The leaf nodes are the plain text tokens. The non-leaf nodes are POS tags, where "S", "VP", "NP" represents a sentence, verb phrase and noun phrase respectively. The phrase structure tree is used in combination with the dependency tree in our analysis phase to obtain more accurate results. For example in Figure 4, we identify that the object is "symbol" from the dependency tree. The phrase structure tree in Figure 5 provides the complementary information that the "symbol" is an attribute of the "order". This kind of attributive information is useful in comparing action tuples.

Table 1: Rules to Extract Action Tuples

	Rules
Main Verb	(1) *HAVE* {ROOT}+(*NOT*)+*BE* +(JJ&)+ VB& (2) *BE* {ROOT}+(*NOT*)+ (JJ&) + VB& (3) *BE* {ROOT} +(*NOT*)+ JJ& {PRD} (4) *MODAL* {ROOT}+(*NOT*)+*BE*+(JJ&)+ VB& (5) ROOT {PI=-1}
Subject	(6) $PI.L$ = ROOT{SUB}
Object	(7) $PI.L$ = ROOT{OBJ/PRD/PRP} (8) $PI.L$ = ROOT{SBAR/ VC/ VMOD}

Table 2: Templates to Extract Condition Predicates

Templates
EX*/NN*/PRP*[a_1] MD*VB* (*not/no*)[R] (DT*JJ*)NN*/PRP*[a_2] (.)*
EX*/NN*/PRP* [a_1] MD*VB* (*not/no*) VB*[R] (.)*
EX*/NN*/PRP* [a_1] MD*VB* (*not/no*) (DT) NN*/JJ* [R] (.)*

3.3 Analyzing Parse Trees

The goal of analyzing parse trees is to extract the sentence structure S as defined in Definition 3. The sentence step number ($s\#$), step start (n_s) and join (n_j) nodes are extracted based on keyword matching. The methods to extract the action tuple α and the condition predicate c are discussed as follows.

Extract Action Tuples An action tuple, as defined in Definition 1, contains a subject, an object and a main verb of a sentence. These parts are immediately available in a dependency tree. For example, in Figure 4, the dependency labels SUB, OBJ and ROOT indicate that the subject, object and main verb are "GSYS", "symbol" and "accepts" respectively. However, dependency parsing suffers from a common problem of natural language parsing, i.e., fragile to ambiguities and noises. Thus relying only on the dependency labels may not provide good accuracy due to the deviations in the dependency trees caused by the diversity of sentence patterns, tenses and subordinate structures. To improve the parsing accuracy, we (1) provide 8 adjusting rules based on general English grammar. (2) rely on the phrase structure parsing result to identify related context information. The rules, shown in Table 1, are general in the sense that they are based on natural language grammars and do not contain any document-specific patterns or key words. There are 3 kinds of information, i.e., plain text, POS tags and dependency label, used in our rules. The plus symbol + composes constraints for consecutive words. We use braces {} to represent compulsory information when more than one kind of information is used on one word. Brackets () are used to represent optional information and the slash / symbol is used to represent a choice among the candidates. For example, rule (7) requires that the parent of the word should be labeled as ROOT and the word itself should be labeled as OBJ or PRD or PRP , both constrains are compulsory.

Extract Condition Predicates We notice that sentences which contain conditions are often complex, e.g., with sub-clauses. The dependency parser is likely to produce a random dependency tree, especially for the condition sub-clause. We also notice that the condition sub-clauses are usually written in simple formats. Therefore we extract the condition predicates through template matching. For example, to process the alternative flow sentence in Figure 1, our method first truncates the condition-containing sub-clause, i.e., "if there is no such constraint in the system". The sub-clause matches the first template in Table 2, and condition predicate $(2, is\_no, there, constraint)$ is obtained.

3.4 Building Activity Diagram

The main idea to build an activity diagram is to link action tuples with control flow information. There are two kinds of control flow indicators. One is the control flow information, such as "go/jump to step #", that we identified by analyzing the content of a sentence. The other is the structure of use cases, i.e., consecutive sentences in each section of the use case represent the ordering in the control flow. Sentences in the alternative flow section are the branch flows of sentences in the main flow section. We build an activity diagram for each use case based on the identified information in step 3. Figure 2 shows the activity diagram that is generated from the use case in Figure 1 by our approach. The action node is labeled with the step number and the action tuple extracted from the corresponding sentence. The decision node (diamond) is labeled with the step number of the sentence in which it is generated. The guards, edges and nodes in dashed line in figure 2 represent the missing flow step that our method detected. The main flow step labeled 47 in Figure 1 does not have a corresponding action node in the activity diagram since it does not describe an action step, thus is removed during the activity diagram building procedure.

3.5 Formal Definition for Use Case Defects

To guide our analysis towards finding defects in use cases, we formally define the defects originally proposed by Törner et.al. [9] and develop algorithms to systematically find them. Specifically, we focus on defects which are objective and have high defect intensity. The defects we focus on are defined below.

Inconsistent step numbering captures the situation where the sentence numbers of main flow or alternative flow are not consistent. This may lead to incorrect step referencing. For example in Figure 1, the step number 3 is missing in the main flow. As a result, the alternative flow has referred to a non-existing main flow step.

DEFINITION 8 (D_1). *Given $uc \in UC$, if $\exists s, (uc.MF.cont(s) \vee uc.AF.cont(s)) : (s.n_s \neq NULL \wedge \neg uc.MF.cont(s.n_s) \wedge \neg uc.AF.cont(s.n_s)) \vee (s.n_j \neq NULL \wedge \neg uc.MF.cont(s.n_j) \wedge \neg uc.AF.cont(s.n_j))$, the use case is said to have inconsistent step numbering defect. The function $cont()$ checks whether the item is a member of the list.*

In some use cases, the starting step (in main flows) of the alternative flow is not clearly specified. This may lead to ambiguity when merging the alternative flows with the main flow.

DEFINITION 9 (D_2). *Given $uc \in UC, if \exists s, uc.AF.cont(s) : s.n_s = NULL$, then the use case contains the unclear alternative flow starting step defect.*

An overly-strong precondition is one such that inconsistencies between the precondition and the guard conditions of an edge may occur. For example in Figure 1, the sentence in the precondition has already restricted the order to be legal, thus the second sentence of the main flow, which checks the validity of the order, is redundant.

DEFINITION 10 (D_3). *The pre-condition of an activity diagram ad is overly-strong if given an activity diagram $ad \in AD$, $\forall prec \in ad.PreC, \exists e \in ad.AE : Conflict(e.g, prec)$, where $Conflict$ is a function deciding whether two predicates conflict.*

Missing of alternative flows is the case when the main flow defines some action under some specific condition, however not all the other possible conditions are addressed. For example in Figure 1, step 43 in the main flow specifies the condition "if the order is legal". But it is not specified what if that condition does not hold.

DEFINITION 11 (D_4). *Given an activity diagram $ad \in AD$, $\forall n \in ad.N$, if $n \in N_c \wedge n.t = decision \wedge OutG(n) = 1$. The use case contains the missing alternative flow defect. $OutG(n) \triangleq |e \in ad.E \wedge e.sn = n|$ returns the number of edges out going from a given node. $||$ is the cardinal number operation on a set.*

Algorithm 1: Check Unnecessary Strong Precondition

Input : ad: activity diagram
Output: whether find an over strong precondition or not

```
1  let n be the initial node
2  while There are unvisited nodes in ad do
3      mark n as visited
4      if n ∈ N_a then
5          if InChangeStatusDict(n.α.vb) then
6              return false
7      if n ∈ N_c ∧ n.t = decision then
8          if guard condition of any edges outgoing from n is Conflict with any
           p ∈ ad.PreC then
9              Report an over strong precondition defect
10             return true
11     set n to the next node following the edge in ad
12 return false
```

3.6 Finding Defects

Following the definitions in Section 3.5, we discuss each defect finding method in details in this section.

Inconsistent step numbering (D_1) is checked based on the use case structure. We check all the sentences in MF and AF sections of a use case. If we find an n_s field referring to a sentence that is not in $MF \cup AF$, an inconsistent step numbering defect is reported.

To detect an unclear alternative flow starting step (D_2) defect, we check all the sentences in the AF section of a use case. If we find a sentence with its n_s field not specified, an unclear alternative flow starting step defect is reported.

The process of detecting unnecessary strong preconditions (D_3) is shown in Algorithm 1. The rationale is that the pre-condition of a use case is the required initial status of the use case. If the status is not changed by the action steps, it should be preserved. The input to Algorithm 1 is an activity diagram ad. We traverse the activity diagram ad (starting from the initial node) in the while loop. For each node n, if it is an action node (line 4), we first check whether the action verb $n.\alpha.vb$ associated with the node is a status-changing verb (line 5). If yes, we stop and return false. Otherwise, if it is a decision node (line 7), we further check all the predicates associated with the edges outgoing from the decision node and see whether they conflict with any precondition predicate of ad (line 8). If yes, an over strong precondition defect is reported. We manually defined a status-changing dictionary based on all the main verbs in the action tuples that we extract in step 3 (in Figure 3). For example, the action "Save the order" may change the status of the order and "check the order" will not change the order status. The manually dictionary defining process is liberated and reinforced by referring to the WordNet [3] lexical database for English.

To detect missing alternative flows (D_4), we traverse each activity diagram to check all the decision nodes and see whether they have branch edges. If no branch edge is present for a decision node with guard conditions, a missing alternative flow error is reported.

3.7 Training Dependency Parser

To handle the problem caused by document-specific factors, such as grammar errors and specific sentence structures, we provide a way to train a domain-adapted dependency parser. In the case of the stock trading system, we manually labeled 6% of wrongly labeled sentences randomly selected from the document to train a domain-adapted dependency parser. This is shown in the dashed box (step 7) in Figure 3. The trained dependency parser will replace the ZPar dependency parser in the dependency parsing step. This is an optional step in our overall procedure and is only needed in order to achieve higher accuracy on document specific patterns.

Table 3: Accuracy of parsing

doc	type	# wrong	# PC	# correct	# total	*prec*
sts	Action(w/o)	155	22	137	314	43.6%
	Action(w)	82	23	209	314	66.6%
	Predicate	31	17	91	139	65.5%
as	Action(w/o)	105	52	131	291	45.0%
	Action(w)	69	14	208	291	71.5%
	Predicate	2	1	14	17	82.4%

4. EVALUATIONS

To test the applicability of our approach, we evaluate our methods with 219 use cases, which cover different application domains (financial, health care, machinery, monitoring and e-commerce systems), adopted from real industry systems as well as academic publications. Due to the space constraints, we put all the experiment data and the implementation information on our website [1].

4.1 Accuracy of Free Text Parsing

The accuracy of the free text parsing is measured by the accuracy of the action tuples and predicates generated. Table 3 shows the evaluation results. The columns, from left to right, represent the document used ("doc"); the evaluation type ("type"); the number of wrongly ("# wrong"), partially correctly ("# PC") and correctly ("# correct") identified action/predicate; the total number of sentences ("# total") and the precision ("*prec*") of the corresponding evaluation type. The precision is calculated by the formula $prec = \frac{\#correct}{\#total}$. An action tuple is identified as correct iff all the three fields of it are correctly extracted. Due to the large number of sentences in the stock trading system, we randomly sampled 18% of the sentences in the document to check the accuracy.

To show how robust/extensible the dependency parsing can be, we check the accuracy on identifying action tuples based only on the dependency label. The results are shown in the "Action(w/o)" rows in Table 3. We can see from the results that, even without applying any of the adjusting rules, we can achieve more than 43% accuracy on identifying action tuples. The accuracy of the analysis results (Action(w)) with our provided adjusting rules is much higher. For the stock trading system, since it is written by non-native English speakers, there are many grammar errors, which lead to the incorrect result. The academic system documents are comparatively well written, thus ZPar achieves a higher accuracy.

From the experiment results, we notice that (1) the rules used to adjust extraction of action tuples are indeed useful. An increase in accuracy of 23% for the stock trading system and 26.5% for the those academic use cases is seen. (2) the rules are generalizable to different documents written by different development groups. It is promising to increase the accuracy by providing more rules.

4.2 Accuracy of the Activity Diagram Builder

The accuracy of the activity diagram building method is measured by (1) whether the nodes and edges are correctly generated and linked in the activity diagram and (2) whether the guard conditions are correctly associated with the edges that are correctly generated in the activity diagrams. For the stock trading system, 166 out of 188 use cases have correct nodes and edges generated. All the guard conditions are correctly associated with those edges. The use cases which do not have activity diagrams correctly generated either contain alternative flow steps which do not have clear starting steps; or have multiple conditions described within one/consecutive steps. Actually, this kind of writing style is not consistent with the majority of other use cases in the document, which follow the Cockburn style. For the academic use cases, 30 out of 31 use cases are correctly generated.

Table 4: Experiment Results of Defect Detection

ID	I_f	I_r	$I_f \cap I_r$	*prec*	*rec*
D_1	18	18	18	100%	100%
D_2	22	20	20	90.9%	100%
D_3	19	21	19	100%	90.5%
D_4	83	59	59	71.08%	100%

4.3 Accuracy of the Defect Finder

To evaluate the accuracy and effectiveness of our defect finding methods, we adopt the standard metrics of precision (*prec*) and recall (*rec*). We define $prec = \frac{|I_f \cap I_r|}{I_f}$ and $rec = \frac{|I_f \cap I_r|}{I_r}$, where I_f represents the set of items automatically identified by our method and I_r represents the set of defects that are manually detected from the document, which act as the baseline in the evaluation.

Table 4 shows the evaluation results on the stock trading system. 22 alternative flow steps are detected with defect type D_2, i.e., do not have clear starting step number. Two of them are false positives. The reason is because of the irrelevant sentence presented in that step. Our method found 19 cases where the precondition is inconsistent with the guard conditions on the flow. Our manual detection finds 21 such cases. The reason for the missing cases is that our predicate extraction method fails to extract the correct predicates for the two cases. For detecting missing alternative flows (D_4), our tool found 83 potential defects, out of which 59 are real defects. The 24 false positives appear in 8 use cases, which have different writing styles with the majority of the other use cases. Thus our method failed to generate correct activity diagrams for those use cases, which further leads to those false positives. Actually, the document is loosely written such that use case pre-conditions and post-conditions do not couple with each other well. Therefore there are very limited information we can use to do the checking. This is also the reason why the accuracy of the free text parsing does not affect too much on the accuracy of the defects detection.

5. DICUSSIONS

There are some limitations, manual efforts and threads to validity of our approach. We discuss them in this section.

Limitations We only assign coarse semantic meanings, such as synonym/conflict/status-changing, to words in our method. This may lead to missing of cases in defect checking. For example, in Figure 2, the action $(save, \_, order)$ changes the status of the order, but does not affect the legality of the order in this case. However, our method will ignore all the branch conditions after the $(save, \_, order)$ action node, since the verb "save" indicates changes of status on order. Assigning fine-grained semantic meanings to words may solve this problem.

Manual Efforts In our approach, there are three steps which may require human intervention. (1) If the input use case document does not follow the Cockburn writing style, some efforts of rewriting the use case document into the Cockburn style are needed. (2) To decide conflict predicates, three domain-specific dictionaries, i.e., the synonym dictionary, the conflict dictionary and the status-changing verbs dictionary, need to be manually categorized. Our method provides all possible candidates for the dictionaries based on our automatically extracted subject, object and main verbs for each sentence. Then the WordNet lexical database is inspected to provide the preliminary dictionaries. Therefore the only manual effort is to check and decide the dictionaries based on the preliminary dictionaries. (3) If a user wants to train a domain-adaptive parser, manual efforts on labeling sentences into dependency trees are required. However, this step is optional in our approach. Our method achieves good accuracy without the training process.

Threats to validity (1) In the evaluation, we manually inspect each kind of defects and use the manual inspection results as the baseline. The manual defects detecting is subjective to the experimenter's understanding. To reduce this factor in our evaluation, the documents are checked by two PhD students in School of Computing, NUS, who have requirement engineering and natural language processing background. (2) For the stock trading system, since it has more than 1700 sentences, we did not check all the sentences when evaluating the accuracy of free text parsing. To reduce the possible threads to validity caused by this, we randomly sampled 18% of the sentences and manually inspect the results.

6. CONCLUSION AND FUTURE WORK

In this work, we proposed a method to automatically detect defects in use case documents. Our method leverages on dependence parsing technique which allows document-independent rules to be provided and is more adaptable to documents of different writing styles than shallow parsing techniques. We formally defined common use case defects. The evaluation with 5 different use case documents shows that our method is effective in finding possible defects. Our method provides horizontal links to the original document to enable easy manual validation.

For future works, we explore to enrich our method by considering semantic level meanings, which may improve the conflict checking precision as discussed in Section 5. Another possible direction is to investigate statistical classification models, e.g. logistic regression, to improve the accuracy of action tuple extraction.

7. REFERENCES

[1] *Experiment data and source code information.* `www.comp.nus.edu.sg/~lius87`.

[2] OMG unified language superstructure specification (formal). Version 2.4.1, 2011-08-06.

[3] *Princeton University "About WordNet." WordNet. Princeton University. 2010.* `http://wordnet.princeton.edu`.

[4] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2000.

[5] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2003.

[6] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol.*, 14(3):277–330, July 2005.

[7] A. Sinha, S. M. S. Jr., and A. Paradkar. Text2test: Automated inspection of natural language use cases. In *ICST*, pages 155–164, 2010.

[8] L. Tan, Y. Zhou, and Y. Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE*, pages 11–20, 2011.

[9] F. Tørner, M. Ivarsson, F. Pettersson, and P. Öhman. Defects in automotive use cases. In *ISESE*, pages 115–123, 2006.

[10] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *ESEC/FSE*, pages 12:1–12:11, 2012.

[11] Y. Zhang and S. Clark. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151, 2011.

[12] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *ASE*, pages 307–318, 2009.