11-2016

Improving quality of use case documents through learning and user interaction

Shuang LIU

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

Hao XIAO

Bimlesh WADHWA

Jin Song DONG

See next page for additional authors

Citation

LIU, Shuang; SUN, Jun; XIAO, Hao; WADHWA, Bimlesh; DONG, Jin Song; and WANG, Xinyu. Improving quality of use case documents through learning and user interaction. (2016). *Proceedings if the 21st International Conference on Engineering of Complex Computer Systems, Dubai, United Arab Emirates, November 6-8*. 101-110. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4941

2016 21st International Conference on Engineering of Complex Computer Systems

Improving Quality of Use Case Documents through Learning and User Interaction

Shuang Liu*, Jun Sun†, Hao Xiao‡, Bimlesh Wadhwa§, Jin Song Dong§ and Xinyu Wang¶
*Singapore Institute of Technology †Singapore University of Technology and Design ‡Nanyang Technological University §National University of Singapore ¶ZheJiang University, China

***Abstract*—Use cases are widely used to capture user requirements based on interactions between different roles in the system. They are mostly documented in natural language and sometimes aided with graphical illustrations in the form of use case diagrams. Use cases serve as an important means to communicate among stakeholders, requirement engineers and system engineers as they are easy to understand and are produced early in the software development process. Having high quality use cases are beneficial in many ways, e.g., in avoiding inconsistency/incompleteness in requirements, in guiding system design, in generating test cases. In this work, we propose an approach to improve the quality of use cases using techniques including natural language processing and machine learning. The central idea is to discover potential problems in use cases through active learning and human interaction and provide feedbacks in natural language. We conduct user studies with a real-world use case document. The results show that our method is helpful in improving use cases with a reasonable amount of user interaction.**

I. Introduction

Use cases are one of the primary ways to capture user requirements. It is like the hub of a wheel [13] which binds together many software development activities throughout the system development lifecycle, including requirement analysis, system design, development, testing and maintenance. Having high quality use cases are beneficial in many ways, e.g., in avoiding inconsistency/incompleteness in requirements, in guiding the system design, in helping generating test cases. Validating and maintaining a high quality use case document is thus crucial, which unfortunately is also subjective and labor-intensive. One of the reasons why it is hard to have high quality use cases is that stakeholders usually do not describe the requirements clearly, consistently or completely. Common problems with use cases include ambiguity and inconsistency in the requirements [12], [21] and, perhaps more importantly, missing information (e.g., precondition, scenario) [4].

Given the importance of high quality use cases, in this work we aim to develop techniques and tools which focus on improving the quality of use cases in practice. The design of our method and tool is guided by three observations. Our first observation is that user interactions are crucial to improve the quality of use cases. Given that many problems of the use cases are caused by incomplete or inconsistent user requirements [12], and there is no better way to obtain the information than interacting with the stakeholders, our method/tool shall try to make the best use of user interactions. We should require information from the stakeholder or requirement engineer in a way which is easy to understand so that they are not overwhelmed or confused. The second observation is that due to the continuous communication request with stakeholders, use cases are mainly documented in natural language [21], and thus inevitably informal. Therefore our methods must be able to handle use cases written in natural language, live with certain level of ambiguity, and help to reduce the ambiguity through automatic analysis and user interaction. The last observation is that problems in use cases, if manifest later in the system design or implementation stages, will cause more efforts (up to 100 times more expensive [6]) to correct than to find and correct them in the early stages.

Based on the above observations, we propose to improve the quality of use cases using techniques including natural language processing (NLP) and machine learning. Central to our idea is to discover potential problems which could manifest during implementation and report the problems at the level of use cases so as to improve the quality of use cases.

Figure 1 shows the high-level workflow of our approach. Firstly, we adopt advanced natural language parsing techniques [30] to extract structured format from individual use case written in English, from which we obtain information on behaviors of each actor in the system in a particular scenario. Next we attempt to answer the question on whether there would be a concise implementation of the system such that the requirements are satisfied. To do that, we need to, for each actor in the system, not only figure out the relationship between its behavior in different use cases, but also check whether the behaviors in different use cases can be grouped into a meaningful and succinct implementation. For the former, we extract predicates from preconditions and postconditions of each use case and use those predicates as guidance to construct a use case relation graph. For the latter, we adopt active learning techniques from the machine learning community to incrementally learn a Deterministic Finite-state Automaton (DFA) from the behaviors in individual use cases. The use case relation graph is then used to compose the learned automata for every actor to obtain a plausible implementation for the actor. We remark that the requirement engineers are involved throughout the process. For instance, we would automatically infer relationships between preconditions and postconditions of different use cases as much as possible. When ambiguity arises, we generate questions in English to consult the requirement engineers, e.g., whether a certain precondition is satisfied by certain postconditions; or whether a behavior anticipated through learning (for instance, an implementation

978-1-5090-5527-2/16 $31.00 © 2016 IEEE
DOI 10.1109/ICECCS.2016.33

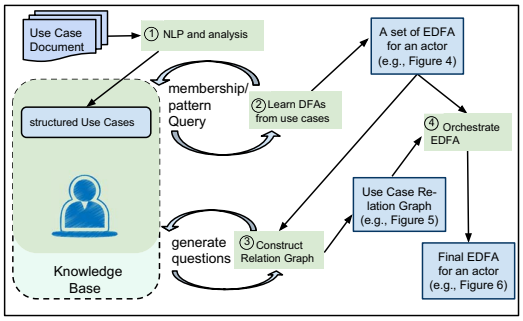


Fig. 1. Overview of our approach

with a small number of states would probably allow this additional behavior) is indeed allowed but is missing from the current set of use cases.

In this way, we are able to elaborate the use cases incrementally and interactively with requirement engineers to improve their quality, for instance, by reducing ambiguity in precondition and postcondition descriptions, or by identifying missing use cases. We remark while we attempt to synthesize a plausible DFA implementation for each actor in the system, it is not the goal of this work. Rather it is a way of identifying problems in use cases, which is more realistic from our point of view. Nonetheless, some of the artifacts generated in our method could be useful on its own. For instance, the use case relation graph shares the same utility as the use case charts or high-level Message Scequence Charts and could be used by existing scenario based requirement validation approaches [25], [27]. Therefore, our approach can be used in combination with those approaches. We conduct a case study with a real industry use case document (of a financial system actively used by a financial institute in Boston), which contains more than 100 use cases. We identified more than a dozen missing scenarios and dozens of other problems. The evaluation results show that our approach is effective in improving the quality of use cases. A user study with 15 software engineers show that the amount of user interactions required by our tool is acceptable.

II. A Running Example

In this section, we illustrate the overall process of our approach through an example. The example is adopted from our industry collaborator. For confidentiality, the use cases presented in this paper have been slightly modified, e.g., the sensitive words have been replaced. Nonetheless, the use cases remain largely faithful. Figure 2 shows four sample use cases of the system, written in English. All these use cases describe the valid behaviors of the actor "Ticker Monitor" and serve as input to our method.

There are four major steps in our approach. The first step is to "understand" the description of the use cases. We adopt NLP techniques to parse the use case documents and obtain formal structures of the use case. In the second step, we formalize the behaviors of each actor in the use cases using DFA and make reasonable guesses on how the behaviors can be realized. In particular, we adopt an active learning algorithm L^* [5] to learn an Extended Deterministic Finite Automaton (EDFA) based on the actor's behaviors in the use cases. Different use cases might have very different preconditions and postconditions, in order to understand the relations between different use cases, we construct a use case relation graph for each actor based on the preconditions and postconditions of each EDFA and then compose the EDFAs to obtain an overall EDFA for each actor. This step allows us to reduce ambiguity as well as identify missing scenarios in the use cases.

Step 1 : Natural Language Parsing We first adopt natural language processing techniques, i.e., dependency parsing and phrase structure parsing [30], to parse the sentences in a use case description into parse trees. Then we conduct rule matching based on general grammar rules (proposed in [16]), which are extracted from the documents, on the parse trees. We identify all the actions which are related to the actor of concern based on the parsed action tuples. For example, in use case 2 of Figure 2, the action tuples for the main flow sentences are *selects(Ticker monitor, symbol information), sends(Ticker monitor, message to delete symbol information)*. Both action tuples have actor "ticker monitor" as subject. We thus consider both of them as actions related to the actor.

Then the structured sentences are linked based on the control flow information (e.g., predecessor/successor relation or the "go to" statement) described in the flow steps to obtain a raw Nondeterministic Finite State Automaton (NFA), which captures the actions of one actor described in the use case. For example the NFAs[1] shown in Figure 3 (a) and Figure 3 (b) are constructed from use case 2 and use case 3 in Figure 2, respectively. We merge all those NFAs which share the same preconditions and describe the actions of the same actor to obtain one NFA. The NFAs in Figure 3 (a) and Figure 3 (b) are merged to obtain the NFA shown in Figure 3 (c). Then we determinize the NFA in Figure 3 (c) to obtain a DFA[2] shown in Figure 3 (d), which serves as a part of the knowledge base during the active learning process. In our approach, we learn the DFA which is prefix-closed with the assumption that the system can stay in any of the state after conducting an action. Therefore, we set all states in the DFA to be accepting states.

Step 2 : Learn Local EDFA We group those structured use cases based on their preconditions and postconditions. For all the use cases which describe the actions of one actor, if they have the same preconditions, they are put together as one group (e.g., use case 2 and use case 3 in Figure 2). All the actions appear in those use cases in the same group are fed to the L^* algorithm as the alphabet to learn one local DFA. For example, in Figure 2, use case 1 and use case 4 correspond to DFAs shown in Figure 4 (a) and Figure 4 (c) respectively. The DFA in Figure 4 (b) is generated based on the traces from use case 2, use case 3 and some other use cases (refer to [2] for the full list of use cases) within the same group. One goal of the learning is to gradually discover

[1]We simplify the action tuple representation to save space.

[2]We only show the transitions which lead to accepting states in the DFAs for clarity. The same applies to Figure 4 and Figure 6.

Use Case 1: Ticker Monitor Connects to GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor can monitor the change of tick information and the connection status with TickFeed.
Main Flow
1. Ticker monitor connects to GSYS.
2. GSYS sends all Exch information and symbol information records in the database to ticker monitor.
3. Ticker monitor displays the records.
4. This ends the use case.
Alternative Flow
N/A
Post-Conditions
N/A

Use Case 2: Delete a Symbol Information from GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor has connected to GSYS.
Main Flow
1. Ticker monitor selects a symbol information.
2. Ticker monitor sends a message to delete the symbol information.
3. This ends the use case.
Alternative Flow
N/A
Post-Conditions
1. The symbol information is deleted.

Use Case 3: Update a Symbol Information in GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor has connected to GSYS.
Main Flow
1. Ticker monitor selects a symbol information.
2. Ticker monitor update some fields of the symbol information.
3. Ticker monitor sends the updated symbol information to GSYS
4. This ends the use case.
Alternative Flow
N/A
Post-Conditions
1. GSYS update the symbol information in the database if it is valid.

Use Case 4: Undo Delete a Symbol Information from GSYS
Initiating Actor: Ticker Monitor
Pre-Conditions
1. The ticker monitor has connected to GSYS.
2. GSYS has deleted one or more symbol information.
Main Flow
1. Ticker monitor undos delete the symbol information.
2. Ticker monitor sends the most recently deleted symbol information to GSYS.
3. This ends the use case.
Alternative Flow
1. In step 1, if ticker monitor has not deleted any symbol information, do nothing.
Post-Conditions
1. GSYS restore the symbol information.

Fig. 2. Sample use cases

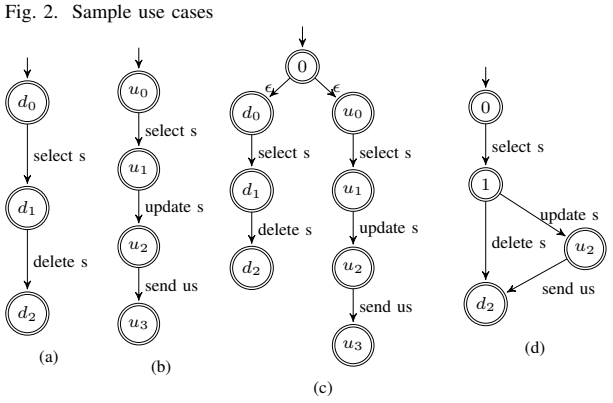


Fig. 3. The NFA for use case 2 (a) and use case 3(b) in Figure 2, the merged NFA (c) and the corresponding DFA (d)

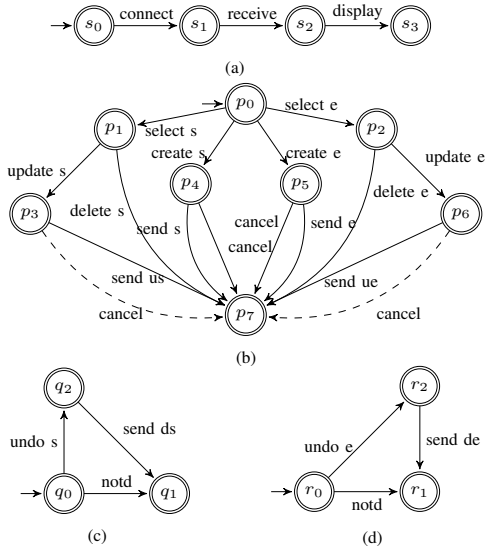


Fig. 4. The partial DFAs for Ticket Monitor

missing scenarios by generating questions to users. Using an active learning algorithm allows us to 'control' the number of questions required. For instance, the dashed lines in the DFA shown in Figure 4 (b) represent the traces that are added during the interactive learning process. These traces are generated by our learning algorithm and are confirmed to be valid by stakeholders.

We then assign each DFA with preconditions and postconditions of the use cases that compose it. The DFA with

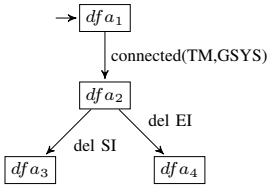


Fig. 5. Relation graph of Ticket Monitor EDFAs

preconditions and postconditions is called an Extended DFA (EDFA). The postconditions of an accepting state are set on a trace basis, i.e., only the accepting states, e.g., p_7 in Figure 4 (b), which correspond to ending of the traces have postconditions. For the trace $\langle$*select s, delete s*$\rangle$, the postcondition is set to be *deleted(symbol informtion)*. The trace-based preconditions and postconditions are used for use case relation graph generation and EDFA composition, when we decide how to split the traces.

Step 3 : Construct Use Case Relation Graphs We observe that there are often "happen before" or "in parallel" relations between use cases. Those relations can be inferred from the precondition and postcondition sections in the use case documentation. At the same time, analyzing those relations between use cases can identify ambiguity in precondition/postcondition descriptions as well. Recall that in step 2, we learn one EDFA for the use cases with the same precondition. Therefore usually multiple EDFAs are learned for one actor. In order to show the overall view of all behaviors of an actor, we build a usage relation graph for those EDFAs based on their corresponding preconditions and postconditions. The usage relation graph for the EDFAs in Figure 4 is shown in Figure 5. The nodes represent the EDFAs and directed edges represent the precedence of usage relations between two EDFAs. The labels on the edges are the conditions based on which we infer the relation between two EDFAs. For example, dfa_3 "happens after" dfa_2 based on the common condition *the symbol information is deleted* (*del SI* in Figure 5), which is the postcondition of dfa_2 and the precondition of dfa_3.

We construct one usage relation graph for each actor. For those use cases with trivial linking conditions, we link them directly. For example dfa_2 and dfa_3 can be linked directly based on the predicate *del SI*. For those use cases which do not have clear linking references, or miss preconditions/postconditions, we raise questions to query the users

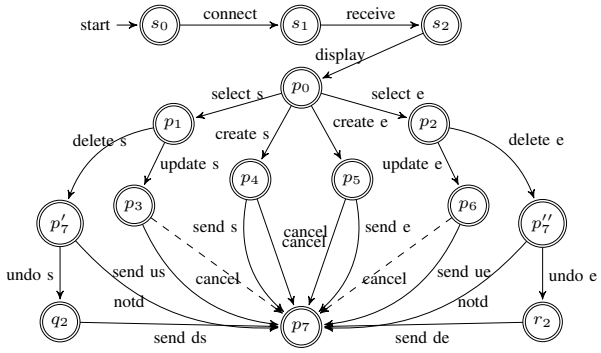


Fig. 6. The overall DFA for Ticket Monitor

about the relations between those use cases. For example, the EDFA in Figure 4 (a) (corresponds to use case 1 in Figure 2) does not have postconditions specified. Our method raises questions based on the preconditions of existing use cases for users' confirmation. In this example, the precondition *connected(TM, GSYS)* ("*The ticker monitor has connected to GSYS*"), from all the existing use cases, is confirmed to be a legal postcondition for use case 1.

Step 4 : Orchestrate Local EDFAs Afterwards, we orchestrate all the EDFAs based on the usage relation graph obtained in step 3. We traverse the usage relation graph in a breadth-first manner and link a node with all its child nodes on the common conditions labeled on the corresponding edges. For example, according to the relation graph shown in Figure 5, dfa_1 (Figure 4 (a)) and dfa_2 (Figure 4 (b)) are linked based on the common condition *connected(TM, GSYS)*, which is the precondition of dfa_2 and the postcondition of dfa_1.

There are cases where we need to split a final state during the orchestration. Since each trace has a set of corresponding postconditions, those traces which have matched postconditions with the preconditions of a given EDFA are split to link with the initial state of that EDFA. For example, according to the graph shown in Figure 5, dfa_2 links with dfa_3 on the common condition *del SI* and link with dfa_4 on the common condition *del EI*. The result of the orchestration is shown in Figure 6. Traces $\langle$*select s, delete s*$\rangle$ is split to link with dfa_3 based on the common condition *del SI*. Similarly traces $\langle$*select e, delete e*$\rangle$ is split to link with dfa_4.

III. Preliminary on Active Learning

Active learning refers to a model of instruction in which a student interacts with a teacher by actively asking questions in order to learn the knowledge. Angluin proposed the *L** algorithm [5] to learn an unknown DFA U (i.e., the knowledge) from the teacher, who knows the DFA, by asking *membership queries* and *candidate queries*. For a membership query, *L** asks the teacher whether a string s is a member of the accepted languages of the DFA, i.e., whether s is accepted by U. The teacher answers yes(1)/no(0) accordingly. After a set of membership queries, *L** conjectures a candidate DFA C from his current knowledge and asks the teacher a candidate query whether the candidate DFA is equivalent to the DFA, i.e., $C \equiv U$. If the teacher answers *yes*, *L** successfully learned the

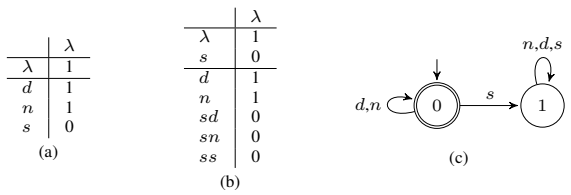


Fig. 7. The observation tables (a) and (b) in the first learning round and the first candidate DFA (c)

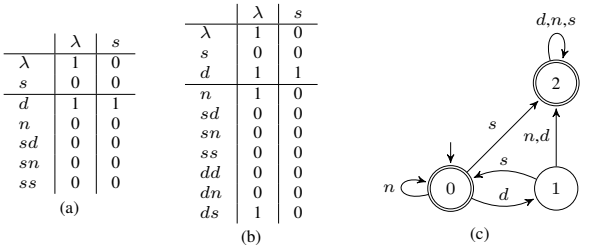
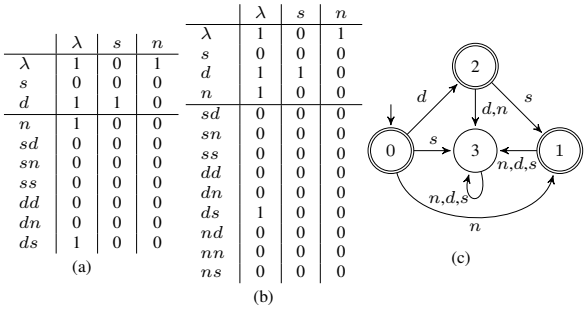


Fig. 8. The observation tables (a) and (b) in the second learning round and the second candidate DFA (c)

DFA, which is equivalent to the current candidate DFA. If the teacher answers *no*, it provides a counterexample trace which is either accepted by C or U but not both. *L** then extracts knowledge contained in the counterexample and starts asking membership queries. *L** is guaranteed to terminate and the student always learns U within polynomial time [5].

We illustrate how *L** works to learn the DFA in Figure 4 (c). To simplify the presentation, we use the symbol d, s, n to represent the alphabet symbol *undo s*, *send ds* and *notd*, respectively. Initially, the observation table is shown in Figure 7 (a). This table is not closed because the row indexed by string s appears only once in the table. *L** moves this row to the upper part and extends the table with each alphabet symbol by asking membership queries for strings sd, sn and ss. The extended table is shown in Figure 7 (b) which is closed. The first candidate DFA constructed from the table is shown in Figure 7 (c). Then *L** asks a candidate query with the candidate DFA, for which the teacher returns a counterexample string ds. The table contains string d which is the maximum prefix of ds. Thus the suffixes of string s are $\{\lambda, s\}$. Only string s is added to the table column because λ already exists in the column.

Then *L** asks several membership queries to fill up the cells due to the addition of the column s. The observation table is shown in Figure 8 (a). This table is not closed because the row with valuation 11 appears only once. *L** moves the row indexed by string d (the first row corresponding to row 11) to the upper part and extends the table with rows indexed by string dd, dn and ds by membership queries. The extended table is shown in Figure 8 (b), which is closed. Then *L** constructs the second candidate DFA shown in Figure 8 (c). For this candidate query, the teacher returns a counterexample string nn. Then the string n is added to the table column. *L** repeats the previous steps to obtain the third candidate DFA shown in Figure 9 (c) and asks a candidate query. The teacher finds that the candidate DFA is equivalent to the DFA to be leaned. Thus *L** successfully learns the DFA.



	λ	s	n
λ	1	0	1
s	0	0	0
d	1	1	0
n	1	0	0
sd	0	0	0
sn	0	0	0
ss	0	0	0
dd	0	0	0
dn	0	0	0
ds	1	0	0

(a)

	λ	s	n
λ	1	0	1
s	0	0	0
d	1	1	0
n	1	0	0
sd	0	0	0
sn	0	0	0
ss	0	0	0
dd	0	0	0
dn	0	0	0
ds	1	0	0
nd	0	0	0
nn	0	0	0
ns	0	0	0

(b)

(c)

Fig. 9. The observation tables (a) and (b) in third learning round and the third candidate DFA (c)

IV. Detailed Approach

A. Natural Language Parsing and Analysis

There is no standard template for writing use case documents as concluded by Fowler [10]. We thus focus on one of the widely used writing styles in the literature and in practice, which is "the single-column, numbered, plain text, full sentence form" [7]. Since the input of our approach is a use case specification document written in English, we adopt advanced NLP techniques, i.e., dependency parsing and phrase structure parsing [30] to parse the document.

We first conduct pre-processing on the input use case document to filter noises so as to improve the accuracy of the dependency parser. The pre-processing step contains standard operations such as truncating, segmenting, and removing the irrelevant information and formatting symbols, such as parenthesized comments and bullets, which may affect the parsing accuracy. The pre-processed use case document is passed to ZPar [30], a statistical natural language parser, for dependency parsing. Then we analyze the parse trees by rule matching to extract action tuples and predicates from the flow steps and precondition/postcondition sections. We adopt the approach proposed in [16] for action tuples/predicates extraction. We skip the details and refer interesting readers to [16] for details.

The formal definition of an action tuple and a predicate is defined in Definition 1 and Definition 2, respectively.

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.

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.

After analyzing the parse trees, each sentence in the use case description is mapped to the formal structure defined in Definition 3. The use case is organized based on the sections to which those sentences belong, as is defined in Definition 4.

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 $n_j \in N$ represent the logical previous and succeeding sentence of the current sentence, respectively.

Definition 4 (Use Case): A *use case* is defined as a tuple $UC \triangleq (UCName, Prec, Post, MF, AF)$, $UCName$ is the name of the use case; $Prec \subset P$ and $Post \subset P$ are the predicates extracted from sentences in the precondition and postcondition sections; MF and AF are the list of sentences S in the main flow/alternative flow sections of the use case.

Definition 5 (NFA): An NFA is defined as NFA$\triangleq$ $\{S, \Sigma, \delta, init, AS\}$, in which S is a non-empty finite set of states; Σ is a non-empty finite set of alphabet; $\delta = S \times \Sigma \rightarrow \mathbb{P}S$ is a transition relation; $init \in S$ is the initial state and $AS \subseteq S$ is the set of accepting states.

A Deterministic Finite Automaton (DFA) is a special NFA where there is no ϵ in the alphabet and there is at most one outgoing transition labeled with any one of the actions in the alphabet from any state. Formally, the transition relation of a DFA is a function $\delta = S \times \Sigma \rightarrow S$. For example, in Figure 4, the DFA on the top can be represented as $\{\{s_0, s_1, s_2, s_3\}$,{connect, receive,display},$\{s_0 \xrightarrow{connect} s_1, s_1 \xrightarrow{receive} s_2, s_2 \xrightarrow{display} s_3\}$,$s_0$, $\{s_0, s_1, s_2, s_3\}\}$.

There are two kinds of control flow information in a use case description as is exemplified in use case 4 in Figure 2. The first kind is captured by the sequential ordering of sentences in each section, i.e., the $s\#$ field in a sentence S. The second kind is enrolled in the conditional statements, which are usually indicated by the keywords such as "if", "whether", "else", in a sentence. The control flow information is extracted during analyzing the parse trees. We adopt the approach proposed in [16] to parse the sentences. Since parsing use cases is not the focus of this work, we refer the readers to the work [16] for details on how to extract the control flow information.

We then compose an NFA from a use case defined in Definition 4 following the steps shown in Algorithm 1. We first create a state for each sentence (line 1–3) and then link the states based on the previous (n_s) and succeeding fields (n_j) of the corresponding sentences (line 4–11). If we find the sentence step number $st_i.s\#$ is the same with the succeeding node step number $st_k.n_j$ of another sentence; or the sentence step number $st_k.s\#$ is the same with the previous node step number of another sentence ($st_i.n_s$), then we add a transition between the corresponding nodes accordingly (line 6–7)). All the actions labeled with the transitions are added into the NFA alphabet (line 8)). If the action field is empty, we label the transition with ϵ. Lastly, we set the initial state and accepting states based on the sequence of the sentences in the $UC.MF$ section. The state corresponding to the first sentence in $UC.MF$ is set as the initial state, and the states corresponding to the last sentences in $UC.MF$ and $UC.AF$ are set as the accepting states.

After obtaining an NFA for each use case, we construct an NFA for all the use cases sharing common preconditions, by adding one unique initial state and an ϵ transition from it to each initial state of all the NFAs for those use cases. Then we convert the NFA to an equivalent DFA with the anti-chain improved powerset construction algorithm [28], which is often efficient despite its worst-case exponential complexity.

After this step, we obtain a set of DFAs for each actor. Each DFA corresponds to a set of use cases which have the same precondition and compatible postconditions. Those DFAs are part of the knowledge base in our learning process.

Algorithm 1: Generate an NFA from a Structured Use Case

```
Input  : UC: a use case
Output: nfa: an NFA
for each st ∈ UC.MF ∪ UC.AF do
    s :=create a state for st
    nfa.S.add(s)
for each pair of states (s_i,s_k) from nfa.S do
    Let st_i and st_k be the corresponding sentences
    if st_i.s# = st_k.n_j or st_k.s# = st_i.n_s then
        nfa.δ.add(s_k,s_i,st_k.α)
        nfa.Σ.add(st_k.α)
    else if st_k.s# = st_i.n_j or st_i.s# = st_k.n_s then
        nfa.δ.add(s_i,s_k,st_i.α)
        nfa.Σ.add(st_i.α)
nfa.Σ.add(ϵ)
set nfa.init and nfa.AS
return nfa
```

B. Learn the DFAs

In the second step, we adopt the *L** algorithm to learn a DFA representation of the actor's behavior. In our setting, the teacher is composed of the structured use cases (i.e., the DFA we obtained in step 1) and the user interacting with our tool. The alphabet is set as all the actions extracted from the structured use cases. The membership query is answered based on traces from the use cases and suggestions from the users. The candidate query is decided by checking the equivalence of two DFAs, i.e., the DFA learned with *L** algorithm and the DFA generated from the knowledge base (in step 1).

1) Membership Query: The membership query checks whether a trace generated by *L** is a valid trace. To answer the query, we first check whether the given trace is a valid trace in the DFA obtained from step 1 by a depth-first search. If it is not, we raise a question to the requirement engineer to ask whether the trace should be accepted or not. If the user answers "no", we reject the trace. Otherwise, we accept the trace and add the trace to the DFA generated in step 1.

The number of membership queries generated by *L** is linear to the size of alphabet, states in the DFA as well as the length of the returned counterexample, and thus may be rather large. We propose five filtering techniques to filter out those traces which are unlikely to be valid so as to reduce the amount of user interactions required.

(1) We only allow the traces that start with actions which are initial actions of some known valid traces. The justification is these initial actions are often 'special' and it is unlikely that the user would completely forget certain functionality of the system. (2) Recall that we assume the learned DFA to be prefix closed. Therefore we can conclude that if a prefix of a trace is not accepted, the trace is not accepted. Thus we record all the traces that are denied by users to avoid asking questions regarding traces whose prefix have been denied. (3) Actions or predicates with conflicting semantics are unlikely to reside in the same trace. We extract conflicting action/predicate pairs automatically from the use cases to filter unlikely traces. (4) Traces must not violate the precedence order (defined in Definition 7).

Definition 6 (Trace): A trace is defined as $T \triangleq \{ < A >, \text{Post}\}$. Post is the postconditions of the trace and $< A >$ is a list of actions.

Definition 7 (Precedence Order): Two actions α_1 and α_2 have a precedence order relation $\alpha_1 \prec \alpha_2$ iff $\exists t \in T : \alpha_1 \in t. < A > \wedge \alpha_2 \in t. < A > \wedge index(\alpha_1) < index(\alpha_2)$. The function *index($\alpha$)* returns the index of an action in the trace. The precedence order is a partial order relation, which satisfies the transitive property, i.e., if we have two pairs of actions $\alpha_1 \prec \alpha_2$ and $\alpha_2 \prec \alpha_3$, then we can infer that the precedence order $\alpha_1 \prec \alpha_3$ holds. In practice, events in a systems are often ordered, for instance, login often precedes authenticate. Knowing the ordering between different events would help to greatly reduce the number of user interactions needed. In our work, we infer likely ordering based on the given use cases and only in the presence of conflicts, we would consult the requirement engineer.

(5) During the process of membership query, we raise questions (in the form of a sequence of actions) to users and they response with "yes" or "no". From the sequences that are rejected by users, we mine frequent patterns with the Apriori algorithm [3]. We present the mined patterns to users for confirmations so as to check whether such sequences which contain those patterns are always not accepted. Those patterns that are confirmed by users are used in the membership queries for filtering.

All these filtering techniques are proposed based on the assumption that the valid traces in the use cases should share common features, such as the partial ordering between actions, common prefix, etc., which can be mined from the existing knowledge. With all these filtering techniques, the number of questions generated is controlled in a reasonable amount.

If a trace generated in a membership query is confirmed to be valid by users, we merge the valid trace with the existing DFA. To do so, we find the state which shares the longest prefix with the given trace t from the initial state of the DFA by a depth-first search on the DFA. Then we add a sequence of transitions which captures the suffix of t from this state of DFA. The obtained DFA is not guaranteed to be minimal, but is guaranteed to be deterministic. By interacting with the users, we are able to find missing scenarios which are not captured by the use case documents.

2) Candidate Query: To conduct candidate query, we check the equivalence of two DFAs, i.e, the DFA which is learned by the *L** algorithm and the DFA that is constructed from the knowledge base (in step 1). To check the equivalence of two DFAs, we construct a product $P_{\times}$of the two DFAs and check whether the accepting states of the product DFA are composed of pairs of accepting states from the two DFAs. Since the algorithm to check the equivalence of two DFAs is quite standard. We skip the details due to space limitations and refer readers to our website [2] for details.

3) Set Precondition and Postcondition for EDFA: To enable further orchestration of DFAs, we set preconditions and postconditions for those DFAs learned by the *L** algorithm to obtain an Extended DFA (EDFA), which contains precondition and postconditions, as is formally defined in Definition 8.

Definition 8 (Extended DFA): An Extended DFA is defined as EDFA$\triangleq$ $\{$DFA, $Prec$, $PostM\}$, where DFA is a DFA; $Prec \subset P$ is the set of precondition of the DFA; $PostM$ is

a mapping from each final state in DFA to their trace based postconditions $\hat{T} \subset T$ that compose this DFA.

The precondition of a DFA is set as the union of the precondition sets of all the traces which are used to generate the DFA. (Recall that in our approach, all the use cases which have the same preconditions are grouped to learn one DFA.) The postconditions of accepting states in the DFA are set to be the set of postconditions of the traces reaching the accepting state. Note that the postconditions are set on a trace basis, meaning each trace ending in the same accepting state has its own postconditions. For example, in Figure 4 (b), only state p_7 has postconditions and it is set as the set $\{\langle\langle$*select s, update s, send us*$\rangle$,*update(GSYS, symbol information)*$\rangle$, $\langle\langle$*select s, delete us*$\rangle$,*deleted(symbol information)*$\rangle$,$\dots\}$. The first two elements in the set represent the left-most two traces in Figure 4 (b). We omit the postconditions for the other traces for confidentiality. This is critical for the splitting of traces during the orchestration of DFAs.

C. Construct Relation Graphs

We first construct a usage relation graph based on the preconditions and postconditions of EDFAs learned for an actor. The relation graph is formally defined in Definition 9.

Definition 9 (Relation Graph): A relation graph is defined as $G \triangleq \{N, n_i\}$. where $n_i \in N$ is the initial node of the graph and N is the set of nodes in the graph. Each node in the graph is defined as n$\triangleq$ $\{Prec,\ PostM,\ EDFA,\ cc,\ ch\}$, where $Prec$ and $PostM$ are the precondition and postconditions of the EDFA the node represents. $EDFA$ is the EDFA that the current node represents. cc is the common condition that the current node shares with its parent node. ch is the list of child nodes of the current node. A node in a usage relation graph represents an EDFA which is learned with the *L** algorithm.

Initially, we have a set of nodes which represent the EDFAs learned for a set of use cases describing the usage scenarios of one actor. We link two EDFAs if the postconditions of an accepting state of an EDFA match the preconditions of the other EDFA. This case represents a succession in behavior.

The necessary condition for the orchestration of two EDFAs with succession behaviors is defined as follows:

Definition 10 (Link Condition): Let dfa_a, $dfa_b \in EDFA$ be two extended DFA, they can be linked if and only if $\exists$ accs$\in dfa_a$.AS: $\exists$t ends in accs$\wedge$ t.Post$\subset dfa_b$.Prec$\vee$ $\exists$ accs$\in dfa_b$.AS: $\exists$t ends in accs$\wedge$ t.Post$\subset dfa_a$.Prec.

We decide whether two predicates are compatable based on a direct comparison on the corresponding fields of the predicates (Definition 2). We also rely on the semantic dictionary to decide whether two words are semantically equal. If the postcondition of one EDFA is equal to or is a subset of the precondition of another EDFA, then they can be linked directly. Given an EDFA, if we cannot find any other EDFA for the same actor that satisfies the link condition, we raise questions for user suggestions. This case usually implies there is some missing precondition/postcondition.

Algorithm 2 describes the procedure to construct a relation graph for a set of EDFAs. We first construct a single-node graph for each input EDFA and add them to the set of graphs (line 1–3). When the graph set has more than one

Algorithm 2: Build Relation graph

```
   Input  : a set of extended DFA widehat{DFA} ={D_1 ... D_n}
   Output: an usage relation graph g for DFAs in widehat{DFA}
1  for D_i in widehat{DFA} do
2      g_i:=construct a single-node graph for D_i
3      Ĝ.add(g_i)
4  while true do
5      while |Ĝ| > 1&& !stabilized do
6          for (g_i,g_j) in Ĝ do
7              stabilized:=true
8              if Comp(g_i.Prec, g_j) then
9                  g_j.ch.Add(g_i, cc)
10                 Ĝ-{g_i}
11                 stabilized:=false
12             else if Comp(g_j.Prec, g_i) then
13                 g_i.ch.Add(g_j, cc)
14                 Ĝ-{g_j}
15                 stabilized:=false
16     if |Ĝ| > 1 then
17         ret:=RaiseQuestion()
18         if ret=false then
19             modify the conditions based on answers from users
20             if find compatable graph pair then
21                 stabilized:=false
22                 continue
23             break
24         else
25             g:=Merge(∀g_i ∈ Ĝ)
26             break
27     else
28         break
29 return g
```

graphs and the graph set is not stabilized (line 5–15), we try to find a pair of graphs which can be linked based on the link conditions defined in Definition 10. The function *Comp(condition, graph)* checks whether the given predicates "condition" are compatible with any node in the given "graph" based on Definition 10. If such a pair of compatible graphs (line $8, 12$) is found, we set one graph as the child of another (line $9, 13$) and remove the child graph from the graph set (line $10, 14$). As long as the graph set is changed, it is said to be not stabilized. If there are more than one graph in the graph set after it is stabilized, which indicates the preconditions and postconditions of those EDFAs are loosely coupled, we raise questions (line 17) to query whether we can merge those graphs by adding a single root node. If the reply is negative, the user can modify the preconditions/postconditions of the graphs, which can be traced back to the use case documents. If the modifications from the user enable new parent-child relation between those graphs, we continue to link those graphs (line 20–22). Otherwise we merge all the graphs in the graph set $\hat{G}$ to obtain a final relation graph (line $24, 25$).

D. Orchestrate EDFAs

Given a set of EDFAs and the relation graph for those EDFAs, we conduct a breadth-first traverse on the relation graph and link the EDFA represented by a graph node with the EDFAs represented by all its child nodes. The procedure is intuitive. What should be noted is that we need to split an accepting state when the preconditions of the child EDFA

match a subset of the postconditions of its parent EDFA (Recall that the postcondition of an EDFA is stored based on traces). For example in Figure 4 (b), the accepting state p_7 is split twice (into p_7' and p_7'') when orchestrating with the EDFA in Figure 4 (c) and the EDFA in Figure 4 (d), respectively. The final EDFA is a visualization of the overall dynamic behaviors of an actor. It can serve as an initial behavior model for its corresponding actor, which can aid the system design.

V. Evaluation and Discussions

We have implemented our method in a prototype tool in Python and Java[3]. We adopt ZPar [30] to analyze syntactic information. To evaluate our approach, we conduct an experiment with a use case document for a real world stock trading system. This use case document contains 106 use cases, which is considerably big. The use cases were written following the Cockburn template [7]. The document contains many grammar errors and is inconsistent in writing style (due to multiple authors), which is not uncommon in practice. These problems also cause difficulties in natural language parsing. Nevertheless our method achieved good accuracy in parsing the document. In this work, we focus on the part of active learning and user interaction. We refer interesting readers to the work [16] on the results and discussions of accuracy in natural language parsing. In this evaluation, we try to answer the following questions:

1) Is our method helpful in improving the quality of use case specifications?
2) How is the user experience with our tool?

Question 1: Improvement on Use Case Quality To answer the first question, we conduct an experiment with the use cases. Since the document is confidential (i.e., one has to get clearance before accessing it) and requires quite some effort to understand, this experiment is conducted with 2 researchers (who have the clearance). The 2 researchers were asked to use our tool to analyze the use cases. During the experiment, if a question is raised by the tool, we first screen the question to see whether it has an obvious answer which is missed by the parser or it indeed requires an answer from the author of the system. If it is the latter, the question is directed to the requirement engineer for clarification. The experiment results are summarized in Table I. The first column shows the actors in the system. Columns 2−4 are the number of use cases for the corresponding actor, the number of nodes of the generated use case relation graph, and the number of states in the final DFA obtained for each actor. Columns 5−7 show the total number of membership queries, the total number of pattern queries raised to users during the learning process, and the total number of queries raised to users during generating of use case relation graphs. Column 8 and 9 show the number of missing scenarios and missing/redundant preconditions/postconditions that we identified during the process. We identifies a total of 17 missing use cases as well as more than 50 other problems. All identified issues are confirmed by our industrial collaborator.

[3]The source code is available on [2]

TABLE I
Results of the case study

Actor	UC	node	state	Q(*L**)	Q(P)	Q(G)	miss s	problem
Broker	3	3	8	0	0	2	0	0
C Monitor	3	2	6	2	1	2	0	0
Exchanger	3	3	12	5	5	1	0	0
T Monitor	10	5	17	34	23	1	2	1
E Monitor	16	4	15	28	15	2	0	14
Trader	36	15	73	75	46	7	5	14
Server	35	24	132	136	125	3	10	23
Total	106	56	263	280	215	18	17	52

TABLE II
Statistics of Volunteers

Occupation	Year of Experience	# of person
Postdoc (software engineering)	≥ 10	5
Industry software engineer	≥ 10	5
PhD (software engineering)	≥ 7	5

There are two common oversights which lead to missing scenarios. The first one is that different ordering of some actions in a use case can lead to different results, however not all possible orderings are considered. For example, for one certain trading strategy, the ordering of "set timer" and "price order" may result in different pricing, and thus different matching on orders. However this kind of ordering-sensitive scenarios are often inadequately specified and the reason, as confirmed with the authors of the use case document, is that they simply did not consider all those situations. Four of the missing scenarios found by our approach belong to this category. The second one is missing some actions in a trace, such as the dashed lines shown in Figure 4. We have found 13 such kind of missing scenarios in the use case document. Those missing scenarios may cause barrier in understanding the system functionality throughout the software development life cycle. The authors of the use cases usually assume background knowledge, which results in those missing scenarios.

Relying only on the preconditions and postconditions to obtain usage relations is inadequate, especially when the use cases are loosely coupled (which is usually the case). But we can still find some cases where the conditions are missing or redundantly stated. Missing preconditions/postconditions affects the integrity of use case documents. Redundant preconditions/postconditions, meaning we can infer the remaining preconditions/postconditions from the given preconditions/postconditions, may cause confusions and understanding barriers. The reason is that not all repeated conditions are redundant. For example, in one use case, the precondition is *login(trader)*$\wedge$*create(server, match)*$\wedge$ *receive(server, order)* and we can infer *create(server, match)*$\rightarrow$*login(trader)* and *create(server, match)*$\rightarrow$*receive(server, order)* from some existing use cases. In this case, *receive(server, order)* is redundant for the use case while *login(trader)* is not since *login(trader)* is the precondition for any operation the trader can conduct in the system. In our method, we find this situation during the generation of relation graphs and raise questions for user confirmation. Among all the 52 use cases which have the stated problems, 50 use cases have missing preconditions, and, 2 use cases have redundant preconditions.

TABLE III
RESULT OF USER STUDY

Find Manually	Find by Tool	#Q Acceptable	Q Meaningful	Tool Useful
2	12	13	10	14

Question 2: User Experience with our tool To answer the second question, we first conduct a user study to test the usability of our tool. We find 15 volunteers for our user study and their statistics are shown in Table II. The volunteers all major in software engineering and have coding experience in related areas for more than 7 years.

We selected 9 use cases (available in [2]) of one actor (i.e., Ticker Monitor) from the industry use case document. The 9 use cases are selected because they capture common system behaviors, such as record creation, update and deletion, and thus it is easier for those volunteers to understand. In the user study, we first ask the participators to read through 9 use cases. Then we ask them to answer 2 questions: (1). Do you understand the functionalities that the use cases describe? (2). Can you find any missing steps or use cases, i.e., actions that you think should have appeared, in any of the use cases?

Then we ask them to use our tool to analyze the use cases they have just read and answer the questions that is generated by our tool. Then they are required to answer the following questions: (1). What do you think of the number of questions asked for each use case, too many or acceptable? (2). Do you think the traces recommended by our tool reasonable? (3). Do you find the tool useful for understanding the use cases or for finding the potential missing actions/traces?

Each volunteer is given 30 minutes to finish the tasks and all of them finished within the time limit. As is shown in Table III, 2 of the 15 volunteers find the potential missing scenarios after reading the use cases. However, 12 of them are able to find the missing traces with the help of our tool. The reason is because our tool recommends the missing traces to the user and it is easier to say yes/no to a trace when it is present to them, than trying to find the missing ones on their own. 13 of them consider the number of questions raised by the tool to be acceptable with respect to the information gained. The other two volunteers recommend that more traces should be filtered. 10 of them rate the generated questions as reasonable. The others find some of the traces recommended are a little redundant or confusing. 14 of them regard the tool as useful for them to elicitate/understand use cases and to find potential missing scenarios.

The manual efforts required in our approach is to answer three kinds of queries. (1) When the active learning algorithm reports some likely missing scenarios, the user need to manually check whether the scenario is a real missing scenario or a false negative. In general, the number of such questions is polynomial to the number of states in the learned DFA and the length of the counterexample in answering candidate queries [5]. Thus, it can be very large, especially when the set of use cases do not share similar alphabets. To reduce the number of membership queries raised to stakeholders, we proposed filtering techniques to filter the traces returned by L^* before raising it to the users, which dramatically reduce the manual checking efforts. (2) Users are required to check the patterns mined from the negative counterexamples as answered by users. (3) During the DFA orchestration, users may be consulted when we find potential missing or redundant preconditions/postconditions.

As we can see from Table I, the average number of all three kinds of questions raised for each use case is less than 5. According to our user study, the majority of volunteers regard this as a reasonable amount of manual effort (with respect to the information gained).

VI. RELATED WORK

Generate behavior models from scenarios Whittle et al. [27] proposed to map a use case charts [26] to a hierarchical state machine. A set of mapping rules are defined from the notation of each level of the use case charts to state machine features. One potential problems with this approach is that the state machine may have many levels, which affect its readability. In [24], Uchitel et al. proposed to synthesis behavior models represented by Model Transition Systems (MTS) [15] from both safety properties, which specify the upper bound of system behaviors, and scenarios, which describe the lower bound of the system behaviors. MTS can properly capture the lower and upper bound of system behaviors simultaneously, which provide guidance for requirement elicitation. Mäkinen et al. [18] adopted the L^* algorithm to learn statecharts from MSC. However the desired language is expressed as a set of traces, which is insufficient to express loops. Moreover, their approach does not consider to reduce the number of membership queries. Our approach captures the desired language with an automaton, which naturally captures loops. We also propose filtering techniques to safely reduce the number of membership queries raised to users. The above approaches all take scenarios captured by MSC as input. However, MSC is a formal structure and is not easy to obtain at first. Usually strong knowledge and experience on UML modeling are required to construct MSC from raw natural language descriptions, which is the initial form of scenarios. Moreover, it is hard for stakeholders to get involved with such a formal structure, which causes difficulties of specification validation. Our approach works directly on scenarios captured by natural language, which facilitates the involvement of stakeholders.

Another kind of work [14], [20], [22], [29], [21] generates UML behavior diagrams [1], e.g., statecharts and activity diagrams, from natural language use case descriptions. However, these approaches only process a single use case and do not consider relations between use cases. Moreover, they rely either on manual rewritten from use cases in natural language to some structured format [20], [29], or on heuristics to process natural language sentences [14].

Find potential problems in scenarios Gervasi and Zowghi [12] 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. Then inconsistencies are checked by reasoning on the propositional formulas. This

work relies on a domain-specific natural language parser CICO [11], which requires the MAS (Model, Action and Substitution) parsing rules to provide domain specific patterns. It also assumes the writing style of the sentences to be consistent with the provided MAS rules. Damas et al. [8] proposed to synthesize LTS from MSCs. They synthesis a global LTS and then project it into local LTS based on different agents. The method modified an existing learning algorithm RPNI [19] to add interactions with users. The learning algorithm does not conduct candidate query and thus suffers from over-generalization problems. To overcome this problem, Damas et al. [9] proposed to inject goals in the form of fluent-based assertions into the synthesize process. Sharing the similar idea with the work by Uchitel et al. [24], the goals/constraints extracted from the domain knowledge help to control the scale of the synthesized model and reduce the number of membership queries. Uchitel et al. [25] proposed an approach which took scenarios in MSC and relations between scenarios described in hMSC as input, then behavior models are synthesized and are used to find implied scenarios. There have been approaches that adopt logic-based learning for the extraction of LTS from scenarios [4]. Different from our approach which process directly on natural language use cases, those approaches take requirement specifications represented by LTL as input and adopt logical reasoning to find missing event/trigger preconditions. Rather than generating models/prototype implementations, which is the main purpose of the above revealed approaches, our work aims at improving the quality of use case scenarios captured in natural language. We value the involvement of stakeholders, which is critical to the integrity of use case documents. Our work is also related to approaches [17], [23] in terms of techniques used. However our work focuses on the early development stages.

VII. Conclusion

We proposed an approach to improve use case documents written in natural language interactively with the guidance of requirement engineers. Our approach adopts natural language processing techniques and an active learning algorithm L^*. We conduct evaluations with an industry case study and results show that our method is able to find missing scenarios and redundant conditions. A user study with 15 software engineers show that the user interactions required by our method is acceptable with respect to the information gained.

References

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

[2] *Project website, https://sites.google.com/site/usecaseanalyzer/*.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large DataBase*, pages 487–499, 1994.

[4] D. Alrajeh, O. Ray, A. Russo, and S. Uchitel. Using abduction and induction for operational requirements elaboration. *Journal of Applied Logic*, 7(3):275–288, September 2009.

[5] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, Nov. 1987.

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

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

[8] C. Damas, B. Lambeau, P. Dupont, and A. Van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[9] C. Damas, B. Lambeau, and A. Van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 197–207. ACM, 2006.

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

[11] V. Gervasi. The cico domain-based parser. Technical report, 2001.

[12] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering Methodol.*, 14(3):277–330, July 2005.

[13] I. Jacobson. Use cases – yesterday, today, and tomorrow. *Software and Systems Modeling*, 3(3):210–220, 2004.

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

[15] K. G. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science, Proceedings of the Third Annual Symposium on*, pages 203 – 210, 1988.

[16] S. Liu, J. Sun, Y. Liu, Y. Zhang, B. Wadhwa, J. S. Dong, and X. Wang. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 785–790. ACM, 2014.

[17] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 345–354. ACM, 2009.

[18] E. Mäkinen and T. Systä. Minimally adequate teacher synthesizes statechart diagrams. *Acta Informatica*, 38(4):235–259, 2002.

[19] J. Oncina and P. Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, volume 5 of *Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

[20] A. Raji and P. Dhaussy. User context models - a framework to ease software formal verifications. In *ICEIS (3)*, pages 380–383. SciTePress, 2010.

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

[22] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *DSN*, pages 327 – 336, 2009.

[23] C. Sun, H. Zhang, J.-G. Lou, H. Zhang, Q. Wang, D. Zhang, and S.-C. Khoo. Querying sequential software engineering data. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 700–710. ACM, 2014.

[24] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Transactions on Software Engineering*, 35(3):384–406, 2009.

[25] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behaviour models using implied scenarios. *ACM Transactions on Software Engineering Methodoloy*, 13(1):37–85, Jan. 2004.

[26] J. Whittle. Specifying precise use cases with use case charts. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2006.

[27] J. Whittle and P. Jayaraman. Generating hierarchical state machines from use case charts. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, number pp. 16-25. IEEE Computer Society, 2006.

[28] M. Wulf De, L. Doyen, T. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In T. Ball and R. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer Berlin Heidelberg, 2006.

[29] T. Yue, L. Briand, and Y. Labiche. An automated approach to transform use cases into activity diagrams. *Modelling Foundations and Applications*, pages 337–353, 2010.

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