Extraction of Message Sequence Charts from Software Use-Case Descriptions

by

Girish Palshikar, Nitin Ramrakhiyan, Sangameshwar Patil, Sachin Pawar, Swapnil Hingmire

in

Conference of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies

Report No: IIIT/TR/2019/-1

Centre for Language Technologies Research Centre
International Institute of Information Technology
Hyderabad - 500 032, INDIA
June 2019

Extraction of Message Sequence Charts from Software Use-Case Descriptions

Girish K. Palshikar **Nitin Ramrakhiyani** **Sangameshwar Patil**
Sachin Pawar **Swapnil Hingmire**
{gk.palshikar,nitin.ramrakhiyani}@tcs.com
{sangameshwar.patil,sachin7.p,swapnil.hingmire}@tcs.com
TRDDC, TCS Research and Innovation, India

Vasudeva Varma
vv@iiit.ac.in
IIIT Hyderabad, India

Pushpak Bhattacharyya
pb@cse.iitb.ac.in
IIT Patna, India

Abstract

Software Requirement Specification documents provide natural language descriptions of the core functional requirements as a set of use-cases. Essentially, each use-case contains a set of actors and sequences of steps describing the interactions among them. Goals of use-case reviews and analyses include their correctness, completeness, detection of ambiguities, prototyping, verification, test case generation and traceability. Message Sequence Charts (MSC) have been proposed as an expressive, rigorous yet intuitive visual representation of use-cases. In this paper, we describe a linguistic knowledge-based approach to extract MSCs from use-cases. Compared to existing techniques, we extract richer constructs of the MSC notation such as timers, conditions and alt-boxes. We apply this tool to extract MSCs from several real-life software use-case descriptions and show that it performs better than the existing techniques.

1 Introduction

Software Development Life Cycle (SDLC) processes generate large and complex natural language text documents, which provide a rich playground for NLP tecnhiques. In particular, NLP techniques have been extensively applied to analyze requirements specifications for early detection of problems such as ambiguity and incompleteness during reviews and inspections; e.g., (Gervasi and Zowghi, 2005; Chantree et al., 2006; Kiyavitskaya et al., 2008; Yang et al., 2011; Ferrari et al., 2017a; Rosadini et al., 2017). Another line of research is concerned with automatically translating software requirements in natural language to various formal models, in order to provide assistance in downstream SDLC tasks like prototyping, verification, test case generation and traceability. Specifically, use-cases provide a textual description of the core functional requirements as sequences of interactions among actors. Hence, Message Sequence Charts (MSC) have been proposed as an expressive, rigorous yet intuitive visual representation of use-cases (Feijs, 2000; Kof, 2008; Yue et al., 2015).

In extracting the MSC from a use-case description, we have to first identify *actors*, which refer to human users, physical objects, systems, subsystems and components. Next, we need to identify *interactions* among the actors in the form of messages of the MSC. The actor which initiates an interaction i.e. sends a message is called the sender and the actors which receive (or experience) the interaction are called receivers. NLP techniques face various challenges in these steps. Firstly, an actor (or an interaction) may be referred in different ways (actor or event co-reference). Secondly, since there is no standardized way of writing use-cases, there is tremendous variety in expressing various aspects of the functionality; e.g., main and alternate flows. While restrictions such as templates or structured English have been imposed for writing use-cases e.g., (Arora et al., 2015), we assume that a use-case is written as a sequence of numbered steps in the main flow, and an alternate flow for any steps in the main flow is specified separately. We impose no linguistic restriction in writing each step in the use-case. Finally, MSC is a rich notation with many complex facilities apart from representing actors and messages. Some of these include alt-boxes, conditions

Use Case: Move to Station
Actors: `Supervisory System, AGV system, motor, vehicle, arrival sensor, robot arm`

Main Flow:
`1. The Supervisory System sends a message to the AGV system requesting it to move to a factory station and load a part.`
`2. The AGV System commands the motor to start moving.`
`3. The motor notifies the AGV System that the vehicle has started moving.`
`4. The AGV System sends a Departed message to the Supervisory System.`
`5. The arrival sensor notifies the AGV System that it has arrived at factory station.`
`6. The AGV System determines that this station is the destination station and commands the motor to stop moving.`
`7. The motor notifies the AGV System that the vehicle has stopped moving.`
`8. The AGV System commands the robot arm to load the part within 10 seconds.`
`9. The arm notifies the AGV System that the part has been loaded.`
`10. The AGV System sends an Arrived message to the Supervisory System.`

Alternate Flow: `1. Step 6 to Step 10: If the vehicle arrives at a different station from the destination station , the vehicle passes the station without stopping and sends a Passed factory station without stopping message to the Supervisory System.`

Table 1: Sample use-case description

and timers. It is challenging to detect appropriate parts of the input text which can be mapped to these constructs. Table 1 shows an example use-case and Figure 1 shows the corresponding MSC.

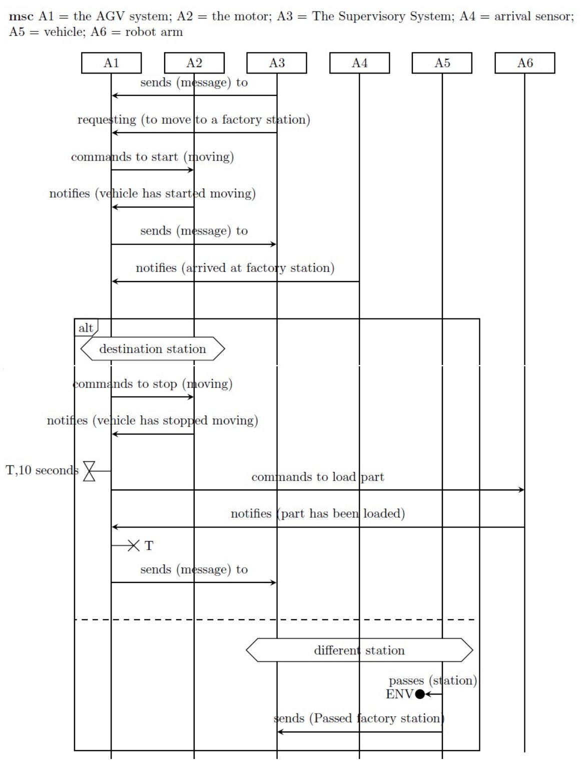
msc A1 = the AGV system; A2 = the motor; A3 = The Supervisory System; A4 = arrival sensor; A5 = vehicle; A6 = robot arm
A1
A2
A3
A4
A5
A6
sends (message) to
requesting (to move to a factory station)
commands to start (moving)
notifies (vehicle has started moving)
sends (message) to
notifies (arrived at factory station)
alt
destination station
commands to stop (moving)
notifies (vehicle has stopped moving)
T,10 seconds
commands to load part
notifies (part has been loaded)
T
sends (message) to
different station
passes (station)
ENV
sends (Passed factory station)

Figure 1: MSC corresponding to the use-case description in Table 1

In this paper, we describe our approach to extract MSCs from use-cases based on Open Information Extraction (OpenIE) (Mausam, 2016). OpenIE extracts structured information from a sentence in the form of relation tuples representing a relation between its subject and one or more objects. We use OpenIE to extract candidate messages and their senders / receivers. We further use WordNet and dependency parsing[1] for filtering the OpenIE candidates and obtaining the final set of messages to be depicted on the MSC.

The MSC representation supports richer constructs such as *timers*, *conditions* and *alt-boxes*. *Timers* represent start and end of specific time durations related to the messages and are shown by hourglass shaped markers. For instance, to capture the information that part loading should be completed in 10 seconds (Step 8 in Table 1), a timer can be used. Accordingly, Figure 1 shows a timer of duration 10 seconds on the timeline of the actor "the AGV system". *Conditions* represent state or situation of one or more actors and are shown as text labels inside hexagonal boxes. In Figure 1, a condition is shown on the timeline of the actors - "the AGV system" and "the motor" denoting the state of the system to be at the destination station. *Alt-boxes* are used to represent an alternate control flow with respect to a certain condition applicable to a set of actors. They are denoted using demarcating lines separating the normal and alternate set of messages. In the example (Table 1), based on the condition of the station at which the vehicle has arrived, the step mentioned as alteranate flow or the steps 6 to 10 of the normal flow need to be followed. This branching is depicted using the alt-

[1]In this paper, we use the Stanford CoreNLP (Manning et al., 2014) pipeline for dependency parsing.

box shown in Figure 1.

The paper is organized as follows. Section 2 covers the related work; Section 3 describes the MSC extraction approach; Section 4 describes the experiments and discusses one of the use-cases in detail; Section 5 concludes the paper.

2 Related Work

Feijs (2000) studies the relationship between natural language use cases and message sequence charts. He uses context-free generative grammars for natural language description of use-cases that use object-oriented system development methods. However, this study focuses on simple sentences and sentences with coordination and subordination. Moreover, this study does not discuss empirical evaluation of the use-case to MSC conversion.

Kof (2007a) proposes a method for representing a software requirement document using a MSC. This method takes a software requirement document and a list of valid actors as its inputs. For a verb in a sentence it generates a message such that the valid actor appearing before the verb is identified as the sender, the valid actor appearing immediately after the verb is identified as the receiver and the text starting from the verb and ending before the receiver is identified as the message label. Kof (2007b) further extends this approach to include sentences with multiple verbs, conjunctions and passive voice. Kof (2007a) also proposes a set of heuristics to handle missing sender or receiver of a message. A key limitation of this method is that it ignores syntactic as well as semantic relations between the verb and its corresponding sender or receiver. Hence it is possible that some of the <sender, message label, receiver> tuples may not be valid semantically. It is important to note that Kof (2007a; 2007b) do not focus on other features of MSC such as conditions, timers and alt-boxes.

3 MSC Extraction

In this section, we describe our approach for creation of an MSC for a given use-case description. The approach involves running OpenIE based extraction on the input use-case text and processing it further for extracting the MSC elements namely actors and messages. Additionally, the approach applies a set of linguistic rules to extract conditions, alt boxes and timers.

Processing input using OpenIE: Use-case descriptions generally use well-written English. Over such text, generation of candidate messages requires a simple relation and argument extractor. We thus propose use of the OpenIE framework which provides tuples of the form (left_argument, relation, right_argument_1, right_argument_2, ...) along with confidence scores with each tuple. We first process the input use-cases through the OpenIE technique described in (Mausam et al., 2012; Mausam, 2016) and obtain a list of candidate messages.

Defining Actors in Software Engineering Use-Cases: In general (Bedi et al., 2017; Patil et al., 2018; Palshikar et al., 2019), the notion of actors in MSC is based on named entities of the types - PERSON, LOCATION and ORGANIZATION. However, in the software requirements domain, these may not be the only entities interacting with each other. This criteria is extended and is proposed to include:

- PERSONS (Human SYSTEM users), ORGANIZATIONS and LOCATIONS
- SYSTEMS (such as Supervisory System, Library System)
- Persistent components of SYSTEMS (such as servers, databases, customer accounts)
- Persistent processes in the SYSTEM (such as schedulers, daemons)
- SYSTEM components (GUI elements like buttons, menus, and similar)

Also, as part of the definition of actors, it is important to exclude the following entities seen in software requirements text.

- Attributes of the above entities (such as usernames, passwords)
- Transient entities and business processes (such as requests, responses, registration process)

To realize this definition of actors we propose a WordNet based approach and check if a candidate actor is a valid actor. We refer to this verification while identifying messages.

We consider all noun phrases as candidate actors and then filter them based on two criteria. If the head word of the noun phrase has certain specific senses as part of its WordNet hypernym hierarchy, we consider it as a valid actor. These specific senses are gathered manually through observation and consist of “Physical Object” (sense 1), “thing” (sense 4 and sense 12), “matter” (sense 3), “substance” (sense 1 and 4) and “Causal agent” (sense 1). This filters out abstract entities which are not actors such as `request`, `response` and `message`.

To allow computer and system related nouns which are abstract in nature and have none of the above specified senses in their hypernym path, we apply the second criteria. We allow nouns with head words having the following senses in their hypernym hierarchy: “Computer”, “Computer File” and “database”.

Identifying Messages: For identifying messages, we begin by considering output of the OpenIE tool on the use-case text which gives us a set of tuples in each sentence. These candidate message tuples are given input to the message identification approach and the list of filtered messages with their senders and receivers are obtained as output. The message identification approach is described in Algorithm 1.

The algorithm iterates over the set of OpenIE tuples. For each tuple, it checks if the left argument is a valid actor which is the sender of the message. The valid actor check is based on the proposed definition of actors. If the left argument is not a valid actor the tuple is ignored. This ensures that each message has a valid sender.

For each right argument, if it has a single dependency subtree and is a valid actor, the tuple is regarded as a valid message. If the subtree is not a valid actor but is dependent to the relation it is appended to the relation string. In the case when there are multiple subtrees in a right argument, subtrees with their head words as valid actors are considered receivers of the message.

Algorithm 1: $identify\_messages$

```
Input: T = Set of tuples given by OpenIE for a
       sentence. T = t; t_confidence >= threshold
       where each t = (left argument, rel, right
       arguments...)
Output: set of messages $M$ with each message $m$
        having three attributes: sender, message label
        (msg_label) and set of receivers
1  foreach $tuple\ t \in T$ do
2      m := ()
3      if $t.left\_arg$ is a valid actor then
4          m.sender := t.left_arg
5      else
6          continue
7      m.msg_label := t.rel
8      foreach $arg \in t.right\_arguments$ do
9          if arg contains only one dependency
           subtree then
10             if arg is a valid actor then
11                 m.receivers ∪ arg
12             else if head of t.rel is governor
               of head of arg then
13                 m.msg_label := append(m.msg_label,
                   arg)
14         else
15             foreach $dep\_st \in arg's$
               disjoint dependency subtrees do
16                 if dep_st headed by a noun
                   and is valid actor then
17                     m.receivers ∪ arg
18                 else
19                     if head of t.rel is governor of
                       head of dep_st then
20                         m.msg_label =
                           append(m.msg_label,
                           dep_st)
21     foreach r in m.receivers do
22         new_tuple := (m.sender, m.msg_label, r)
23         M = M ∪ (new_tuple)
24 return M
```

Extracting Conditions: We consider a text fragment in a use-case as a *condition*, if it denotes a state or property and the set of associated actors are those that are “involved” in that condition. A condition text is often either a verb phrase (VP) (`has stopped moving`), a noun phrase (NP) (`power failure`) or an adjective phrase (ADJP), though sometimes a prepositional phrase (PP) is also observed. Since not all VPs or NPs are conditions, we have designed a set of linguistic rules to identify conditions: (1) Any ADJP connected to a copula (or copula-like) verb and also to an actor through **nsubj** dependency relation (`The valve is open`). (2) Any VP connected to a copula (or copula-like) verb through **auxpass** dependency relation and also connected to an actor through **nsubjpass** dependency relation (`The page is refreshed`). (3) Any VP connected to a copula-like verb v (e.g., `remains`) through an **xcomp** dependency and v is connected to an actor through **nsubj** (`The engine remains idling`). (4) A VP V connected to cue phrases like `if`, `upon`, `in case of`, where the head verb of V has an actor as a part of its dependency subtree (`If power to the pump fails`). (5) If a NP is predicative nominal of another NP, the text segment comprising these NPs is considered as a condition (`This station is the destination station.`)

Extracting Alt-Boxes: Alt boxes in the MSC representation are useful to represent paths alternate to the normal flow of events. This is an important construct especially for use-case descriptions as alternate steps of action occur frequently with software systems. For identifying these alternate paths and representing them as alt boxes in MSCs, we harness the structure of use-case descriptions. As per our observation, there are generally two styles in which alternate paths are specified in use-case descriptions. In one format, the alternate flow is specified separately from the normal. The alternate flow provides a pointer to the set of steps in the normal flow to which it is alternate to. We use this structural information and capture messages from the alternate flow to be shown in "alt" to the corresponding normal flow messages. In another format, the alternate step is specified just after the normal flow step marked by a set of conditional prepositions such as `if` or prepositional phrases such as `in case, in the event of` or adverbs such as `else, otherwise`. We identify these markers with the help of regular expression based patterns and separate the alternate step from the normal step. The later format however, does not provide scope information of alternate steps. Based on observation of multiple use-cases of real software systems, we make an assumption that the scope of the alternate flow begins at the normal flow step with which it is specified and lasts till the last step in the normal flow.

Extracting Timers: MSC supports an important construct - the "timer", to capture interactions which must happen in a time-bound manner. A timer may be attached to an actor's time-line or a condition. Timers in a use-case description are extracted through two steps. Firstly, we identify the durative time expressions in the input use-case description using regular expression based patterns such as $[0-9]+(milliseconds|seconds|minutes|hours|days)$ (numbers followed by time period expressions like milliseconds, seconds, minutes, hours, days). Secondly, using the OpenIE output, we identify the relation for which the time expressions identified in the previous step appears as a temporal argument. We attach a timer to time-line around the message.

4 Experimentation Details

4.1 Dataset and Evaluation

We report results in this paper on a set of 4 use-cases obtained from publicly available Software Requirement Specifications (SRS) of real life software systems (Ferrari et al., 2017b). These use-cases are AGV (Automated Guided Vehicle System), G6 & G16 (gamma-j web order system) and EMS (Electronic Monitoring System). Additionally, we use one more use-case (TRAIN) from an internal project dealing with Automatic Train Control systems.

We manually create the gold standard MSC for each use-case and use them to evaluate our extraction system. As part of the evaluation, we compare the performance of our system with a baseline technique proposed in (Kof, 2007a,b).

We evaluate the proposed approach on five levels of increasing complexity starting from actor identification to complete message extraction:

1. Actors : At this level we evaluate the predicted actors with respect to the gold actors. A predicted actor is a true positive if its complete phrase is present exactly in the set of gold actors. False positives and false negatives are accordingly computed.

2. Message label : At this level we evaluate only the message label of each predicted message with respect to labels of gold messages. A predicted message label from a sentence is considered as a true positive if the main verb of the label matches the main verb of a gold message from the same sentence. False positives and false negatives are accordingly computed.

3. Message label + Sender : At this level we evaluate the combination of message label and sender of each predicted message with respect to the same combination for gold messages. A predicted combination from a sentence is considered as a true positive if it matches the combination from a gold message from the same sentence.

4. Message label + Receiver : At this level we evaluate the combination of message label and

Dataset		Actors	Message Label	Sender	Receiver	Complete Message
AGV	B	0.552	0.083	0.080	0.071	0.071
	M	0.800	0.741	0.741	0.581	**0.581**
G6	B	0.667	0.667	0.667	0.571	0.571
	M	0.778	0.947	0.947	0.727	**0.727**
G16	B	0.667	0.769	0.769	0.571	0.571
	M	0.800	1.000	1.000	0.933	**0.933**
EMS3	B	0.727	0.333	0.333	0.333	0.333
	M	0.909	0.750	0.750	0.750	**0.750**
TRAIN	B	0.556	0.667	0.444	0.526	0.421
	M	0.941	0.800	0.800	0.706	**0.706**

Table 2: Comparative performance for MSC Extraction. M: Our approach described in Algorithm 1. B: Baseline approach based on Kof (2007a; 2007b)

receiver of each predicted message with respect to the same combination for gold messages. We compute F1-measure on similar lines as above.

5. Message label + Sender + Receiver : At this level we evaluate the complete message i.e. combination of message label, sender and receiver with respect to the complete gold messages. We compute F1-measure on similar lines as above.

As each level's performance, we report the F1-measure in Table 2. Our approach outperforms the baseline on all datasets on all evaluation levels.

For extraction of the complex constructs like conditions, alt-boxes and timers, we employ a set of simple rules described earlier. The baseline technique proposed in (Kof, 2007a,b) does not focus on identifying these constructs.

4.2 Analysis

It is important to note that performance of the proposed approach is dependent on the performance of tools in the NLP processing backend (which are WordNet, Stanford CoreNLP and OpenIE in this case). We highlight a few error cases to explain the NLP challenges encountered.

OpenIE fails to identify any relation in some sentences. For example, in the sentence `System stores order confirmation and order details.`, OpenIE does not generate any relation of the form <`System, stores, order confirmation`>, because it identifies "stores" as a noun. Hence our approach fails to identify the messages in this sentence.

In the context of use-case description texts, there are multiple words which have domain specific senses which are not captured by resources like WordNet, e.g., "flag", "ticket", "turnkey", etc. As our approach depends on WordNet to identify valid actors, it may identify certain spurious messages. For example, in the sentence `System appends cookie with flag for completed checkout process.`, our approach identifies "flag" as a valid actor even though in this context it is not. Hence, it creates an incorrect message with sender as "System" and "flag" as receiver.

In use-cases we frequently observe that the actors are part of multiple interactions and a single actor is referred by multiple lexical mentions. Hence, it is necessary to perform coreference resolution to group multiple mentions of an actor and represent it using a canonical mention in the MSC. However, we observed that performing coreference resolution to group coreferring actor mentions together, degrades the performance. The average (over all use-cases) complete message identification performance degrades from an F1 of 74% to 45%. This is because of several incorrect coreference links identified by the Stanford CoreNLP toolkit. E.g., consider the following extract from a use-case: `The Supervisory System sends a message to the AGV system requesting it to move to a factory station and load a part. The AGV System commands the motor to start moving.` Here, Stanford CoreNLP incorrectly identifies `The Supervisory System` in the first sentence and `The AGV System` in the second sentence as coreferences.

These examples point out the NLP challenges faced in automated extraction of MSC and the need for further research.

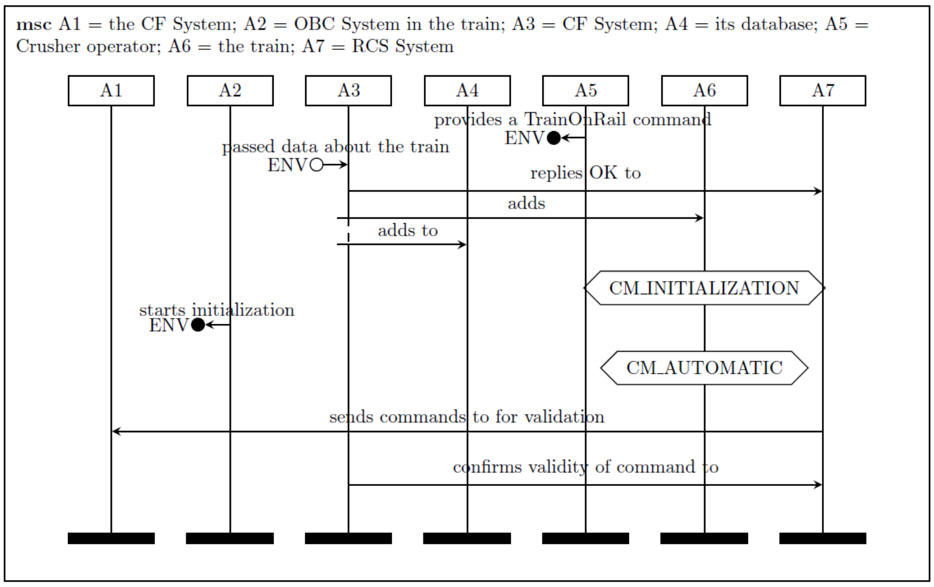
msc A1 = the CF System; A2 = OBC System in the train; A3 = CF System; A4 = its database; A5 = Crusher operator; A6 = the train; A7 = RCS System
A1
A2
A3
A4
A5
A6
A7
provides a TrainOnRail command
ENV
passed data about the train
ENV
replies OK to
adds
adds to
CM_INITIALIZATION
starts initialization
ENV
CM_AUTOMATIC
sends commands to for validation
confirms validity of command to

Figure 2: MSC for sample use-case about addition of a new train in the automated train system

4.3 Discussion on the TRAIN Use-Case

We have used the MSC to analyze the software use-cases for an industrial computer-controlled automated train system. It is used to move ores from mines. The automated train system has following key components: (i) the Railway Controller Software (RCS) system, (ii) the Checker Functionality (CF) system, (iii) the On-Board Computer (OBC) system. The locomotive train is controlled by the RCS. This system is used by various operators such as the crusher operator, etc. A brief use-case about addition of a new locomotive train into the the set of trains already under control of the RCS is described below:-

1. Crusher operator provides a *TrainOnRail* command.
2. The data about the train is passed to CF System.
3. CF System replies OK to RCS System.
4. CF System adds the train to its database.
5. At the beginning the state of the train is CM_INITIALIZATION.
6. OBC System in the train starts initialization.
7. After initialization, the state of the train becomes CM_AUTOMATIC.
8. RCS System sends commands to the CF System for validation.
9. CF System confirms validity of command to the RCS System.

Figure 2 shows the MSC for this use-case. Using the MSC, we are able to identify the gaps in the use-case specification as well as generate test-cases from the use-case. For instance, in the above use-case, the expected behaviour of the automated train system is not specified if the RCS sends a command while the train is still in the CM_INITIALIZATION state. The domain expert then clarified that the RCS should never send the commands till the train state becomes CM_AUTOMATIC. Based on the tools and techniques (Alur et al., 1996) developed for verifying MSC properties, it can be verified that such a sequence of infeasible actions is not specified in the set of given use-cases. Also, it can be verified that there are no contradictions in the given set of use-cases. It is also possible to develop test-case outlines using the MSCs prepared from the use-case descriptions. Such test-case generation is part of our future work.

5 Conclusions

We explored automatic extraction of Message Sequence Charts (MSC) from use-case descriptions in Software Requirement Specification documents. In this paper, we described an Open IE based approach which uses linguistic knowledge such as dependency parsing and WordNet hypernyms to extract MSCs from use-cases. Compared to existing techniques, we also extracted richer constructs of the MSC notation such as timers, conditions and alt-boxes. Our approach outperforms the baseline on a dataset of five real-life use-cases.

References

Rajeev Alur, Gerard J Holzmann, and Doron Peled. 1996. An analyzer for message sequence charts. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 35–48. Springer.

Chetan Arora, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2015. Automated Checking of Conformance to Requirements Templates Using Natural Language Processing. *IEEE Trans. Software Eng.*, 41(10):944–968.

Harsimran Bedi, Sangameshwar Patil, Swapnil Hingmire, and Girish K. Palshikar. 2017. Event Timeline Generation from History Textbooks. In *Proceedings of the 4th Workshop on Natural Language Processing Techniques for Educational Applications, NLP-TEA@IJCNLP 2017, Taipei, Taiwan, December 1, 2017*, pages 69–77.

Francis Chantree, Bashar Nuseibeh, Anne N. De Roeck, and Alistair Willis. 2006. Identifying Nocuous Ambiguities in Natural Language Requirements. In *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*, pages 56–65.

Loe M. G. Feijs. 2000. Natural language and message sequence chart representation of use cases. *Information & Software Technology*, 42(9):633–647.

Alessio Ferrari, Felice Dell'Orletta, Andrea Esuli, Vincenzo Gervasi, and Stefania Gnesi. 2017a. Natural Language Requirements Processing: A 4D Vision. *IEEE Software*, 34(6):28–35.

Alessio Ferrari, Giorgio Oronzo Spagnolo, and Stefania Gnesi. 2017b. PURE: A Dataset of Public Requirements Documents. In *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*, pages 502–505.

Vincenzo Gervasi and Didar Zowghi. 2005. Reasoning About Inconsistencies in Natural Language Requirements. *ACM Trans. Softw. Eng. Methodol.*, 14(3):277–330.

Nadzeya Kiyavitskaya, Nicola Zeni, Luisa Mich, and Daniel M. Berry. 2008. Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requir. Eng.*, 13(3):207–239.

Leonid Kof. 2007a. Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 121–130.

Leonid Kof. 2007b. Treatment of Passive Voice and Conjunctions in Use Case Documents. In *Natural Language Processing and Information Systems*, pages 181–192, Berlin, Heidelberg. Springer Berlin Heidelberg.

Leonid Kof. 2008. From textual scenarios to message sequence charts: inclusion of condition generation and actor extraction. In *6th IEEE international requirements engineering conference, (RE'08)*, pages 331–332.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60.

Mausam. 2016. Open Information Extraction Systems and Downstream Applications. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 4074–4077.

Mausam, Michael Schmitz, Stephen Soderland, Robert Bart, and Oren Etzioni. 2012. Open Language Learning for Information Extraction. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 523–534.

Girish K. Palshikar, Sachin Pawar, Sangameshwar Patil, Swapnil Hingmire, Nitin Ramrakhiyani, Harsimran Bedi, Pushpak Bhattacharyya, and Vasudeva Varma. 2019. Extraction of Message Sequence Charts from Narrative History Text. In *Proceedings of the Workshop on Narrative Understanding*.

Sangameshwar Patil, Sachin Pawar, Swapnil Hingmire, Girish K. Palshikar, Vasudeva Varma, and Pushpak Bhattacharyya. 2018. Identification of Alias Links among Participants in Narratives. In *ACL 2018*.

Benedetta Rosadini, Alessio Ferrari, Gloria Gori, Alessandro Fantechi, Stefania Gnesi, Iacopo Trotta, and Stefano Bacherini. 2017. Using NLP to Detect Requirements Defects: An Industrial Experience in the Railway Domain. In *Requirements Engineering: Foundation for Software Quality - 23rd International Working Conference, REFSQ 2017, Essen, Germany, February 27 - March 2, 2017, Proceedings*, pages 344–360.

Hui Yang, Anne N. De Roeck, Vincenzo Gervasi, Alistair Willis, and Bashar Nuseibeh. 2011. Analysing anaphoric ambiguity in natural language requirements. *Requir. Eng.*, 16(3):163–189.

Tao Yue, Lionel C. Briand, and Yvan Labiche. 2015. aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models. *ACM Trans. Softw. Eng. Methodol.*, 24(3):1–52.