Pattern-based Analysis of Interaction Failures in Systems-of-Systems: a Case Study on Platooning

Sangwon Hyun, Jiyoung Song, Seungchyul Shin, Young-Min Baek, and Doo-Hwan Bae
School of Computing
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, South Korea
{swhyun, jysong, scshin, ymbaek, bae}@se.kaist.ac.kr

***Abstract*—Interactions between software components play a major role in the achievement of goals in complex systems, such as platooning System-of-Systems (SoS). A platooning SoS groups vehicles in order to increase their fuel efficiency and alleviates traffic congestion by enabling driving in close proximity using operation protocols. In a platooning SoS, the execution of typical operations, such as `Leave` or `Merge`, consists of 20 micro-operations on average. Owing to this overabundance of sub-operations, interaction failures in a specific operation sequence can occur in an SoS execution. Further, analyzing the root cause of such failures is highly time-consuming, due to the density of the constituent interactions. Existing techniques suffer from two limitations: (1) The majority of the root cause analysis techniques are not capable of isolating faulty interaction sequences, because they do not directly utilize interaction data; (2) The majority of the fault diagnosis techniques assume the pre-examined fault knowledge base, which needs too high cost due to limited knowledge in an SoS. To effectively analyze interaction failures in an SoS, we propose a pattern-based faulty interaction analysis technique. To this end, an interaction model is first defined for an SoS, followed by the proposal of a suspicious interaction pattern mining algorithm. During the case study using a platooning simulator, the technique automatically abstracts interaction data from logs and extracts faulty interaction patterns, thereby enabling the identification of seven new unreported interaction failure scenarios. The conclusions of this study can enrich the general fault knowledge base for platooning SoS.**

***Keywords*—Platooning System-of-Systems, Interaction Failure, Fault Analysis**

I. Introduction

Recently, platooning System-of-Systems (SoS) has garnered huge attention in research due to its positive impacts on fuel efficiency and traffic congestion, leading to various environmental and business benefits [1], [2]. An SoS is a complex, heterogeneous system comprising independent Constituent Systems (CSs) to achieve common goals that cannot be achieved by a single CS [3]. A platooning SoS operates by grouping vehicles in close proximity into one unit and manages them using operation protocols, such as `Leave` and `Merge`, to achieve SoS-level goals [4]. One of the characteristics of an SoS is that most SoS-level operations involve intricate interactions among its CSs. Because of this characteristic, there exist certain types of failures, called interaction failures, caused by a specific faulty sequence of interactions.

Even though many global companies, such as Daimler (Mercedes-Benz Trucks) [5], Volvo [6], and Hyundai [7], have succeeded in operating platoons on real roads, it is difficult to guarantee the faultless functioning of such a highly complex software with respect to all possible interactions. Fig. 1 depicts an example of the interaction failures, which is induced by simultaneous `Leave` and `Merge` requests. In this example, the leader of platoon-1, v1, tries to leave the platoon, but the leader of platoon-2, v5, transmits a `Merge` request to v1 before v1's `Leave` operation is completed. This situation inhibits the normal execution of `Merge` operations and adversely affects the execution time of the ongoing `Leave` operation. In order for SoS to work without interaction failures, it is important to systematically identify and analyze faulty interaction sequences in advance, which are fundamental causes of interaction failures.

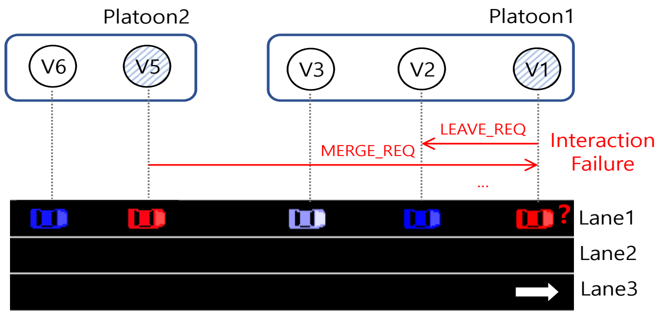


Fig. 1. An example interaction failure scenario in Platooning SoS

Recent studies have applied a Spectrum-based Fault Localization (SBFL) to isolate the most suspicious components of failures in large-scale systems [8], [9]. Shin *et al.* [8] applied the SBFL technique to disaster-response SoS to identify faults injected within it. Arrieta *et al.* [9] also applied the SBFL technique to localize the most suspicious feature in a software product line. However, existing techniques suffer from certain limitations in the identification of suspicious interaction sequences. Firstly, the techniques do not directly utilize detailed interaction data to analyze the fundamental cause. Instead, they only abstract high-level information from systems, such as participant CSs, and the existence of the connections between CSs. Secondly, due to the infinite number of combinations of interaction sequences, it is infeasible to apply SBFL to extract faulty interaction sequences.

Other existing studies have focused on applying machine learning techniques to diagnose root causes underlying failures in an SoS [10], [11]. Kleyko *et al.* [10] constructed a matrix-based abstraction model of a nuclear power plant and proposed a hyperdimensional vector-based diagnostic approach to efficiently match patterns based on a fault knowledge base. Cai *et al.* [11] suggested a fast Object-Oriented Bayesian Network (OOBN)-based fault diagnosis model for a subsea production system. The primary limitation common to these techniques is that they only consider the "known faults" assuming the existence of the pre-examined fault knowledge base of a system. In an SoS, each CS is regarded as a black-box system because of its operational and managerial independence. Thus, assuming the well-specified fault data of an SoS is practically infeasible. The existing techniques that require a fault knowledge base are not suitable for an SoS, which is difficult to expect pre-examined fault data.

To overcome the aforementioned limitations of existing approaches, we propose a pattern-based failure-inducing interaction analysis technique, which mines the most suspicious interaction patterns from the logs of failed executions. The major contributions of this study are as follows:

- We suggest an interaction model for an SoS, which abstracts interaction features and operation sequences on each SoS execution log.
- We define an approach to automatically mine faulty interaction patterns by extending the Longest Common Subsequence (LCS) algorithm.
- We identify detailed interaction failure scenarios in a platooning SoS, which can be used as testing benchmarks and a fault knowledge for general platooning systems.

By applying the interaction model and pattern mining algorithm, we conduct a case study on a platooning SoS using *StarPlateS* [12]. In the case study, we categorize the detected failures into three classes and develop seven detailed interaction failure scenarios without any existing fault knowledge on the platooning SoS. To the best of our knowledge, this is the first study to provide interaction failure scenarios with respect to the platooning SoS.

The remainder of this paper is organized as follows: Section 2 explains the background. Section 3 elucidates the proposed approach, and Section 4 presents the case study on the platooning SoS and evaluates the case study result. Section 5 describes works related to this research. Section 6 discusses the implications of our findings, recommends directions for future work, and concludes the paper.

II. Background

A. StarPlateS: Statistical Verification Framework for Platooning SoS

A platooning SoS is a highly complex system, which should be capable of functioning in as many environments as possible to ensure reliable operation. Since it is difficult to test a platooning SoS in all possible real-world scenarios, a simulator is necessary [4]. Several recent studies have investigated simulation and verification of platooning SoS [12], [13], [14], [15]. In order to focus on interactions between platooning vehicles during various operations, we used StarPlateS [12] for this purpose, which focuses on platooning operations and realistic scenario generation.

StarPlateS is a VENTOS [4] extended framework that considers internal and external uncertain factors enabling it to deal with realistic scenarios. The implementation of roads and vehicles in StarplateS is based on the SUMO simulator and communication between vehicles is based on OMNET++ simulator. With the integration of the simulators, StarplateS generates random platooning configurations and scenarios, generates execution logs for the scenarios, and verifies platooning goals using the logs. In this context, a platooning configuration denotes a set of platoon generation features, and a scenario denotes a sequence of basic operations in the platooning SoS- `Merge`, `Split` and `Leave`. During the Merge operation, two distinct platoons merge into a single platoon, while the Split operation does exactly the opposite. The Leave operation is executed when a member of a platoon wishes to leave the platoon. During `Merge` operation, two distinct platoons merge into one platoon, while `Split` operation does exactly the opposite. `Leave` operation is executed when a member of the platoon wants to leave out of the platoon.

In this paper, we used two primary modules in StarplateS- the scenario generation and the simulation module in the case study. The scenario generation module generates random platooning configurations and scenarios, then the simulation module runs the VENTOS simulator to generate execution logs for the scenarios.

B. Spectrum-Based Fault Localization (SBFL) Technique

To identify failure-inducing interactions based on the data present in logs of failures, we propose a pattern mining-based fault localization technique. A fault localization technique pinpoints suspicious locations in a program, such as statements, that merit the programmers' attention [16]. Program locations that appear to be erroneous are called suspicious locations. Corresponding to any set of test cases, including failed cases, a list of suspicious locations in the program is produced.

The SBFL techniques utilize code coverage corresponding to each test case to determine suspicious locations. Code coverage, which is also called the program spectrum is an execution trace of a program [16] with respect to a specific input. The basic concept of SBFL techniques for determining suspicious locations is that the more program location is executed in failed cases, the more it is considered suspicious.

In the case of large and complex SoS, identification and correction of failures are effort-intensive tasks. Since fault localization techniques help engineers to analyze the root causes and occurrence contexts of failures, SBFL techniques have been used for localizing faults in large-scale complex systems, such as a disaster-response SoS and a software product line [8], [9]. However, our study deals with interaction failures in a platooning SoS, in particular. In order to effectively extract faulty interactions from logs of failures, we propose an interaction model for the SoS and a Longest Common Subsequence (LCS)-based pattern mining algorithm.

III. Failure-inducing interaction pattern mining

A. Overall Process

We propose a pattern-based analysis technique to process the interaction data present in SoS execution logs and extract failure-inducing interaction patterns. Fig. 2 depicts the overall process of the proposed approach. It comprises three major phases — interaction model generation, Dynamic Programming for Longest Common Subsequence (DP-LCS)-based pattern mining, and analysis of the identified patterns.

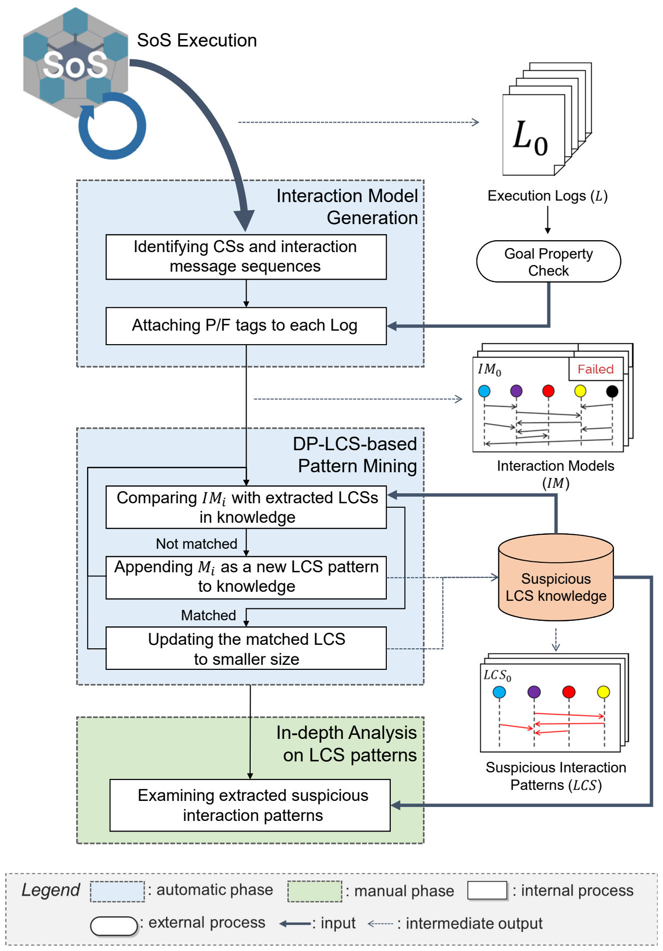
The technique uses two inputs: execution logs and their *Passed/Failed* results of goal property checking. Based on the data present in the logs, the proposed technique first identifies the CSs and their interactions that are executed during a single run of the SoS. Interactions refer to sequences of communication traces (e.g., messages) that are captured in a communication network. Then, the proposed technique attaches the goal property check result of each log to each Interaction Model (IM). Each interaction model is generated by aggregating a set of identified CSs, a sequence of messages, and a *Passed/Failed* tag. We assume the existence of an external goal verification module, due to the scalability for diverse domains. For instance, in the case study presented later, we used SIMVA-SoS [17] to inspect each execution log. Further details about IMs and IM-generation have been included in the next subsection.

The next phase comprises the extraction of patterns of suspicious message sequences from the generated IMs. Since it is inadequate to assume the existence of a single bug in the SoS due to its large scale and complexity, we assume that the SoS suffers from multiple faults and each failed IM may contain one or more faulty patterns. The proposed technique searches for the existence of LCSs between each pair of failed IMs and assigns such pairs to the same category. Via this process, each category eventually becomes populated by failed IMs, which can be expected to have been induced by similar root causes. The extracted LCS pattern corresponding to each category, which is a specific message subsequence observed to be common to all the IMs belonging to that category, can be used to analyze the fundamental cause and occurrence context of the associated failure.

By analyzing the output patterns in detail, SoS engineers can gain an understanding of the root causes and occurrence contexts of failures by analyzing the categorized LCS patterns and IMs. We demonstrate detailed examples of outputs at the end of this section and outline the manual analysis process and its results in the next section.

B. Interaction Model Generation

The system abstraction model plays an important role in fault analysis in complex systems because the root causes of failures can be isolated based solely on the abstracted information of a system model. For instance, existing studies have utilized high-level information of system execution [8], [10]. They abstracted the participating CSs and the communication between them. However, the existing abstraction



SoS
SoS Execution
L_0
Execution Logs (L)
Goal Property Check
Interaction Model Generation
Identifying CSs and interaction message sequences
Attaching P/F tags to each Log
IM_0
Failed
Interaction Models (IM)
DP-LCS-based Pattern Mining
Comparing IM_i with extracted LCSs in knowledge
Not matched
Appending M_i as a new LCS pattern to knowledge
Matched
Updating the matched LCS to smaller size
Suspicious LCS knowledge
LCS_0
Suspicious Interaction Patterns (LCS)
In-depth Analysis on LCS patterns
Examining extracted suspicious interaction patterns
Legend
: automatic phase
: manual phase
: internal process
: external process
: input
: intermediate output

Fig. 2. Overall process of the pattern-based interaction failure analysis

models do not include internal sequences of interactions and contexts of execution, rendering the existing models unsuitable for interaction-failure analysis. In order to utilize sufficient interaction features, such as occurrence context and execution flow, in fault analysis, we define the interaction model (IM) and construct an automated IM generation module from execution logs. The IM is defined as follows:

$$IM_i = (CS_i,\ M_i,\ tag_i)$$
$$CS_i = \{cs_j \mid cs_j \text{ is a participating CS in } L_i,\ i\text{-th Log}\}$$
$$M_i = \{msg_k\}_{k=0}^{l_i},\ \text{which are delivered in } L_i,$$
$$\text{where } msg_k = \langle continuity,\ synchronization,\ sender,\ receiver,\ content,\ time\rangle$$
$$tag_i = Passed \text{ or } Failed \text{ result of } L_i \text{ on a goal property}$$

IM_i, which is an instance of IM generated from L_i, consists of three components: CS_i, M_i, and tag_i. CS_i is a set of cs_j instances that participate in an SoS during the execution of the log, L_i. The participation of a cs_j comprises serving a specific function for the achievement of common SoS-level goals. For instance, in the platooning SoS, we marked vehicles capable of executing platooning operations as participating CSs.

M_i denotes an ordered sequence of messages, with its length l_i that are communicated during the execution of L_i. In other

TABLE I
FEATURES OF COMMUNICATION MESSAGES FOR INTERACTION MODEL

Feature	Type	Example
Continuity	Enumeration of Continuous Communication (CC) and Temporary Communication (TC)	TC
Synchronization	Enumeration of Synchronous (Sync) and Asynchronous (Async)	Sync
Initiating Entity	String	CS_A
Receiving Entity	String	CS_D
Content	Message	Operation Request
Start Time	Time	2019/12/24:000000
End Time	Time	2019/12/24:000159
Delay	ms	159

words, M_i is the formal representation of a sequence diagram that is executed in L_i. Each message msg_k in the M_i denotes a vector of SoS interaction features. To extract interaction features in an SoS, we refer to existing studies that identify features of interaction and communication in various domains, such as autonomous vehicles [18], telecommunication systems [19], and web services [20], [21].

Table I depicts the features that are used to represent message-based interactions and their types. The *Continuity* feature determines whether a sender sends a single message (Temporary Communication, TC) or a stream of messages (Continuous Communication, CC). The *Synchronization* determines whether a delivered message is synchronous (Sync) or asynchronous (Async). The *Continuity* and *Synchronization* features together capture the concurrency properties among the CSs. The *Initiating/Receiving Entity* refers to the sender and the receiver of any message. The *Content* feature describes the contents of a message, while *Start Time*, *End Time*, and *Delay* are used to record the sending and receiving times of the message in the IM. Example values of the *Content* feature include 17 micro-commands defined in a platooning operation protocol [4], such as $LEAVE\_REQ$, $LEAVE\_ACCEPT$. The *Content* feature can be used to assess certain types of conflicts, such as interface conflicts and goal conflicts between CSs. The *Initiating/Receiving Entity* and *Content* features can be used to assess direct and indirect resource conflicts during the system operations by analyzing the relationships of *Initiating Entity* and *Receiving Entity* with *Content* [18].

The final component of the interaction model is tag_i. Each execution log can be evaluated by whether it satisfies a certain goal of the SoS. The goal satisfaction result corresponding to the i-th execution log, $Passed/Failed$, is assigned to tag_i.

We implemented the aforementioned IM generation module, which automatically identifies participating CSs and interaction features of L_i. Following identification, the module attaches a tag_i to each IM_i. The external module analyzes the logs to determine whether L_i passes a certain goal. Then, the $Passed/Failed$ result is stored in the tag_i in IM_i.

C. DP-LCS-based Pattern Mining Algorithm

The next phase involves the mining process of suspicious interaction sequences based on Dynamic Programming for

Algorithm 1: Suspicious interaction pattern mining

```
Input  : A set of $IM$s with size $N$
Output: A set of extracted $LCS$s with detected counts
1  suspKnowledge ← ∅;
2  matched$IM$s[][] ← ∅;
   // Compare $IM_i$ with existing
      knowledge set
3  for $i \leftarrow 0$ to $N-1$ by 1 do
4      for $j \leftarrow 0$ to $Size_k$ by 1 do
           // $IM_i$ matched with a pattern
5          if DP-LCS($IM_i$, $LCS_j$) exists then
6              update_knowledge_set(min_length(DP-
                LCS($IM_i$, $LCS_j$), $LCS_j$))
               matched$IM$s[j].append($IM_i$);
7          else
               // Not matched with patterns
8              add_knowledge_set($IM_i$.M);
9              matched$IM$s.add([$IM_i$]);
10         end
11     end
12 end
13 return (suspKnowledge, matched$IM$s);
```

Longest Common Subsequence (DP-LCS) algorithm. Algorithm 1 depicts the proposed DP-LCS-based mining algorithm that isolates failure-inducing interactions and categorizes them. We assume the existence of multiple faults in the SoS and multiple faulty interaction patterns within a single execution of each IM. We construct the algorithm based on the following branches- (1) adding a given IM to the existing category, (2) creating a new category with the IM. Lines 5 to 7 illustrate the IM-addition case. The algorithm decides to add an IM to an existing category when an LCS exists between the IM and existing LCS pattern. Line 5 depicts the decision-making process. If a common interaction subsequence exists, the given IM is assigned to the category and the interaction pattern of the updated category is newly extracted involving the given IM. Each IM can be assigned to multiple categories if it possesses multiple subsequence patterns satisfying the aforementioned condition. Lines 8 to 9 illustrate the process of a category creation in case no LCS exists between the IM and any of the existing patterns in the knowledge base. In this case, the IM is added to a new category and its message sequence M is firstly assigned as the new pattern for that category.

The LCS generation algorithm is explained in complete detail in Algorithm 2. We use a Dynamic Programming (DP) solution to generate the LCS because it demonstrates a feasible performance of $O(Size_p * Size_k)$. The LCS generation algorithm begins by generating an LCS table by comparing two IMs, as illustrated in lines 6 to 17. Lines 18 to 24 illustrate the process of extraction of a sequence of commonly observed messages from the table. The basic flow of the algorithm is similar to the existing DP-LCS algorithm for strings. The existing algorithm compares all characters in the strings to generate the LCSs. We expand the string-based comparison

Algorithm 2: LCS extraction algorithm for IM

```
Input  : $IM_p$, $IM_k$
Output: An extracted $LCS$
1  $Size_p \leftarrow$ Size($IM_p.M$);
2  $Size_k \leftarrow$ Size($IM_k.M$);
3  LCSTable $\leftarrow$ [][];
4  current $\leftarrow 0$;
5  LCS_ret $\leftarrow \emptyset$;
   // Generate LCSTable
6  for $i \leftarrow 0$ to $Size_p$ by $1$ do
7      for $j \leftarrow 0$ to $Size_k$ by $1$ do
8          if $i == 0 \mid\mid j == 0$ then
9              LCSTable[i][j] = 0;
10         end
11         if compare_message($IM_p.M[i]$, $IM_k.M[j]$) == $True$ then
12             LCSTable[i][j] = LCSTable[i-1][j-1] + 1;
13         else
14             LCSTable[i][j] = MAX(LCSTable[i][j-1],
                 LCSTable[i-1][j]);
15         end
16     end
17 end
   // Extract LCS from the LCSTable
18 for $i \leftarrow 1$ to $Size_p$ by $1$ do
19     for $j \leftarrow 1$ to $Size_k$ by $1$ do
20         if LCSTable[i][j] > current then
21             current++;
22             add_LCS_ret($IM_i.M$[i]);
23         end
24     end
25 end
26 return LCS_ret;
27
28 Function compare_message($msg_i$, $msg_j$):
29     ret $\leftarrow false$;
30     if continuity, synchronousness, sender,
         receiver, content are identical then
31         ret = $true$
32     end
33     return ret;
```

metrics to message comparison metrics in the IM. As defined in lines 28 to 33, two messages are considered to be identical if the *Continuity*, *Synchronousness*, *Sender and Receiver*, and *Content* features of the two messages share identical values. We do not employ time-related features for the comparison, because the same message may be delivered at different times during a simulation. The same issue exists in the case of *Sender/Receiver* comparison. To address this, we compare *Sender/Receiver* by abstract CS classes, such as $Follower$ and $Leader$ in the platooning SoS, not by concrete CS instances, like vehicle ID. The abstract classes can be defined by the types of roles in an SoS. The implementation of the proposed algorithm has been included in our repository [22].

In other words, the proposed algorithm returns a set of LCS patterns and categorized IMs that contain the patterns common to each category. The suspicious patterns involving the contexts of failures aid SoS engineers to analyze the failures of the categories, and understand their contexts of occurrence and root causes.

D. In-depth Analysis based on an Illustrative Example

Fig. 3 depicts an illustrative example of the process, the interaction model structure, and the outputs of the algorithm. Given the logs of an SoS, the proposed technique first automatically produces interaction models (IM_i) corresponding to each log (L_i), which contains system traces pertaining to the i-th execution. In the example, IM_0 contains an ordered sequence of four messages (M_0), a set of four participating CSs (CS_0), and a $Passed$ tag (tag_0). Each message in M_0 is a vector consisting of the specific features that are explained in Table I. The example illustrates that msg_0 comprises the values: *TC*, *Sync*, sender *A* and receiver *B*, `SPLIT_REQ` content, and time values of *2019/12/24:000000000*, *2019/12/24:000159000*, and *159000*. This indicates that msg_0 is transmitted to convey a `SPLIT_REQ` from *A* to *B* as a *temporal* and *synchronous* message with a delay of *159000* ms.

By using the generated interaction models, the technique runs a DP-LCS-based suspicious interaction pattern mining algorithm. The output of this algorithm is a knowledge table comprising all suspicious LCS patterns, categorized IMs, and further information for subsequent analysis, such as detection counts. In the example, two LCS patterns are listed. The first one, LCS_0, consists of three messages, and this pattern is observed in 35 IMs — IM_1 , IM_9 , $\cdots$, IM_{n-1}. The second suspicious pattern, LCS_1, is a sequence of four messages, and it is detected in 13 different IMs.

The last phase of the technique involves the analysis of the results by SoS engineers. Each row of the output table in Fig. 3 includes categorized IMs and corresponding LCS patterns, which might cause interaction failures in the SoS. In other words, each row denotes an independent category, and IMs in the category contain a discriminative interaction pattern, which causes a specific interaction failure. SoS engineers can utilize the output to analyze the detected interaction failures in great detail.

IV. Case Study

A. Study Design

The goal of our case study is to demonstrate the effectiveness of the proposed technique using a platooning simulation and verification framework, StarPlateS [12], and to analyze unknown faulty interactions in the platooning SoS. The following research questions are used to evaluate the technique:

1) **RQ1.** Does the proposed technique effectively extract suspicious patterns including a sufficient understanding of the contexts and interactions?
2) **RQ2.** Can SoS engineers utilize mining results to analyze faulty scenarios of a platooning SoS?

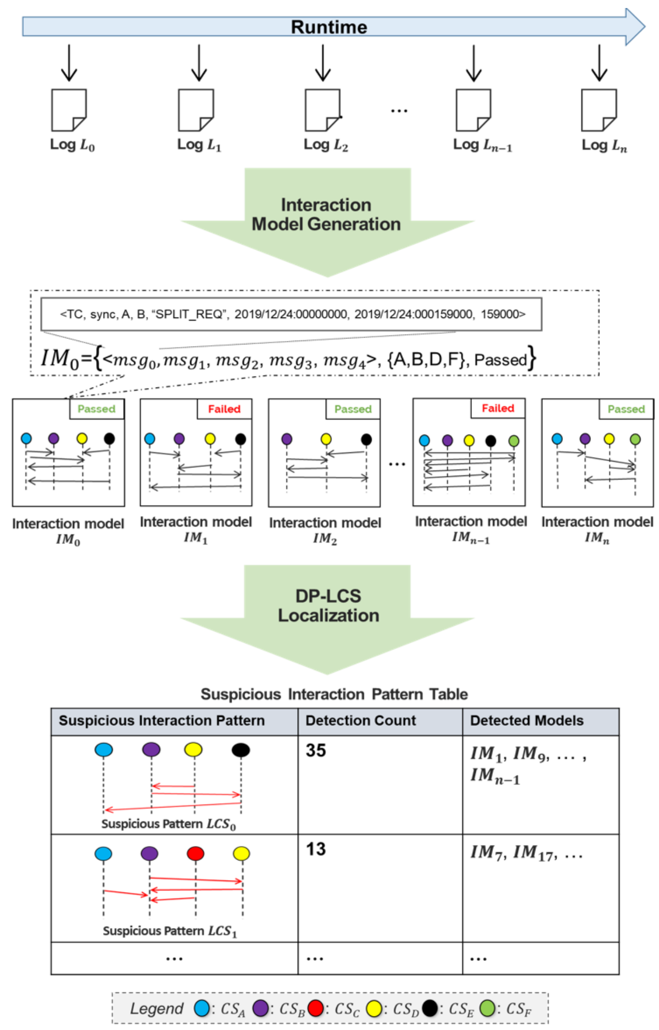

Runtime
Log L_0
Log L_1
Log L_2
Log L_{n-1}
Log L_n
Interaction Model Generation
<TC, sync, A, B, "SPLIT_REQ", 2019/12/24:00000000, 2019/12/24:000159000, 159000>
IM_0={<msg_0, msg_1, msg_2, msg_3, msg_4>, {A,B,D,F}, Passed}
Passed
Failed
Interaction model IM_0
Interaction model IM_1
Interaction model IM_2
Interaction model IM_{n-1}
Interaction model IM_n
DP-LCS Localization
Suspicious Interaction Pattern Table
Suspicious Interaction Pattern
Detection Count
Detected Models
35
IM_1, IM_9, … , IM_{n-1}
Suspicious Pattern LCS_0
13
IM_7, IM_{17}, …
Suspicious Pattern LCS_1
Legend ●: CS_A ●: CS_B ●: CS_C ●: CS_D ●: CS_E ●: CS_F

Fig. 3. Illustrative example of the interaction model-based fault localization

RQ1 is aimed at the high-level analysis of the DP-LCS-based pattern mining results. To answer this question, we analyzed the detailed contexts and execution flows of the extracted interaction patterns. **RQ2** is aimed to determine the practicality of the technique by subdividing the categories into detailed scenario cases. In practice, we performed a manual investigation of each IM based on the high-level analysis results and identified seven interaction failure scenarios in the platooning SoS.

In this case study, we used two StarPlateS modules — the scenario generation module and the simulation module. Using these modules, we generated random scenarios of the platooning SoS and produced corresponding execution logs. The proposed technique also requires a goal property checking result for each log. We modified a verification module present in SIMVA-SoS [17] to construct an independent goal verification module. We selected the following criterion from StarPlateS, which is the most relevant one in the evaluation of the correct execution of operations in the platooning system:

- $P =? \; [(op\_success\_rate > x) \cup sim\_Terminate]$

This criterion determines the number of requested operations that are successfully executed in a simulation. For example, if 10 operations are requested in or between platoons, and 9 out of 10 requested operations are successfully executed, the $op\_success\_rate$ of the simulation is taken to be 90%.

The detailed settings of platooning scenarios for the case study are described as follows:

- Total number of generated scenarios: 600 scenarios
- Duration of a Single simulation: 100 seconds
- Number of generated platoons: 2∼4 platoons
- Size of each platoon: 2∼6 vehicles
- Environmental objects: 1 Human-Driven Vehicle (HDV) generated every 5 seconds

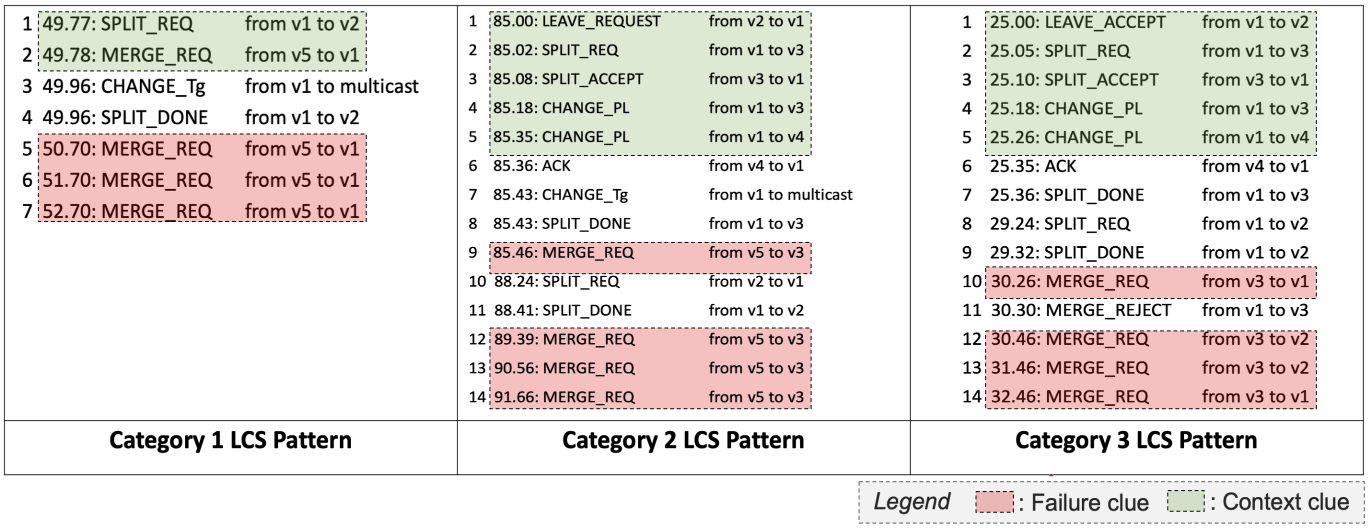
In total, we randomly generated 600 scenarios with a duration of 100 logical seconds for a single simulation. In every simulation, the number of generated platoons was randomly decided, from two to four, with randomly assigned sizes between two to six vehicles. Moreover, we defined a generator for HDVs, which were not connected with platoons and randomly changed lanes and speed. The generator added an HDV into simulation every five seconds. Then, we generated 600 IMs based on the logs and extract suspicious interaction patterns via the proposed algorithm. In the next section, we elucidate the mining and analysis results in detail.

B. Evaluation

In order to adequately answer **RQ1**, we analyzed the qualitative meanings of mined patterns extracted from the 600 simulated scenarios in terms of the $op\_success\_rate$ criterion. We set the success rate threshold as 80% and extracted LCS patterns from 258 failed scenarios. The mining results demonstrate that there exist three major patterns in the $op\_success\_rate$-related failures. We examined the patterns in detail to determine (1) root causes of the failure categories and (2) the contexts of the failed executions. Fig. 4 depicts the sequence patterns of faulty interactions corresponding to each category. Green areas represent the contexts of the executions, which aid the understanding of the failure scenarios, and red areas provide clues for root causes behind failures.

All three LCS patterns are observed to contain the `Merge` operation. Each pattern contains at least three `Merge` requests from the same vehicles, as illustrated in the "Failure clue" of each category in Fig. 4. Because $op\_success\_rate$ represents the proportion of executed operations among all requested operations, continuously requested operations adversely affect the reaction time and the proper execution of other operations in the receiver. The `Split` operation is also common to the LCS patterns. In category 1, the first reported message is a `Split` request from v1 to v2. In categories 2 and 3, the second messages are observed to request `Split` operation. Based on these two common features, we concluded that the operation protocol suffered from certain problems in the target platooning SoS and that these problems were highly correlated with the `Split` and `Merge` operations.

The suspicious interaction pattern in category 1 consists of seven sequential messages, as illustrated in Fig. 4. The first two messages record simultaneous requests of `Split` and `Merge`. Line 1 indicates that the leader vehicle, v1, transmitted `Split`



```
1  49.77: SPLIT_REQ      from v1 to v2
2  49.78: MERGE_REQ      from v5 to v1
3  49.96: CHANGE_Tg      from v1 to multicast
4  49.96: SPLIT_DONE     from v1 to v2
5  50.70: MERGE_REQ      from v5 to v1
6  51.70: MERGE_REQ      from v5 to v1
7  52.70: MERGE_REQ      from v5 to v1
```

Category 1 LCS Pattern

```
1  85.00: LEAVE_REQUEST  from v2 to v1
2  85.02: SPLIT_REQ      from v1 to v3
3  85.08: SPLIT_ACCEPT   from v3 to v1
4  85.18: CHANGE_PL      from v1 to v3
5  85.35: CHANGE_PL      from v1 to v4
6  85.36: ACK            from v4 to v1
7  85.43: CHANGE_Tg      from v1 to multicast
8  85.43: SPLIT_DONE     from v1 to v3
9  85.46: MERGE_REQ      from v5 to v3
10 88.24: SPLIT_REQ      from v2 to v1
11 88.41: SPLIT_DONE     from v1 to v2
12 89.39: MERGE_REQ      from v5 to v3
13 90.56: MERGE_REQ      from v5 to v3
14 91.66: MERGE_REQ      from v5 to v3
```

Category 2 LCS Pattern

```
1  25.00: LEAVE_ACCEPT   from v1 to v2
2  25.05: SPLIT_REQ      from v1 to v3
3  25.10: SPLIT_ACCEPT   from v3 to v1
4  25.18: CHANGE_PL      from v1 to v3
5  25.26: CHANGE_PL      from v1 to v4
6  25.35: ACK            from v4 to v1
7  25.36: SPLIT_DONE     from v1 to v3
8  29.24: SPLIT_REQ      from v1 to v2
9  29.32: SPLIT_DONE     from v1 to v2
10 30.26: MERGE_REQ      from v3 to v1
11 30.30: MERGE_REJECT   from v1 to v3
12 30.46: MERGE_REQ      from v3 to v2
13 31.46: MERGE_REQ      from v3 to v2
14 32.46: MERGE_REQ      from v3 to v1
```

Category 3 LCS Pattern

Legend 🟥: Failure clue 🟩: Context clue

Fig. 4. Extracted representative LCS patterns of interaction sequences.

request to its follower, v2. Line 2 indicates that the leader of the rear platoon, v5, requests `Merge` to the front leader, v1. Fig. 5 illustrates the failure pattern pertaining to category 1. The problem in this scenario is that because of the request for the `Split` operation, the front platoon leader in front of v5 was changed from v1 to v2. However, even though the rear front leader, v5, continuously requests `Merge` to v1, they could not be executed. Thus, we concluded that one of the observed failures is related to the operation request logic in the protocol and this failure is triggered by the simultaneous requests of `Split` and `Merge`.

In the category 2 pattern, two platoons are observed — a front platoon with leader, v1, and a rear platoon with leader, v5, — in the same lane. In this case, v5 transmitted continuous `Merge` requests. However, in category 3, vehicles v1 to v4 comprised a single platoon. One of its followers, v3, transmitted the requests to v1 and v2.

Fig. 6 depicts the example execution flow corresponding to category 2 pattern failure. In the pattern, one of the followers in the front platoon, v2, requests a `Leave` operation as illustrated in lines 1 to 3 in Fig. 4. The `FollowerLeave` operation comprises three steps in the target platooning operation protocol [4] — dividing, leaving, and merging. First, since v2 requested `Leave`, v3 and v4 were divided into a new platoon, in lines 4 to 8. Then, the rear platoon suddenly transmitted a `Merge` request to the divided platoon. However, the divided platoon needed to remerge with the original leader, v1, after v2 leaves. The problem here is that the rear platoon leader, v5, kept transmitting messages containing `MERGE_REQ` to the intermediate leader, v3. We identified the reason behind the continuous `Merge` requests by v5 to be the `Optimal size` configuration. If the rear platoon size is smaller than the `Optimal size` and the sum of the rear platoon size and the intermediate platoon size is coincidentally equal to the `Optimal size`, the rear platoon is observed to continuously transmit `Merge` requests to the intermediate platoon.

The final LCS pattern is observed to begin with a `Leave` operation. Line 1 to 5 in category 3 in Fig. 4 illustrates that v2 requested `Leave` and that the rear vehicles, v3 and v4, were divided into an intermediate platoon to execute the `Leave` operation. A discriminative feature of this pattern was that the leader of the intermediate platoon, v3, transmitted a `Merge` request to the leaving vehicle, v2. Originally, it was expected to send requests for v1 to remerge with the original platoon. However, the v3 sends messages to both the leader, v1 and the leaving vehicle, v2. To explain this case, we reproduced similar cases in a simulation with different `Optimal size` settings. We found that the redundant `Merge` requests were transmitted to leaving vehicles irrespective of `Optimal size` or `Leader/Follower Leave` settings. This failure case is frequently observed when the `FollowerLeave` operation is executed on a platoon consisting of more than three vehicles.

TABLE II
EXAMINED FAILURE-INDUCING INTERACTION SCENARIOS IN VENTOS

Category	Related Operations	Detailed Failure Scenarios
Simultaneous Operation Request	*Split* and *Merge*	*Split* and *Merge*
		OptSizeChange and *Merge*
		LeaderLeave and *Merge*
Configuration Conflict	*Leave* and *Merge* with *Optsize*	*LeaderLeave*
		FollowerLeave
Single Operation Failure	*Leave* and *Merge*	*LeaderLeave*
		FollowerLeave

In order to answer the **RQ2**, we further analyzed each IM within each category and subdivided the seven failure scenarios from the existing categories. Detailed failure reports of the seven scenarios were developed including the preconditions, execution flows, illustrative examples, and extracted patterns. This analysis demonstrates that SoS engineers with limited knowledge can satisfactorily utilize the data provided by the proposed approach. Table II presents the details of the analysis. For instance, in Table II, three different cases — simultaneous `Split` & `Merge`, `OptSizeChange` & `Merge`, and `LeaderLeave` & `Merge` — may cause the category 1 failure. We observed these concrete scenario cases from IMs belonging to category 1. Similarly, we identified two types of interaction failure scenarios — `LeaderLeave` and `FollowerLeave` — in each of categories, 2 and 3. We recorded seven failure scenarios in detail and compiled them into failure reports for the platooning SoS considered in this study. To the best of our knowledge, the aforementioned failure scenarios have not been previously reported for VENTOS [4] or any other related platooning simulator repository [14], [15], [23], [24]. We expect the reported failure scenarios to be used as testing benchmarks and to enrich fault knowledge databases for general platooning systems. The detailed failure reports have been included in our repository[1].

C. Threats to Validity

Interaction Features. Even though we defined interaction features based on a thorough investigation of various system domains [18], [19], [20], [21], it is difficult to conclude that the features considered in this study are sufficient to analyze all possible types of failures in an SoS. Certain exceptional cases of failure are induced by external root causes. For instance, multiple collision-related failures are observed on the part of Human-Driven Vehicles (HDVs) in the simulation of the platooning SoS. However, since the platooning simulator used in the case study, VENTOS only logs internal micro-commands of platooning operations and does not log sensor-level information, it cannot provide enough information to identify external causes behind collisions. In future works, we intend to generalize the interaction model and the simulator to account for failures due to external factors as well.

Limited Evaluation Resource. To the best of our knowledge, it is the first study to report interaction failures in large-scale systems, especially platooning SoS. As a result, appropriate testing benchmarks could not be obtained for the platooning SoS being considered, to provide execution logs, oracles, and data on failure scenarios and corresponding causes. Thus, in our case study, we manually investigated the outputs and the generated fault knowledge about the interaction failures. We expect the fault knowledge obtained in this study to be used as benchmarks or a general fault knowledge for platooning SoSs in future studies.

We also used the *op_success_rate* property with 80% of the empirical value. We examined studies that simulated and verified platooning systems, but most of them used basic testing criteria for platoons, such as maintenance of platoons until the end of simulation [23], [24], or verification of a single operation execution [14], [15]. Instead, we attempted to generate cogent properties for the platooning SoS based on international standards, such as ISO26262 [25]. However, certain recent studies have reported that the existing standards, such as ISO26262, focus on autonomous driving, and thus they cannot fully meet the requirements of platooning SoS [13], [26], [27]. In this study, the *op_success_rate* was benchmarked based on the Percentage of Successful Request (PSR) used in the testing of the cloud system [28] and modified for application to the simulation logs of the platooning system.

[1]https://github.com/abalon1210/StarPlateS

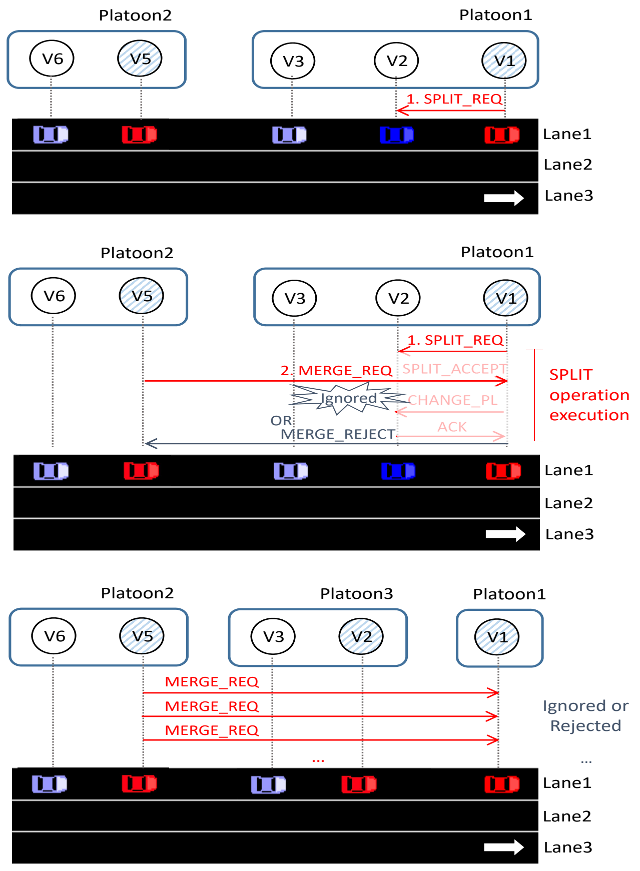


Fig. 5. Example execution flow of category 1 pattern

V. Related Works

Recent studies on the analysis of root causes underlying system failures have proposed several abstraction models and techniques for various SoS domains, such as power plant systems, flight control systems, mass casualty incident response systems, and underwater vessels [8], [9], [10], [11], [29]. These studies can be classified into two major categories — those that diagnose fault patterns based on a fault knowledge base and those that deduce suspicious components of a system using testing/verification results.

Fault Diagnosis based on Fault Knowledge Base. The majority of diagnosis techniques have applied machine learning techniques to identify fault patterns from the system state. Kleyko *et al.* [10] suggested a matrix-based system model to represent features of the power plant system. In order to improve the accuracy of real-time faulty pattern matching, the technique converts feature matrices of time, t, into hyper-dimensional vectors and uses the hamming distance to calculate the similarity between input system state vectors and fault vectors within the knowledge base. Cai *et al.* [11] proposed an Object-Oriented Bayesian Network (OOBN) to handle uncertainty in the fault diagnosis process. Using OOBN model, which is widely regarded as effective to solve uncertainty problems, the authors developed fault diagnosis models for a subsea production system. Yang *et al.* [29] proposed an integrated fault diagnosis technique that combines a lightweight

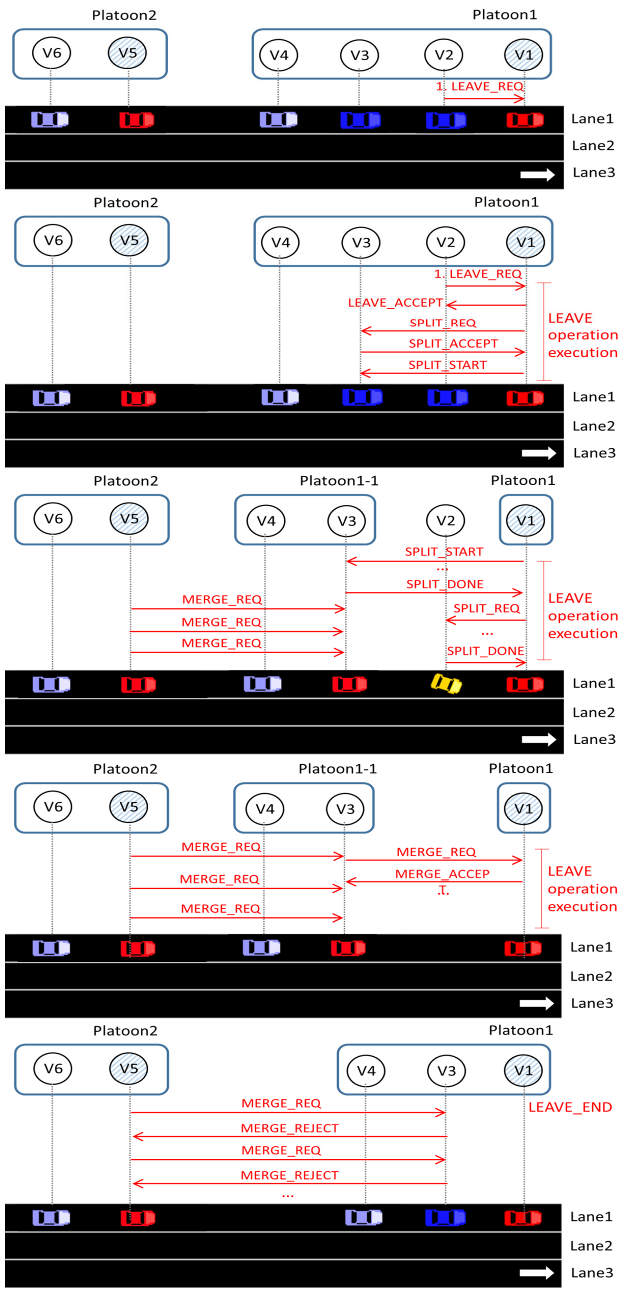


Fig. 6. Example execution flow of category 2 pattern

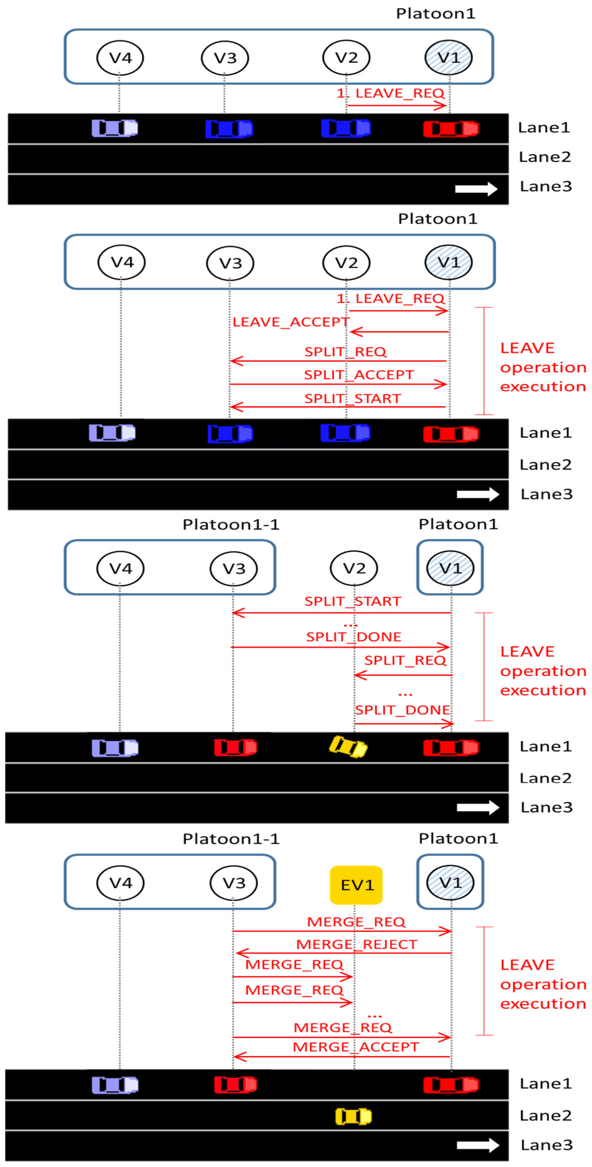


Fig. 7. Example execution flow of category 3 pattern

diagnosis with a Bayesian network-based approach to support low-level and deep-level analysis on a flight control system. They developed a framework to investigate the root causes of failures by determining the most relevant description in each case from the fault knowledge database. Most diagnosis techniques classify failures by referring to an existing fault knowledge database, and therefore, only focus on the accuracy and efficiency of the diagnosis. The framework developed by Yang *et al*. can cover the unknown fault situation, but it also assumes a pre-built fault knowledge base.

SBFL Technique on Large-Scale Systems. Techniques that focus on the determination of suspicious components of a system generally apply the Spectrum-Based Fault Localization (SBFL) technique. Shin et al. [8] applied the SBFL technique to disaster response SoS. They modeled the SoS as a collaboration graph that represents the participation and connection of CSs. By injecting faulty components and connections into the SoS, they localized suspicious components and connections based on the execution logs. Arrieta et al. [9] also applied the SBFL technique on the software product line. They used software feature models, employing each feature as a single spectrum in the SBFL. However, existing studies using the SBFL technique suffer from such limitations — (1) inability to utilize sequential interaction data from the logs, (2) infinite combinations of interaction sequences.

The approach proposed in this study is capable of dealing with the aforementioned issues. Firstly, our study provides a systematic approach to analyze failure-inducing interactions by analyzing interaction data obtained from logs and, subsequently, mining suspicious patterns. In addition, it does not require high-level domain knowledge and can deal with unknown fault scenarios.

VI. Conclusion

In this paper, we proposed a pattern mining technique to effectively analyze interaction failures in an SoS. The technique addresses two issues present in existing studies — inadequate use of interaction data without proper algorithm and unknown fault issues. To this end, we first defined an interaction model for an SoS using the message features used in diverse systems. We also proposed an interaction pattern mining algorithm to isolate and categorize suspicious interaction sequences. By applying the technique on the platooning SoS and investigating the output results manually, we identified three separate categories of operation failures and seven detailed interaction failure scenarios affecting the success rate of the platooning operation without referring to any existing knowledge. We expect the outputs of this study to be used as benchmarks or a fault knowledge base of a general platooning SoS in future studies. To the best of our knowledge, this study presents the first attempt to determine precise interaction failure scenarios based on concrete preconditions, execution flows.

We expect the scope of the proposed approach to be extended in three directions. First, we plan to generalize the interaction model to cover external causes in addition to internal fundamental causes. We will modify the target simulator to enable sensor-level logging and update the interaction model to utilize the relevant information for analyzing external root causes. Secondly, we intend to improve the pattern mining algorithm to categorize IMs and extract patterns more effectively by applying certain heuristic metrics. Finally, we intend to generate a testing benchmark of the platooning SoS by defining further goal properties with oracles.

Acknowledgment

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System) and Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (2017M3C4A7066212).

References

[1] A. Davila, E. del Pozo, E. Aramburu, and A. Freixas, “Environmental benefits of vehicle platooning,” tech. rep., SAE Technical Paper, 2013.

[2] H. L. Humphreys, J. Batterson, D. Bevly, and R. Schubert, “An evaluation of the fuel economy benefits of a driver assistive truck platooning prototype using simulation,” tech. rep., SAE Technical Paper, 2016.

[3] C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, “Systems of systems engineering: basic concepts, model-based techniques, and research directions,” ACM Computing Surveys (CSUR), vol. 48, no. 2, p. 18, 2015.

[4] M. Amoozadeh, H. Deng, C.-N. Chuah, H. M. Zhang, and D. Ghosal, “Platoon management with cooperative adaptive cruise control enabled by vanet,” Vehicular communications, vol. 2, no. 2, pp. 110–123, 2015.

[5] M. Splittgerber, “Daimler now testing platooning technology for more truck efficiency also in japan.” [Online; accessed 21-Jan-2020].

[6] F. O. Staff, “Volvo trucks tests on-highway three-truck platooning.” [Online; accessed 21-Jan-2020].

[7] J. hyuk Lee and M. Kim, “Hyundai motor demonstrates first successful truck platooning on highway.” [Online; accessed 21-Jan-2020].

[8] Y.-J. Shin, S. Hyun, Y.-M. Baek, and D.-H. Bae, “Spectrum-based fault localization on a collaboration graph of a system-of-systems,” in 2019 14th Annual Conference System of Systems Engineering (SoSE), pp. 358–363, IEEE, 2019.

[9] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, “Spectrum-based fault localization in software product lines,” Information and Software Technology, vol. 100, pp. 18–31, 2018.

[10] D. Kleyko, E. Osipov, N. Papakonstantinou, and V. Vyatkin, “Hyperdimensional computing in industrial systems: the use-case of distributed fault isolation in a power plant,” IEEE Access, vol. 6, pp. 30766–30777, 2018.

[11] B. Cai, H. Liu, and M. Xie, “A real-time fault diagnosis methodology of complex systems using object-oriented bayesian networks,” Mechanical Systems and Signal Processing, vol. 80, pp. 31–44, 2016.

[12] S. Hyun, J. Song, S. Shin, and D.-H. Bae, “Statistical verification framework for platooning system of systems with uncertainty,” in 2019 26th Asia-Pacific Software Engineering Conference (APSEC), pp. 212–219, IEEE, 2019.

[13] M. Elgharbawy, “A big testing framework for automated truck driving,” Urban transportation and construction, vol. 4, no. 1, pp. e27–e27, 2019.

[14] M. Kamali, S. Linker, and M. Fisher, “Modular verification of vehicle platooning with respect to decisions, space and time,” in International Workshop on Formal Techniques for Safety-Critical Systems, pp. 18–36, Springer, 2018.

[15] P. Mallozzi, M. Sciancalepore, and P. Pelliccione, “Formal verification of the on-the-fly vehicle platooning protocol,” in International Workshop on Software Engineering for Resilient Systems, pp. 62–75, Springer, 2016.

[16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” IEEE Transactions on Software Engineering, vol. 42, no. 8, pp. 707–740, 2016.

[17] S. Park, Y. Shin, S. Hyun, and D. Bae, “SIMVA-SoS: Simulation-based verification and analysis for system-of-systems,” in 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE), pp. 575–580, 2020.

[18] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, “Testing autonomous cars for feature interaction failures using many-objective search,” in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 143–154, ACM, 2018.

[19] D. O. Keck and P. J. Kuehn, “The feature and service interaction problem in telecommunications systems: A survey,” IEEE Transactions on Software Engineering, vol. 24, no. 10, pp. 779–796, 1998.

[20] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen, “A feature-interaction benchmark for in and beyond,” IEEE Communications Magazine, vol. 31, no. 3, pp. 64–69, 1993.

[21] M. Weiss, B. Esfandiari, and Y. Luo, “Towards a classification of web service feature interactions,” Computer networks, vol. 51, no. 2, pp. 359–381, 2007.

[22] S. Hyun, “Starplates github repository.” [Online; accessed 13-Jul-2020].

[23] B. Vieira, R. Severino, A. Koubâa, and E. Tovar, “Towards a realistic simulation framework for vehicular Platooning Applications,” arXiv preprint arXiv:1904.02994, 2019.

[24] K. Meinke, “Learning-based testing of cyber-physical systems-of-Systems: a platooning study,” in European Workshop on Performance Engineering, pp. 135–151, Springer, 2017.

[25] I. O. of Standardization, “Iso 26262: Road vehicles – functional safety.” [Online; accessed 23-Jan-2020].

[26] P. E. Memebers, “Platooning ensemble.” [Online; accessed 23-Jan-2020].

[27] S. Achrifi, “Coverage verification framework for ADAS models,” March 2017.

[28] C. Sauvanaud, M. Kaâniche, K. Kanoun, K. Lazri, and G. D. S. Silvestre, “Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned,” Journal of Systems and Software, vol. 139, pp. 84–106, 2018.

[29] S. Yang, C. Bian, X. Li, L. Tan, and D. Tang, “Optimized fault diagnosis based on fmea-style cbr and bn for embedded software system,” The International Journal of Advanced Manufacturing Technology, vol. 94, no. 9-12, pp. 3441–3453, 2018.