The Journal of Systems & Software 198 (2023) 111613

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Timed pattern-based analysis of collaboration failures in system-of-systems☆

Sangwon Hyun, Jiyoung Song, Eunkyoung Jee *, Doo-Hwan Bae

School of Computing, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea

ARTICLE INFO

Article history:
Received 12 August 2022
Received in revised form 26 October 2022
Accepted 4 January 2023
Available online 7 January 2023

Keywords:
Failure-inducing interaction
Pattern-based clustering
Fault localization
System-of-systems

ABSTRACT

A system-of-systems (SoS) tries to achieve prominent goals, such as increasing road capacity in platooning that groups driving vehicles in proximity, through interactions between constituent systems (CSs). However, during the collaboration of CSs, unintended interference in interactions causes collaboration failures that may lead to catastrophic damage, particularly for the safety–critical SoS. It is necessary to analyze the failure-inducing interactions (FIIs) during the collaboration and resolve the root causes of failures. Existing studies have utilized pattern-mining techniques to analyze system failures from logs. However, they have three limitations when applied to collaboration failures: (1) information loss caused by the limited capabilities of handling interaction logs; (2) limitations in identifying multiple failure patterns in a log; (3) absence of an end-to-end solution mapping patterns to faults. To overcome these limitations, we propose an FII pattern mining algorithm covering the main features of SoS interaction logs, an overlapping clustering technique for multiple pattern mining, and a pattern-based fault localization method. In experiments conducted on platooning and mass casualty incident-response SoS, the proposed approach exhibited the highest pattern mining and clustering accuracy and achieved feasible localization performance compared with existing methods. The findings of this study can facilitate the accurate analysis of collaboration failures.

1. Introduction

Software systems have become more complex and massive, concomitant with the increasing number and sophisticated requirements of society. This trend is prevalent in the social and business aspects of smart factories (Shi et al., 2020), smart cities (Trencher, 2019), and intelligent platooning transportation systems (Zambrano et al., 2020). A platooning system, in which vehicles are driven in groups in close proximity, increases fuel efficiency and road capacity beyond that achievable by a single vehicle (Amoozadeh et al., 2015). The mass casualty incident-response (MCI-R) system (Organization et al., 2007) increases the rescue and treatment rate of patients in the catastrophic situation of smart cities through the collaboration of various constituent systems (CSs) such as ambulances, firefighters, and drones. These systems that try to achieve high-level goals through the interactions between CSs can be considered system-of-systems (SoS), which contain distinguishing features of independence and autonomy. For example, in platooning, vehicles from different vendors have operational and managerial independence as CSs, and vehicles autonomously `Join` or `Leave` platoons to participate in SoS-level goal achievement (Hyun et al., 2019).

SoS collaboration protocols define the concrete interaction processes of CSs to achieve SoS-level goals (Petitdemange et al., 2018; Lü et al., 2017). In this study, interaction denotes communication to exchange information in SoS and collaboration denotes a CS–CS interaction. In fact, VENTOS (Amoozadeh et al., 2015) and ENSEMBLE (Group, 2022) provide platooning collaboration protocols defining interaction processes of vehicles (e.g., `Leave`). SIMVA-SoS (Park et al., 2020) defines the collaboration processes of five different CSs (e.g., SoS manager, firefighters) for MCI-R SoS.

However, the design and development of the SoS collaboration protocol are conducted under limited knowledge owing to the independence of CSs. The limited knowledge denotes that CSs have black-boxed and preexisting modules for their functionalities (Kazman et al., 2013). Thus, SoS developers do not have access to investigate and modify the detailed implementation of the internal rules and structures of CSs. For example, in platooning, the collaboration protocol is developed in a situation where the internal details of distance or speed management of vehicles from different vendors are black-boxed. Consequently, the SoS collaboration protocol inherently contains uncertainty about the mutual effect between the protocol and existing modules of CSs. The uncertainty in the collaboration protocol may cause unexpected failures during execution.

☆ Editor: Prof W. Eric Wong.

* Corresponding author.
E-mail addresses: swhyun@se.kaist.ac.kr (S. Hyun), jysong@se.kaist.ac.kr (J. Song), ekjee@se.kaist.ac.kr (E. Jee), bae@se.kaist.ac.kr (D.-H. Bae).

https://doi.org/10.1016/j.jss.2023.111613
0164-1212/

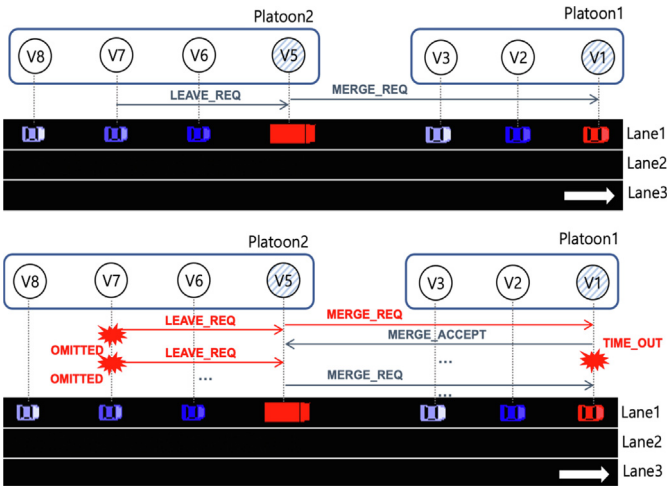


Fig. 1. An example failure detected in platooning SoS simulation.

Collaboration failures in SoS are defined as the failures to achieve SoS-level goals caused by unintended inter-functional interference during the intricate interactions (Augustine et al., 2012). Fig. 1 describes the deadlock-like failure of the platooning operations found in this study's simulation using VENTOS simulator. The scenario has two platoons: Platoon 1 with a size of three and Platoon 2 with a size of four, where *V1* and *V5* are marked as the leaders of each platoon. The failure is caused by the simultaneous requests of `Merge` from *V5* and `Leave` from *V7*. In this situation, *V5* continually ignores the `Leave` from *V7* because *V5* is in the `Merge` operation. Moreover, because of the communication between *V5* and *V7*, the wait time for the `Merge` is exceeded; thus, the `Merge` is also not properly executed in *V5*. Consequently, both operations fail and are repeatedly requested. These failures caused by such complex interactions are a significant challenge to achieving SoS goals and may lead to serious collisions.

SoS developers should resolve root causes (i.e., bugs) before deployment to prevent such failures. To effectively analyze the root causes of failures in such intricate data, existing studies have focused on mining patterns for log anomaly detection (Landauer et al., 2018; Schmidt et al., 2020; Amar and Rigby, 2019; Sauvanaud et al., 2018; Du et al., 2017; Zhang et al., 2019), time-series data analysis (Liu et al., 2019; Soleimany and Abessi, 2019; Choong et al., 2017; Kleyko et al., 2018), and sequence data analysis (Hyun et al., 2020; Millham et al., 2021) for various systems. Our investigation of the applicability of existing methods to SoS collaboration failure analysis indicated that they (1) do not cover the major features of SoS interaction logs (e.g., multidimensional and temporal features); (2) have limitations in terms of identifying multiple failure patterns in a single log; and (3) do not provide an end-to-end solution from pattern analysis to code-level fault localization.

First, existing approaches are limited to dealing with the features of SoS interaction logs, such as multidimensional structure and time sensitivity, so that serious information loss occurs when mining failure patterns. Although SoS interaction logs consist of multidimensional communication messages involving heterogeneous data types (e.g., contents, time), existing studies have mainly focused on the pattern mining of single-dimensional or single-type data (e.g., numbers or text). They commonly use Euclidean or Levenshtein distance for analyzing vehicle trajectory (Choong et al., 2017) and power plant sensor data (Kleyko et al., 2018). The similarity metrics of extant studies can only cover parts of multidimensional data. In addition, the interactions defined in the collaboration protocol generally have time and order sensitivity (Liu et al., 2018; Muhammad and Safdar, 2018). Although temporal features, such as message intervals, are important factors for analyzing interactions, extant studies do not fully consider them. Such information loss adversely affects the accuracy of the extracted patterns. A pattern mining algorithm that covers the major features of interaction logs is needed.

Second, most pattern analysis studies concentrate on extracting a single pattern from a single data element (e.g., a log). However, in SoS, one failure can cause other cascading failures (Nakarmi et al., 2020; Meango and Ouali, 2020). Thus, multiple failure patterns can occur in a single log, and this should be considered in the pattern mining and classification process to extract all the failure patterns involving edge-cases.

Finally, few studies provide an end-to-end solution to map the patterns to the root causes in the collaboration protocol code. Patterns are flagged logs that can effectively explain the occurrence of failures (Amar and Rigby, 2019). Developers need to localize the bugs in the collaboration protocol from the patterns to resolve the collaboration failures. Because the cost required for developers to localize bugs remains expensive, extant studies have argued the necessity of fault localization from patterns (Amar and Rigby, 2019; Jiang et al., 2017). Nonetheless, studies proposing the localization methods from patterns to reduce the analysis cost of failures are limited.

To overcome these limitations, we propose a failure-inducing interaction (FII) pattern-based overlapping clustering and fault localization as an extension of our previous study (Hyun et al., 2020). The proposed approach provides three key contributions to rectify the aforementioned issues in analyzing the root causes of SoS collaboration failures. First, we define a TImed Message pattern Extraction-Longest Common Subsequence (*TIME-LCS*) pattern mining algorithm that accurately extracts FII patterns by covering the multidimensional and temporal features of the interaction logs. Second, we propose a *TIME* overlapping clustering to classify and extract all FII patterns that have occurred during the SoS execution. Finally, we provide a pattern-based fault localization method that calculates the suspiciousness of collaboration protocol codes. Further, to facilitate compatibility with the limited knowledge of SoS, our approach only utilizes communicated message logs and the collaboration protocol code as accessible inputs, without considering any data related to the black-boxed codes, such as internal state changes of vehicles, in CSs.

We conducted experiments on the platooning (Hyun et al., 2021) and MCI-R SoS (Park et al., 2020) dataset. The results obtained verified that our approach (1) generates the most accurate FII patterns from the platooning interaction logs among existing pattern mining techniques; (2) exhibits significantly high overlapping clustering precision; and (3) achieves a higher efficacy of the debugging cost reduction compared with existing spectrum-based fault localization (SBFL) methods.

The remainder of this paper is organized as follows. Sections 2 and 3 explains the background and the proposed approach based on platooning scenarios. Section 4 describes the experiment conducted. Section 5 discusses related work. Section 6 recommends directions for future works and provides concluding remarks.

2. Background

This section provides background information on *LCS* algorithm and interaction model for SoS interaction logs.

2.1. LCS algorithm

LCS algorithm finds the longest subsequence that two strings have in common, which is often used to analyze gene sequence data in bioinformatics (Jones et al., 2004) and sensor data in

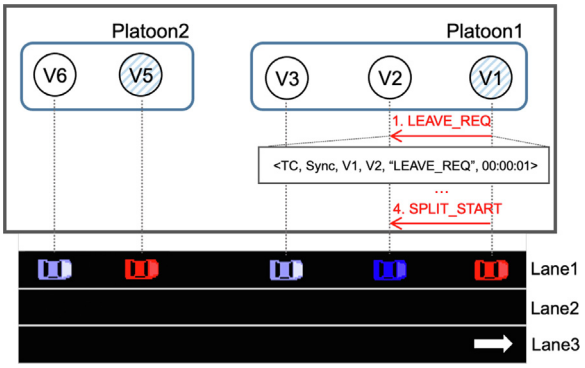


Fig. 2. Example interaction model of Platooning SoS.

system engineering (Choong et al., 2017; Soleimany and Abessi, 2019). Let $m, n \in \mathbb{N}_{\geq 0}$, $x_m, y_n \in Char$, $s_m = x_1, x_2, x_3, \ldots, x_m$ and $s_n = y_1, y_2, y_3, \ldots, y_n$ be *String*s having lengths m and n, respectively. The function $LCS : String \times String \rightarrow String$ maps two input strings to the longest common subsequence involved in both strings. The *LCS* function can be defined as follows:

$$LCS(s_m, s_n) = \begin{cases} \phi, & \text{if } m = 0 \text{ or } n = 0 \\ LCS(s_{m-1}, s_{n-1}) \oplus x_m, & \text{if } x_m = y_n \\ maxLenS(LCS(s_m, s_{n-1}), & \text{otherwise} \\ \quad LCS(s_{m-1}, s_n)) & \end{cases} \quad (1)$$

The function $maxLenS : String \times String \rightarrow String$ selects the longer string between the two input strings. If the length of both strings is zero, the function *LCS* outputs an empty string. The operator $\oplus$ implies the concatenation of the operands. When two input strings have a common character, the function recursively concatenates the character to $LCS(s_{m-1}, s_{n-1})$. We extend the *LCS* function in Section 3 to be applied to SoS message sequences.

2.2. Interaction model of SoS

This study concentrated on the interaction logs of CSs to comprehensively analyze SoS collaboration failures. The SoS interaction model (Hyun et al., 2020) represents interaction logs as sequences of messages between CSs. The definitions of a message and a message sequence are as follows:

$$Msg \ni msg = \langle continuity, synchronization, sender, receiver, content, time \rangle,$$
$$M \ni m_n = msg_1, msg_2, msg_3, \ldots, msg_n,$$

where m_n is a finite sequence of messages of length n. Each msg_i in the sequence is a tuple consisting of *continuity*, *synchronization*, *sender*, *receiver*, *content*, and *time*. For example, Fig. 2 shows a sequence of communication messages between vehicles in a platooning SoS. In the example scenario, $V1$ wants to leave Platoon1; thus, it sends a message to $V2$. The message, msg_1, sent to $V2$ is (*TC*, *Sync*, $V1$, $V2$, `LEAVE_REQ`, 00:00:01), which implies temporary (*TC*) and synchronous (*Sync*) communications from $V1$ to $V2$ with `LEAVE_REQ` command at 00:00:01. The example message sequence from `LEAVE_REQ` to `SPLIT_START` can be written as $m_4 = msg_1, msg_2, msg_3, msg_4$. In this study, we consider the time windows of interaction logs in pattern mining. Hence, m_n is expanded to m_n^t, where t denotes the onset time of a time window.

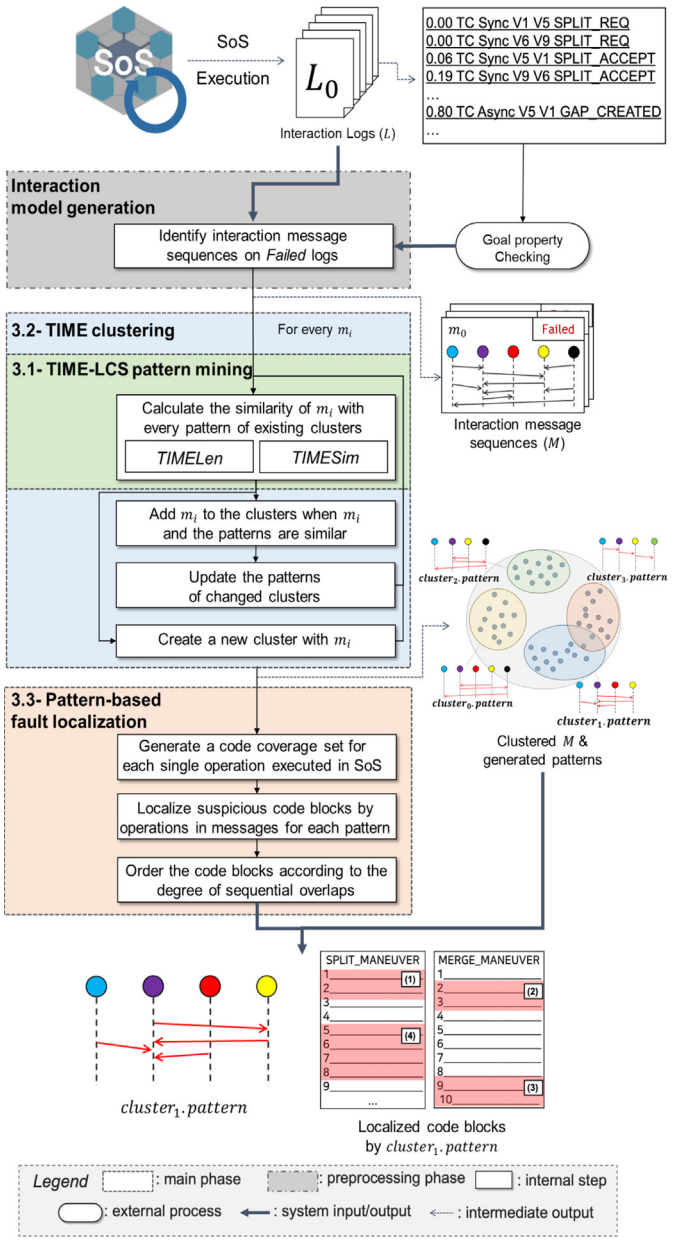


Fig. 3. Overall process of the proposed failure analysis technique.

3. TIME-LCS pattern-based collaboration failure analysis

Fig. 3 depicts the overall process of our approach, which comprises three effective techniques— *TIME-LCS* pattern mining, *TIME* overlapping clustering, and pattern-based fault localization. First, we define the *TIME-LCS* algorithm to extract SoS collaboration failure patterns by covering the multidimensional structure of interaction logs. Further, the algorithm considers the temporal features of interactions to increase the accuracy of pattern mining. The second technique, *TIME* overlapping clustering, classifies and extracts multiple failure patterns. In *TIME* clustering,[1] failure patterns are used as the centroid for each group. Hence, the metrics required for the clustering are defined based on the *TIME-LCS* algorithm. Finally, our approach localizes the root causes of

[1] In this study, *TIME* clustering refers to *TIME* overlapping clustering and is abbreviated for the convenience of reading.

collaboration failures based on the patterns. The implementation of the proposed approach is fully opened in our repository.[2]

The proposed approach uses two main inputs: SoS interaction logs and the *Passed*/*Failed* tag on each log to check whether the logs satisfy certain goal properties. As described in Section 2, we follow the SoS interaction model format. Thus, during the preprocessing phase, our technique abstracts all message-based interactions on each *Failed* log and returns a set of interaction message sequences (M). For example, the first message in the example log seen at the top of Fig. 3 consists of time (0.00), continuity (*TC*), synchronization (*Sync*), sender (*Leader*), receiver (*Follower*), and contents (`SPLIT_REQ`). Herein, we do not use the concrete instance id for sender and receiver, like *V1* or *V5*, but we do abstract the id to the roles of CSs (i.e., *Leader* or *Follower*) in platooning SoS to extract patterns occurring in different vehicles and time zones.

3.1. TIME-LCS pattern mining

With the generated M as input, the primary goal of this approach is to extract accurate collaboration failure patterns. The algorithm is focused on covering the multidimensional and temporal features of interaction logs so as not to cause information loss in the pattern mining.

First, to deal with the multidimensionality, we extend the string-based *LCS* function in Eq. (1) to the longest common message subsequence (LCMS) function based on the definition of message sequences m_n in Section 2.2. We define the function $LCMS : M \times M \rightarrow M$ which maps two input message sequences, p_k and q_n, to the longest common message sequence as follows:

$$LCMS(p_k, q_n) \triangleq \begin{cases} \phi, & \text{if } k = 0 \text{ or } n = 0 \\ LCMS(p_{k-1}, q_{n-1}) \oplus msg_k^p, & \text{if } MCT(msg_k^p, msg_n^q) \\ maxLenM(LCMS(p_k, q_{n-1}), LCMS(p_{k-1}, q_n)) & \text{otherwise} \end{cases} \quad (2)$$

The function $maxLenM : M \times M \rightarrow M$ selects the longest message sequence among the two inputs. In Eq. (1), the comparison of two characters forming the strings is self-explanatory. However, a special function is required to determine the identity of two messages. Hence, we define a message comparison with time function, *MCT*, which enables us to check the identity of two messages. Let us assume $pre_p, pre_q \in \mathbb{N}_{\geq 0}$, which denotes the previously matched message ids in p_k and q_n, respectively. Function $MCT : Msg \times Msg \rightarrow \mathbb{B}$ maps two input messages to the Boolean value of the message identity as follows:

$$MCT(msg_k^p, msg_n^q) \triangleq \begin{cases} (msg_k^p.sender = msg_n^q.sender) & pre_p = 0 \\ \wedge\ (msg_k^p.content = msg_n^q.content) & or \\ \wedge\ (msg_k^p.receiver = msg_n^q.receiver) & pre_q = 0 \\ \wedge\ (msg_k^p.continuity = msg_n^q.continuity) & \\ \wedge\ (msg_k^p.synchronization = msg_n^q.synchronization) & \\ & \\ (msg_k^p.sender = msg_n^q.sender) & \\ \wedge\ (msg_k^p.content = msg_n^q.content) & \\ \wedge\ (msg_k^p.receiver = msg_n^q.receiver) & \\ \wedge\ (msg_k^p.continuity = msg_n^q.continuity) & \text{otherwise} \\ \wedge\ (msg_k^p.synchronization = msg_n^q.synchronization) & \\ \wedge\ ((msg_k^p.time - msg_{pre_p}^p.time) & \\ -(msg_n^q.time - msg_{pre_q}^q.time) \leq delay\_threshold) & \end{cases} \quad (3)$$

[2] https://github.com/KAIST-SE-Lab/StarPlateS

1	53.03: MERGE_REQ	from v6 to v5	1	85.00: LEAVE_REQUEST	from v2 to v1
2	53.53: MERGE_REQ	from v6 to v5	2	85.02: SPLIT_REQ	from v1 to v3
3	54.03: MERGE_REQ	from v6 to v5	3	85.08: SPLIT_ACCEPT	from v3 to v1
4	54.06: MERGE_ACCEPT	from v5 to v6	4	85.18: CHANGE_PL	from v1 to v3
5	54.53: MERGE_REQ	from v6 to v5	5	85.43: CHANGE_Tg	from v1 to multi
6	54.06: MERGE_ACCEPT	from v5 to v6	6	85.43: SPLIT_DONE	from v1 to v3
7	55.26: MERGE_DONE	from v6 to v5	7	85.46: MERGE_REQ	from v5 to v3
8	85.43: SPLIT_DONE	from v1 to v3	8	88.24: SPLIT_REQ	from v2 to v1
9	85.46: MERGE_REQ	from v5 to v3	9	89.46: MERGE_REQ	from v5 to v3
10	85.96: MERGE_REQ	from v5 to v3	10	90.46: MERGE_REQ	from v5 to v3
11	86.46: MERGE_REQ	from v5 to v3			
12	86.96: MERGE_REQ	from v5 to v3			
13	87.46: MERGE_REQ	from v5 to v3			
Extracted Pattern 1 from LCMS(p_k^0, q_n^0)			Extracted Pattern 2 from LCMS(p_k^0, q_n^{60})		

Fig. 4. Example abstracted LCS patterns extracted from the same message sequences but different time windows.

The *MCT* function not only covers the multidimensionality, but also covers the temporal features of interactions. The function compares the delivery intervals of two messages, to exclude the situation in which certain subsequences occur at significantly different time intervals. Assume that msg_{10}^p contains a 1-s interval with its previously matched message and that msg_{20}^q contains a 20-s interval. Even if all other values of msg_{10}^p and msg_{20}^q are the same, determining that two messages having significantly different delivery intervals are identical may adversely affect the accuracy of extracted patterns. We, therefore, divide the process of checking message identities into two cases. If no message is matched during the *LCS* pattern extraction, we check only the identity of the message contents, such as *sender*, *content*, and *receiver*. Otherwise, we further check whether the difference of delivery intervals of the messages is within the margin of the *delay_threshold*.

The proposed *LCMS* function in Eqs. (2) and (3) can extract the *LCS* of interaction messages between any two message sequences. However, it is difficult to conclude that the proposed function always extracts the most "critical" message subsequence that accurately contains the information needed to identify the root causes of failures. The term "critical" indicates the quality of information owned by a specific FII pattern regarding collaboration failures. Because the *LCS*-based algorithms start from the "firstly matched" instance, *LCS* patterns may include completely meaningless parts prior to the critical points.

Fig. 4 depicts two example *LCS* patterns extracted from the same message sequences with different time windows. Example pattern 1, starting from 53.03 s, consists of repetitive `MERGE_REQ` messages and a few other messages. Otherwise, pattern 2, starting from 85.00 s, contains various messages between *V1*, *V2*, *V3*, and *V5*, such as `LEAVE_REQ`, `SPLIT_REQ`, and `MERGE_REQ`s. Pattern 2 provides a more critical understanding of the failure that *V5* repetitively requests `Merge` to *V3* when *V3* is still in the `Leave` operation between *V2* and *V1*.

To accurately extract the critical FII patterns, we define the algorithm that extracts *LCS* patterns according to several time windows of input message sequences and selects the most "critical" *LCS* among the *LCS*s. Let $t_1, t_2 \in T = \{t \in \mathbb{R}_{\geq 0} \mid t \text{ is a time window starting time}\}$, M be the set of message sequences, and $n, k \in \mathbb{N}$ be the length of message sequences. We define the sub-message sequences starting from t_1 and t_2 as follows:

$$p_k^{t_1} \triangleq msg_1^p, msp_2^p, msg_3^p, \ldots, msg_k^p$$
$$q_n^{t_2} \triangleq msg_1^q, msg_2^q, msg_3^q, \ldots, msg_n^q$$

To properly select the most "critical" *LCS* among the generated ones, we define the parameters needed to evaluate the quality

of the *LCS*s as the number of content types in the *LCS* and their lengths. We assume that an *LCS* is more "critical" than other *LCS*s if it contains more *Content* types and its length is shorter than others, so that it contains more informative FII sequences with fewer redundant messages. This definition is based on the priority among the contexts and symptoms of failures. The context denotes the conditions and execution flows in which failures occur. Symptoms denote the results of failures and are frequently used in fault detection techniques as failure indicators (Sauvanaud et al., 2018; Zhang et al., 2019). However, our study focuses on the analysis of collaboration failures, especially identifying root causes. During this analysis process, the failure occurrence context provides more meaningful knowledge to help understand the root causes. Therefore, we prioritize the *LCS*s with various types of interaction *Content*s providing more contextual information.

Let $k, n \in \mathbb{N}$. We define the aforementioned methods to *T-LCS* and *TIME-LCS* as follows:

$$T\text{-}LCS(p_k, q_n) \triangleq \bigcup_{t_1, t_2 \in T} LCMS(p_k^{t_1}, q_n^{t_2}) \tag{4}$$

$$TIME\text{-}LCS(p_k, q_n) \triangleq \underset{m \in T\text{-}LCS(p_k, q_n)}{argmax}\ NumContentTypes(m), \tag{5}$$

where function $NumContentTypes : M \rightarrow \mathbb{N}$ maps an input *LCS* to the number of independent content types contained in messages belonging to the given *LCS*. When the algorithm calculates *T-LCS*, it generates a set of *LCS*s with each sub-message sequence of p_k and q_n by a discrete time window starting at t_1 and t_2 in T. As described above, the function, *TIME-LCS*, first selects an *LCS* among *T-LCS* by the number of content types in Eq. (5). If the number of content types is the same, we use the lengths of *LCS*s as a tie-breaking rule.

3.2. TIME clustering

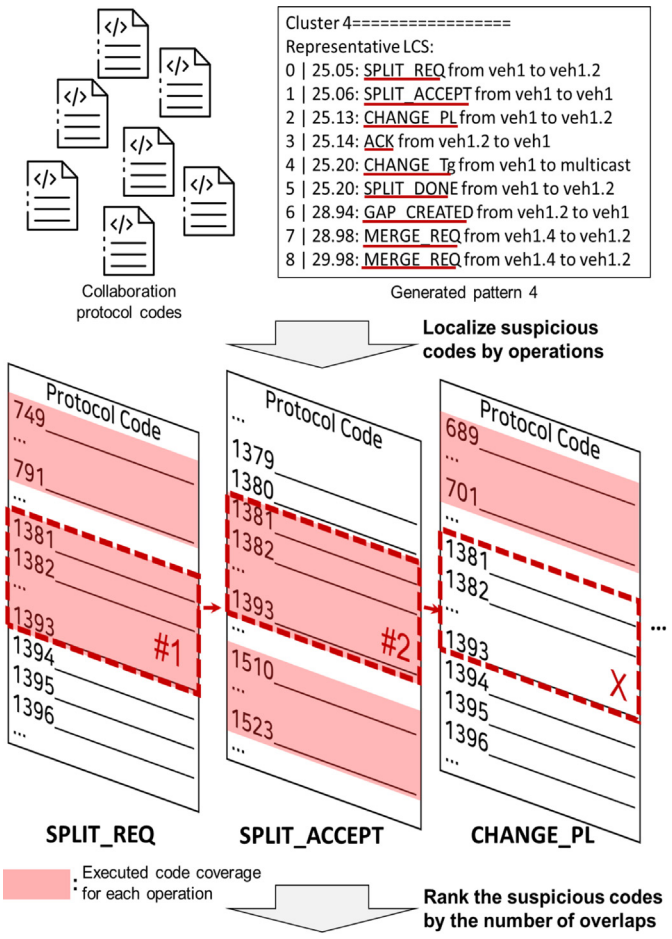
Based on the *TIME-LCS* algorithm, this study proposes a *TIME* overlapping clustering technique that effectively groups similar failed logs and extracts the common patterns of collaboration failures. Because we assume that an overlapping clustering is necessary to extract multiple patterns in a single message sequence, m_i, our technique adds an m_i to several clusters, if the m_i is regarded to have several faulty patterns simultaneously.

To achieve this, we define two similarity calculation metrics (i.e., *TIMELen* and *TIMESim*) based on the *TIME-LCS* function. The reason for suggesting these two metrics is that *LCS*-based algorithms are basically pair-wise algorithms. Hence, we divide the cases into (a) a cluster with a single element (i.e., m_i) and (b) a cluster with two or more elements, and we use different similarity metrics for each case. In the case of (a), because there exist not enough elements to generate *LCS* pattern for the cluster, the subjects of similarity calculation are a message sequence in an existing $cluster_j$ and a given m_i. For instance, the first message sequence, m_0, always forms a new cluster, $cluster_0$. The second given message sequence, m_1, will be compared to the m_0 in $cluster_0$. Thus, in the case of (a), we define the similarity of two message sequences as the length of the *LCS* between the message sequences.

$$TIMELen \triangleq len(TIME\text{-}LCS(m_{cluster_j},\ m_i)) \tag{6}$$

$$TIMESim \triangleq \frac{len(TIME\text{-}LCS(cluster_j.pattern,\ m_i))}{len(cluster_j.pattern)} \tag{7}$$

In Eq. (6), $m_{cluster_j}$ denotes a sole message sequence located in the $cluster_j$ and m_i denotes a message sequence given to the technique as an input. The *TIMELen* similarity value becomes the length of the *LCS* extracted from the *TIME-LCS* function.



Lines of Codes	Overlapped #	Suspicious Orders
853-859	6	1
870-871	5	2
729-744	3	3
…	…	…

Fig. 5. Example of pattern-based fault localization.

Otherwise, when there exist more than two elements in a cluster, *TIMESim* in Eq. (7) is applied. Subjects of the *TIMESim* in the case of (b) include the *LCS* pattern in an existing cluster, $cluster_j.pattern$, and a given message sequence m_i. The *TIMESim* metric calculates the LCS-based sequence similarity between the m_i and $cluster_j.pattern$.

Using these two metrics with the threshold values, *len_ threshold* and *similarity_threshold*, the technique determines whether to add the given m_i to existing clusters or to create a new cluster. For example, if the *similarity_threshold* is 0.8 and the *TIMESim* value is 0.75 for m_{10} and $cluster_3$, m_{10} is not assigned to $cluster_3$. This process also enables the simultaneous extraction of multiple patterns in an m_i. If the given m_i is similar enough to be added to $cluster_j$, the m_i is added to $cluster_j$ and $cluster_j.pattern$ is updated using m_i. Otherwise, if there exists no cluster that is similar enough to add the m_i, our technique creates a new cluster with the given m_i. This clustering process is repeated until no m_i exists and finally returns FII patterns corresponding to the cluster sets.

3.3. Pattern-based fault localization

Based on the generated patterns, we propose a suspicious code localization technique for SoS collaboration protocol codes that can reduce the significant cost needed to localize the root causes of collaboration failures. A pattern contains a sequence of communication messages involving CS-level operations as their *Content*s. The localization technique infers suspicious codes by using the code coverage calculation method for CS-level operations, such as `SPLIT_REQ` and `MERGE_REQ`. By executing every single CS-level operation and measuring the code coverage, we built a code coverage set that records the actually executed code lines for each operation. For example, lines 1381–1431 are executed in the single execution of *SPLIT_REQ.*

By using the code coverage set of each CS-level operation, our approach ranks the collaboration protocol codes by the suspiciousness of causing the failures. There are numerous studies of spectrum-based fault localization (SBFL) for calculating the suspiciousness of code lines from the code execution coverage (Jones et al., 2001; Abreu et al., 2006; Naish et al., 2011; Wong et al., 2013, 2016). However, SBFL techniques only consider the coverage of operations without considering the sequential orders of the execution. Thus, we propose a suspiciousness ranking method, *SeqOverlap*, based on the number of sequential overlaps of the codes. *SeqOverlap* method is aimed to prioritize the most repeatedly executed code statements in the sequence of CS-level operations in FII patterns. For example, the codes related to the execution of `SPLIT_REQ`, the first message at the top of Fig. 5, contain lines 1381–1393. The next message is `SPLIT_ACCEPT`, and the related codes contain lines 1381–1393. Hence, lines 1381–1393 have two sequential overlaps. In this manner, we calculate the number of sequential overlaps and order the code snippets depicted as the bottom part in Fig. 5. Even though the spectrum of the proposed study is a single statement as in the SBFL methods, the rankings are determined in units of code snippets because the overlapping is counted according to the execution of a single operation of the collaboration protocol. In the example, lines 853–859 are ranked first, because the code snippet is frequently executed by `MERGE_REQ` and `ACK` in the platooning simulations.

4. Experiment

4.1. Experiment design

The goal of our experiment is to demonstrate the efficacy and accuracy of the proposed approach based on the three major outputs: extracted patterns, clustered failed logs, and localized codes. We have utilized two target systems in this experiment: platooning SoS and mass casualty incident-response (MCI-R) SoS.

Platooning SoS. Table 1 explains the overall summary of the two target systems. For the platooning SoS, we utilized the PLTBench dataset (Hyun et al., 2021), which provides the investigation results (e.g., failure occurrence context, symptoms, and code-level bugs) and classified logs of collaboration failures in platooning SoS. The PLTBench dataset provides 10 scenarios (i.e., classes) and six bugs of platooning collaboration failures (Hyun et al., 2021) from the analysis of operation success rate property on about 8000 randomly generated simulation logs. In addition to the failures and bugs provided by PLTBench, we discovered two new failure scenarios and identified one bug through this experiment. Thus, a total of 12 failure scenarios and seven bugs were used in the evaluation. The details of the new failures and the bug are presented in Section 4.3. The size of LoCs of the platooning collaboration protocol is 3596 out of 540,277 in VENTOS (Amoozadeh et al., 2015), encompassing codes only related to executing the platooning collaboration except for the simulation, configuration, logging, and rendering.

Table 1
Overall statistics of the target systems in experiment.

Scenario	Platooning SoS (Hyun et al., 2021)	MCI-R SoS (Park et al., 2020)
Types of CSs	6 types of vehicles	5 independent CSs
Number of logs	7,935	2,034
Number of failed logs	3,985	1,005
Verification property	Operation success rate in platooning	Treatment rate of patients
Threshold value	0.8	0.9
Failure scenarios	12* types of scenarios	5 types of injected scenarios
Number of faults	7* faults	5 faults
Collaboration-related LoCs	3,596	2,634
Benchmark/Simulator	PLTBench dataset[a]/VENTOS[b]	SIMVA-SoS simulator[c]

*Including the number of newly detected failure scenarios and faults.
[a]https://sites.google.com/se.kaist.ac.kr/pltbench
[b]https://maniam.github.io/VENTOS/
[c]https://github.com/psumin/SoS-simulation-engine

MCI-R SoS. The goal of MCI-R SoS is to rescue and treat as many patients in MCI situations as possible through the collaboration of firefighters, ambulances, SoS managers, bridgeheads, and hospitals. SIMVA-SoS (Park et al., 2020) provides simulation and verification modules for MCI-R SoS, defining various types of failure-inducing stimuli, such as communication loss, delay, and specific bugs in code. We found five types of bugs among the defined stimuli, which are specifically located in the collaboration protocol code, in SIMVA-SoS. For example, the deadlock fault among the Bridgehead-SoS manager-Ambulance prevents the three CSs from sharing patient arrival information, resulting in the failure of achieving a 90% patient treatment rate. We generated 2034 logs including 1005 failure logs with the faults injected. The size of LoCs of MCI-R collaboration protocol is 2364 out of 8691 in SIMVA-SoS.

We defined the following research questions for the experiment:

- **RQ1.** Does the proposed approach accurately extract FII patterns that explain collaboration failures?
- **RQ2.** Do the clustering results yield better clustering precision considering multiple FII patterns?
- **RQ3.** Does the localization method precisely infer the bug location in the protocol from the patterns?

RQ1 aims to check the accuracy of the proposed pattern mining technique, which prevents the information loss during the collaboration failure pattern mining by considering the major features of SoS interaction logs. Here, we evaluated the accuracy of the generated patterns by calculating the similarity with the manually created FII patterns, *ideal patterns*. The *ideal patterns* contain critical information about failures, such as failure context, triggering events, and symptoms. We created the *ideal patterns* for all collaboration failure classes of platooning SoS based on the analysis results from PLTBench in advance.

We defined the pattern identity with weight (PITW) score by extending the difference-based accuracy measure, MAE (Cobos et al., 2013; Tarus et al., 2018), to SoS interaction sequences. PITW is calculated by dividing the number of identical messages between the *ideal* and generated patterns (true-positive (TP)) by the length of *ideal patterns* (TP + true-negative (TN)). Additionally, instead of giving the same point to all messages in patterns, weights were given to messages essential for understanding the SoS failures. For example, if the length of an *ideal pattern* is 10 and

3 of them are essential and the number of identical messages is 7 with 1 essential message, the PITW score is 0.62 ((7+1)/(10+3)). The details of the *ideal patterns* are presented in Section 4.3.

We evaluated a total of four approaches: *TIME*, *BASE* (Hyun et al., 2020), *SPADE* (Zaki, 2001), and *LOGLINER* (Amar and Rigby, 2019). *BASE* is our previously proposed *LCS*-based pattern mining and clustering method that does not fully consider the features of interaction logs. *SPADE* is one of the most commonly used frequent sequence mining algorithms in various domains (Millham et al., 2021; Huynh et al., 2018; Maylawati et al., 2018). *LOGLINER* is a recently proposed log-flagging algorithm to detect the most suspicious log lines. We conducted a total of six experimental cases based on the approaches: *Single-TIME*, *Single-BASE*, *Single-LOGLINER Multi-TIME*, *Multi-BASE*, and *Multi-SPADE*. *Single* indicates that the approaches were applied to each set of categorized logs for 12 failure classes respectively, thus focused only on mining patterns without classification. *Multi* means that the approaches were applied to the whole log set, including the logs of all failure classes. We repeated the above experimental cases 30 times according to the random order of input logs.

RQ2 aims to evaluate the overlapping clustering precision regarding the existence of the multiple failure patterns (e.g., cascading failures in SoS). With the comparison of clustering precision, we intend to show that our technique is effective in classifying failed logs that include multiple patterns. Indeed, hundreds of platooning SoS logs contain multiple failure patterns in PLTBench (Hyun et al., 2021). We compared *Multi-TIME* and *Multi-BASE* clustering results by considering all possible combinations of hyperparameter values.

To properly evaluate the overlapping clustering precision, we implemented the recently proposed F1P score (Lutov et al., 2019). The F1P is defined as follows:

$$F1P(C', C) = \frac{2F_{C',C}F_{C,C'}}{F_{C',C} + F_{C,C'}}, \text{ where}$$

$$F_{X,Y} = \frac{1}{|X|}\sum_{x_i \in X} pprob(x_i, g(x_i, Y)),$$

$$g(x, Y) = \{\underset{y}{argmax}\ pprob(x, y) | y \in Y\},$$

$$pprob(c', c) = \frac{matched^2}{|c'| * |c|}.$$

C denotes a set of formed clusters consisting of clusters $c \in C$, and C' denotes a set of ground-truth clusters (i.e., a set of categorized failed logs in the PLTBench dataset) consisting of clusters $c' \in C'$. $|c|$ denotes the sum of contributions of elements in c, considering the number of clusters with which an element is involved. *matched* refers to the sum of contributions of matched elements of c and c'.

Further, we set the range of hyperparameters used in the overlapping clustering as follows:

- Message delay (*delay_threshold*): 0.1–1.0 by 0.1
- *LCS* minimum length (*len_threshold*): 5–20 by 1
- *LCS* similarity (*similarity_threshold*): 0.6–1.0 by 0.01

We compared the performance of the two scenarios in all combinations of the hyperparameter ranges with a time window set, $T = \{0, 20, 40, 60, 80\}$.

RQ3 aims to show the feasibility of end-to-end localization methods that map extracted patterns to bugs in collaboration protocol code. We evaluated the localization results by the EXAM (Wong et al., 2008) and Top-K (Parnin and Orso, 2011) scores that estimate the reduced cost required in debugging and the accuracy of the localization. In this study, the EXAM score is defined by $(N-n)/N$ where N is the total rank of the code statements, and n is the number of code statements to be investigated for resolving the failures. The Top-K score is defined by the number of faulty statements localized within K-ranks (K = 10, 50, 100).

Here, we compared the proposed fault localization method, *SeqOverlaps*, with several SBFL methods: Tarantula (Jones et al., 2001), Ochiai (Abreu et al., 2006), OP2 (Naish et al., 2011), Barinel (Abreu et al., 2009), and DStar (Wong et al., 2013). To the best of our knowledge, this is the first study to present a localization method based on patterns. Therefore, we indirectly evaluated the localization performance with the conventionally used SBFL methods (Pearson et al., 2017). The spectra of the SBFL methods are defined by the code coverage of executed statements in the collaboration protocol files of the target system. Based on the spectra of each failed and passed scenarios, we applied the SBFL methods to calculate the suspiciousness of each code statement.

Collaboration failures are predominantly caused by the omission of specific logic in the protocol (Hyun et al., 2021). The PLTBench dataset contains a few multi-statement bugs, where bug fixes span multiple statements (Pearson et al., 2017). In calculating the EXAM score in these types of bugs, we applied the worst-case debugging scenario, thus all the buggy statements needed to be fixed. Additionally, we followed the average-rank strategy (Steimann et al., 2013; Wong et al., 2016) when multiple statements have the same suspiciousness score, then all of the codes are treated as the average rank of the statements.

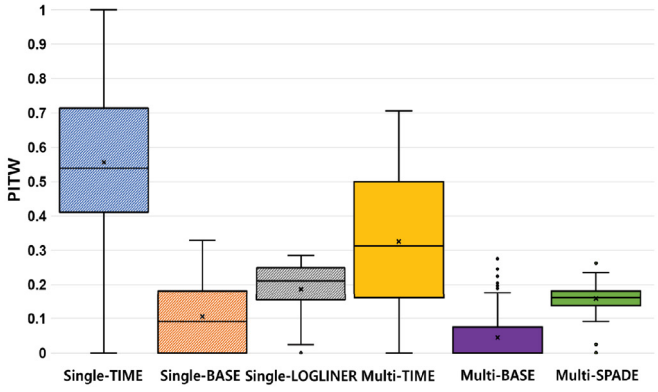
4.2. Experimental results

RQ1. Fig. 6 depicts the PITW scores of six experimental cases on the platooning and MCI-R SoS dataset. We found that our approach, *TIME*, showed a significantly high accuracy in FII pattern mining. The first three plots are *Single* cases that were applied to the categorized sets of logs based on each of the failure classes in the datasets. The next three results are *Multi* cases generated by using all the failed logs as inputs. For the best and average of the PITW values, the *TIME* approach showed the highest accuracy in both of the target systems. In particular, the *Single-TIME* case only achieved the extraction of FII patterns containing 100% of the *ideal patterns* in both target systems. The *Multi-TIME* case showed higher performance than all of the other approaches. It also exhibited the most comparable accuracy to *Single-TIME* in the best and average cases, even though the clustering of failed logs was also considered. *Multi-TIME* results in Fig. 6 indicate that the proposed approach extracted FII patterns from failed logs that contain 70 to 100% of the fault knowledge in the best case. In the average case, the proposed approach automatically extracted multiple FII patterns containing 30 to 50% of the fault knowledge.

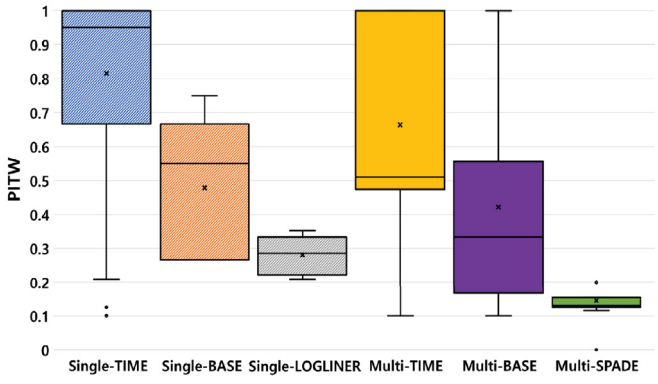
Even though the *BASE* approach depicts higher performance than *LOGLINER* and *SPADE* in MCI-R SoS data, the performance of the *BASE* approach fell short of expectation for both the *Single* and *Multi* cases in the platooning dataset. We decided that the point where *BASE* did not consider the temporal features of interactions had serious effects on the analysis of a considerable number of interaction logs. Consequently, *BASE* exhibited a performance difference compared to *TIME* in both of the *Single* and *Multi* case in both target systems.

The results of *LOGLINER* and *SPADE* were also lower than those of *TIME* but exhibited a lower deviation of accuracy despite the random input order. This is because the key algorithm in the approaches is the counting of specific elements and sequences, thus the results are scarcely affected by the input order.

Further, the *TIME* approach demonstrated a high deviation of PITW values, such as deviation values of 0.9 for *Single* and 0.8 for *Multi* cases on average, as shown in Fig. 6. This demonstrates that the proposed approach is relatively sensitive to the order



(a) PITW evaluation results on the platooning SoS dataset



(b) PITW evaluation results on the MCI-R SoS data

Fig. 6. PITW evaluation results.

of inputs. This technical issue requires to be solved to increase the average performance of FII pattern mining. Nevertheless, the *TIME* approach presented the highest pattern mining accuracy on average. We also found that the *TIME* approach exhibited the highest performance of mining FII patterns in most of the failure classes in Fig. 7. For some MCI-R failure scenarios, *Multi-BASE* depicts higher PITW values than *Multi-TIME*. This is because the *Multi-BASE* approach generated patterns with hundreds of lengths; thus, the parts of the ideal patterns were included with a high probability. This issue is explained in detail in Section 4.4.

> *Findings.* The *TIME* approach presented the highest accuracy on the mining of FII patterns with and without including the classification of patterns.

RQ2. We evaluated the efficacy of our clustering approach considering the extraction of multiple failure patterns in a log. We utilized the F1P score, which calculates the precision of overlapping clustering results based on the number of pair-wised TP, TN, false-positive (FP), and false-negative (FN) elements compared with the ground-truth clustering results (Lutov et al., 2019).

Fig. 8 demonstrates the F1P evaluation results of 30 random inputs of the two clustering approaches on the two target systems. The *y*-axis denotes the F1P score values described in Section 4.1. In the overall performance distribution according to the hyperparameter options, the *TIME* surpassed the *BASE* approach. Particularly, *TIME* clustering showed an F1P score (0.78) that was almost four times higher than that of *BASE* (0.22) for the best score on the platooning dataset and achieved 90% of overlapping clustering precision on the MCI-R SoS dataset. This indicates that the proposed *TIMESim* and *TIMELen* metrics and the *TIME* clustering process fully considered the characteristics of SoS interaction logs, enabling sophisticated classification of multiple failure patterns.

We discovered two new failure classes in the platooning dataset based on the clustering results. We found that the average number of clusters for the best F1P options was 11.67, while the PLTBench dataset only contains 10 types of failure scenarios. Consequently, we inferred two new failure scenarios that have distinguishing features on context and symptoms compared to existing scenarios. The details of the new patterns are described in Section 4.3.

> *Findings.* The proposed *TIME* similarity metrics and clustering process showed significantly high overlapping clustering precision than the *BASE*.

RQ3. As the last evaluation factor, we checked the feasibility of the pattern-based fault localization by comparing the proposed method with the existing SBFL methods. We applied the SBFL methods to the code coverage of each failed log category with all coverage of passed logs. Similarly, our localization method, *SeqOverlap*, was applied to each pattern of the failure classes.

Fig. 9 depicts the EXAM scores of the localization methods on each failure class of platooning and MCI-R SoS. The higher the EXAM score, the lower the cost required for finding bugs, which means that the buggy codes are accurately localized at high ranks. Based on Fig. 9, we confirmed that *SeqOverlap* has the highest EXAM score in all failure classes in both target systems. The difference in the EXAM score between the *SeqOverlap* and SBFL methods was 24% on average. In particular, the SBFL methods achieved lower EXAM scores on average in MCI-R SoS than those in platooning SoS. This is because most of the collaboration protocol codes are covered by both failed and passed scenarios; thus, the SBFL methods calculate a myriad of the same rank codes in results. This indicates that the pattern-based fault localization method showed feasible performance in localizing the root causes of collaboration failures in the two target systems.

SBFL methods are based on the code execution coverage. Coverage-based methods do not consider the execution order and timing of operations during the collaboration, but only consider whether a code line is covered according to the execution of operations. However, most of the collaboration protocol code can be covered in a SoS simulation regardless of pass and failure results because several CS agents (e.g., vehicles and firefighters) conduct various collaborative operations autonomously. Consequently, coverage-based SBFL methods cannot infer the significant difference of code coverage of the passed and failed executions of collaboration protocols. This explains why SBFL methods showed lower localization accuracy than that of our method on the time/order-sensitive collaboration protocols.

The EXAM scores of platooning failure class 9 and new class 2 are lower than other results in Fig. 9. This is because the buggy code lines of the failure classes are multiple lines distributed among various functions. For example, the buggy code lines of the platooning failure class 9 are located in lines 1332, 1473, and 1769, which are placed in all different function blocks. The corresponding failures can be resolved by fixing all the buggy statements; thus, we used the last rank in which all statements were found to calculate the EXAM score. In the multi-statement bugs, our pattern-based code localization method achieved a 40.39% higher EXAM score on average than other SBFL methods.

Table 2 describes the Top-K results of the SBFL and the proposed localization methods. Higher values are better for this

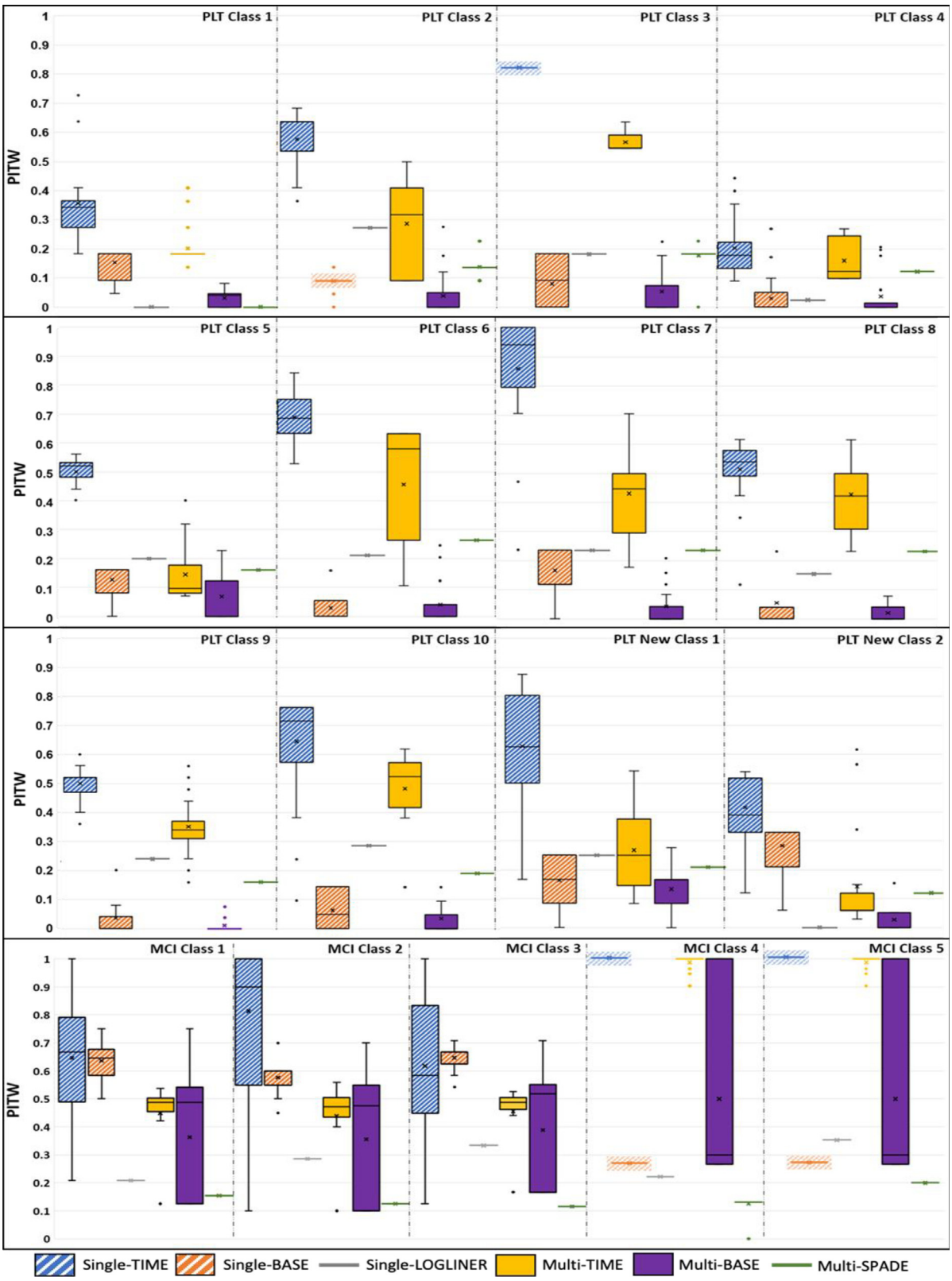


Fig. 7. PITW evaluation results of all failures scenarios in platooning and MCI-R SoS.

metric. The proposed *SeqOverlap* method achieved the best Top-K score for every K value in Table 2. Among all the methods, *SeqOverlap* solely ranked the buggy statements in the Top 10 and ranked most of the failure cases in Top 50 and 100, except for the two distributed multi-statement bugs. Ochiai and OP2 accomplished the highest Top-50 and Top-100 score among the SBFL methods, respectively. Tarantula failed to rank the bugs within Top 100 for all failure classes.

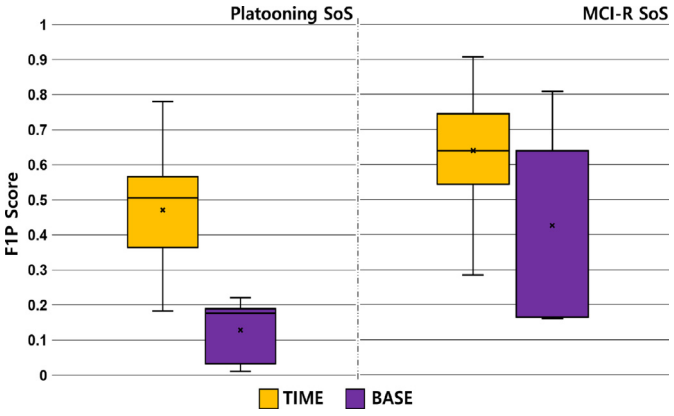


Fig. 8. F1P evaluation results of the *TIME* and *BASE* clustering.

Findings. The pattern-based fault localization approach achieved the highest EXAM and Top-K scores for all failure classes, including the multi-statement bugs.

4.3. Discussion

New Failures and a Bug. The PLTBench dataset was built via a systematic process involving the manual investigation of thousands of logs (Hyun et al., 2021). However, we found that the optimal clustering results of *Multi-TIME* contained 11.67 clusters on average, which differed from the 10 failure classes in PLTBench. Through the detailed analysis of the results, we discovered two new failure scenarios related to the existing failure class 4, simultaneous `Leave` and `Merge`, and a new bug that caused one of the failures.

Fig. 10(a) describes the *ideal pattern* for the failure class 4 in PLTBench. In the pattern, a yellow box depicts the failure occurrence context and blue boxes denote the symptoms. In lines 1 to 2, *V1* got requests of `Leave` and `Merge` from *V1.2* and *V1.4*, respectively. Lines 3, 4, and 8 specify that the `Leave` operation is accepted and two `Splits` are requested: the first `Split` is to make space for *V1.2* and the second is to make *V1.2* left. In this process, several `Merges` are continuously requested from *V1.4* to *V1* and *V1.3*, such as lines 6, 7, 10, and 11. These meaningless, repetitive requests cause not only delays in the `Leave` operation, but also the result that *V1.4*, the `Merge` sender, cannot execute other operations.

However, the patterns we found in the experiment have features that are distinct from those in Fig. 10(a). The new FII pattern 1 in Fig. 10(b) seems similar to the *ideal pattern 4*, but the new pattern only has one `SPLIT_DONE` operation. This difference results from the execution of a different `Leave` operation. The original failure class 4 is in the context of `MiddleFollowerLeave` and the new pattern describes the `EndFollowerLeave`.

Fig. 10(c) presents a different kind of failure scenario. It has the same context as *ideal pattern 4*, where both `Leave` and `Merge` are requested. However, in the new failure, `Merge` is the firstly requested operation, as depicted in the first red box, and none of the two operations are normally executed at once as depicted in lines 3 to 6, 10, and 11. This failure has totally different symptoms from *ideal pattern 4*, in that both of the operations are omitted a few times and the `Leave` is eventually executed at a significantly delayed time. The reason why it was difficult to detect this failure scenario in the manual analysis is that the failure class is a great edge-case, where only four out of every 8000 logs contain the failure. However, our approach has strength in accurately mining such unique sequences containing several requests of `Leave` and `Merge` from thousands of logs. The application of overlapping clustering contributes to the classification and extraction of the unique failure pattern.

Table 2
Top-K analysis results on the bugs of collaboration failures.

K	Tarantula	Ochiai	OP2	Barinel	Dstar	SeqOverlap
10	0	0	0	0	0	2
50	0	3	1	1	1	14
100	0	6	8	3	6	15

Moreover, we identified a new bug causing the new failure class 2. Fig. 10(d) illustrates the bug location and an example patch of the bug. Line 815 of the "*05_PlatoonMg.cc*" file in VENTOS simulator (Amoozadeh et al., 2015), which has a target platooning protocol code, contains one of the buggy codes. The patch example makes continuous `MERGE_REQ` not to be repeated so that other operations such as `Leave` are not omitted by checking the non-response vehicles' ids.

Ideal Patterns and PITW Score. *Ideal patterns* for the platooning collaboration failures were used as key factors for evaluating the accuracy of FII pattern mining results. We manually created the *ideal patterns* by investigating the detailed fault knowledge of the failures in PLTBench. For example, the fault knowledge corresponding to the *ideal pattern 4* in Fig. 10(a) is as follows:

> Context: During the `MiddleFollowerLeave` operation,
> Triggering events: The rear platoon leader requests `Merge` to the same leader who is simultaneously requesting `Middle FollowerLeave` operation.
> Symptoms: The rear platoon leader constantly sends `MERGE_REQs` to the original leader.

The pattern in Fig. 10(a) follows the context of the `Middle FollowerLeave` operation having two `Splits`; thus, the pattern contains the sequence of `LEAVE_REQ`, `LEAVE_AC-CEPT`, and two occurrences of `SPLIT_REQ`, etc. In Fig. 10(a), `MERGE_REQ` in line 2 indicates the triggering events of simultaneous request, and repetitive `MERGE_REQs` in lines 6 to 7 and 10 to 11 represent the failure symptoms.

During the creation process of *ideal patterns*, we realized that not all messages constituting the patterns are essential for understanding the failures. For example, `MERGE _REQs` were actually repeated three to hundreds of times in simulations. We limited the repetition of the same message in a pattern to five, but we could classify whether the specific messages are repeated only by the two messages as presented in Fig. 10. Likewise, there exist a few CS-level operations for executing `Split`, such as `CHANGE_PL`, `GAP_CREATED`, and commonly used messages like `ACK`. However, the information about the failure situation that should be known through the patterns is not the execution of internal CS-level operations, but the fact that a specific vehicle was in the process of `Split`.

Accordingly, we checked the essentialness of the messages based on the specific description of platooning collaboration failures in PLTBench. In the above fault knowledge, the CS-level operations corresponding to the underlined parts were checked as essential. The example *Ideal pattern 4* in Fig. 10(a) only showed the essential parts among the total of 29 messages. We gave weights to the essential messages in the PITW calculation.

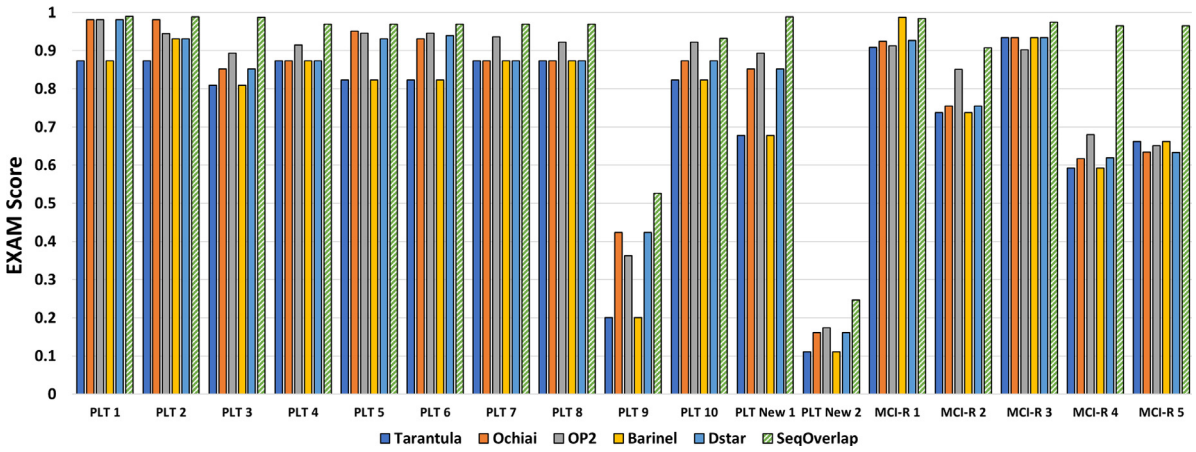


Fig. 9. EXAM analysis results on bugs causing collaboration failures in platooning and MCI-R SoS.

4.4. Threats to validity

We explained the primary threats to the validity of the experiment using the internal, construct, and conclusion validity (Wohlin et al., 2012). We elucidated the internal validity of the PITW score utilized in the RQ1 evaluation and the case of bugs in the CSs. The construct validity is explained by the target systems used in the experiment. Finally, we explicated the conclusion validity of the number of logs with the hyperparameter settings of the proposed approach.

Internal Validity. We defined the PITW score to evaluate the accuracy of the generated patterns with the manually created *ideal patterns* based on the fault knowledge. In Section 4.1, we represented the PITW score by TP / (TP + TN), where TP indicates the number of identical messages in both patterns and TN indicates the number of messages in *ideal patterns*, but not in the generated patterns. Here, we decided not to add FP value in the PITW score, which is the number of messages in the generated patterns, but not in the *ideal* patterns. It is because the *ideal patterns* are not the axiom for describing the failure scenarios and the generated patterns could point to the unpredictable context or symptoms.

We found one technical issue with the PITW score in Section 4.2. In Fig. 7, the *Multi-BASE* approach in MCI-R scenarios achieved higher PITW scores than the proposed *Multi-TIME* approach because *Multi-BASE* generated patterns with lengths of hundred, highly increasing the probability of matching with *ideal patterns*. However, the patterns with lengths of several hundred are narrowly advantageous for SoS managers to build fault knowledge on collaboration failures. We confirmed that the PITW score must include the lengths of the generated patterns for evaluating the practicality. We plan to design and utilize the improved PITW score metric in future studies.

Further, even though the scope of this study is focused on the bugs in the collaboration protocol codes, there might be some situations where CSs have bugs. We classified this case into two sub-cases: one where a CS has a bug in executing its autonomous functioning and the other where a collaboration bug is located in the codes of the functioning of CSs by implementation issues. The first case is not in the scope of this study because we assume that each CS has been sufficiently tested to execute its autonomous behaviors. In the second case, there could be several issues with implementing SoS based on black-boxed CSs as described in Section 1. We also found the same scenario where the collaboration failures occurred because of a bug statement located in the firefighter's searching code in MCI-R SoS. Because the firefighter's searching is the autonomous function of CSs, not included in collaboration with other CSs, the scenario was not used in this experiment. For this second case, it is assumed that the extracted patterns will be delivered to the manager of the CS by the independence assumption in SoS.

Construct Validity. One of the most difficult problems faced is the lack of available benchmark data wherein target systems satisfy the main characteristics of SoS. Hence, we focused on the prevalent and open platooning simulator, VENTOS, and generated the experimental dataset, PLTBench, in advance by the thorough analysis of the collaboration failures (Hyun et al., 2021). SIMVA-SoS, a simulation-based verification tool providing MCI-R SoS scenario execution, defined concrete stimuli involving a few examples of code-level faults that adversely affect the performance of MCI-R SoS (Park et al., 2020). We generated thousands of logs for the experiment by injecting the most relevant types of faults and stimuli in the MCI-R collaboration protocol.

In the evaluation, among the various types of logs provided by VENTOS and SIMVA-SoS, only message-based interaction logs delivered in the network channel were used as input logs for the evaluated approaches. This indicates that the experiment did not consider logs of individual CSs' state and internal variable values, such as vehicle states. Additionally, in evaluating the localization methods, only collaboration protocol codes were utilized, except for other codes of the CS's autonomous functioning, simulation and verification. The approaches were evaluated in an experimental setting that thoroughly considered the operational and managerial independence of SoS.

Conclusion Validity. The proposed approach has three primary thresholds, *delay_thres — hold*, *len_threshold*, and *similarity_threshold*, that impact the accuracy of the approach. In the experiment, we set the ranges of the three hyperparameters by referring to the message request duration setting in the VENTOS simulator (Amoozadeh et al., 2015), using the LCS length and similarity parameters in existing studies (Soleimany and Abessi, 2019). Based on the ranges, there exist 6560 combinations of the three hyperparameters. As we repeated the experiment thirty times, all the combinations were tested.

In addition, we found that the order of the input logs affects the accuracy of the proposed approach in the RQ1 evaluation result in Section 4.2. Because the proposed clustering method is based on the prevalently used subsequent time-series (STS) clustering for time-series data analysis (Soleimany and Abessi, 2019; Madicar et al., 2013; Rodpongpun et al., 2012), the clustering

```
Ideal Pattern 4=================
1 | 25.00: LEAVE_REQ from V1.2 to V1
2 | 25.07: MERGE_REQ from V1.4 to V1
3 | 25.07: LEAVE_ACCEPT from V1 to V1.2
4 | 25.07: SPLIT_REQ from V1 to V1.3
...
5 | 25.30: SPLIT_DONE from V1 to V1.3
6 | 26.07: MERGE_REQ from V1.4 to V1
7 | 26.17: MERGE_REQ from V1.4 to V1.3
...
8 | 29.75: SPLIT_REQ from V1 to V1.2
...
9 | 29.92: SPLIT_DONE from V1 to V1.2
10 | 30.17: MERGE_REQ from V1.4 to V1.3
11 | 31.17: MERGE_REQ from V1.4 to V1.3
...
```

(a) Ideal pattern for failure class 4 (simultaneous `Leave` & `Merge` requests)

```
Extracted Pattern 1=================
0 | 65.00: LEAVE_REQ from V.3 to V.2
1 | 65.06: MERGE_REQ from V.4 to V.2
2 | 65.09: LEAVE_ACCEPT from V.2 to V.3
3 | 65.09: SPLIT_REQ from V.2 to V.3
4 | 66.06: MERGE_REQ from V.4 to V.2
...
5 | 66.36: SPLIT_DONE from V.2 to V.3
6 | 67.06: MERGE_REQ from V.4 to V.2
7 | 67.16: MERGE_REQ from V.4 to V.3
8 | 68.16: MERGE_REQ from V.4 to V.3
...
```

(b) New FII pattern 1 extracted in experiment

```
Extracted Pattern 2=================
1 | 44.86: MERGE_REQ from V.4 to V.3
2 | 45.00: LEAVE_REQ from V.6 to V.4
3 | 45.26: MERGE_REQ from V.4 to V.3
...
4 | 46.00: LEAVE_REQ from V.6 to V.4
5 | 46.66: MERGE_REQ from V.4 to V.3
...
6 | 47.00: LEAVE_REQ from V.6 to V.4
7 | 47.08: LEAVE_ACCEPT from V.4 to V.6
8 | 47.08: SPLIT_REQ from V.4 to V.6
...
9 | 46.37: SPLIT_DONE from V.4 to V.6
10 | 50.16: MERGE_REQ from V.4 to V.3
11 | 50.26: MERGE_REQ from V.4 to V.3
...
```

(c) New FII pattern 2 extracted in experiment

Bug location

```
811 if(vehicleState == state_platoonLeader)
812 {
813     // can we merge? // Merge Request by OptimalSize
814     if(!busy && plnSize < optPlnSize)
815     {
816         if(isBeaconFromFrontVehicle(wsm))
817         {
                int finalPlnSize = wsm->getPlatoonDepth() + 1 + plnSize;
```

Patch version

```
811 if(vehicleState == state_platoonLeader)
812 {
813     // can we merge? // Merge Request by OptimalSize
814     if(!busy && plnSize < optPlnSize)
815     {
816         if(isBeaconFromFrontVehicle(wsm) && isNonResponseVehicle(wsm->getSenderID()))
817         {
                int finalPlnSize = wsm->getPlatoonDepth() + 1 + plnSize;
```

(d) Bug location and patch example for FII pattern 2 failures

Fig. 10. Example FII patterns and bug location in code.

results are prone to be changed by the input orders. Significantly, the proposed clustering technique is an overlapping clustering meant to classify cascading failures in SoS; thus, the effect of the input orders was revealed. We plan to improve the proposed clustering technique to fuzzy clustering in future work to mitigate the impact of the input orders.

5. Related work

We investigated several studies that focused on log anomaly detection and pattern mining in various domains and studies of concurrency bug analysis and graph mining-based fault localization. We examined the applicability of the techniques, including their essential assumptions to SoS collaboration failure analysis.

Log Anomaly Detection. Most log anomaly detection and analysis studies have applied clustering techniques for system text logs. Landauer et al. (2018) proposed a cluster evolving technique to effectively detect security attacks. Schmidt et al. (2020) also suggested a technique to detect critical time spans of security attacks in cyber–physical system logs. The studies utilized Euclidean distance and dynamic time warping (DTW) methods to detect security attacks. Finally, Amar and Rigby (2019) introduced two log-flagging techniques: *LOGLINER*. Those techniques are based on the term frequency-inverse document frequency (TF-IDF) approach. They applied the TF-IDF approach to each log line and calculated the uniqueness of the log lines in *LOGLINER*. The major limitation of these studies is that their similarity metrics are not applicable to the nominal and high-dimensional interaction logs in platooning SoS. In our experiment, we extend the *LOGLINER* to analyze each log line in the SoS interaction logs.

Several studies have applied supervised approaches to detect anomalies from logs. Sauvanaud et al. (2018) proposed a supervised anomaly detection technique, followed by the monitoring and data processing of cloud services. Du et al. (2017) suggested an integrated anomaly detection technique comprising long-short term memory-based log-key anomaly detection model, and parameter-level anomaly detection model. Zhang et al. (2019) proposed a supervised anomaly detection technique for unstructured log data focusing on the noise processing and unknown log sentence management. The major limitation of these supervised approaches is that they assumed the well-defined fault knowledge of the target domain for training. The goal of our study was to reduce the overall analysis cost of collaboration failures; thus, the techniques requiring detailed fault knowledge are not appropriate for this study.

Time-Series Pattern Analysis. Other studies generated anomaly patterns from time-series sensor logs by proposing pattern-based clustering approaches. Liu et al. (2019) proposed a k-means clustering approach for detecting faults in a solar power system. They defined a DTW metric for the assessment of element similarities. Soleimany and Abessi (2019) proposed *LCS*-based clustering approaches that enabled the extraction of *LCS* from sensor time-series data. Choong et al. (2017) applied fuzzy k-means clustering to spatial vehicular trajectory data. Lastly, Kleyko et al. (2018) applied hyperdimensional computing-based learning techniques to the fault analysis for power plant sensor data. The major limitation of these studies is that they mostly used Euclidean distance-based similarity metrics which cause information loss by not covering the features of SoS interaction logs. In addition, they did not consider overlapping clustering for extracting multiple patterns in a single log and assumed the manual identification of root causes after the pattern extraction.

Sequence Data Analysis. *SPADE* is one of the commonly used sequence mining algorithms in various domains (Millham et al., 2021; Huynh et al., 2018; Maylawati et al., 2018). *SPADE* algorithm generates unique ids from the dataset and calculates the frequency of the pair of the unique ids and data elements with internal databases. We applied the *SPADE* algorithm to SoS

interaction logs by generating unique ids of sequence items using contents, sender and receiver roles, such as `SPLIT_REQ-Leader-Foll ower`. However, the *SPADE* approach exhibited lower pattern mining accuracy than the *TIME* approach.

Besides, in our previous study, *BASE* (Hyun et al., 2020), we proposed a pattern mining and clustering technique for platooning SoS. In that study, we extracted FII patterns from interaction logs and manually analyzed the occurrence contexts of failures. Nonetheless, the *BASE* technique has the following limitations. Even though the *BASE* extends the *LCS* algorithm as the core metric of mining patterns, it does not consider the temporal features of interactions, such as delivery intervals and time windows. This adversely affects the accuracy of the extracted patterns, as depicted in Section 4.2. Furthermore, the approach proposed in our previous study is fully dependent on manual fault identification after pattern mining. In this study, we improve the accuracy of pattern extraction in several aspects and provide further steps for code-level fault localization.

Concurrency Bug Analysis. Furthermore, we investigated the concurrency bug analysis that have certain similarities to collaboration failure analysis in resolving failures caused by the unintended interactions of system components (i.e., threads). Cai et al. (2021) suggested a concurrency bug prediction technique by modeling branch events and checking the event feasibility. Li et al. (2019) presented a thread-safety violation detector (TSVD) in the testing phase by applying a happens-before (HB) analysis. Liu and Huang (2018) proposed an automated concurrency bug detection technique based on the incremental control-flow graph (CFG) update. However, the extant studies depend on the CFG of target systems, which cannot be utilized in SoS analysis due to the state-explosion problem (Park et al., 2020). Moreover, existing studies focused on analyzing the shared memory conflict. However, we focused on analyzing specific sequences of interactions between CSs in SoS. We concluded that the concurrency bug analysis studies have a different scope than the failure analysis.

Graph Mining-based Fault Localization. Several fault localization techniques localize the suspicious code statements and extract the failure occurrence context to reduce the software debugging cost and improve the localization results. These graph mining-based fault localization (GMFL) techniques mainly focused on code/function-level failure context; thus, most methods defined the context model based on the CFG (Zhong and Mei, 2020; Henderson and Podgurski, 2018; Chu et al., 2022). Zhong and Mei (2020) proposed a fault location prediction technique based on a convolutional neural network (CNN). When building CNN models that can predict the fault location, they generated the CFG from Java code to train the models. Henderson and Podgurski (2018) proposed a depth-first search-based CFG mining technique to extract the critical sub-graphs according to suspicious fault behaviors. Chu et al. (2022) applied the GMFL approach to concurrent programs. They defined inter-thread CFG models that can link memory access patterns that frequently occurred in the failing executions to diagnose the concurrency bugs better.

A few studies (de Souza et al., 2018; Yu et al., 2016; Zhang et al., 2020) defined their function-level context model to analyze the software failures effectively. de Souza et al. (2018) described the code hierarchy and integration coverage model to extract the contextual information of failures. Notably, the integration coverage model used the method call pairs to relate the caller and callee methods to extract suspicious techniques. Yu et al. (2016) applied a Bayesian network to fault localization. They used the Bayesian network-based program dependence graph to calculate the probability of each program entity in specific failure scenarios. Zhang et al. (2020) proposed an abstract syntax tree model-based fault localization to link the bug reports to the faults in the source files.

The primary difference between our work and these studies is that the GMFL techniques mainly focused on analyzing code/function-level failure context. This difference causes two limitations in the SoS collaboration failure analysis. (1) Code/function-level context models are not appropriate for system-level SoS collaboration analysis. The code/function-level context models are not appropriate to catch the failure context of the SoS collaboration because the internal code executions of the CSs are black-boxed. (2) Most of the GMFL techniques face the state/path explosion problem due to the increasing complexity of the CFG modeling on the SoS collaboration.

6. Conclusion

We proposed a pattern-based clustering and localization technique based on the SoS interaction logs for effectively analyzing collaboration failures in SoS. We addressed three issues associated with existing studies, namely, information loss caused by the limited capabilities of handling interaction logs, no consideration for the existence of multiple failure patterns, and lack of the end-to-end solution from failure patterns to buggy codes. To address the issues, we proposed a *TIME-LCS* pattern mining algorithm, *TIME* overlapping clustering, and a pattern-based fault localization method, *SeqOverlap*. In the experiment on platooning and MCI-R SoS collaboration failures, the proposed approach achieved the highest pattern mining accuracy compared with existing pattern mining studies and yielded an overall improvement in clustering precision compared with our previous study. Lastly, the localization method achieved a 15% higher localization accuracy than SBFL methods on average. We expect that the conclusions of this study can enrich the accurate analysis of SoS collaboration failures. This study is a first attempt at checking the feasibility of pattern-based fault localization for such communication-intensive SoS.

We plan to improve the proposed approach in three ways. As described in Section 4.3, we intend to improve the overlapping clustering approach to minimize the effect of the input order. We will also propose an enhanced suspiciousness calculation metric focusing on multi-statement and omission bugs. Moreover, we plan to generate additional SoS scenarios, such as drone swarming or smart warehouses, extending the collaboration failure analysis scope to include CS–Environment interactions.

CRediT authorship contribution statement

Sangwon Hyun: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Visualization, Writing – original draft, Writing – review & editing, Resources. **Jiyoung Song:** Conceptualization, Formal analysis, Writing – review & editing, Validation. **Eunkyoung Jee:** Conceptualization, Investigation, Writing – review & editing, Resources, Supervision. **Doo-Hwan Bae:** Conceptualization, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: co-author serving as a senior associate editor of Journal of Systems and Sofware - Doo-Hwan Bae.

Data availability

I have shared the links to the code and experimental data in the manuscript.

Acknowledgments

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2023-2020-0-01795), (No. 2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation), and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2022R1I1A1A01072004).

References

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing. PRDC'06, IEEE, pp. 39–46.

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2009. Spectrum-based multiple fault localization. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 88–99.

Amar, A., Rigby, P.C., 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 140–151.

Amoozadeh, M., Deng, H., Chuah, C.-N., Zhang, H.M., Ghosal, D., 2015. Platoon management with cooperative adaptive cruise control enabled by VANET. Veh. Commun. 2 (2), 110–123.

Augustine, M., Yadav, O.P., Jain, R., Rathore, A., 2012. Cognitive map-based system modeling for identifying interaction failure modes. Res. Eng. Des. 23 (2), 105–124.

Cai, Y., Yun, H., Wang, J., Qiao, L., Palsberg, J., 2021. Sound and efficient concurrency bug prediction. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 255–267.

Choong, M.Y., Angeline, L., Chin, R.K.Y., Yeo, K.B., Teo, K.T.K., 2017. Modeling of vehicle trajectory clustering based on LCSS for traffic pattern extraction. In: 2017 IEEE 2nd International Conference on Automatic Control and Intelligent Systems. I2CACIS, IEEE, pp. 74–79.

Chu, J., Yu, T., Huffman Hayes, J., Han, X., Zhao, Y., 2022. Effective fault localization and context-aware debugging for concurrent programs. Softw. Test. Verif. Reliab. 32 (1), e1797.

Cobos, C., Rodriguez, O., Rivera, J., Betancourt, J., Mendoza, M., León, E., Herrera-Viedma, E., 2013. A hybrid system of pedagogical pattern recommendations based on singular value decomposition and variable data attributes. Inf. Process. Manage. 49 (3), 607–625.

de Souza, H.A., Mutti, D., Chaim, M.L., Kon, F., 2018. Contextualizing spectrum-based fault localization. Inf. Softw. Technol. 94, 245–261.

Du, M., Li, F., Zheng, G., Srikumar, V., 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1285–1298.

Group, P.E., 2022. Enabling safe multi-brand platooning for Europe (ENSEMBLE). URL https://platooningensemble.eu/. (Online; Accessed 26 May 2022).

Henderson, T.A., Podgurski, A., 2018. Behavioral fault localization by sampling suspicious dynamic control flow subgraphs. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 93–104.

Huynh, B., Trinh, C., Huynh, H., Van, T.-T., Vo, B., Snasel, V., 2018. An efficient approach for mining sequential patterns using multiple threads on very large databases. Eng. Appl. Artif. Intell. 74, 242–251.

Hyun, S., Liu, L., Kim, H., Cho, E., Bae, D.-H., 2021. An empirical study of reliability analysis for platooning system-of-systems. In: 2021 IEEE International Symposium on Autonomous Vehicle Software. IEEE AVS 2021, IEEE.

Hyun, S., Song, J., Shin, S., Bae, D.-H., 2019. Statistical verification framework for platooning system of systems with uncertainty. In: 2019 26th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 212–219.

Hyun, S., Song, J., Shin, S., Baek, Y.-M., Bae, D.-H., 2020. Pattern-based analysis of interaction failures in systems-of-systems: A case study on platooning. In: 2020 27th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 326–335.

Jiang, H., Li, X., Yang, Z., Xuan, J., 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 712–723.

Jones, J.A., Harrold, M.J., Stasko, J.T., 2001. Visualization for fault localization. In: Proceedings of ICSE 2001 Workshop on Software Visualization. Citeseer.

Jones, N.C., Pevzner, P.A., Pevzner, P., 2004. An Introduction to Bioinformatics Algorithms. MIT Press.

Kazman, R., Schmid, K., Nielsen, C.B., Klein, J., 2013. Understanding patterns for system of systems integration. In: 2013 8th International Conference on System of Systems Engineering. IEEE, pp. 141–146.

Kleyko, D., Osipov, E., Papakonstantinou, N., Vyatkin, V., 2018. Hyperdimensional computing in industrial systems: The use-case of distributed fault isolation in a power plant. IEEE Access 6, 30766–30777.

Landauer, M., Wurzenberger, M., Skopik, F., Settanni, G., Filzmoser, P., 2018. Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection. Comput. Secur. 79, 94–116.

Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R., 2019. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 162–180.

Liu, B., Huang, J., 2018. D4: Fast concurrency debugging with parallel differential analysis. ACM SIGPLAN Not. 53 (4), 359–373.

Liu, G., Zhu, L., Wu, X., Wang, J., 2019. Time series clustering and physical implication for photovoltaic array systems with unknown working conditions. Sol. Energy 180, 401–411.

Liu, C., Zou, D., Luo, P., Zhu, B.B., Jin, H., 2018. A heuristic framework to detect concurrency vulnerabilities. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 529–541.

Lü, Z., Lü, Y., Yuan, M., Wang, Z., 2017. A heterogeneous large-scale parallel SCADA/DCS architecture in 5G OGCE. In: 2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics. CISP-BMEI, IEEE, pp. 1–7.

Lutov, A., Khayati, M., Cudré-Mauroux, P., 2019. Accuracy evaluation of overlapping and multi-resolution clustering algorithms on large datasets. In: 2019 IEEE International Conference on Big Data and Smart Computing. BigComp, IEEE, pp. 1–8.

Madicar, N., Sivaraks, H., Rodpongpun, S., Ratanamahatana, C.A., 2013. Parameter-free subsequences time series clustering with various-width clusters. In: 2013 5th International Conference on Knowledge and Smart Technology. KST, IEEE, pp. 150–155.

Maylawati, D., Aulawi, H., Ramdhani, M., 2018. The concept of sequential pattern mining for text. IOP Conf. Ser.: Mater. Sci. Eng. 434 (1), 012042.

Meango, T.J.-M., Ouali, M.-S., 2020. Failure interaction model based on extreme shock and Markov processes. Reliab. Eng. Syst. Saf. 197, 106827.

Millham, R., Agbehadji, I.E., Yang, H., 2021. Pattern mining algorithms. In: Bio-Inspired Algorithms for Data Streaming and Visualization, Big Data Management, and Fog Computing. Springer, pp. 67–80.

Muhammad, M., Safdar, G.A., 2018. Survey on existing authentication issues for cellular-assisted V2X communication. Veh. Commun. 12, 50–65.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 1–32.

Nakarmi, U., Rahnamay Naeini, M., Hossain, M.J., Hasnat, M.A., 2020. Interaction graphs for cascading failure analysis in power grids: A survey. Energies 13 (9), 2219.

Organization, W.H., et al., 2007. Mass Casualty Management Systems: Strategies and Guidelines for Building Health Sector Capacity. World Health Organization.

Park, S., Shin, Y.-j., Hyun, S., Bae, D.-H., 2020. Simva-SoS: Simulation-based verification and analysis for system-of-systems. In: 2020 IEEE 15th International Conference of System of Systems Engineering. SoSE, IEEE, pp. 575–580.

Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 199–209.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 609–620.

Petitdemange, F., Borne, I., Buisson, J., 2018. Modeling system of systems configurations. In: 2018 13th Annual Conference on System of Systems Engineering. SoSE, IEEE, pp. 392–399.

Rodpongpun, S., Niennattrakul, V., Ratanamahatana, C.A., 2012. Selective subsequence time series clustering. Knowl.-Based Syst. 35, 361–368.

Sauvanaud, C., Kaâniche, M., Kanoun, K., Lazri, K., Silvestre, G.D.S., 2018. Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. J. Syst. Softw. 139, 84–106.

Schmidt, T., Hauer, F., Pretschner, A., 2020. Automated anomaly detection in CPS log files. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 179–194.

Shi, Z., Xie, Y., Xue, W., Chen, Y., Fu, L., Xu, X., 2020. Smart factory in industry 4.0. Syst. Res. Behav. Sci. 37 (4), 607–617.

Soleimany, G., Abessi, M., 2019. A new similarity measure for time series data mining based on longest common subsequence. Am. J. Data Min. Knowl. Discov. 4 (1), 32.

Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 314–324.

Tarus, J.K., Niu, Z., Kalui, D., 2018. A hybrid recommender system for e-learning based on context awareness and sequential pattern mining. Soft Comput. 22 (8), 2449–2461.

Trencher, G., 2019. Towards the smart city 2.0: Empirical evidence of using smartness as a tool for tackling social challenges. Technol. Forecast. Soc. Change 142, 117–128.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2013. The DStar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.
Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
Wong, E., Wei, T., Qi, Y., Zhao, L., 2008. A crosstab-based statistical method for effective fault localization. In: 2008 1st International Conference on Software Testing, Verification, and Validation. IEEE, pp. 42–51.
Yu, X., Liu, J., Yang, Z.J., Liu, X., Yin, X., Yi, S., 2016. Bayesian network based program dependence graph for fault localization. In: 2016 IEEE International Symposium on Software Reliability Engineering Workshops. ISSREW, IEEE, pp. 181–188.
Zaki, M.J., 2001. SPADE: An efficient algorithm for mining frequent sequences. Mach. Learn. 42 (1), 31–60.
Zambrano, A., Zambrano, M., Ortiz, E., Calderón, X., Botto-Tobar, M., 2020. An intelligent transportation system: The Quito city case study. Int. J. Adv. Sci. Eng. Inform. Technol. 10 (2), 507–519.
Zhang, J., Xie, R., Ye, W., Zhang, Y., Zhang, S., 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In: Proceedings of the 28th International Conference on Program Comprehension. pp. 219–229.
Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z., et al., 2019. Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 807–817.
Zhong, H., Mei, H., 2020. Learning a graph-based classifier for fault localization. Sci. China Inf. Sci. 63 (6), 1–22.

Sangwon Hyun is a Ph.D. candidate in School of Computing at KAIST in Republic of Korea. He received his B.S. degree in Department of Computer Science at Hanyang University. His research interest includes software testing, automated software debugging, cyber–physical system, and system-of-systems engineering.

Jiyoung Song is a Postdoctoral Researcher in Electronics and Communications Research Institute in Republic of Korea. She received her Ph.D. degree in Computer Science from KAIST. Her research interest includes software analysis, software testing, software verification, and system of system engineering.

Eunkyoung Jee is a Research Associate Professor in School of Computing at KAIST in Republic of Korea. She was a Postdoctoral Researcher in the Computer and Information Science Department at the University of Pennsylvania. She received her B.S., M.S., and Ph.D. degrees in Computer Science from KAIST. Her research interest includes safety–critical software, software testing, formal verification, and safety analysis.

Doo-Hwan Bae is a Professor at the School of Computing, KAIST in Republic of Korea. He had served as the Head of the School of Computing, KAIST from 2012 to 2016. He received his BS degree in the Seoul National University, Korea in 1980 and his Ph.D. degree in Computer and Information Sciences in the University of Florida in 1992. Since 1995, he has been with the School of Computing (formerly Department of Computer Science until 2014), KAIST. His research interests include model-based software engineering, quality-driven software development, and modeling & verification of system of systems.