박 사 학 위 논 문
Ph.D. Dissertation

사이버 물리 시스템 오브 시스템즈의 협력 실패 분석을 위한 컨텍스트 마이닝 기반 오류 분석 기법

Context Mining-based Fault Analysis of Collaboration Failures in Cyber-Physical System-of-Systems

2023

현 상 원 (玄 相 原 Hyun, Sangwon)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

사이버 물리 시스템 오브 시스템즈의 협력 실패 분석을 위한 컨텍스트 마이닝 기반 오류 분석 기법

2023

현 상 원

한 국 과 학 기 술 원

전산학부

사이버 물리 시스템 오브 시스템즈의 협력 실패 분석을 위한 컨텍스트 마이닝 기반 오류 분석 기법

현 상 원

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2022년 11월 28일

심사위원장 배 두 환 (인)

심 사 위 원 고 인 영 (인)

심 사 위 원 유 신 (인)

심 사 위 원 민 상 윤 (인)

심 사 위 원 지 은 경 (인)

Context Mining-based Fault Analysis of Collaboration Failures in Cyber-Physical System-of-Systems

Sangwon Hyun

Advisor: Doo-Hwan Bae

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
November 28, 2022

Approved by

Doo-Hwan Bae
Professor, School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS
20197043

현상원. 사이버 물리 시스템 오브 시스템즈의 협력 실패 분석을 위한 컨텍스트 마이닝 기반 오류 분석 기법. 전산학부 . 2023년. 86+iv 쪽. 지도교수: 배두환. (영문 논문)
Sangwon Hyun. Context Mining-based Fault Analysis of Collaboration Failures in Cyber-Physical System-of-Systems. School of Computing . 2023. 86+iv pages. Advisor: Doo-Hwan Bae. (Text in English)

초 록

CPSoS는 여러 CPS들이 상호작용하여 단일 시스템으로는 달성할 수 없는 상위 목표를 달성하고자 하는 시스템이다. 이런 복잡한 상호작용 과정에서 CPSoS의 실패가 발생하며, 이런 실패는 소프트웨어적 환경적 상호작용의 간섭에 의해 발생된다. 기존 연구들은 패턴 마이닝 기법들을 활용하여 시스템 실패를 분석하였다. 하지만 기존 기법들은 협력 실패 분석에 있어서 (1) CPSoS logs를 위한 data model의 부재; (2) CPSoS logs의 연속적 분석을 위한 주요 기술적 특징에 대한 한계; (3) 단일 로그에 대한 멀티 패턴 추출 불가; (4) 패턴 분석 후 버그 분석까지 기법의 부재라는 한계점들을 가진다. 본 연구에서는 이러한 한계점들을 해결하고자 CPSoS logs를 위한 컨텍스트 모델을 정의하고, 해당 모델로부터 정확하게 실패 컨텍스트를 마이닝하기 위한 패턴 마이닝과 클러스터링 기법을 제안하였다. 마지막으로 추출한 실패 컨텍스트 패턴으로부터 코드상의 오류 위치를 추정하는 기법을 제안하였다. 여러 CPSoS를 대상으로 하는 실험에서 본 연구가 실패 컨텍스트 패턴 추출과 클러스터링에서 가장 높은 정확도를 보였다. 또한, 모든 실패 케이스에 대해서 가장 높은 오류 위치 추정 정확도를 달성하였다. 추가적으로, 실험 과정에서 기존에 발견되지 않았던 실패 사례와 버그들을 발견하였다. 본 연구의 향후 연구로 오류 위치 추정 기법의 두가지 확장에 대해 제시한다.

핵 심 낱 말 실패 컨텍스트 마이닝, 오류 위치 추정, 사이버 물리 시스템 오브 시스템즈, 퍼지 클러스터링

Abstract

A cyber-physical system-of-systems (CPSoS) 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 CPSoS. It is necessary to analyze the failure-inducing interactions (FII) 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 four limitations when applied to collaboration failures: (1) limited data model to handle discrete and continuous logs generated from CPSoS; (2) limited coverage of main features required to sequentially analyze the logs; (3) limitations on identifying multiple failure patterns in a single log; and (4) an absence of an end-to-end solution from pattern analysis to the fault identification. To overcome these limitations, we define a context model for CPSoS logs and propose an FII pattern mining algorithm covering the main features of the sequential analysis, an overlapping clustering technique for multiple pattern mining, and a pattern-based fault localization method. In experiments conducted on several CPSoS examples, we found that the proposed approach achieved the highest context mining accuracy and clustering precision. We also checked that the proposed localization method presented the highest fault localization efficacy. We newly detected undiscovered failure scenarios and bugs in this study. The findings of this study can facilitate the accurate analysis of collaboration failures.

Keywords Failure context mining, fault localization, cyber-physical system-of-systems, fuzzy clustering

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Systems are becoming more complex and massive according to the increasing requirements of society. This trend is prevalent in the social and business aspects of smart factories [2], smart cities [3], and intelligent platooning transportation systems [4]. These systems are called Cyber-physical system-of-systems (CPSoS), which are large and complex physical systems that interact with constituent systems (CSs) to achieve high-level goals beyond the single system's capability. A platooning, where independent vehicles from different vendors are driven in groups with close proximity, increases fuel efficiency and road capacity that a single vehicle cannot achieve [5]. The goal achievements of a platooning SoS are based on the interactions between CSs (i.e., vehicles). The interactions include the execution of platooning operations, such as two platoons `Merging` into a platoon and a vehicle `Leaving` from a platoon.

SoS collaboration is defined by the concrete interaction processes of CSs to achieve SoS-level goals [6, 7] and implicit logic of indirect interactions with different CSs [8]. In this study, interaction denotes direct and indirect information exchange in/around CPSoS and collaboration denotes a CS–CS interaction. In fact, VENTOS [5] and ENSEMBLE [9] provide platooning collaboration protocols that define interaction processes of vehicles (e.g., `Leave`, `Merge`). SIMVA-SoS [10] defines the collaboration processes of five different CSs (e.g., SoS manager, firefighters) for MCI-R SoS. SwarmLab [11] also provides the bio-inspired implicit collaboration algorithm and simulation of drone swarming based on MATLAB.

However, the design and development of the SoS collaboration 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 [12]. 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, and uncertainty of executions in various CS configurations. The uncertainty in the collaboration protocol may cause unexpected failures during execution.

Collaboration failures in SoS are defined as the failures to achieve SoS-level goals that are caused by unintended inter-functional interference during the intricate interactions [13]. Figure 1.1 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. Especially, failure occurrence context (i.e., failure context) should be provided to effectively analyze the details of CPSoS failures and resolve the root causes. To effectively analyze the root causes of failures in such intricate data, existing studies have focused on mining patterns for log anomaly detection [14, 15, 16, 17, 18, 19], time-series data analysis [20, 21, 22, 23], sequence data analysis [24, 25], and graph

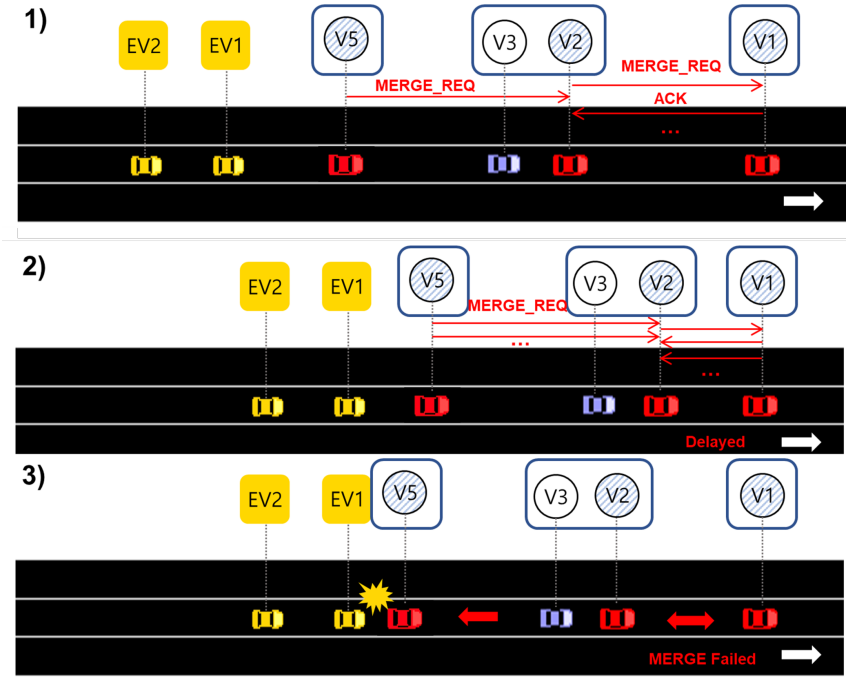


Figure 1.1: An example collision failure in platooning scenario

mining-based fault localization [26, 27, 28, 29, 30, 31, 32, 33, 34, 35] for various systems. Concentrating on the general fault analysis process, including fault detection, understanding on failure occurrence context generation, root cause localization, root cause identification, our investigation of the applicability of existing methods to CPSoS collaboration failures indicated that (1) their data models do not handle both of the discrete and continous data generated in CPSoS execution; (2) they do not cover the major features required to the sequential analysis of the discrete and continuous data; (3) have limitations in terms of identifying multiple failure patterns in a single log; and (4) do not provide an end-to-end solution from failure pattern analysis to the root cause identification.

First, discrete and continuous data are required to be analyzed in CPSoS collaboration failures. One of the primary features of CPS is the utilization of continuous data from diverse types of sensors, such as LIDAR [36], ultrasonic wave, wind, and temperature sensors. Moreover, CPSs communicate to execute collaborative operations for achieving SoS-level goals. This communication is mainly conducted through message-based data exchange. Both the discrete and continuous data generated from CPSoS execution provide a valuable understanding of failure occurrence context of collaboration failures. However, existing failure analysis techniques in various domains did not utilize a proper data model (i.e., data element) that has the capability of analyzing message-based communication and environmental states.

Second, existing approaches are limited to dealing with the features of sequential analysis of the CPSoS logs, such as serialization, temporality, multidimensionality, comprehensiveness, and series relationship, so that serious information loss occurs when mining failure patterns. For example, CPSoS communication logs consist of multidimensional communication messages involving heterogeneous data types (e.g., contents, time). However, 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 [22] and power plant sensor data [23]. 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 [37, 38]. 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.

Next, 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 [39, 40]. 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 code. Patterns are flagged logs that can effectively explain the occurrence of failures [16]. Developers need to localize the bugs in the collaboration code from the patterns to identify the root causes of 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 [16, 41]. 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. The proposed approach provides four main contributions to rectify the aforementioned issues in analyzing the root causes of CPSoS collaboration failures. First, we define an Interaction and Environment Model (IEM) to handle the discrete message logs and continuous sensor logs in CPSoS. Second, we proposed a Context-Aware Failure pattern-based Clustering Approach (CAFCA) in this study. CAFCA-Longest Common Subsequence ($CAFCA\text{-}LCS$) pattern mining algorithm that accurately extracts FII patterns by covering the main features of sequential analysis of the CPSoS logs. Next, the CAFCA contains a Fuzzy-based 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 an experiment in which we applied the proposed approach to a platooning SoS dataset, PLTBench [42], MCI-R SoS data, SIMVA-SoS, and DroneSwarming simulation, SwarmLab. The results obtained verified that our approach 1) generated the most accurate failure context patterns from the platooning interaction logs among existing pattern mining techniques; 2) exhibited significantly high overlapping clustering precision; and 3) achieved a 15% higher EXAM score on average compared with spectrum-based fault localization (SBFL) methods, which indicates the higher efficacy of the debugging cost reduction than SBFL methods. Moreover, we newly discovered a failure pattern and a bug that causes frequent collisions of drones in SwarmLab.

The remainder of this paper is organized as follows: Section 2 works related to this research; Section 3 explains the background; Section 4 elucidates the proposed approach; Section 5 describes the experimental dataset and simulator utilized in this study; Section 6 explains an experiment on the platooning SoS; and Section 7 and 8 recommends directions for future works and concludes the study.

Chapter 2. Related Work and Contributions of this Dissertation

2.1 Cyber-Physical System-of-Systems and Models

We have investigated existing studies concentrating on formal or semi-formal modeling of CPS [43, 44, 45, 46, 47, 48, 49, 50, 51], SoS [52, 53, 54, 55, 56, 57, 58, 59, 12, 60], and distributed systems [61, 62, 63, 64, 65, 66] to define the representation of CPSoS model in this study. We have found several common points of the existing modeling studies for CPS, SoS, and distributed systems. First, most of the existing studies utilized the graph-based representation of their models. In addition, this also indicates that all of the models contain connections (i.e., interactions) between system instances. Next, the studies described the attributes of capabilities and tasks in the models. Finally, most of the existing models assumed the heterogeneous component modules in a system. The distinguishing attributes and features of existing CPS models are the existence of continuous management attributes and timer attributes. As described in Section 1, handling continuous sensor data is essential in the execution of CPS. Hence, I/O interfaces and timer attributes are particularly depicted in CPS models. In distributed system models, multi-layered graph models, such as functional and service layers, are one of the distinctive features. The studies for distributed systems also intended to concentrate on the load-balancing issues of task allocation. In SoS models, the limited knowledge assumption on the constituent systems (CSs) is the distinguishing feature compared with CPS and distributed system models.

CPS Models. Drozdov et al. [43] formally defined a Syntax and Semantic of CPS model based on IEC61499 standard. The syntax is defined by a set of function blocks, I/Os, and their connections. The semantic defines the I/O examples by the function block types, followed by the description of priorities for executable components. Zhou et al. [44] defined a hybrid UML-based CPS model. Halba et al. [45] specified the spaces of IoT devices and capabilities and defined Relations between the IoT and capability space. Calvaresi et al. [46] focused on the in-time property of CPS timing errors; thus, defined a negotiation rule model including the initiator, contractor, task, and starting and finishing time. Zhao et al. [47] proposed a modeling method for CPS that enables the analysis of failure propagation in CPS based on the relevant relations and orders of input and output ports. Sun et al. [48] designed a five-layered timed automata model for CPS and defined interactions and functions according to each layer. Lee et al. [49] interpreted a CPS from the viewpoint of SoS. They defined discrete and continuous components of the CPS model and their relations with the coupling of the components. Chen et al. [50] represented a hybrid automaton model for CPS considering the uncertainty of internal actions in CPS. Bouheroum et al. [51] described control agent and bigraphical reactive systems formalism that combines agent and bigraphical reactive systems to deal with software, physical, and behavior levels in CPS. Existing CPS models have common features that they defined I/O interfaces and timer attributes to effectively handle the continuous data in CPS.

SoS Models. Luna et al. [52] defined a network graph model, consisting of a set of CSs in SoS and a set of links that depict the communication element between the CSs. Akhtar et al. [53] modeled a flood monitoring SoS based on the colored Petri-net (CP-Net). CP-Nets has a graphics language that has the advantage of easy understanding of formalism. Moreover, Rehman et al. [54] elaborated the CP-Net model for the flood monitoring SoS by defining the collaboration protocol and event models to

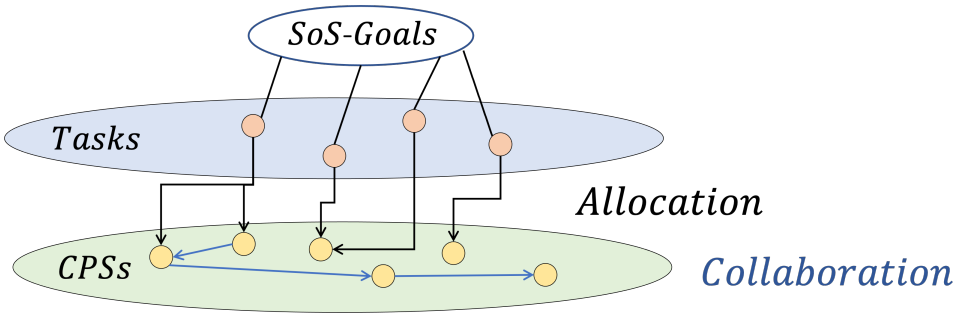


Figure 2.1: Visualized CPSoS model defined in this study

represent the integration of participating CSs. Fitzgerald et al. [55] applied a SysML-based modeling approach to SoS. They modeled interaction behaviors in SOS based on the communication sequential process formalism. Payne et al. [56] defined interface specification and contract modeling for CSs in major incident response SoS. Wiecher et al. [57] proposed a hierarchical graph-based representation of general SoS and CS structure, followed by defining an SoS scenario specification. Bryans et al. [58] applied SysML-based modeling to the travel agent problem for SoS architecture modeling. Zhou et al. [59] provided an SoS development context for design space modeling and implementation and integration patterns, such as one-directional information change, bi-directional information change, control, and negotiation for SoS architecture. Baek et al. [60] proposed a meta-model for general SoS and an example of MCI-R SoS applying the meta-model. In the existing SoS models, the limited knowledge assumption on the constituent systems (CSs) is commonly utilized.

Distributed System Models. Shchurov et al. [61] defined multi-layered graph models for functional, service, logical, and physical architecture layers. The multi-layered graph model is composed of a non-empty set of system components, a non-empty set of component-to-component connections, and a non-empty set of component-to-component inter-layer mapping for each layer. Srivastava et al. [62] proposed a timed automata-based modeling of distributed software-defined networks (SDN). The SDN contains the main activities of time passing, transactions, and synchronized transitions. The SDN model mainly focuses on analyzing load balancing and security issues of distributed systems. Kubiuk et al. [63] also utilized graph-based modeling to represent an efficient orchestrator for distributed systems. Bin et al. [64] proposed a graph-based reliability calculation method for distributed systems by the topological sorting algorithm. They calculated the reliability value for each component in distributed systems. Beschastnikh et al. [65] provided a graph-based search and global ordering algorithm for distributed systems. The global ordering algorithm chronologically orders the events conducted in different hosts. After the global ordering of the events, the graph-based search algorithm extracts specific patterns, such as a request-response pattern, and broadcast pattern. They applied the approach to the bully leader election and distributed two-phase commit transaction systems. Neves et al. [66] also proposed a graph pattern search algorithm and global ordering method based on log messages and kernel-level operations. The common points of the studies for distributed systems are (1) they commonly defined the model with hierarchical network models; and (2) their study concentrated on the analysis of load balancing in the execution of distributed systems.

CPSoS Model. Through the investigation of existing CPS, SoS, and distributed system models, we defined a general CPSoS model that contains common features of CPS and SoS and has distinguishing features from general distributed systems. Figure 2.1 described the abstracted structure of CPSoS consisting of *SoS-Goals*, *Tasks*, *CPSs*, *Allocation*, and *Collaboration*. *SoS-Goals* is a set of high-level

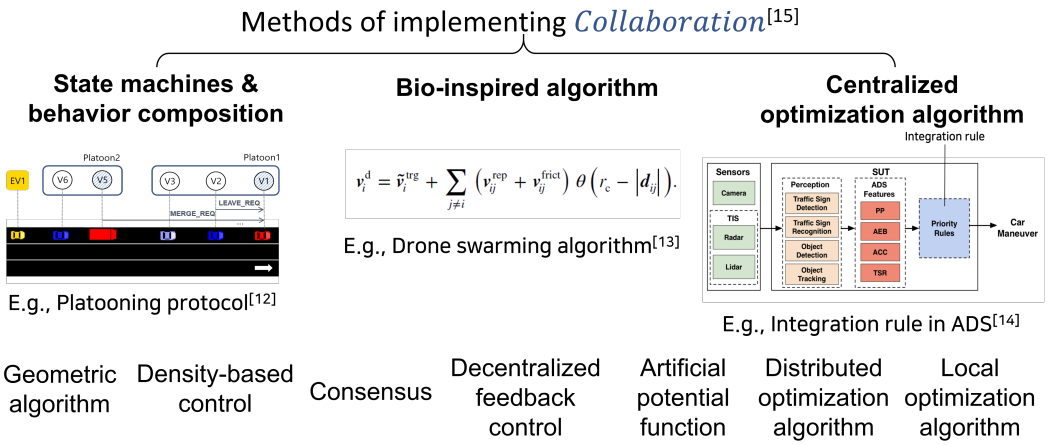


Figure 2.2: Categorization and examples of implementation of collaboration

goals that can be achieved by the CPSoS. *Tasks* is a set of defined tasks required to be accomplished to achieve or maintain the *SoS-Goals*. *CPSs* contain a set of constituent CPSs in CPSoS. A CPSoS and a single CPS are modelled as follows:

$$CPSoS = (SoS\text{-}Goals,\ Tasks,\ CPSs,\ Allocation,\ Collaboration)$$
$$CPS = (ComponentModels,\ Inputs,\ Outputs,\ Integration,\ Capabilities)$$

A CPS is composed of five attributes in this study. *ComponentModels* denotes software component modules in CPS, such as function blocks, services, or automata. *Inputs* indicates values from sensors or initial states of the components involving the data from other components while *Outputs* represents the actions of actuators and data to other components in CPS. *Integration* defines the connection of components, followed by the definition of transitions and priority rules in CPS. *Capabilities* literally indicates the attribute of feasible performance or behaviors conducted by the CPS. The *Capabilities* of each CPS is utilized in defining *Allocation* relation between *Tasks* and *CPSs* in a CPSoS model. *Allocation* defines the task allocation to a specific CPS or a set of CPSs that can effectively achieve particular tasks in CPSoS. *Collaboration* determines concrete approaches to accomplish the allocated tasks by a single CPS or a set of CPSs. Because our study is focusing on the failures of *Collaboration* during the execution of CPSoS, we will explain the details of *Collaboration* in the next subsection.

2.2 Implementation of Collaboration

We have investigated existing studies that implemented collective behaviors or collaboration of CPSs or IoTs [8, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78]. Particularly, Rossi et al. [8] classified existing existing implementation methods of collective behaviors of complex systems in 10 categories, such as state machines & behavior composition, bio-inspired algorithm, and centralized optimization algorithm depicted in Figure 2.2. This classification can be elaborated by two different dimensions: Interactive and Directive dimension; and Explicit interaction and Implicit interaction dimension. For example, centralized optimization algorithm is a directive implementation of collaboration where the example of the centralized algorithm is an integration rule of autonomous driving systems [70]. In the directive types of collective behaviors, there would be no autonomous interaction between each component, but exist only the order-based execution. Therefore, we focused on the interactive types of

collaboration which can be subdivided by Explicit and Implicit interactions. An explicit collaboration indicates the interaction process by the direct data exchange. For example, platooning collaboration protocol [5] based on the message-based communication is one of the examples of explicit collaboration. An implicit collaboration defined an autonomous rule for component systems, primarily utilizing indirect interactions through environmental objects. Representative examples of the implicit collaboration are drone swarming algorithms [67, 68, 69, 70, 71, 72, 73] and other bio-inspired algorithms for complex systems [74, 75, 76, 77, 78]. In this study, we utilized three experimental dataset for covering both of the explicit and implicit collaboration in the evaluation section. MCI-R and platooning SoS are examples of the explicit collaboration. Drone swarming algorithm is a type of implicit collaboration.

2.3 Taxonomy of Interaction Failures

We have investigated how the existing studies defined the interaction of systems in failure analysis and structured a taxonomy of interaction failures by this survey. We searched existing studies that focused on interaction failures in diverse domains. We defined the taxonomy of the interaction failures by the bottom-up approaches based on the example failure scenarios in the existing studies. The taxonomy of interaction failures are defined in Figure 2.3.

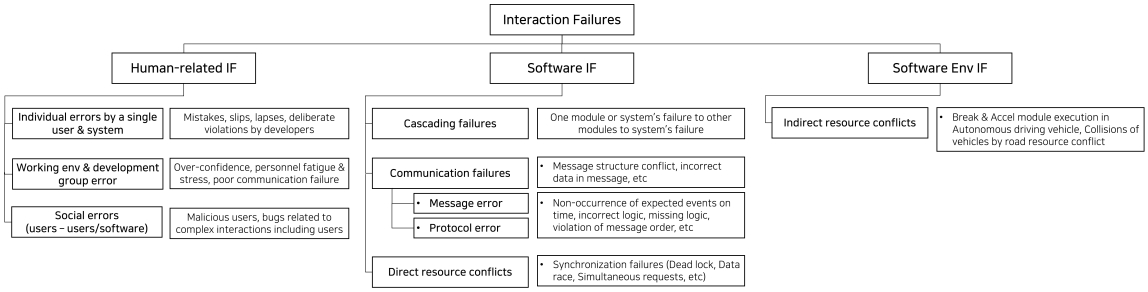


Figure 2.3: Taxonomy of interaction Failures

There exist three major types of interaction failures in the proposed taxonomy: human-related interaction failures, software interaction failures, and software-environment interaction failures. We have found the software-software-environment interaction failures in the empirical analysis of platooning SoS, but did not add the cases into the taxonomy, since the failures could be too domain-specific. We will add the failure classes into the taxonomy after extending the target domain to other CPSoS scenarios.

First, human-related interaction failures literally denotes the failures caused by the developers and users in the development and deployment phases [79]. The individual errors indicates mistakes, slips, lapses, and deliberate violations [79]. The working environment and development group errors are caused by the over-confidence, personnel fatigue and stress, poor task allocation, and human communication failures [80]. These errors are occurred in the development phase of software. The social errors are caused by the user-user interactions or multiple users-software interactions. This type of error includes malicious users, bugs relating to complex interactions between users and software [81].

Second, software interaction failures are caused by the interactions between software modules/subsystems to other modules/subsystems. Most of existing studies that focused on interaction failures targeted the cascading failures, which denote one module or systems' failure cause other modules or systems' failures [39, 13, 82, 40, 83, 84, 85, 86, 87]. In the studies, they defined the interactions between software to the propagation relationship of the failures. Communication failures can be classified into the communication message error and protocol error. Message error indicates the mess loss or damage, incorrect data

in a message, message structure conflict [79, 88]. The protocol error denotes the logic error, especially the violation of expected message orders, non-occurrence of expected events(messages) on expected time, incorrect logic, missing logic, wrong implementation [79]. The last type of software interaction failures are resource conflict failures. Direct resource conflict, a.k.a synchronization error, is caused by the simultaneous access to specific data. Data race, deadlock, atomicity violation, livelock, and starvation are included in this interaction failure type [79, 89, 90, 91, 92, 93, 94].

Finally, software-environment interaction failures are caused by the indirect influence by the environment or other software modules. We have found few examples in the software-environment interaction failures [95], which illustrate the interference of autonomous emergency breaking and adaptive cruise control modules in autonomous driving vehicles.

After building the taxonomy of interaction failures, we utilized the concrete examples and taxonomy when defining the context models of CPSoS failure analysis.

2.4 Graph Mining-based Fault Localization

Graph mining-based fault localization (GMFL) aims to extract the failure occurrence context (i.e., failure context) and suspicious locations of the root causes of the failures. GMFL techniques mainly focused on code/function-level failure context; thus, most of the techniques defined the context model based on the control flow graph (CFG) [26, 27, 28, 29, 30, 31]. Gaber et al. [26] proposed a CFG-based node and edge ranking method to calculate not only the suspiciousness of each statement, but also the suspiciousness of the transition between the statements. Parsa et al. [27] defined an weighted CFG-based spectrum-based fault localization (SBFL) to extract the discriminative CFG from the failed and passed test results. Zhong et al. [29] proposed a supervised convolutional neural network (CNN)-based fault location prediction technique. When building the CNN model that can predict the fault location from the failure data, they generated the CFG from java code and converted the CFG to Word2Vec format to train the CNN model. Henderson et al. [30] proposed a depth-first search(DFS)-based CFG mining technique to extract the critical sub-graphs according to suspicious fault behaviors. Chu et al. [31] applied the GMFL techniques on concurrent program. They defined interthread control flow path models that can link memory access patterns that occurred frequently in the failing executions to better diagnose the concurrency bugs. Major limitations of the CFG-based GMFL techniques on CPSoS failure analysis are (1) limited failure mode coverage to software-software interaction failures in CPSoS failure modes; (2) code/function-level context models are difficult to be applied in the black-box/gray-box constituent systems in CPSoS; (3) CFG-based GMFL techniques directly face the state/path explosion problem due to the increasing complexity of the CFG on CPSoS failure modeling. Furthermore, the last two studies [30, 31] did not provide the causality linking methods (i.e., suspiciousnes calculation methods) from context models to the code-level root causes.

A few studies [32, 33, 34, 35] defined their own function-level context model to effectively analyze the software failures. De et al. [32] defined the code hierarchy model and integration coverage model to extract the contextual information of failures. Particularly, the integration coverage model used the method call pairs to relate the caller and callee methods. Based on the models, they extracts the most suspicious methods and the code blocks that should be inspected at each reltaed method. Yu et al. [33] applied a bayesian network to fault localization. They defined the bayesian network-based program dependence graph to calculate the probability of each program entity in specific failure scenarios. Zhang et al. [34] proposed an abstract syntax tree model-based fault localization to link the bug reports to the

faults in source files. He et al. [35] formally defined the fault influence networks that mainly utilize the method call pair data. Based on the fault influence networks, they calculates the improved suspiciousness scores of each statement. These studies do not have the state/path explosion problem due to the model complexity, but they still have (1) limited failure mode coverage on CPSoS failures and (2) difficulties on applying limited knowledge characteristics of CPSoS. Additionally, the techniques mainly focusing on the association rule mining for code/function-level elements. Association rule mining provides only limited information to understand the failure context of software-software interactions, software-environment interactions, and software-software-environemnt interactions.

2.5 Time-series/Sequential Knowledge Mining

We have investigated several studies that focused on log anomaly detection and distinguished pattern extraction in various domains. We examined the applicability of the techniques, including their essential assumptions, for CPSoS failure analysis.

2.5.1 Log Anomaly Detection and Analysis

The majority of the log anomaly detection and analysis studies have applied clustering techniques. Landauer et al. [14] proposed a cluster evolving technique to effectively detect security attacks. Their technique parsed each line of logs for each time window, evolved clustering results of the prior time window, and applies forecasting methods to several types of attack scenarios. Schmidt et al. [15] also suggested a technique to detect critical time spans of security attacks in Cyber-Physical System logs. They integrated multiple time-series data analysis methods, such as Euclidean distance and Dynamic-Time Warping (DTW) with K-means clustering algorithm to detect suspicious time spans of security attacks. Wu et al. [96] applied an association mining technique to reversely engineer the software behaviors and their transitions based on the behavior logs. Finally, Amar et al. [16] introduced two log-flagging techniques: *LOGLINER*, and LogFaultFlagger. 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*. LogFaultFlagger added fault and bug counts on the *LOGLINER* by calculating the log line similarity using cosine similarity. These studies basically used text-based logs for anomaly detection and analysis. Nevertheless, they have commonly utilized conventional similarity metrics, such as Euclidean distance which can not be applied to the high-dimensional message sequences. Their studies basically did not consider any interactions during anomaly detection or analysis, and they did not consider the existence of multiple bugs or patterns in a single time window or log file. Lastly, their studies assumed highly-detailed logs that included details of behaviors, state-transitions, or failures. Witn an SoS, it is difficult to expect the well-ordered logs of CS-level behaviors or state-transitions, because the CSs are regarded as the black-box or gray-box systems. Thus, in our study, we fully utilized the observable execution traces (i.e., communication logs) in interaction failure analysis.

Other studies generated anomaly patterns from system sensor data by proposing clustering approaches. Liu et al [20] 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. Madicar et al. [97] suggested an approach that extracts faulty motifs among system sensor data. Motif refers to a commonly observed sequential data pattern. Rodpongpun et al. [98] proposed a search-based clustering approach

with clustering operations of addition, creation, and merging. The technique searched the optimized order of clustering operations on specific time-series data. Lastly, Soleimany et al. [21] proposed *LCS*-based clustering approaches that enabled the extraction of *LCS* from sensor time-series data. The major limitation of these studies is that they mostly used Euclidean distance-based similarity metrics which are not applicable to the nominal and high-dimensional communication logs in SoS. Additionally, they did not consider the overlapping clustering for extracting multiple faulty patterns. Furthermore, these studies did not consider the process of identifying root causes of failures after the pattern extraction.

Several studies have applied supervised approaches to detect anomalies from logs. Sauvanaud et al. [17] proposed a supervised anomaly detection technique, followed by the monitoring and data processing of cloud services. They applied several supervised learning approaches, such as Bayesian network and neural networks for anomaly detection. Du et al. [18] suggested an integrated anomaly detection technique comprising Long-Short Term Memory-based log-key anomaly detection model, workflow generator, and parameter-level anomaly detection model. The study firstly parsed each log by N-gram model, then applied the LSTM-based and parameter-level approaches to predict the probability of security attacks. Zhang et al. [19] proposed a supervised anomaly detection technique for unstructured log data, which contains previously unknown log sentence or noises in log. The study majorly focused on noise processing and unknown log sentence management during the anomaly detection process. The major limitation of these supervised approaches is that they assumed the well-defined fault knowledge of the target domain for training, such as concrete security attack patterns. The goal of our study was to reduce the overall analysis cost of interaction failures, wherein the final deliverable of the analysis would be the fault knowledge of the interaction failures. Therefore, the techniques requiring the fault knowledge are not appropriate for this study.

2.5.2 Multi-Variate Time-Series Pattern Mining

Pattern extraction studies can be classified into two categories based on the target data types: time-series and sequence data. The primary difference between the categories is that sequence data is nominal and unstructured, whereas time-series data consist of numerical values. Several studies extracted patterns from various time-series data for interpreting in-depth analysis on the data. Choong et al. [22] and Zhou et al. [99] respectively applied fuzzy k-means clustering to spatial vehicular trajectory data and household electricity consumption data, respectively. Their studies extracted meaningful patterns from both data and provided new implications for analysis on the data. Zhang et al. [100] proposed a hierarchical periodic pattern mining approach to effectively analyze real-world animal tracking data. Lastly, Kleyko et al. [23] applied hyperdimensional computing-based learning techniques to the fault analysis for power plant sensor data. The primary limitation pertaining to apply these pattern mining approaches in CPSoS is that all time-series data analysis studies used similarity metrics based on the Euclidean distance. Hence, the metrics cannot be applied to calculate similarities of nominal and high-dimensional interaction message sequences. Furthermore, all studies, except the study from Zhou et al. [99], did not provide overlapping clustering, which is necessary for analyzing multiple patterns from a single execution log.

Other time-series pattern analysis studies consider multiple variables together to investigate the failures of systems [101, 102, 103, 104, 105]. Hallac et al. [101] proposed a multi-variate time-series (MTS) clustering method for multi-sensor system such as autonomous vehicles. They segmented the multi-variate time-series data and clustered the data by calculating the independence of subsequences by the Markov random field. Sürmeli et al. [102] proposed a variable order markov models to cluster the

multi-variate time-series data. They mainly utilized the averaging and principal component analysis for dimension reduction and cluster the discretized data. Li [103] also utilized common principal component analysis to reduce the dimension of the data and generate the initial prototype clusters. Then, the technique searched the optimal cluster set by the reconstruction and error calculation processes. D'Urso et al. proposed two MTS techniques: observation-based clustering that analyzes the MTS data in unit-slices, variable-slices, and time-slices [106]; Dynamic time warping and partitioning around medoids-based clustering that have strengths on outlier effects by the trimming process [105]. These existing MTS studies have common points with CPSoS failure analysis in that they need to consider the dependency of values from multiple variables. However, MTS studies still have limitations on calculating the similarities of interaction message sequences and on the application of overlapping clustering for analyzing multiple failures occurred in a single log.

2.5.3 Sequence Data Analysis

SPADE is one of the commonly used sequence mining algorithms in various domains [25, 107, 108]. *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`.

We also examined a recent base study [24] that analyzed failure-inducing interaction patterns in platooning SoS. This highly relavant study first defined interaction models for SoS, which abstracted the message-based interaction data from communication logs. Then it suggested an *LCS*-based pattern mining approach that used the interaction models. The study extracted three *LCS* patterns of interaction failures and manually analyzed seven failure-inducing scenarios for platooning SoS. The study presented an initial idea for analyzing interaction failures in SoS, by applying *LCS*-based pattern extraction process. However, the study simply applied the existence of *LCS* as a similarity metric in pattern mining, but did not completely utilize the temporal features of interaction sequences. For instance, the study evaluated the identities of two messages without considering the delivery intervals between previous messages. Additionally, the base study did not take into account various time window sizes in *LCS* generation. The omission of time-related information in the pattern mining process adversely affected the quality of extracted patterns and increased the probability of false-alarm patterns. Furthermore, the base study was fully dependent on the manual fault identification process without providing any code-related information about the corresponding failures.

2.5.4 Concurrency Bug Analysis

Furthermore, we investigated concurrency bug analysis studies that have a common point with SoS failure analysis in resolving failures caused by the unintended interactions of system components (i.e., threads). Cai et al. [109] suggested a concurrency bug prediction technique by modeling branch events and checking the event feasibility. Li et al. [110] presented a thread-safety violation detector, TSVD in the testing phase by applying happens-before (HB) analysis. Liu et al. [111] proposed an automated concurrency bug detection technique based on the incremental control-flow graph (CFG) update. However, the extant studies are dependent on the CFG of target systems, which cannot be utilized in SoS analysis due to the state-explosion problem [10]. Additionally, existing studies focused on analyzing the shared memory conflict, but in SoS, we focused on analyzing specific sequences of

interactions between CSs. We concluded that the concurrency bug analysis studies have a different scope with SoS failure analysis.

2.6 Surveys on Existing Context Models & Similarity metrics

To define the appropriate context model for representing the CPSoS failures, we have investigated the existing context models in various system domains.

2.6.1 Interaction Data Model in System Analysis

The goal of the interaction data model analysis is to define the interactions in various systems and set the scope of interactions. First, we investigated how the interactions are defined in various domains and standards, such as CPS [95, 112, 113], agent-based systems, web-services [114], and telecommunication systems [115, 116].

In automotive domains, which is one of the representative examples of CPS, several studies defined interactions as "The activation of two or more features sending requests to the mechanical processes (physical actuators) that create contradictory physical forces." [95, 115, 113]. The studies abstracted the interactions of internal modules of autonomous vehicles based on the feature-driven development, where a feature denotes a collection of units of functionality [95]. They modelled the interactions as a vector of features, such as containing the feature elements in the system or environment. Especially, they classified the interactions in autonomous vehicles into two main features: by the conflict type and by the duration. First, by the conflict type, they classified the interactions into direct interactions and indirect interactions by the other system components or environments. Second, by the duration, they classified the interactions into immediate interactions and temporal interactions.

The studies focused on the interactions on telecommunication systems and web services also utilized the feature-based modeling of interactions. In telecommunication systems, they defined interactions as "a situation where a combination of these services behaves differently than expected from the single services' behaviors, is called service interaction" [116]. They also classified the emergence level of interactions into *logical level*, *abstract architecture*, and *concrete architecture*. Cameron et al. [115] also considered the user values and shared resources as features in the interaction representation. Juraez et al. [113] defined the interactions in web services as "a feature invokes or influences another feature directly or indirectly". They also characterized the key features of the interactions to goal of features, allocated and available resources, assumptions of features, interface, and concurrency.

Based on this analysis of the interaction data model in various system domains, we defined the interaction model for CPSoS failure analysis in Section 4.2.

2.6.2 Environment Data Model in System Analysis

To cover not only the software-software interaction failures, but also the software-environment and software-software-environment interaction failures, we have investigated the existing model definition of environment in system analysis process [117, 118, 119, 120, 121, 122]. In summary, the majority of the existing studies defined the environment surrounding the system to the vector (matrix) of sensor data in a system. Choong et al. [117] defined the environment of intelligent transportation system as a chronologically arranged vector of multiple sensor data. Harada et al. [118] defined the automatic aquarium management system to a high dimension vector which can deal with both discrete and continuous data.

Lee et al. [121] abstracted the environment of the ship engine system into a vector with 10 sensor dimensions. Serdio et al. [122] also modelled the environment of multi-sensor network system as a vector of sensors. Fontes et al. [119] abstracted the states of the gas turbine system as a matrix of sensor values. huang et al. [120] also generated environmental state matrix from vectors of multi sensor data from an independently operated data sources.

Based on the analysis of the existing environmental state representation models, we defined the environment model for CPSoS failure analysis in Section 4.2.

2.6.3 Similarity Calculation Methods

Principal Component Analysis Similarity metric (SPCA index)

The principal component analysis similarity metric (SPCA index) was utilized for many MTS mining cases [123, 124, 125, 119]. The SPCA index measures the similarity level between two matrices (MTS items) based on principal components (eigenvectors) of each matrix. The SPCA index is obtained from the square of the cosine values for all combinations of their first p principal components from two matrices [124]. p is determined by how many percentages of the data the principal components can represent. X and Y are the two matrices (MTS items). The SPCA index of X and Y is defined as $SPCA(X,Y) = \frac{1}{p}\sum_{i=1}^{p}\sum_{j=1}^{p} cos^2\theta_{ij}$ [126], where θ_{ij} is the angle between the ith principal component of X and the jth principal component of Y. The SPCA index can be applied when lengths of time series are inconsistent. Since the dimension can be reduced to number of principal components through principal component analysis (PCA), the SPCA index is suitable to high-dimensional data. However, it makes harder to interpret processed data after PCA.

Cosine similarity

Cosine similarity is commonly used in high dimensional positive data sets (such as text mining). It was also applied for the cases including fault identification in power systems [127] and knowledge extraction in energy consumption data [128]. Cosine similarity measures the similarity of two vectors by computing the cosine of the angle between the vectors. Given two vectors (time series), $X_i(i = 1, ..., n)$ and $Y_j(j = 1, ..., n)$, the cosine similarity of X and Y is defined as follows: $cos(\theta_{XY}) = \frac{X \cdot Y}{\|X\|\|Y\|} = \frac{\sum_{i=1}^{n} X_i \times Y_i}{\sqrt{\sum_{i=1}^{n}(X_i)^2} \times \sqrt{\sum_{i=1}^{n}(Y_i)^2}}$. θ_{XY} denotes the angle between the vectors X and Y, and n is the length of time series. The cosine similarity has a linear time complexity. However, the cosine similarity cannot deal with the cases that lengths of two time series are different. The cosine similarity is based on the directions of vectors, so the magnitudes of vectors are not considered.

Dynamic Time Warping (DTW)

Dynamic Time Warping (DTW) is one of the most common methods for similarity measure in time series clustering [129] and was adopted in [130, 131, 15]. DTW is trying to find the alignment, which is making each point of a time series correspond to another point of the other time series, to minimize the difference of two time series. Thus, DTW can deal with the temporal drift (drift in time dimension) and has better accuracy than Euclidean distance. The DTW distance is the minimum cost of the warping path to best match two time series. Given two time series, $X_i(i = 1, ..., m)$ and $Y_j(j = 1, ..., n)$, the DTW

distance of X and Y is defined as follows: $D(X_m, Y_n) = E(X_m, Y_n) + min \begin{cases} D(X_{m-1}, Y_{n-1}) \\ D(X_{m-1}, Y_n) \\ D(X_m, Y_{n-1}) \end{cases}$. m and n are the lengths of time series X and Y, respectively. The function $E(a, b)$ represents the Euclidean distance between two points a and b. The time complexity of DTW is $O(n^2)$. The DTW distance can be applied on time series of different length. In the case when the domain information is missing, the DTW distance measure can be a good candidate to use since the classification accuracy of DTW was experimentally shown to be comparable to those of LCSS, EDR and ERP [132].

Short Time Series (STS) distance

The short time series (STS) distance was proposed by Möller-Levet et al. [133] to address the problem that intervals between time points varied. The STS distance is a shape-based method and measures the difference of slopes between data points. The slopes are calculated by the relative change of amplitude and time difference between data points. A time series dataset is denoted as $X_j^i (i = 1, ..., m; j = 1, ..., n)$, where m is the number of time series in the data set, and n is the length of time series. t_k denotes the time that the kth data point was sampled in time series. The STS distance of two time series a and b is defined as $STS_{ab} = \sqrt{\sum_{k=1}^{n-1} (\frac{X_{k+1}^b - X_k^b}{t_{k+1} - t_k} - \frac{X_{k+1}^a - X_k^a}{t_{k+1} - t_k})^2}$. The STS distance have a time complexity of $O(n)$. The STS distance is suitable for the case when the sampling intervals are various, but it may be not suitable for long time series. Moreover, the STS distance is sensitive to scaling [129].

Maximum Shifting Correlation Distance (MSCD)

Two similarity measurements, Maximum Shifting Correlation Distance (MSCD) and the second distance of MSCD (MSCD-2nd), were proposed by Jiang et al. [134] aiming to deal with both the temporal and amplitude drifts. Since the correlation coefficient does not affect by the amplitude drift, this method finds the maximum correlation coefficient of two time series by temporally shifting time series. A time series data set is denoted as $X_{T_i}^i (i = 1, ..., m; T_i = \{t_1^i, ..., t_{n_i}^i\})$, where m is the number of time series in the data set, and n_i is the length of time series $X_{T_i}^i$. $T_i + s = \{t_1^i + s, ..., t_{n_i}^i + s\}$, where s is the shifting time constant. The MSCD of two time series a and b is defined as $MSCD_{ab} = 1 - \max_{s \in \{-k+u, ..., k-u, k\}} correlation(X_{T_i - s}^a, X_{T_i}^b)$. The correlation function represents the Pearson correlation coefficient distance. $k = \min\{[\frac{n_a}{10}], [\frac{n_b}{10}]\}$,where [] is a round function. if $k < 50$, u equals to 1, otherwise $u = [\frac{k}{50}]$. The MSCD distance matrix is denoted as $D = (d_{ab})_{m \times m} = (MSCD_{ab})_{m \times m}$. The MSCD-2nd of two time series a and b ($MSCD\text{-}2nd_{ab}$) is the Euclidean Distance (ED) between the ath and bth column vectors in the distance matrix D. The MSCD and MSCD-2nd both have a time complexity of $O(n)$. In the experiments, Jiang et al. evaluated the accuracies and efficiencies of the MSCD, MSCD-2nd, and 8 other similarity measures, including ED, ARMA [135], LCSS, DTW, fastDTW [136], SOM [137], GARCH [138], and GEV [106]. The DTW and the MSCD-2nd had much better clustering accuracies than other measures. Furthermore, the MSCD-2nd had the best clustering accuracy. In view of time consumption, the ED, MSCD and MSCD-2nd had the least time cost.

2.7 Experimental Framework for CPSoS: Platooning SoS

One of the important and difficult issues on CPSoS failure analysis is the lack of available open benchmark data wherein target systems should otherwise satisfy the characteristics of SoS. Furthermore, there exists no official standard for SoS examples (e.g., platooning SoS).

Several previous studies have provided tools for verifying the platooning system for specific goals, such as safety and resilience [139, 140]. Vieira et al [139] provided an integrated simulator consisting of the robot operating system (ROS) based simulator [141] and OMNET++ to test a communication network model of the platooning system. In the present work, the network frequency of a communication channel was used as test input to verify the maintenance of a generated platoon. Kamali et al. [140] focused on the spatial and timing constraints of the platooning system using an agent-based model with an integrated simulator of TORCS and MATLAB/Simulink [142]. However, they did not use the simulator to verify their model; instead, they used the UPPAAL model checker with a formal model. Both studies used a single platoon and did not consider the environment in the simulation and verification. Vieira et al. considered infrastructural settings in the system by changing the network frequency, but this work may not have sufficiently covered the uncertainty factors in the platooning system. In addition, their verification techniques were not compatible with verifying the platooning system in scalable situations.

In other previous studies [143, 144, 145, 146], formal models of the platooning system were designed and verified using existing tools, such as UPPAAL and MATLAB VnV Toolbox [147]. Elgharbawy et al. [143] verified the safety of an automated truck driving system by including unexpected environmental situations. They added stochastic properties to environmental sensing modules and exhaustively verified the system in several scenarios. However, their definition of a scenario was different from ours. They used a scenario to represent the parameter settings in a vehicle. We infer that they used a single vehicle to verify their system. Achrifi et al. [144] also focused on environmental uncertainty issues. They mainly described the advantages of the MATLAB VnV framework for verifying advanced driving assistant (ADAS) models. In this study, they also used a single ADAS model and did not include environmental factors in testing it. Mallozzi et al. [145] proposed a formal model for selecting platoon leaders in UPPAAL. They used diverse scenarios with different numbers of vehicles and platoon operations to verify the system in UPPAAL. Although they partially covered the internal uncertainty of the platooning system, they only used a single platoon and homogeneous vehicle types in the verification and didn't consider environmental factors. Meinke [146] verified a platooning system in scenarios in which the leaders' speed was changed. Their research focused on the scalability of the platooning system and simulated different numbers of vehicles in a platoon. This research only used a *Speedchange* event in the scenarios, thus they locally cover the internal uncertainties of the system.

To the best of our knowledge, no previous study has simultaneously used various numbers of platoons and vehicles with stochastic environmental objects in the verification. Previous studies only partially covered the uncertainties in the platooning system. Moreover, most of the previous studies used exhaustive verification techniques that could not bypass the state-explosion problem. Based on this investigation, we developed a platooning SoS simulation and evaluation framework that covers the internal and external uncertainties using various factors and applies the SPRT algorithm, which alleviates the state-explosion problem. The details are explained in Section 5.1.

2.8 Contributions of this Dissertation

In this dissertation, we aims to propose an approach to effectively analyze the CPSoS failures. To achieve the research goal, we investigated log anomaly detection, time-series data analysis, sequence data analysis [24, 25], and graph mining-based fault localization studies for various systems. Concentrating on the general fault analysis process, including fault detection, understanding on failure occurrence context generation, root cause localization, root cause identification, our investigation of the applicability of existing methods to CPSoS collaboration failures indicated that (1) their data models do not handle both of the discrete and continous data generated in CPSoS execution; (2) they do not cover the major features required to the sequential analysis of the discrete and continuous data; (3) have limitations in terms of identifying multiple failure patterns in a single log; and (4) do not provide an end-to-end solution from failure pattern analysis to the root cause identification.

To overcome these limitations, we propose a failure-inducing interaction (FII) pattern-based overlapping clustering and fault localization. The proposed approach provides four main contributions to rectify the aforementioned issues in analyzing the root causes of CPSoS collaboration failures. First, we define an Interaction and Environment Model (IEM) to handle the discrete message logs and continuous sensor logs in CPSoS. Second, we proposed a Context-Aware Failure pattern-based Clustering Approach (CAFCA) in this study. CAFCA-Longest Common Subsequence ($CAFCA\text{-}LCS$) pattern mining algorithm that accurately extracts FII patterns by covering the main features of sequential analysis of the CPSoS logs. Next, the CAFCA contains a Fuzzy-based 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.

We conducted an experiment in which we applied the proposed approach to a platooning SoS dataset, PLTBench [42], MCI-R SoS data, SIMVA-SoS [10], and DroneSwarming simulation, SwarmLab [11]. The results obtained verified that our approach 1) generated the most accurate failure context patterns from the platooning interaction logs among existing pattern mining techniques; 2) exhibited significantly high overlapping clustering precision; and 3) achieved a 15% higher EXAM score on average compared with spectrum-based fault localization (SBFL) methods, which indicates the higher efficacy of the debugging cost reduction than SBFL methods. Moreover, we newly discovered a failure pattern and a bug that causes frequent collisions of drones in SwarmLab.

Chapter 3. Background

3.1 Spectrum-based Fault Localization

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 [148]. 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 [148] 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 CPSoS, 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 [149, 150]. In CPSoS, understanding the failure occurrence context is the most important process to resolve the failures. Therefore, we focused on the graph mining-based fault localization techniques for CPSoS failure anlaysis.

3.2 Longest Common Subsequence Pattern Mining

Longest common subsequence (LCS) algorithm finds the longest subsequence that two strings have in common, which is often used to analyze gene sequence data in bioinformatics [151] and sensor data in system engineering [22, 21]. Let $\exists m, n \in \mathbb{N}_{\geq 0},\ x_m, y_n \in Char,\ s_m = x_1, x_2, x_3, ..., x_m$ and $s_n = y_1, y_2, y_3, ..., y_n$ 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:

$$LCS(s_m, s_n) = \begin{cases} \phi, & \text{if } m = 0 \text{ or } n = 0 \\ LCS(s_{m-1}, s_{n-1}) \oplus x_m, & \text{if } x_m = y_n \\ maxLenS(LCS(s_m, s_{n-1}), & \text{otherwise} \\ \quad LCS(s_{m-1}, s_n)) & \end{cases} \tag{3.1}$$

The function $maxLenS : String \times String \rightarrow String$ selects a 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 4.3.1 to be applied to SoS message sequences.

3.3 Fuzzy Clustering Algorithm

3.3.1 Fuzzy Clustering

Classical clustering methods are famous and widely used, but they have some limits since real life data sets are noisy, incomplete, and overlapping. To handle such uncertainty, many uncertainty based models have been proposed over the years. Among the models, the most popular fuzzy logic is introduced by Zadeh [152]. In contrast to classical hard sets where an element may be in a set or may not be in it, fuzzy set used membership of elements in it so that an element can belong to many clusters at same time.

3.3.2 Optimization Methods

Fuzzy C-means (FCM) algorithm [1] is inspired from the classical c-means algorithm. The membership vector which only contained 0 or 1 is replaced to membership matrix which contains a value between 0 and 1. Membership matrix U is n by k matrix, where n is number of the dataset and k is number of clusters. The final target is to minimize objective function $J_m(U, C) = \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ij}^m d_{ij}^2$, which is related to membership value u and distance d. Membership value is powered by m, which is a hyperparameter to choose fuzziness. This hyperparameter is also called fuzzifier. It gets fuzzier when m gets larger, and 2 is mostly used for m. Distance d is Euclidian distance between data and cluster center. The algorithm steps are as follows: 1) Initialize Membership matrix U randomly. 2) Update Cluster centers using $c_i = (\sum_{j=1}^{m} u_{ij}^m x_j)/(\sum_{j=1}^{m} u_{ij}^m)$. 3) Update Membership values using $u_{ij} = \sum_{l=1}^{k} (d_{ij}/d_{lj})^{2(m-1)}$. 4) Calculate Objective value. 5) Repeat 2, 3, 4 until Objective value is less than threshold. To sum up, FCM has strength that it is simple, and always converges. However, it needs long computational time and it is sensitive to initial guess and noise. Further, there is one more limitation at FCM. Because FCM uses Euclidian distance between data and cluster center, it was considered that all data features have equal importance. However, it is not true in many cases.

Xizhao Wang et al. [153] introduced Feature-Weight learning Fuzzy C-means (WFCM) algorithm to have an information about importance. Feature weight is defined as $w = w_1, w_2, \ldots, w_d$, $\forall w_i = [0, 1]$, so that it can express how much the feature is important. FCM is the case when all feature weight is 1. It also suggests the new measurement, called similarity measure $\rho_{ij}^{(w)} = \frac{1}{1+\beta * d_{ij}^{(w)}} \in [0, 1]$ to find appropriate feature weight. Here, β is normalizing factor to make average of $\rho_{ij}^{(1,1,\ldots,1)} (= \rho_{ij})$ is 0.5. WFCM considered feature weight as $d_{ij}^{(w)} = \sqrt{\sum_{k=1}^{n} w_k^2 (x_{ik} - x_{jk})^2}$ to calculate distance rather than Euclidian distance. Then, two data having phi more than 0.5 is considered similar, and less than 0.5 is considered different. Therefore WFCM tried to choose w to maximize similarity measure if phi is larger than 0.5, and minimize if smaller than 0.5. The detail algorithm is as follows: 1) Calculate FCM's similarity measure and normalizing factor β. 2) Calculate w to minimize cost function $E(w) = \sum_{i=1}^{k} \sum_{j=1}^{k} \rho_{ij}^{(w)} (1 - \rho_{ij}) + \rho_{ij} (1 - \rho_{ij}^{(w)})$ using gradient descent. 3) Follow FCM's algorithm using weighted distance. Overall, WFCM introduced feature selection techniques. It made better performance than FCM, but since it needs calculation for feature wight, it has larger time complexity.

To solve this large time complexity problem, Anter et al. [154] introduced Fast Fuzzy C-means (FFCM) algorithm. This algorithm decreased the number of distance calculation by eliminating membership values which are smaller than a hyperparameter threshold T. When T is 0.42, it showed it is saving more than 80% time regardless of number of clusters. However, it made bigger error when

number of clusters gets bigger. Additionally, research has also been conducted to apply fuzzy clustering algorithm to sequential data. The paper Fuzzy Clustering of Sequential Data introduced Kernel and Set Similarity Measure to find similarity of sequential data. This method does not only consider the content of sequential pattern, but also takes an account of the order of items. So they could assume (a, b, c) and (d, a, b) are more similar than (a, b, c) and (a, e, f, g, b), even though their length of longest subsequence is both 2.

In our study, we applied the fuzzy clustering to failure context mining process and defined the new fuzzy algorithm for CPSoS failures. The details are elucidated in Section 4.3.1.

Chapter 4. Proposed Approach

4.1 Overall Process

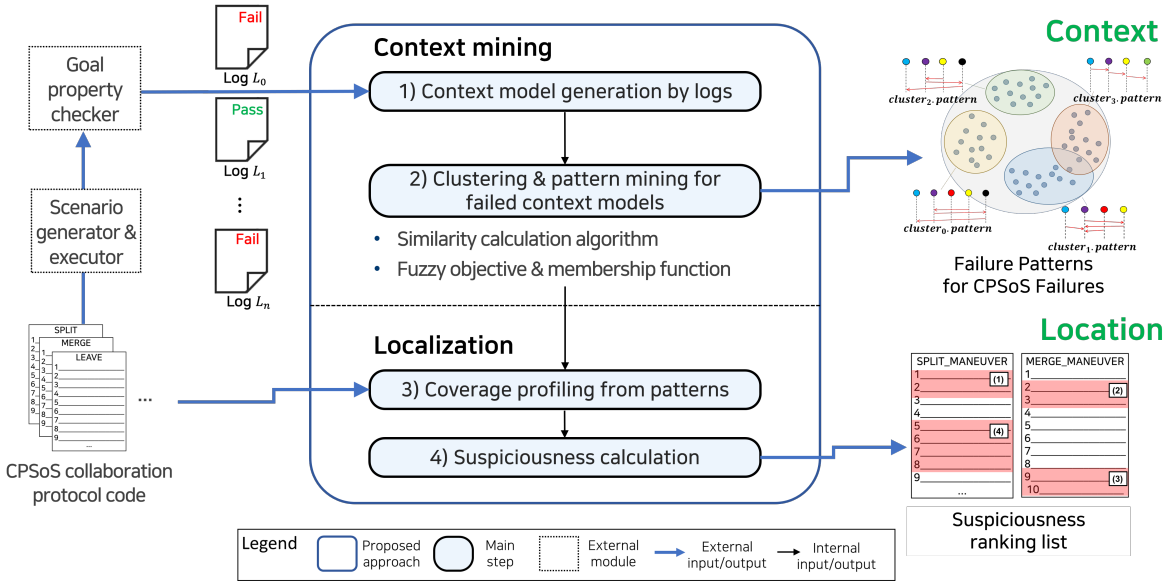


Figure 4.1: Overall Process of the Proposed Approaches

The overall process of the propose approach is described in Figure 4.1. The main inputs of the approach are the execution logs of the CPSoS simulation with *Pass/Fail* tags attached by the external goal property checker module, and the CPSoS collaboration protocol code. The proposed approach consists of two processes: context mining and localization. In the context mining process, there exist two main steps that (1) context model generation and (2) clustering & pattern mining. The output of the context mining process is a set of clusters of failed context models with the representative failure patterns for each cluster. Based on the generated patterns, the localization process calculates the suspiciosuness of each statement in the collaboration protocol code. Therefore, the propose approach returns the two main outputs: failure context patterns for CPSoS failures and suspiciousness ranking list for each failure context pattern. The details for each step in the process are elucidated in the following sections.

4.2 Context Model for CPSoS Failure Analysis

In order to analyze interaction failures in CPSoS, we define the interaction-environment model (*iem*) that captures the communication data and environmental states of the target CPSoS based on the execution logs as follows:

Let $t \in T$ be a time value, and for every vector v, $v[x]$ be the x-th element of the v,

$$Msg \ni msg = \langle t,\ continuity,\ synchronization,\ sender\_id,\ sender\_role,\ receiver\_id,\ receiver\_role,\ content \rangle, \tag{4.1}$$

$$M \ni m_n = (msg_i)_{i=1}^{n}, \text{where } n \in \mathbb{N},\ msg_x[0] < msg_y[0] \text{ for } 1 \leq x < y \leq n, \tag{4.2}$$

$$State \ni state = \langle t, envar_1(t), envar_2(t), envar_3(t), ..., envar_h(t) \rangle, \text{ where } envar_j(t) : \mathbb{R} \to \mathbb{R},\ h \in \mathbb{N}, \tag{4.3}$$

$$E \ni e_l = (state_k)_{k=1}^{l}, \text{where } l \in \mathbb{N},\ state_x[0] < state_y[0] \text{ for } 1 \leq x < y \leq l, \tag{4.4}$$

$$tag \in \{false, true\}, \tag{4.5}$$

$$Iem \ni iem = (m_n, e_l, tag) \tag{4.6}$$

$envar_j(t)$ maps input time t to the value of the observable sensor variable, $envar_j$ at the time t.

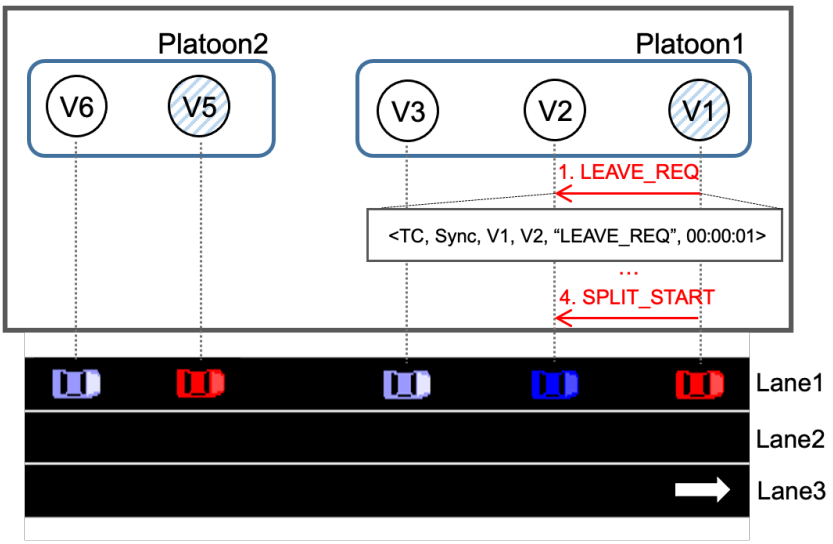


Figure 4.2: Example interaction model of Platooning SoS

4.2.1 Interaction Message Sequence Model

The interaction model (m) represents interaction logs as sequences of messages between CSs. The definitions of a message and a message sequence are as follows:

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

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, Figure 4.2 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 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.

As it is described in Section 2.6, 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 [95], telecommunication systems [116], and web services [115, 114].

Table 4.1 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 [5], 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* [95].

Table 4.1: 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

4.2.2 Environment State Model

The environment state model (e) is also defined by the survey on the existing environment state models in various system domains. As it is described in Section 2.6, the majority of the existing models

use the vector and matrix-based context model for representing environmental state. Therefore, in our approach, we extend the vector model to be compatible with the dynamic reconfiguration feature of CPSoS. In CPSoS, constituent CPSs can join into or leave from the CPSoS by its authority. This makes the dynamic reconfiguration of CPSoS structure and it causes the dynamically changing dimension of sensor variables in the environment state vectors for each time. For example, at time t_1, the vector dimension size is four, but at time t_2, the dimension of the CPSoS environment state vector could be three, four, and five. To cover the unique features of CPSoS, we define the environment state of the CPSoS as a chronologically ordered sequence of vectors that contain the sensor values for specific time t.

4.3 Context Mining

4.3.1 Failure Context Pattern Mining Algorithm

Since our context model (iem) consists of the two independent types of data sequence: m and e, we defined two pattern mining algorithm for each data sequence.

LCS-based Message Sequence Pattern Mining

With the generated M as input, the primary goal of this approach is to extract accurate CPSoS 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 process of pattern mining.

First, to deal with the multidimensionality, we extend the string-based LCS function in Equation (3.1) to the longest common message subsequence (LCMS) function based on the definition of message sequences m_n in Section 3.2. We define the function $LCMS : M \times M \to M$ which maps two input message sequences, p_k and q_n, to the longest common message sequence as follows:

$$LCMS(p_k, q_n) \triangleq \begin{cases} \phi, & \text{if } k = 0 \text{ or } n = 0 \\ LCMS(p_{k-1}, q_{n-1}) \oplus msg_k^p, & \\ & \text{if } MCT(msg_k^p, msg_n^q) \\ maxLenM(LCMS(p_k, q_{n-1}) & \\ \ , LCMS(p_{k-1}, q_n)) & \text{otherwise} \end{cases} \tag{4.7}$$

The function $maxLenM : M \times M \to M$ selects the longest message sequence among the two inputs. In Equation (3.1), 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 $MCT : Msg \times Msg \to \mathbb{B}$ maps two input messages to the Boolean value of the message identity

as follows:

$$MCT(msg_k^p, msg_n^q) \triangleq \begin{cases} (msg_k^p.sender = msg_n^q.sender) & pre_p = 0 \\ \wedge\ (msg_k^p.content = msg_n^q.content) & or \\ \wedge\ (msg_k^p.receiver = msg_n^q.receiver) & pre_q = 0 \\ \wedge\ (msg_k^p.continuity = msg_n^q.continuity) & \\ \wedge\ (msg_k^p.synchronization = msg_n^q.synchronization) & \\ & \\ (msg_k^p.sender = msg_n^q.sender) & \\ \wedge\ (msg_k^p.content = msg_n^q.content) & \\ \wedge\ (msg_k^p.receiver = msg_n^q.receiver) & \\ \wedge\ (msg_k^p.continuity = msg_n^q.continuity) & \text{otherwise} \\ \wedge\ (msg_k^p.synchronization = msg_n^q.synchronization) & \\ \wedge\ ((msg_k^p.time - msg_{pre_p}^p.time) & \\ -(msg_n^q.time - msg_{pre_q}^q.time) \leq delay\_threshold) & \end{cases} \tag{4.8}$$

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 Equations (4.7) and (4.8) 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 SoS failures. Because the LCS-based algorithms start from the "firstly matched" instance, LCS patterns may include completely meaningless parts prior to the critical points.

Figure 4.3 depicts two example LCS patterns extracted from the same message sequences with different time windows. Example pattern 1, starting from 53.03 sec, consists of repetitive `MERGE_REQ` messages and a few other messages. Otherwise, pattern 2, starting from 85.00 sec, contains various messages, such as `LEAVE_REQ`, `SPLIT_REQ`, and `MERGE_REQ`s between *V1*, *V2*, *V3*, and *V5*. 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 LCSs. 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

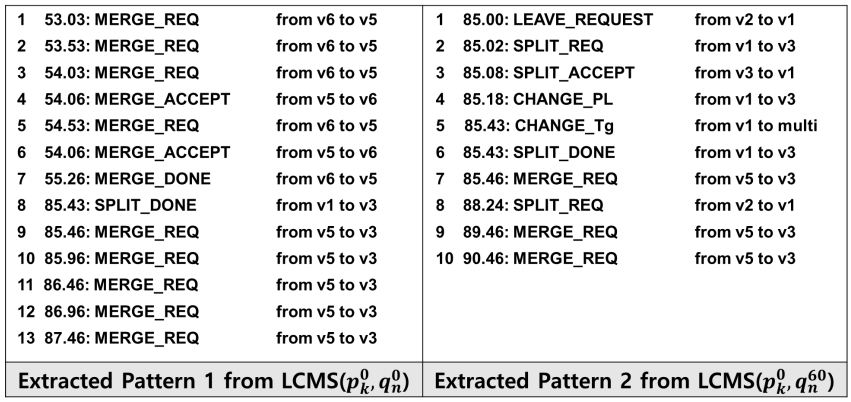


Figure 4.3: Example abstracted LCS patterns extracted from the same message sequences but different time windows

starting from t_1 and t_2 as follows:

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

To properly select the most "critical" LCS among the generated ones, we define the parameters needed to evaluate the quality of the LCSs as the number of content types in the LCS and their lengths. We assume that an LCS is more "critical" than other LCSs 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 SoS 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 [17, 18, 19]. However, our study focuses on the analysis of 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 LCSs with various types of interaction contents providing more contextual information.

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

$$T\text{-}LCS(p_k, q_n) \triangleq \bigcup_{t_1,t_2 \in T} LCMS(p_k^{t_1}, q_n^{t_2}) \tag{4.9}$$

$$TIME\text{-}LCS(p_k, q_n) \triangleq \underset{m \in T\text{-}LCS(p_k,q_n)}{argmax} NumContentTypes(m), \tag{4.10}$$

where function $NumContentTypes : M \to \mathbb{N}$ 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 LCSs 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 Equation (5). If the number of content types is the same, we use the lengths of LCSs as a tie-breaking rule. The algorithm 1 describes the details of executing the proposed *TIME-LCS* pattern mining algorithm. *MsgSim* in Equation (4.11) is defined as the similarity calculation metric for message sequence data.

$$MsgSim \triangleq \frac{len(TIME\text{-}LCS(cluster_j.pattern,\ m_i))}{len(cluster_j.pattern)} \tag{4.11}$$

Subjects of the *MsgSim* include the LCS pattern in an existing cluster, $cluster_j.pattern$, and a given

message sequence m_i. The *MsgSim* metric calculates the LCS-based sequence similarity between the m_i and $cluster_j.pattern$.

Algorithm 1: CA-LCS Algorithm for *iem* models

Input : $iem_i, iem_j \subset Iem$

Output: $iem \subset Iem$

$T \leftarrow \{t_0, t_1, ..., t_k\}$, a set of starting times of time window;
LCSTable $\leftarrow [][]$;
LCS_set, $common\_env\_state \leftarrow \emptyset$;
$prev\_msg_i, prev\_msg_j \leftarrow Null$;
// Extract LCS by each subseq of messages & env states by starting time
for u *in* T **do**
  for v *in* T **do**
    $m^u_{n_i} \leftarrow$ subsequence of m_{n_i} from time u;
    $m^v_{n_j} \leftarrow$ subsequence of m_{n_j} from time v;
    for $x \leftarrow 0$ *to* $len(m^u_{n_i})$ *by* 1 **do**
      for $y \leftarrow 0$ *to* $len(m^v_{n_j})$ *by* 1 **do**
        if $x == 0 || y == 0$ **then**
          LCSTable[x][y] = 0;
        end
        $msg\_identity = MessageIdentityChecking(m^u_{n_i}[x], m^v_{n_j}[y],$
        $prev\_msg_i, prev\_msg_j, delay\_threshold)$;
        $env\_sim, env\_state = EnvSimilarityCalculation(m^u_{n_i}[x], m^v_{n_j}[y],$
        $iem_i.e_{l_i}, iem_j.e_{l_j}, env\_time\_window\_threshold)$;
        if $msg\_identity == True$ *and* $env\_sim \geq env\_sim\_threshold$ **then**
          LCSTable[x][y] = LCSTable[x-1][y-1] +1;
          $prev\_msg_i = m^u_{n_i}[x]$;
          $prev\_msg_j = m^v_{n_j}[y]$;
          $common\_env\_state$.add(env_state)
        else
          LCSTable[x][y] = MAX(LCSTable[x][y-1], LCSTable[x-1][y]);
        end
      end
    end
    LCS_set.add(LCS_of_LCSTable) // by reverse-traversing LCSTable
  end
end
return $iem = (ARGMAX(\text{LCS\_set}, NumContentTypes()), common\_env\_state, False)$;

// Count the number of contents in an LCS
Function $NumContentTypes(m_n)$**:**
  types $\leftarrow \emptyset$;
  for $i \leftarrow 0$ *to* $len(m_n)$ *by* 1 **do**
    if *!types contain* m_n*[i][5]* // 5th elem: contents
    then
      types.add(m_n[i][5])
    end
  end
  return len(types);

Dynamic Cosine Similarity Calculation for CPSoS Environment State

The algorithm 2 describes the detailed execution of the proposed message identity checking function and environment state similarity calculation function. As it is described above, the main issue for environment state pattern mining and similarity calculation is that the dimension of target vectors is not fixed, but changed by the dynamic reconfiguration of CPSoS. Therefore, we decided to extend the existing cosine similarity algorithm for vector similarity calculation to the similarity calculation of vectors with different dimensions.

Figure 4.4 explains the detailed process of the proposed dynamic cosine similarity calculation method. The dynamic cosine similarity calculation algorithm comprise of three steps: (1) discretization of the sensor variable data; (2) sliding-based vector similarity calculation; (3) selection by the threshold values. Discretization indicates that the sensor values should be transformed to the discrete values based on the domain specific knowledge. For example, in Figure 4.4, the discretization step is to transform the distance sensor values to the discrete values of one to five, which denote the qualitative distance classification from “very close" to “very far". The classification standard for each sensor is decided by the platooning inter/intra-distance setting in the target domain simulator.

The second step, sliding-based vector similarity calculation, aims to apply the cosine similarity to different dimension of vectors and to select the biggest value among the sets of the possible similarity values. The algorithm bases on the shorter length vector and calculates the cosine similarity values between the vector with the shorter length and the sliced vector of the longer vector by the size of the shorter one. In Figure 4.4, the example vectors have length five and four, respectively. Then, the length four vector is utilized as the base vector and the cosine similarity of the base vector and the sliced longer vectors by the base size are calculated. In the example, there exist two slices in the long vector, orange and green-colored vectors. From the generated similarity values, the algorithm sets the max value to the similarity of the two vectors with different dimensions.

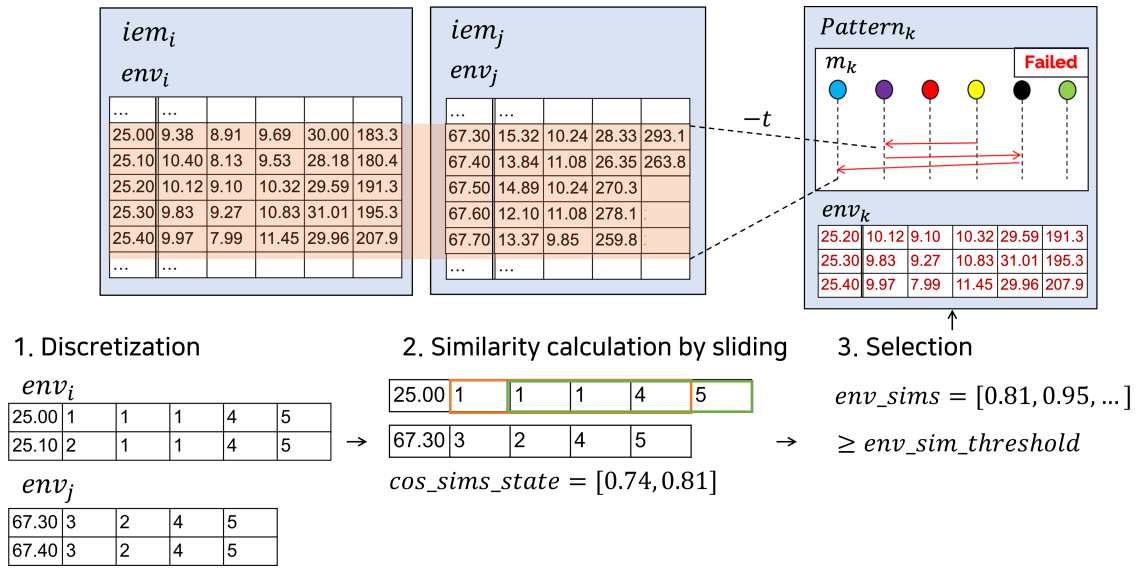


Figure 4.4: Dynamic cosine similarity calculation for CPSoS environment state

The last step is the selection of similar environment states based on the threshold values. As it is described above, an environmental state is defined as a sequence of sensor value vectors. The sliding-based vector similarity calculation is for calculating the similarity of the two environment sensor vectors; thus, we need to calculate the similarity of two environment states. Equation 4.12 defines the similarity

of the two environment states by the average value of similarity value of the constituent sensor vectors.

$$EnvSim(env_i, env_j) = AVG(Dynamic\_cosine\_sim(env_i, env_j)) \tag{4.12}$$

The environment pattern is selected by the *env_sim_threshold* value; thus, the environment states that exceed the *env_sim_threshold* value are added to the environment state in a pattern.

Algorithm 2: Message identity and environmental state similarity calculation function

```
// Message identity checking function
Function MessageIdentityChecking(msg_x, msg_y, prev_msg_i, prev_msg_j, delay_threshold):
    msg_identity ← Null;
    msg_identity = msg_x[2] == msg_y[2] ∧ msg_x[4] == msg_y[4] ∧ msg_x[5] == msg_y[5];
    // 2nd elem: sender_role, 4th elem: receiver_role, 5th elem: contents
    if prev_msg_i != Null and prev_msg_j != Null then
        msg_identity =
         msg_identity ∧ (ABS{(msg_x[0] − prev_msg_i[0]) − (msg_y[0] − prev_msg_j[0])} ≤
         delay_threshold);
    end
    return msg_identity;

// Environmental state similarity calculation function
Function EnvSimilarityCalculation(msg_x, msg_y, e_{l_i}, e_{l_j}, env_time_window_threshold):
    s ← env_time_window_threshold;
    env_sims, env_set ← ∅;
    surrounding_env_i = e_{l_i}.split(msg_x[0] − s, msg_x[0]);// Before s sec from msg_x
    surrounding_env_j = e_{l_j}.split(msg_y[0] − s, msg_y[0]);
    // 0th elem: message delivery time
    for state_i in surrounding_env_i do
        for state_j in surrounding_env_j do
            if len(state_i) < len(state_j) then
                for x ← 0 to len(state_j) − len(state_i) do
                    env_sims.add(cosine_similarity(state_i, state_j.split(x, x + len(state_i));
                end
                env_set.add(state_i);
            else
                for x ← 0 to len(state_i) − len(state_j) do
                    env_sims.add(cosine_similarity(state_j, state_i.split(x, x + len(state_j));
                end
                env_set.add(state_j);
            end
        end
    end
    return MAX(env_sims), env_set;
```

Discriminative Pattern Mining

Even though we proposed a fuzzy clustering approach to extract discriminative patterns from the failed execution logs in the following section, the clustering approach only concentrated on the failed logs except for the passed execution logs of CPSoS collaboration. In this section, we extended existing discriminative pattern mining algorithms [155, 156] to *iem* models for CPSoS to increase the accuracy of the pattern mining process by utilizing both of the passed and failed execution logs. The discriminative algorithm is also utilized as a similarity calculation metric in the fuzzy clustering process.

The primary implication of the discriminative pattern mining is to maximize the difference of occurrences in different item groups. We defined the discriminative pattern mining function, $Disc$, as follows:

$$Disc(Pattern_k, Iem) = \frac{Supp(Pattern_k, Iem_{passed})}{Supp(Pattern_k, Iem_{failed})},$$
$$Supp(Pattern_k, Iem) = \frac{Occ(Pattern_k, Iem)}{|Iem|},$$
$$Occ(Pattern_k, Iem) : \text{Occurrence of } Pattern_k \text{ in group } Iem,$$

where the Occ function calculates the number of occurrence of $Pattern_k$ in a passed or failed group of iem. The proposed functions are utilized as a selection metric for candidate patterns in the pattern update process in fuzzy clustering depicted in Figure 4.5.

4.3.2 Fuzzy Clustering for Failure Context Pattern Mining

Based on the similarity calculation and pattern mining algorithm, we proposed a fuzzy clustering for failure context pattern mining technique to classify different failure context and extract the representative patterns from each cluster. We applied the fuzzy clustering algorithm that has strengths on the overlapping clustering and of less influence by the random input orders. Figure 4.5 depicts the general execution process of the fuzzy clustering. A data element of the fuzzy clustering in this study is a single *iem* model. The first step of the fuzzy clustering is initialization. In the proposed fuzzy clustering, we selects C random *iem* models as initial patterns. Next, the approach calculates the similarity values between the patterns and all *iem* models. Based on the similarity values, membership values are calculated. Then, the *iem*s are clustered to each cluster by the membership values and the patterns are newly extracted for the updated groups of *iem*s. Finally, the algorithm checks the end-condition of the fuzzy clustering by using the objective function. If the end-condition is not satisfied, the steps from similarity calculation to end-condition checking are repeated.

The fuzzy clustering mainly consists of the similarity metric function and objective function. Based on the described similarity calculation and pattern mining algorithm above, we define the similarity function, Sim, and dissimilarity function,$Diss$, as follows:

Let C be the given number of clusters and N be the total number of *iem* elements,

$$Cluster = \{cluster_i | cluster_i \subset Iem, i \in \mathbb{N}_{\leq C}\} \tag{4.13}$$

$$Pattern = \{pattern_i | pattern_i \text{ is a LCS pattern contained in all } iem_j \in cluster_i,\ j \in \mathbb{N}_{\leq |cluster_i|}\} \tag{4.14}$$

$$Sim(iem_i, iem_j) = p * MsgSim(iem_i.m, iem_j.m) + q * EnvSim(iem_i.env, iem_j.env) \tag{4.15}$$

$$Diss(iem_i, iem_j) = 1 - Sim(iem_i, iem_j) \tag{4.16}$$

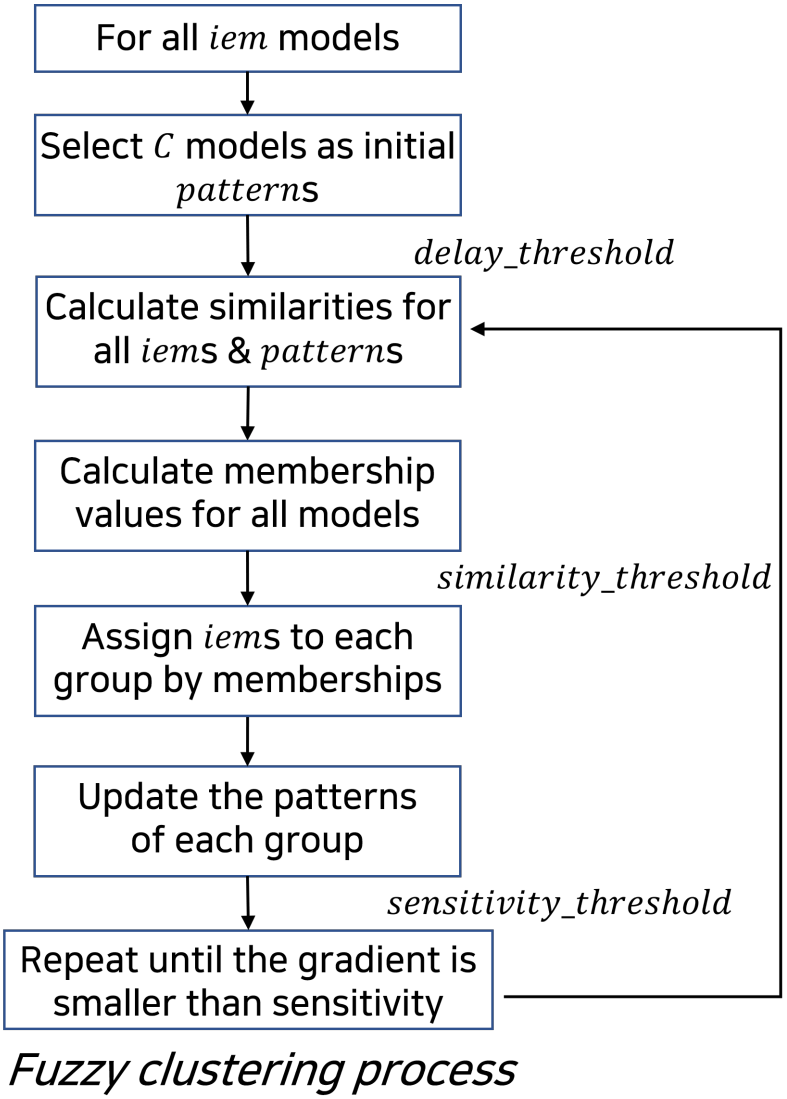


Figure 4.5: Execution process of fuzzy clustering [1]

The objective function, J, for the proposed fuzzy clustering is defined as follows:

$$J = \sum_{j=1}^{C}\sum_{k=1}^{N} u_{kj}^{p} Diss(pattern_j, iem_k)^2 + \frac{\sum_{j=1}^{C}\sum_{k=1}^{N}\sum_{l=1}^{N} u_{kj}^{p} u_{lj}^{p} Diss(iem_k, iem_l)^2}{\sum_{k=1}^{N} u_{kj}^{p}} \tag{4.17}$$

with following the constraints:

$$\sum_{j=1}^{C} u_{kj} = 1 \tag{4.18}$$

The former part of the function J is based on the objective function from fuzzy c-means (FCM) clustering [1]. The objective function evaluates the distance between patterns and data element, *iem*; thus, make the similar patterns and *iem*s closer. The latter part of the function J is introduced by the recent study focusing on the fuzzy clustering of sequential data [157]. The algorithm is newly attached to FCM objective function in this study. The latter part of J indicates the pair-wised analysis of the data element, which means that the algorithm makes the two similar *iem*s closer. This pair-wised similarity evaluation is necessary in this study, because the patterns are the subsequence of the *iem*s,the patterns already have baseline similarity to specific *iem*s; thus, this may result in the biased clustering results

and highly dependent clustering results by the initially selected patterns.

In the fuzzy clustering algorithm, the important point is that the algorithm should converge the cluster sets to the optimal set of clustering results. Therefore, we proved the following theorem that membership function, u, converges the objective function J to (local) optimal value by the Lagrangian function.

Theorem 4.3.1. *The optimal solution of Equations (4.17) and (4.18) is:*

$$u_{kj} = \frac{D(k,j)^{\frac{1}{1-p}}}{\sum_{i=1}^{C} D(k,i)^{\frac{1}{1-p}}}, \text{ where} \tag{4.19}$$

$$D(k,j) = Diss(pattern_j, iem_k)^2 + \frac{\sum_{h=1,h\neq k}^{N} u_{hj}^p Diss(iem_h, iem_k)^2}{\sum_{h=1,h\neq k}^{N} u_{hj}^p} \tag{4.20}$$

Proof. The Lagrangian function with respect to $U \ni u$ with Lagrangian multiplier, λ_k, is

$$L(u) = \sum_{j=1}^{C}\sum_{k=1}^{N} u_{kj}^p Diss(pattern_j, iem_k)^2 + \frac{\sum_{j=1}^{C}\sum_{k=1}^{N}\sum_{l=1}^{N} u_{kj}^p u_{lj}^p Diss(iem_k, iem_l)^2}{\sum_{k=1}^{N} u_{kj}^p} - \sum_{k=1}^{N}\lambda_k(\sum_{j=1}^{C} u_{kj} - 1) \tag{4.21}$$

The necessary condition for a minimum is that the partial derivatives of the Lagrangian function with regard to U vanish. Therefore $\frac{\partial L(u)}{\partial u_{ab}} = 0$, we have:

$$\frac{\partial L(u)}{\partial u_{ab}} = \frac{\partial}{\partial u_{ab}} f_1 + \frac{\partial}{\partial u_{ab}} f_2 - \frac{\partial}{\partial u_{ab}} f_3 = 0, \text{ where} \tag{4.22}$$

$$f_1 = \sum_{j=1}^{C}\sum_{k=1}^{N} u_{kj}^p Diss(pattern_j, iem_k)^2 \tag{4.23}$$

$$f_2 = \frac{\sum_{j=1}^{C}\sum_{k=1}^{N}\sum_{l=1}^{N} u_{kj}^p u_{lj}^p Diss(iem_k, iem_l)^2}{\sum_{k=1}^{N} u_{kj}^p} \tag{4.24}$$

$$f_3 = \sum_{k=1}^{N}\lambda_k(\sum_{j=1}^{C} u_{kj} - 1) \tag{4.25}$$

$$\frac{\partial}{\partial u_{ab}} f_1 = \frac{\partial}{\partial u_{ab}} \sum_{j=1}^{C}\sum_{k=1}^{N} u_{kj}^p Diss(pattern_j, iem_k)^2 = p\, u_{ab}^{p-1} Diss(pattern_b, iem_a)^2 \tag{4.26}$$

$$\frac{\partial}{\partial u_{ab}} f_2 = \frac{\partial}{\partial u_{ab}} \frac{\sum_{j=1}^{C}\sum_{k=1}^{N}\sum_{l=1}^{N} u_{kj}^p u_{lj}^p Diss(iem_k, iem_l)^2}{\sum_{k=1}^{N} u_{kj}^p} \tag{4.27}$$

$= 0$ if $j \neq b$

$+\frac{\partial}{\partial u_{ab}} \frac{u_{ab}^p u_{ab}^p Diss(iem_a, iem_a)^2}{u_{ab}^p} = 0$ ($\because Diss(iem_a, iem_a) = 0$) if $k = a, l = a, j = b$

$+\frac{\partial}{\partial u_{ab}} \frac{u_{ab}^p \sum_{l=1,l\neq a}^{N} u_{lb}^p Diss(iem_a, iem_l)^2}{u_{ab}^p} = 0$ if $k = a, l \neq a, j = b$

$+\frac{\partial}{\partial u_{ab}} \frac{u_{ab}^p \sum_{k=1,k\neq a}^{N} u_{kb}^p Diss(iem_k, iem_a)^2}{\sum_{k=1,k\neq a}^{N} u_{kb}^p} = p\, u_{ab}^{p-1} \frac{\sum_{k=1,k\neq a}^{N} u_{kb}^p Diss(iem_k, iem_a)^2}{\sum_{k=1,k\neq a}^{N} u_{kb}^p}$ if $k \neq a, l = a, j = b$

$+0$ if $k \neq a, l = a, j = b$

$= p\, u_{ab}^{p-1} \frac{\sum_{k=1,k\neq a}^{N} u_{kb}^p Diss(iem_k, iem_a)^2}{\sum_{k=1,k\neq a}^{N} u_{kb}^p}$

$$\frac{\partial}{\partial u_{ab}} f_3 = \frac{\partial}{\partial u_{ab}} \sum_{k=1}^{N} \lambda_k (\sum_{j=1}^{C} u_{kj} - 1) = \lambda_a \tag{4.28}$$

From Equations (4.26) to (4.28),

$$\frac{\partial L(u)}{\partial u_{ab}} = p\, u_{ab}^{p-1} \Big(Diss(pattern_b, iem_a)^2 + \frac{\sum_{k=1,k\neq a}^{N} u_{kb}^p Diss(iem_k, iem_a)^2}{\sum_{k=1,k\neq a}^{N} u_{kb}^p} \Big) - \lambda_a = 0 \tag{4.29}$$

$$u_{ab} = \left(\frac{\lambda_a}{p} \Big(Diss(pattern_b, iem_a)^2 + \frac{\sum_{k=1,k\neq a}^{N} u_{kb}^p Diss(iem_k, iem_a)^2}{\sum_{k=1,k\neq a}^{N} u_{kb}^p} \Big)^{-1} \right)^{\frac{1}{p-1}} \tag{4.30}$$

From Equations (4.18) and (4.30),

$$\sum_{j=1}^{C} u_{kj} = \sum_{j=1}^{C} \left(\frac{\lambda_k}{p} \Big(Diss(pattern_j, iem_k)^2 + \frac{\sum_{h=1,h\neq k}^{N} u_{hj}^p Diss(iem_h, iem_k)^2}{\sum_{h=1,h\neq a}^{N} u_{hj}^p} \Big)^{-1} \right)^{\frac{1}{p-1}} = 1 \tag{4.31}$$

$$\text{Let. } D(k,j) = Diss(pattern_j, iem_k)^2 + \frac{\sum_{h=1,h\neq k}^{N} u_{hj}^p Diss(iem_h, iem_k)^2}{\sum_{h=1,h\neq k}^{N} u_{hj}^p} \tag{4.32}$$

From Equations (4.31) and (4.32),

$$\sum_{j=1}^{C} \Big(\frac{\lambda_k}{p} \frac{1}{D(k,j)} \Big)^{\frac{1}{p-1}} = 1 \tag{4.33}$$

$$\lambda_k^{\frac{1}{p-1}} \sum_{j=1}^{C} \Big(\frac{1}{p \cdot D(k,j)} \Big)^{\frac{1}{p-1}} = 1 \tag{4.34}$$

$$\lambda_k^{\frac{-1}{p-1}} = \sum_{j=1}^{C} \Big(\frac{1}{p \cdot D(k,j)} \Big)^{\frac{1}{p-1}} \tag{4.35}$$

$$\lambda_k = \left(\sum_{j=1}^{C} \Big(p \cdot D(k,j) \Big)^{-\frac{1}{p-1}} \right)^{1-p} = \left(\sum_{j=1}^{C} \Big(p \cdot D(k,j) \Big)^{\frac{1}{1-p}} \right)^{1-p} \tag{4.36}$$

From Equations (4.30), (4.32), and (4.36),

$$u_{kj} = \left(\frac{1}{p} \cdot \left(\sum_{i=1}^{C} \Big(p \cdot D(k,i) \Big)^{\frac{1}{1-p}} \right)^{1-p} \cdot D(k,j)^{-1} \right)^{\frac{1}{p-1}} \tag{4.37}$$

$$= \left(\frac{1}{p} \cdot (p^{\frac{1}{1-p}})^{1-p} \cdot \Big(\sum_{i=1}^{C} D(k,i)^{\frac{1}{1-p}} \Big)^{1-p} \cdot D(k,j)^{-1} \right)^{\frac{1}{p-1}} \tag{4.38}$$

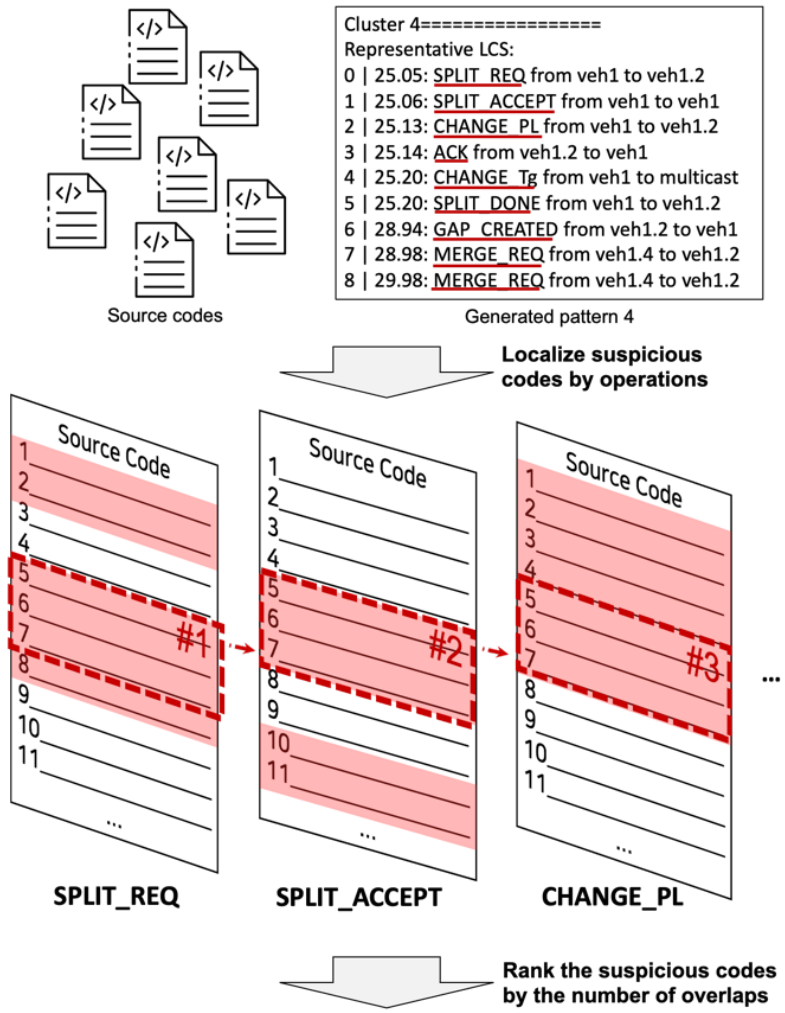
$$= \frac{D(k,j)^{\frac{1}{1-p}}}{\sum_{i=1}^{C} D(k,i)^{\frac{1}{1-p}}} \tag{4.39}$$

□

4.4 Pattern-based Suspiciousness Calculation

Based on the generated patterns, we propose a suspicious code localization technique that can reduce the significant cost needed to localize the root causes of failures. A pattern contains a sequence of communication messages involving CS-level operations as their contents. 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*. Based on the code coverage set, we localize codes that are suspicious to cause SoS failures.



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

Figure 4.6: Example of pattern-based fault localization

By using the code coverage set of each CS-level operation, our approach ranks the 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 [158, 159, 160, 161, 148]. 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 Figure 4.6, 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 Figure 4.6. 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.

The proposed approach finally returns two outputs: identified interaction failure patterns and ranked suspicious code blocks for the patterns. From the outputs, SoS managers can obtain two benefits in fault identification. First, the managers can establish the basic understanding of the failures from the patterns. For example, from pattern 4 in Fig. 4.6, we can infer that the failure is related to the concurrent execution of `Split` and `Merge`. Second, SoS managers can focus on the ranked suspicious lines of codes corresponding to the faulty interaction patterns, thus effectively reducing the cost required to identify the root causes of interaction failures in SoS. Besides, we implemented and opened the automated phases of our approach in our Github repository[1].

We expect that by using the manual fault identification process based on the fault patterns and suspicious code blocks, detailed fault knowledge can be established. The fault knowledge may contain the root causes, failure occurrence contexts with preconditions, frequencies, and severities of the corresponding interaction failures [162]. The fault knowledge can be utilized in various ways, such as run-time monitoring/detection, or prediction of the analyzed failures, and oracle for fault injection. We expect that the analyzed fault knowledge of the CPSoS can be utilized as general failure scenarios or patterns for platooning protocol testing. Further, it can support run-time monitoring or detection for CPSoS.

[1] Unable to describe due to the double blind policy

Chapter 5. Experimental Dataset

5.1 Verification Framework for Platooning SoS

5.1.1 Statistical Verification Framework of Platooning SoS: StarPlateS

In this section, we explain the main features of this framework. Fig. 5.1 shows the architecture of the framework which is composed of the scenario generation module, simulation module, and verification module. The proposed framework, StarPlateS operates as follows. First, the scenario module generates random configurations and scenarios of the platooning SoS using condition-based approach. Then, the simulation module executes the system on the generated scenarios with stochastic environment. Finally, the verification module applies a statistical model checking technique, especially the SPRT, and returns the verification results for each configuration and scenario. The following parts detail each component of the framework in sequence.

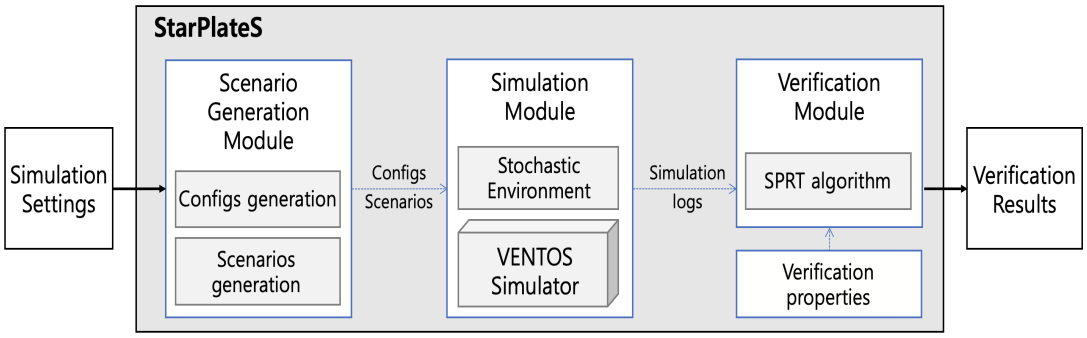


Figure 5.1: Overall architecture of the verification framework.

Simulation settings

There are several options for executing a simulation: simulation time, repetition numbers, verification options, duration between events, and GUI. We define these options as simulation settings. There are two modes of simulation settings according to the use of the StarPlateS framework. First, users can use our framework in 'verification mode'. In this mode, users set high repetition numbers, such as 1,000, and the verification option that is true for statistical verification. Another way of finding configurations and scenarios with specific purpose is called the 'single simulation mode'. In this mode, users assign the repetition number as 1 and set the verification option at false; thus, the framework generates as many scenarios as possible in the given amount of time. For example, we use the single simulation mode to compare two scenario generation approaches and the verification mode to conduct statistical model checking. The simulation settings are utilized in the scenario generation module and simulation module.

Scenario generation module

In the scenario generation module, the main goal is to generate diverse scenarios and configurations to address the internal uncertainty of the platooning system. Prior to explaining the module, we define *a configuration* as *a set of platoon generation features* and *a scenario* as *a sequence of operations on the created vehicles*. This module first generates random platoon configurations, and then generates scenarios based on each configuration.

When generating configurations of the platoons, We assign random values to every parameter of the platooning configuration. For example, in the configuration part of Fig. 5.2, the first generated platoon consists of "6" homogeneous cars with id "veh1" created in the "0" lane of "route1" at position "100". Its optimal size is "4" and its maximum size is "10". The second configuration shows the "4" sizes of the platoon with id "veh2" created in the "1" lane of "route1". Its optimal size is "4", and its maximum size is "8". The pltMgnProt option is used to check whether the vehicles use platooning management protocols or not. In this work, we assume that the platoons always use management protocols; thus, the value of pltMgnProt is always "true". We also add heterogeneous types of vehicles to generate configurations. For example, Fig. 5.3 shows a heterogeneous platoon that consists of a truck leader (V5) and three following passenger vehicles (V6, V7, and V8).

Configuration

```
<vehicle_platoon id="veh1" type="TypeCACC1" size="6" route="route1"
departPos="100" departLane="0" platoonMaxSpeed="0" pltMgmtProt="true"
 optSize="4" maxSize="10"/>
<vehicle_platoon id="veh2" type="TypeCACC1" size="4" route="route1"
departPos="100" departLane="2" platoonMaxSpeed="0" pltMgmtProt="true"
 optSize="3" maxSize="8"/>
```

Scenario

```
<speed id="veh1" begin="5" value="20" />
<speed id="veh2" begin="5" value="30" />
<pltSplit pltId="veh1" splitIndex="3" begin="25" />
<optSize begin="45" value="5" />
<pltLeave pltId="veh2" leaveIndex="1" begin="65" />
<pltMerge pltId="veh1.3" begin="85" />
…
```

Figure 5.2: An example of platoon configuration and scenario.

Next, to generate various scenarios, this module uses a condition-based approach to generate the scenarios for each configuration. A key point of the proposed approach is the pre-simulation of the scenarios to prevent the generation of meaningless and invalid scenarios by using the conditions and actions of each event. For example, if the module selects events in a purely random way, there could be meaningless scenarios, such as executing a *Split* operation on a platoon with size 1, and invalid scenarios, such as executing a *Leave* operation to a vehicle which has already left the platoon.

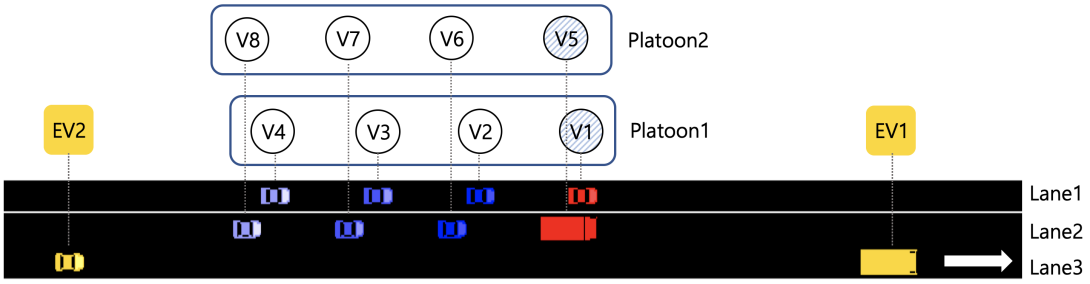


Figure 5.3: Generated platoons and Humman-Driven Vehicles (HDVs) in StarPlateS.

To alleviate this problem, we first define the available events and their conditions and actions in the platooning system. In the framework, vehicle management, platoon management, and policy management events as well as their conditions and actions are utilized. The *Speedchange* operation changes the speed of the leader of the platoon, and the *Optimalsizechange* operation changes the optimal sizes of all platoons. This operation causes *Split* events in the platoons that are larger than the assigned optimal size.

Considering the condition and action of each event, this algorithm generates a status set in each step of the scenario generation. For example, according to the configuration of the two platoons shown in Fig. 5.2, the initial status set is as follows: {“veh1:6", “veh2:4"}, which describes the platoon ids and their sizes in the initial state. Next, the proposed approach randomly selects an available event by comparing the current status set with the condition of each event. In the example, all but *Merge* operations are available, because two platoons in the same lane don’t exist. Then, the algorithm randomly selects the *Split* event among the available events as shown in the Fig. 5.2[1]. Then the attributes of the *Split* operation are randomly assigned. In the example, the *Split* operation is executed in the platoon “veh1" with an index of “3" at “25 seconds". Using these attributes and actions, the algorithm updates the status set. Because the *Split* operation divides “veh1" into two platoons from the “veh1.3" vehicle, the example status is changed to {“veh1:3", “veh1.3:3", “veh2:4"}. In this way, we can select an available event after specific sequences of events to successfully generate valid scenarios. Fig. 5.2 shows a scenario that consists of *Split*, and *Optsize*, *Leave*, *Merge* with the duration of 20 seconds.

Simulation module

After the scenario generation module returns the sets of configurations and scenarios, the simulation module executes the platooning management system on the configurations and scenarios using the VENTOS. This module performs two main functionalities to generate execution traces for each configuration and scenario, and to address the environmental uncertainty of the platooning SoS. In addition, as mentioned in subsection A, there are two types of simulation modes: “verification" and “single simulation". These modes repeat the same configurations and scenarios many times with verification and only one time without verification, respectively. To address the external environmental uncertainty problems in the platooning system, this module adds stochastic environmental objects, such as HDVs, which are not communicatable with C-ACC vehicles; thus, they cannot anticipate the movement of the HDVs in the simulation, which randomly change lanes and speeds, and even stop randomly. These features of HDVs could make collision of vehicles in the simulation. Fig. 5.3 shows generated HDVs with ids of “EV1" and “EV2". “EV1" is a truck vehicle and “EV2" is a passenger vehicle. In addition, in this module, users can change the generation period of the HDVs. Therefore, users can execute the simulation with various environments, such as rush hour or an empty road by changing the vehicle generation period. With stochastic environmental objects, this module finally returns the executions traces of each configuration and scenario to the verification module.

Verification module and results

Lastly, the verification module applies the statistical model checking algorithm to check the achievement rates of specific goals. The verification module needs verification properties of the platooning SoS

[1]In the VENTOS, there is an implementation issue that it returns an error when platoons move at 0 second in a simulation. Thus, in the VENTOS manual, they suggest to assign the platoon with speed 0 at first and change its speed at specific times. Therefore, the first two *Speedchange* events are automatically generated in Fig. 5.2

which are related to the goals of the system. Previous research [140] provided the formal definition of the verification properties in a platooning management system. However, in this definition, the properties are focused on verifying the performance of basic operations of the system without considering the macro-level goals, and assume that there is only one platoon in the simulation. Because our framework assumes that there are more than two platoons in the simulation, and it attempts to verify systems with high-level goals, the existing properties do not match our verification goals. Therefore, we defined new verification properties that are appropriate for verifying the high-level goals in a multi-platoon situation. The verification properties of the platooning system can be written in property specification languages, such as Linear Time Logic (LTL), and Computational Tree Logic (CTL), Probabilistic CTL (PCTL) [163]. We chose the PCTL verification property specification language. The defined properties are as follows:

1. $P =? [F <= t\ num\_passed\_veh\_P > n]$
2. $P =? [(op\_reject\_rate > x)\ U\ sim\_Terminate]$

We defined these two properties to check the CS-level goal, which checks arrival of participant vehicles, and SoS-level goal, which checks the success rate of the operation in the platoon. The meaning of the first verification property is "the probability that more than n cars passed through the specific point P within the first t seconds". This verification property checks whether the average speed of the platoon is maintained by the end of the simulation. For example, in the experiment, we set the average velocity of all generated platoons at 20 m/s and assigned the P value to 1,800m point to check the average speed maintenance. The second property checks "the probability that the rate of *operation_reject* is over x before the simulation terminates". In the experiments, we assigned the value x to 0 for checking the existence of *reject* signal of the operations. Thus, by using this property, we can see how smoothly the platoon operations were done at the occurrence rate of the *reject* signal.

With the two verification properties, we applied a statistical verification technique, the SPRT [164] algorithm which gradually checks the achievement rate of a specific property. Most of the existing research tried to verify the platooning system with simulating a single platoon because of the state-explosion problem. In contrast, we overcome this limitation by applying the statistical approach. Thus, we used 2 to 4 platoons which consist of 3 to 6 vehicles respectively with more than 20 HDVs to verify the platooning system in the VENTOS.

5.2 PLTBench Dataset

In this section, we explain the detailed procedure for generating a benchmark dataset based on the StarPlateS framework, and elucidate the components of the PLTBench[2] in several aspects.

5.2.1 Benchmark Generation Procedure

Fig. 5.4 depicts the overall procedure for generating the benchmark dataset. We performed two full examinations of all failed logs. The goal of the first examination is to establish the fault knowledge base for the failures that occurred in platooning SoS. In the first examination, we specified the failure scenarios in the form of the context, triggering events, and symptoms. We further analyzed the root

[2]https://sites.google.com/se.kaist.ac.kr/pltbench/

causes and failure patterns of each failure class. By generating the fault knowledge base including the aforementioned information, the basis for generating a benchmark dataset was completed.

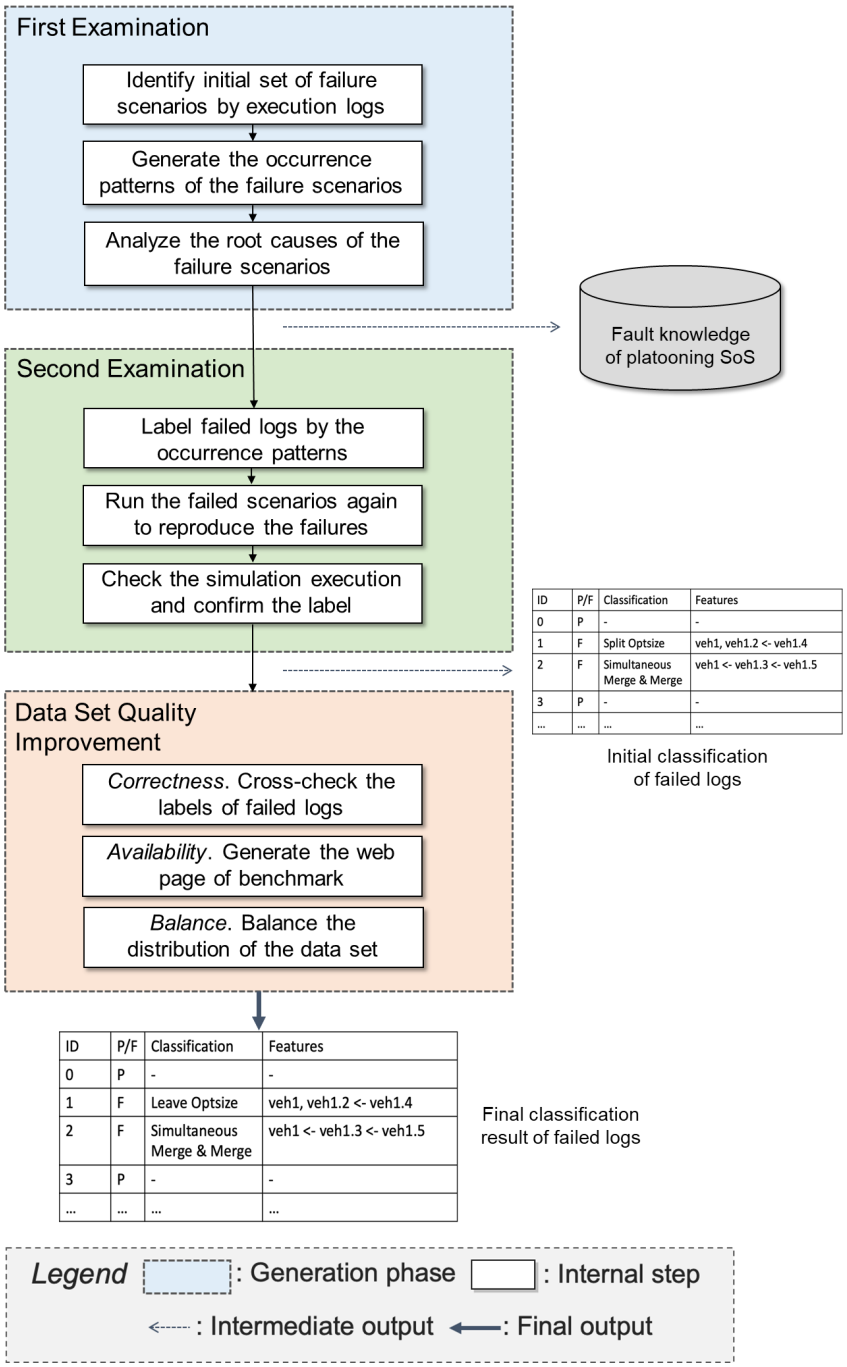


Figure 5.4: Overall process of generating the benchmark dataset for platooning SoS

Using the fault knowledge base, we performed a labeling procedure for all failed logs in the second examination phase. First, we labeled the failed logs by checking whether the occurrence patterns for each failure class are detected. Then, we executed the failed scenarios again and confirmed that the labels were correctly assigned to the failure executions in the GUI simulation. We performed the classification based on the analysis results for *OSR* and *COLL*, respectively.

Subsequently, we analyzed and improved the quality of the generated dataset. We analyzed the data set on three aspects: *Correctness*, *Availability*, and *Balance*. To check the correctness of the data set, we

assigned the failed logs to the project members and cross-checked the labeling result so that every failed log could be checked by at least two people. To increase the availability of the dataset, we implemented a web page for the accessible utilization of the benchmark dataset. Finally, we investigated the balance of the dataset, which checks whether the distribution of data was not too skewed. The details of the composition of the benchmark dataset are described in the next section.

5.2.2 PLTBench Composition

The benchmark dataset is mainly composed of raw logs with scenarios, analysis results, and classification results with statistics. The whole benchmark dataset including the analysis and classification results can be found on our web page. As it is described, the raw logs are approximately 42 GB of simulation execution traces, and consist of vehicle location data, emission data, platooning configuration data, and platooning communication data. Each datapoint was stored in units of 0.5 milliseconds. Additionally, we gathered console messages from VENTOS simulator for each simulation and saved them. The console logs contain information on the state changes of platooning vehicles and collision messages. To provide the reproducibility of the generated logs, we also added the scenarios and configurations of the logs in the dataset. The initial vehicle configurations and scenarios with platooning operation execution at specific times were included. Therefore, users can reproduce the execution traces or check the simulation in GUI by using the scenarios and configurations.

5.2.3 Empirical Analysis of Platooning SoS

In this section, we describe the empirical analysis of the platooning SoS protocol. After introducing the settings for the empirical analysis, we present the analysis results in detail.

Empirical Study Design

The target software of this analysis is the platooning management protocol provided by VENTOS. We utilized the StarPlateS framework to efficiently generate random scenarios for VENTOS execution and check whether the verification property was achieved on the execution logs. The detailed setup for the empirical study is presented in Table 5.1. We generated a total of 6,525 scenarios and execution traces for the scenarios (42 GB). In each scenario, events representing the execution of the platooning operation, such as `Merge` or `Leave` were inserted at 20-second intervals. Thus, in 100 seconds simulation, at least five platooning operations should be executed. For reliability evaluation factors, *operation_success_rate* (*OSR*) and *collision_existence* (*COLL*) were used. The *OSR* property is provided by the StarPlateS framework, and this property is one of the conventional verification properties for cloud systems [17]. The *COLL* property is newly added to verify the existence of collisions in the simulation. Originally, the VENTOS simulator had a collision-free option in the simulation; thus, no collision occurred. We turned off the collision-free setting by following the VENTOS manual and added the collision detection module in StarPlateS. We set the threshold values of 0.8 and 1 to distinguish success from failure for the *OSR* and *COLL* properties, respectively.

A single simulation time was 100 logic seconds of the simulator. It actually takes approximately 10 to 15 seconds to execute in VENTOS. In each simulation, the number of platoons was randomly selected, from 2 to 4, with a randomly assigned size of 2 to 6. The simulation map is assigned by an infinite length of road with three lanes. The starting lanes of the platoons were randomly selected. In addition, to cover environmental uncertainties in the reliability analysis, we added human-driven vehicles

Table 5.1: Overall setups of the empirical study

Parameter	Setting
Scenario setting	
Number of generated scenarios	6,525 scenarios
Duration between events	20 logic seconds
Verification property	
OSR threshold	0.8
COLL threshold	1
Simulation setting	
Duration of a single simulation	100 logic seconds
Number of generated platoons	2−4 platoons
Size of each platoon	2−6 vehicles
Map	An infinite length road with 3 lanes
Environmental objects	Human-Driven Vehicle (HDV) generated every 5 seconds
Vehicle setting	
Types of vehicles	Passenger, Truck
Autonomous driving policy	Krauss, ACC, CACC models
Hardware specification	
CPU	Intel i7-9700K CPU @3.60GHz
Memory	32GB
SSD	Samsung SSD 860 Pro 500GB
Software specification	
OS	Ubuntu 16.04 64-bit
OMNET++	5.4.1
JAVA version	java 1.8.0

(HDVs) that cannot communicate with platoon vehicles and randomly change the speed and lanes in the simulation. The HDVs are generated every 5 seconds; thus, approximately 20 HDVs are generated in a single simulation.

To cover the diversity and heterogeneity of platooning SoS, we added two types of vehicles, passenger and trucks, as platooning vehicles and HDVs. Every vehicle had an autonomous driving model out of the Krauss, adaptive cruise control (ACC), and cooperative ACC (CACC) models provided by SUMO[3]. In the simulation, platooning vehicles use the CACC model as the default option. However, in specific cases, such as persistent communication failures or a sudden loss of leaders by collision, the driving policy of the platooning vehicles is changed to the ACC or the Krauss model.

The hardware and software specifications for the empirical study are also described in Table 5.1. We followed the default settings of the VENTOS installation: Ubuntu 16.04, OMNET++ 5.4.1, and Java 1.8.0. In the next section, we elucidate the empirical analysis results based on these settings.

[3]https://sumo.dlr.de/docs/

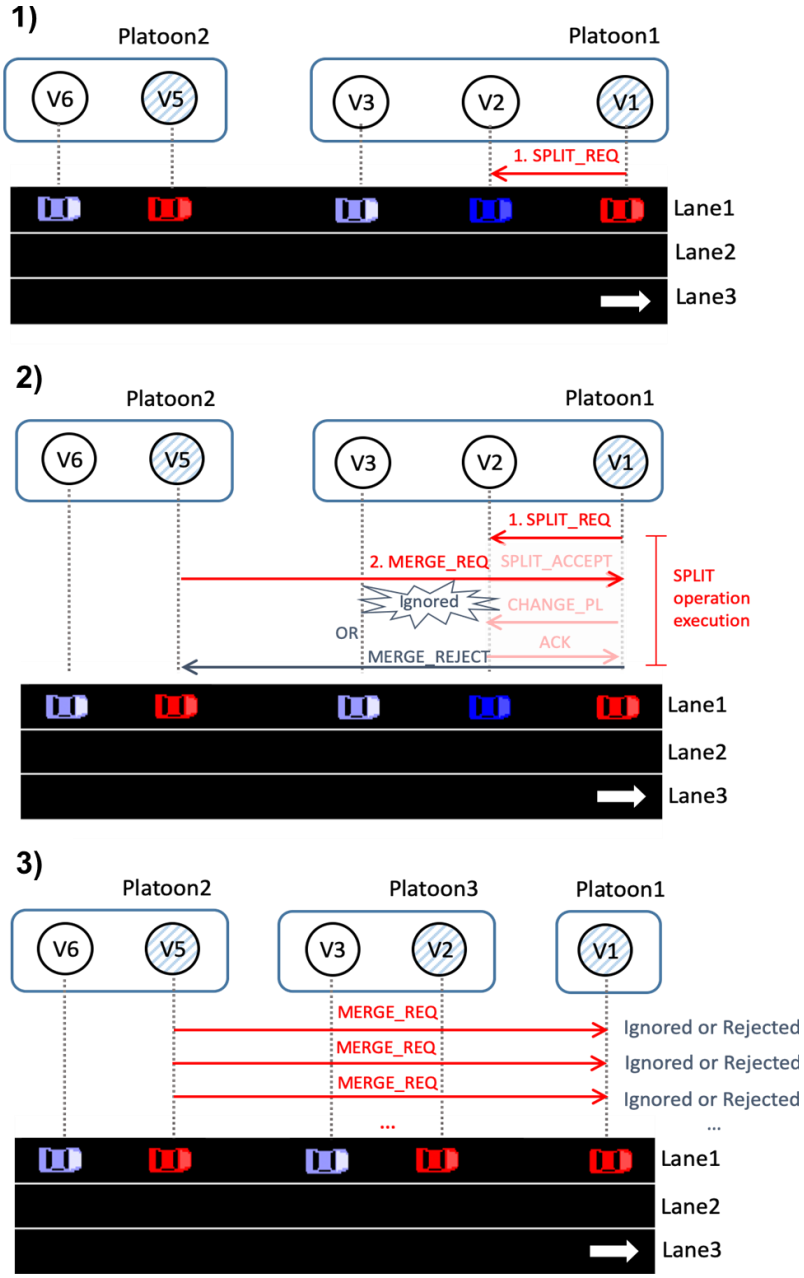


Figure 5.5: Illustrative example of executions of failure class 2 in *OSR* analysis

Operational Success Rate Analysis

Based on the settings described in the previous section, we investigated the failed execution logs using the *OSR* property. We found ten failure situations that always violate the *OSR* property. Table 5.2 elucidates the detailed fault analysis results for each situation, which are organized as failure classes. The failure scenarios for each class are described by the context, triggering event of errors, and symptoms. Furthermore, the failures are categorized into four types: Incorrect logic; Missing logic; Non-occurrence of expected events on expected time; and Communication concurrency error by their root causes and execution context. Based on the existing categorization taxonomy of interaction failures among humans and robots [79], we added new classifications, like Communication concurrency errors or

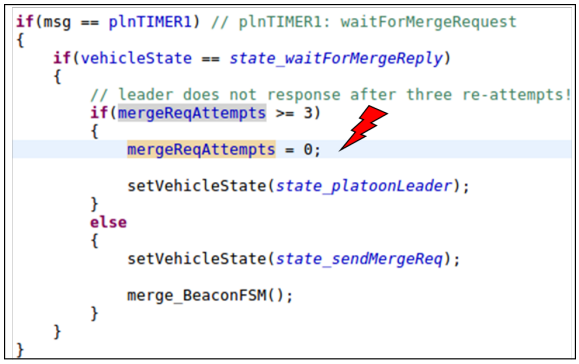
Table 5.2: Empirical analysis results on *OSR* verification property

Class ID	Failure scenario					Faults in code	Lines in code
	Name	Context	Triggering event	Symptoms	Categorization		*05_PlatoonMg.cc*
Class 1	*Simultaneous Merge & Merge*	During the Merge operation	The rear platoon leader requests Merge by OptSize policy to the same leader who simultaneously requests Merge operation.	Constantly requests Merge to the original leader	Communication concurrency error	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744
Class 2	*Simultaneous Split & Merge*	During the Split operation	The rear platoon leader requests Merge by OptSize policy to the same leader who simultaneously requests Split operation.	Constantly requests Merge to the original leader	Communication concurrency error	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744
Class 3	*Simultaneous LeaderLeave & Merge*	During the Leader Leave operation	The rear platoon leader requests Merge by OptSize policy to the same leader who simultaneously requests LeaderLeave operation.	Constantly requests Merge to the original leader	Communication concurrency error	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744
Class 4	*Simultaneous FollowerLeave & Merge*	During the Follower Leave operation	The rear platoon leader requests Merge by OptSize policy to the same leader who simultaneously requests FollowerLeave operation.	Constantly requests Merge to the original leader	Communication concurrency error	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744
Class 5	*Split Optsize*	During the Split operation	The rear platoon leader requests Merge to the newly split platoon by OptSize policy	Constantly requests Merge to the newly split platoon leader	Incorrect logic	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744
Class 6	*LeaderLeave Optsize 1*	During the Leader Leave operation	The rear platoon leader requests Merge to the new platoon leader by OptSize policy	Constantly requests Merge to the new leader	Incorrect logic	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744
Class 7	*LeaderLeave Optsize 2*	During the Leader Leave operation	The new platoon leader requests Merge to the left leader by OptSize policy	Constantly requests Merge to the left leader	Missing logic	- ChangeVehStateLastly bug OR - LeavedVehList bug	- 1397~1424 - 726~745
Class 8	*MiddleFollower Leave Optsize 1*	During the Middle Follower Leave operation	The intermediate platoon leader requests Merge to the left vehicle by OptSize policy	Constantly requests Merge to the left leader	Missing logic	- FollowerLeaveProtocol bug OR - LeavedVehList bug	- 1764~1798 - 726~745
Class 9	*MiddleFollower Leave Optsize 2*	During the Middle Follower Leave operation	The rear platoon leader requests Merge to the intermediate leader by OptSize policy	Constantly requests Merge to the intermediate leader	Non-occurrence of expected events on expected time	- FollowerLeaveProtocol bug AND - LeaveSplitCaller bug	- 1780~1796
Class 10	*EndFollower Leave Optsize*	During the Split operation in the End Follower Leave operation	The rear platoon leader requests Merge to the left vehicle by OptSize policy	Constantly requests Merge to the left leader	Incorrect logic	- BusyReplying bug OR - MERGE_REQUEST attempt bug	- 609~636 - 729 ~744

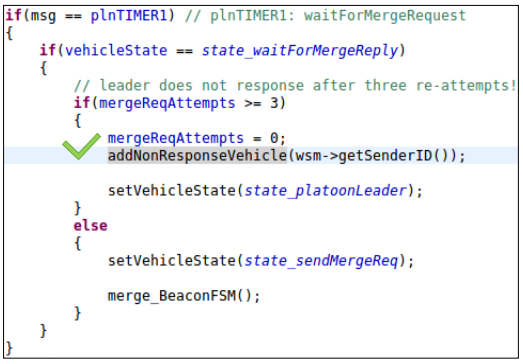
Cascading failure, and applied them in the analysis. For example, the failure classes 5, 6, and 10 are caused by the "MergeRequestAttempt bug", which is an incorrect logic applied during the `MERGE_REQ` process. Similarly, failure classes 7 and 8 were caused by the missing logic of specific protocol execution, such as `LeaderLeave` and `MiddleFollowerLeave` operations. Class 9 denotes that during the `MiddleFollowerLeave` operation, certain events were not correctly executed in a specific condition. Finally, classes 1 to 4 explains the concurrency failure cases, where a platoon leader requests operations, such as `LeaderLeave` or `Merge`, and is also requested to `Merge` by the other platoon leader simultaneously.

Fig. 5.5 illustrates the execution of failure class 2, which is a simultaneous `Split` and `Merge`. When a platoon follower, V2, requests `Split` to V1, the other platoon leader behind the platoon, V5, could simultaneously request `Merge` operation to V1. In the VENTOS protocol, the `Merge` request delivered during the `Split` execution, is ignored or rejected. However, even after the `Split` operation is completed, we observed that the rear platoon leader, V5, continuously requests `Merge` to the same vehicle, V1. This failure situation adversely affects the operation of the related platoon vehicles, V1 and V5, and may cause overall operation delays or execution failures.

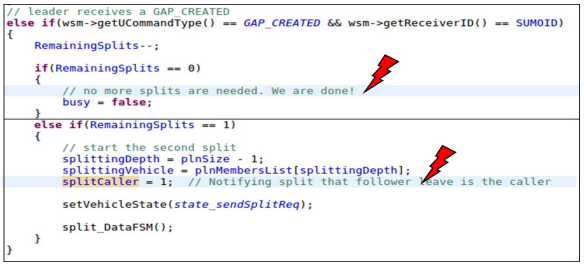
The seventh and eighth columns of Table 5.2 describe the corresponding faults and their locations. For example, "`MERGE_REQUEST` attempt bug" is one where the "MergeRequestAttempts" value is changed to the initial value after the three times requests, but the sender vehicle continuously requests `Merge`. To resolve the bug, the additional data variable is necessary to check the busy vehicles. The bug is located in the code blocks of *05_platoonMg.cc* file in lines 729-744. The *OR* symbol in the seventh column denotes that, if one of the faults is fixed, the failure is resolved. The *AND* symbol indicates that both of the faults must be fixed to resolve the failure. The faults identified in this study are mostly due to the absence of several statements, which are missing logic or incorrect logic flows for unexpected situations and interactions in the protocol. This implies that most solutions for the bugs include adding new conditional code blocks followed by the improvement of the protocol model design. On the web page of the PLTBench, we provided detailed information for all code-level faults and their occurrence patterns with more various illustrative examples.



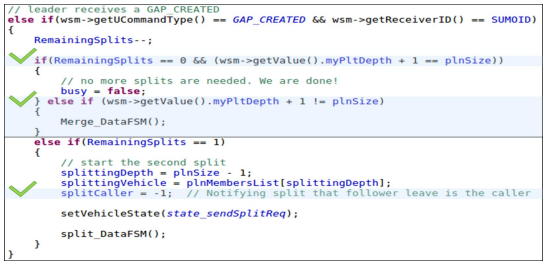
a An example of MergeRequestAttempts bug



b An example solution of MergeRequestAttempts bug



c Examples of bugs related to pattern 3



d An example solution of bugs related to pattern3

Figure 5.6: Example code-level bugs and solutions identified in this study

The `Merge` requests by the rear platoon were caused by the optimal-size maintenance policy in the protocol. However, we focused on the continuous requests of `Merge`, because it badly affected the execution of other operations by continuing to deliver redundant messages. We found the root cause of the continuous `MERGE_REQUEST` in the isolated code blocks depicted in Fig. 5.6a. In the code, if a vehicle requests a `Merge` operation with the same vehicle more than three times, the sender vehicle stops requesting more. However, after the initialization of the mergeReqAttempts variable returns to 0, the sender vehicle again requests `Merge` to the same vehicle. In other words, once a vehicle starts sending a `MERGE_REQUEST`, it constantly sends the request to the vehicles until the end of the simulation. One solution for the "MergeRequestAttempt bug" suggested addressing this is by saving and checking vehicle ids that have already been rejected more than three times, as in Fig. 5.6b.

Fig. 5.6c describes example faults related to the $cluster_3$ pattern; "FollowerLeaveProtocol bug" and "SplitCaller bug". Firstly, we found that the execution of the `MiddleFollowerLeave` differed from the platooning protocol logic [5]. In the logic, the `MiddleFollowerLeave` consists of two `Splits` of the following and left vehicles, and one `Merge` of the intermediately split vehicles. However, in the VENTOS code depicted at the top of Fig. 5.6c, `MiddleFollowerLeave` only consists of two `Splits`, and the `Merge` process is not properly called when there is no remaining `Split` operation. The second bug, described at the bottom of Fig. 5.6c, is related to the execution of the two Split operations. During the `MiddleFollowerLeave` operation, the two `Split` operations must be executed with different internal settings, but the splitCaller variable has the same value in each `Split` call. We explain the example solutions for the described bugs in Fig. 5.6d. Firstly, for solving the "FollowerLeaveProtocol bug", the conditional statement to check whether the case is `MiddleFollowerLeave` or `EndFollowerLeave` in the first "if" statement is added. Then, one more "else if" block must be appended to call the `Merge` operation when it is `MiddleFollowerLeave`. Next, to resolve the "LeaveSplitCaller" bug, -1 must be assigned to the splitCaller variable rather than 1.

Table 5.3: Empirical analysis results on *COLL* verification property

Class ID	Failure scenario					Faults in code	Lines in code
	Name	Context	Triggering event	Symptoms	Categorization		*05_PlatoonMg.cc*
Class 1	*Split*	During the Split operation	The split operation from the platoon results the platoon vehicles to increase the front distance.	Collision occurred	Missing logic	- Missing distance checking during Split	- 1381 ∼ 1393
Class 2	*LeaderLeave*	During the Leader Leave operation	The leave operation from the platoon results the platoon vehicles to increase the front distance.	Collision occurred	Missing logic	- Missing distance checking during LeaderLeave	- 1581 ∼ 1596
Class 3	*MiddleFollowerLeave*	During the Middle Follower Leave operation	The leave operation from the middle of the platoon results in a split operation.	Collision occurred	Missing logic	- Missing distance checking during MiddleFollowerLeave	- 1738 ∼ 1749 - 1764 ∼ 1777
Class 4	*EndFollowerLeave*	During the End Follower Leave operation	The end vehicle prepares to leave the platoon by increasing the front distance.	Collision occurred	Missing logic	- Missing distance checking during EndFollowerLeave	- 1738 ∼ 1749 - 1751 ∼ 1763
Class 5	*SpeedChange*	During the Speed Change operation	The vehicle changes its speed	Collision occurred	Missing logic	- Missing distance checking during SpeedChange	in vehicle driving setting code
Class 6	*Merge*	During the Merge operation	The merge operation is unsuccessful resulting in increasing the front distance.	Collision occurred	Missing logic Cascading failure	- Missing distance checking during Merge - BusyReplying bug OR - MERGE_REQUEST attempt bug	- 727 ∼745

Through the analysis on interaction patterns and isolated codes, we showed that most of the solutions included adding new codes, such as modifying specific logic or adding a new logic. This is a characteristic of omission bugs, and it can be shown that with the proposed fault analysis process, isolation and identification of omission bugs in platooning SoS protocol are possible. Besides, We have issued the identified bugs in the VENTOS platooning protocol to VENTOS developers, and we are waiting for their answers. Detailed descriptions of the “MergeRequesetAttempt bug", “FollowerLeaveProtocol bug", and “LeaveSplitCall bug" are elucidated above, but the detailed explanations for all bugs can be found in the PLTBench website. We expect that the fault identification results could be used as an example fault knowledge base for future studies focusing on testing and analyzing failures in large and complex systems.

Collision Analysis

Similar to the *OSR* property-based analysis, we conducted a detailed analysis on failure cases that violated the *COLL* property. Table 5.3 describes the failure scenarios, code-level faults, and categorization of the root causes. We focused on the six types of failure classes that were caused by platooning operations. There are more types of failure scenarios that cause collisions in Table 5.5. However, we conducted the analysis focusing on failure scenarios that have the root causes inside the platooning system. First, we confirmed that collisions could be caused by all platooning operations that are executable in the VENTOS simulator. The root cause of collisions in the majority of detected failure classes was that the platooning operation logic did not consider the presence of environmental vehicles (i.e., HDVs) or other platoon vehicles in the rear. Therefore, most of the root causes of detected collisions are the omission of the logic that considers the distance from the rear vehicle in each operation. In the case of `SpeedChange` operation, there is no code of function call in the *05_platoonMg.cc* file, but in the other code section. Similarly, in order to solve the majority of the root causes found above, specific codes must be newly added.

Fig. 5.7 depicts one of the interesting failure scenarios detected in the collision analysis. The illustrated failure case belongs to failure class six in Table 5.3 and is a type of comprehensive and cascading failure scenario. First, in a situation where three platoons, V1, V2, and V5, request simultaneous `Merge` to the front leader, illustrated in 1) in Fig. 5.7, the inter-platoon distance between V1 and V2 is reduced to perform the `Merge` operations, as illustrated in 2). However, owing to the simultaneous requests of the `Merge` operations, V2 is overloaded and the `Merge` with V1, which is already in progress, eventually fails.

Figure 5.7: Illustrative example of executions of failure class 6 in *COLL*

Due to the failure of the progressing `Merge` operation, V2 needs to increase the inter-platoon distance to the original distance. Inevitably, in this process, the V5 also slows down to maintain the inter-platoon distance with V2, and V5 collides with the rear environmental vehicle in the end. In this manner, failure situations in platooning SoS occur during concurrent and intricate interactions; thus analyzing the failures is a highly time-consuming task. We generated the benchmark dataset based on the detailed analysis results.

5.2.4 Statistics of the Dataset

In addition to the detailed empirical analysis results for the reliability properties, we provide failure scenario classification results and statistics for all failed logs. The classification results for failure scenarios that violate the *OSR* property are listed in Table 5.4. We found 3,256 numbers of failed logs and 3,830 cases of failure executions. The difference between the total number of failed logs and the number of actual detected failure classes appears because multiple failure scenarios occur simultaneously in a single log. On average, it was confirmed that 1.17 failure classes were found in one log.

It is also observed that the data distribution is affected according to the failure classes. Generally, the simultaneous failure classes were smaller than the other classes. This trend is caused by the generation of random scenarios. Scenarios in which specific operations are executed simultaneously are less likely to be randomly generated than scenarios that do not involve simultaneous operation executions. Therefore, differences in distribution are inevitable because of the difficulties in generating edge cases in the process of generating random scenarios. From the viewpoint of data balancing, uniformly distributed data is not always the best option for a dataset [165]. In the credit card defrauded dataset, only 3.9% of the data are related to the fraud [166], and only 0.4% is positive in the HIV prevalence data set [165]. Nevertheless, we plan to generate more scenarios involving simultaneous operation execution. We will use a guided method by modifying the random scenario generation module in StarPlateS and provide more numbers

Table 5.4: *OSR* analysis statistics

Class ID	**Counts**
Class 1 (Simultaneous `Merge` & `Merge`)	213
Class 2 (Simultaneous `Split` & `Merge`)	62
Class 3 (Simultaneous `LeaderLeave` & `Merge`)	133
Class 4 (Simultaneous `FollowerLeave` & `Merge`)	138
Class 5 (`Split` optsize)	794
Class 6 (`LeaderLeave` optsize 1)	159
Class 7 (`LeaderLeave` optsize 2)	579
Class 8 (`MiddleFollowerLeave` optsize 1)	389
Class 9 (`MiddleFollowerLeave` optsize 2)	1104
Class 10 (`EndFollowerLeave` optsize)	259
Total	3830

of the failure executions corresponding to the simultaneous operation executions.

Table 5.5 describes the statistics of the classification results by the failure classes that violate the *COLL* property. We conducted the classification process according to the collision subjects: "Env vehicle (i.e., HDV) with Plt vehicle", "Env vehicle with Env vehicle", and "Plt vehicle with Plt vehicle" and to the cause of collisions: "by Env vehicle", "*By Plt Op*erations(e.g., `Merge`, `MiddleFollowerLeave`), and "by unknown". The simultaneous occurrence of multiple failure classes showed a similar trend in collision failures. We identified 965 failure cases among the 900 failed logs. However, we found that the logs having multiple failure cases contain more numbers of failure cases at once. For instance, the maximum number of multiple failure cases in a single failed log is six in the *COLL* analysis result.

The failure cases we mainly focus on are collisions caused by platooning operations (*By Plt Op*) or collisions involving platoon vehicles (*Plt_Plt* and *Plt_Env*). All collision cases among platoon vehicles (*Plt_Plt*) are caused by HDVs, which suddenly change their speed or driving lanes just in front of truck platoon vehicles. Most crashes caused by environmental HDVs (*By Env*) have similar failure scenarios to sudden lane change and speed change. In the collision cases caused by the platooning operations (*By Plt Op*), we found 172 cases in total. The most common case is by `Split` operation. This is because the `Split` operation is called during the execution of all `Leave` operations.

The *Unknown* cases are the scenarios that are difficult to reproduce or the cases in which the causes of the collisions are unclear by checking the simulation. For instance, one example of an *Unknown* case is a situation in which an HDV increases the distance to its rear vehicle corresponding to a platoon vehicle in the process of `Split` in the other lane. To analyze the unknown cases, we plan to perform the deep code-level analysis for all simulator codes, as well as the platooning protocol codes in VENTOS.

We utilized the empirical analysis results of the platooning SoS in the evaluation of the proposed approaches with existing techniques. The details are elucidated in Section 6.

Table 5.5: *COLL* analysis statistics

Class ID	***Env_ Env***	***Plt_ Plt***	***Plt_ Env***	**Sum**
By Env	521	9	218	748
By Plt Op	29	–	172	201
- Class 1 (`Split`)	11	–	78	89
- Class 2 (`LeaderLeave`)	6	–	18	24
- Class 3 (`MiddleFollowerLeave`)	3	–	26	29
- Class 4 (`EndFollowerLeave`)	1	–	25	26
- Class 5 (`SpeedChange`)	8	–	23	31
- Class 6 (`Merge`)	0	–	2	2
Unknown	4	–	12	16
Total	554	9	402	965

5.3 Mass Casualty Incident-Response (MCI-R) SoS Dataset by SIMVA-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 [10] 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. Figure 5.8 describes the structure of the MCI-R collaboration protocol logic. Centering on the SoS manager, CSs cooperate with each other to perform patient searching, rescue, transportation, and treatment. We found five types of bugs among the defined stimuli, which are specifically located in the collaboration protocol code, in SIMVA-SoS as depicted in Figure 5.8.

The first bug causes a failure scenario in that the SoS manager delivers not the nearest and available Hospital information, but the position of the nearest Hospital of the requested Firefighters or Ambulances. The second bug (i.e., collaboration fault 2) adds a deadlock fault in the collaboration of the three CSs of the SoS manager, Ambulances, and Bridgehead; thus, ambulances cannot receive the messages from SoS Manager even if there exist patients in the bridgehead. 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. The third one causes a failure scenario in that Bridgehead cannot recognize that the Ambulances arrive at the Bridgehead. The fourth bug scenario injects fault into the logic that more than one firefighter finds the same patient. The last buggy scenario causes a failure that Bridgehead can not recognize that the Firefighters transport patients. The detailed information of the injected faults containing the Context–Triggering event–Symptom analysis results and the specific location of faults in the protocol code are described in the open SIMVA-SoS

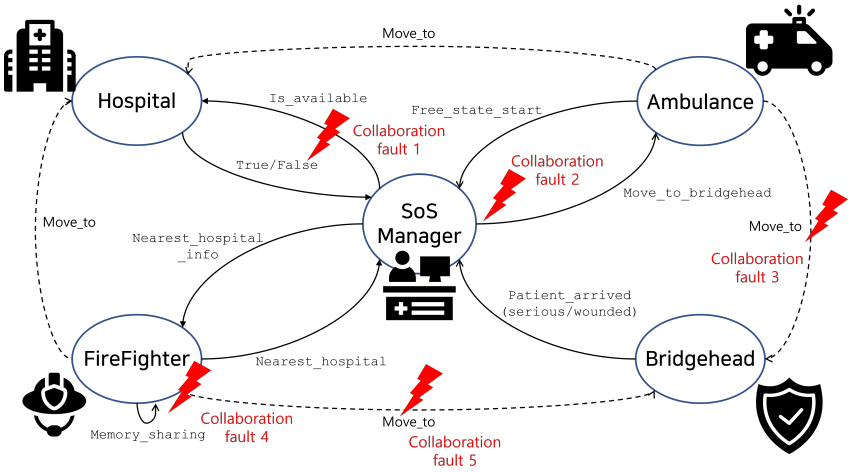


Figure 5.8: Illustrative example of Collaboration Protocol of MCI-R SoS in SIMVA-SoS repository[4].

5.4 Drone Swarming Dataset by SwarmLab

As a representative example of implicit collaboration, we utilized a drone swarming simulator, SwarmLab [11]. SwarmLab provided a MATLAB-based simulation module and GUI setting for a single drone and swarming simulation. In SwarmLab, there exist two drone swarming algorithms: Vasarhelyi [67] and Olfati-saber [167]. For example, the final velocity calculation method, V_i^d, of Vasarhelyi algorithm is defined as follows:

$$V_i{}^d = \frac{V_i}{|V_i|}V^{flock} + V_i^{rep} + V_i^{frict} + \sum_s V_{is}^{wall} + \sum_s V_{is}^{obstacle}$$

$$V_i^d = \frac{V_i{}^d}{|V_i{}^d|}min|V_i{}^d|, V^{max}$$

where V_i is a constant flight value of a drone with id i, V^{flock} is a short-range collision avoidance speed value, V^{firct} is a middle-range speed value to minimize the velocity difference in the given distance, V^{wall} is a long-range speed value to maintain area of swarms based on the local center of mass, $V^{obstacle}$ is a speed value calculated to avoid the obstacle of a drone, and V^{max} is a constant max speed value of a drone. We have utilized both algorithms to generate the experimental dataset for drone swarming.

Because there exists any open dataset that investigates the failures of drone swarming based on the simulation to the best of our knowledge, we generated several drone swarming failures scenarios based on the collision test scenarios in SwarmLab[5]. Based on the collision test scenarios, we have generated several patterns of failure scenarios and investigated the patterns to identify the root causes of the collision of drone swarming in the drone swarming algorithm.

[4]https://github.com/psumin/SoS-simulation-engine
[5]https://github.com/lis-epfl/swarmlab

Chapter 6. Experiment

6.1 Experiment Design

6.1.1 Research Questions and Evaluation Metrics

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 three target systems in this experiment: platooning SoS, mass casualty incident-response (MCI-R) SoS, and drone swarming SoS.

Platooning SoS. Table 6.1 explains the overall summary of the two target systems. For the platooning SoS, we utilized the PLTBench dataset [42], 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 [42] from the analysis of operation success rate property on about 8,000 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 the following explanation on evaluation metric. The size of LoCs of the platooning collaboration protocol is 3,596 out of 540,277 in VENTOS [5], encompassing codes only related to executing the platooning collaboration except for the simulation, configuration, logging, and rendering.

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 [10] 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 2,034 logs including 1,005 failure logs with the faults injected. The size of LoCs of MCI-R collaboration protocol is 2,364 out of 8,691 in SIMVA-SoS.

Drone Swarming SoS. As explained in Section 5.4, we have applied the proposed approach to drone swarming SoS scenario provided by SwarmLab. Because there exists no open dataset or benchmark available for the evaluation, we concentrated on identifying undetected bugs and failure scenarios in existing drone swarming algorithm. We executed the collision test scenarios in SwarmLab and found several patterns of collision failures in drone swarming. We will explain the investigation results in the qualitative analysis of the evaluation in detail.

[1] https://sites.google.com/se.kaist.ac.kr/pltbench
[2] https://maniam.github.io/VENTOS/
[3] https://github.com/psumin/SoS-simulation-engine

Table 6.1: Overall statistics of the target systems in experiment

Scenario	**Platooning SoS** [42]	**MCI-R SoS** [10]
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 and collision existence	Treatment rate of patients
Threshold value	0.8/1.0	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[1] / VENTOS[2]	SIMVA-SoS simulator[3]

* including the number of newly detected failure scenarios and faults

We defined the following research questions for the experiment:

- **RQ1.** Does the proposed approach accurately extract FII patterns that explain SoS failures?
- **RQ2.** Do the clustering results yield better clustering precision considering multiple FII patterns?
- **RQ3.** Does the context mining approach depict the feasible efficiency compared with existing techniques?
- **RQ4.** Does the localization method precisely infer the bug location from the patterns?

RQ1 aims to check the accuracy of the proposed pattern mining technique, which prevents information loss 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 pattern*s contain critical information about failures, such as failure context, triggering events, and symptoms. We created the *ideal patterns* for all platooning SoS failure classes based on the analysis results in PLTBench in advance.

We defined the pattern identity with weight (PITW) score by extending the difference-based accuracy measure, MAE [168, 169], 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)).

*Ideal pattern*s for SoS failures were used as key factors for evaluating the accuracy of FII pattern mining results. We manually created the *ideal pattern*s by investigating the detailed fault knowledge of SoS failures in PLTBench. For example, the fault knowledge corresponding to the *ideal pattern 4* in Figure 6.2a 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_REQ`s to the original leader.

The pattern in Figure 6.2a follows the context of the `Middle FollowerLeave` operation having two `Split`s, thus the pattern contains the sequence of `LEAVE_REQ`, `LEAVE_ACCEPT`, and two occurrences of `SPLIT_REQ`, `SPLIT_DONE`, etc. In Figure 6.2a, `MERGE_REQ` in line 2 indicates the triggering events of simultaneous request, and repetitive `MERGE_REQ`s 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 composing the patterns are essential for understanding the failures. For example, `MERGE _REQ`s 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 Figure 6.2. 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 SoS 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 Figure 6.2a only showed the essential parts among the total of 29 messages. We gave weights to the essential messages in calculating the PITW score.

RQ2 aims to evaluate the overlapping clustering precision regarding the existence of the multiple failure patterns. 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 [42]. We compared *Multi-TIME* and *Multi-BASE* clustering results by considering all possible combinations of hyperparameter values.

To properly evaluate the overlapping clustering precision, an evaluation metric and clustering oracle (i.e., categorized failed logs) are needed. Hence, we used the categorization results of all failed logs in the PLTBench dataset. We also implemented the recently proposed F1P score [170]. The F1P is defined as follows:

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

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

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

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

C denotes a set of formed clusters consisting of clusters $c \in C$, and C' denotes a set of ground-truth clusters 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 cases in all combinations of the ranges with a time window set, $T = \{0, 20, 40, 60, 80\}$.

RQ3 aims to check the context mining efficiency of the proposed approaches with existing techniques. The accuracy of extracted failure context patterns is the most important factor for evaluating the context mining techniques. Nevertheless, the cost to extract the failure context patterns should be figured out to properly evaluate the performance of the proposed approaches. In this study, we calculated the log-scale time of the single execution for each experiment group.

RQ4 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 [171] and Top-K [87] 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 [158], Ochiai [159], OP2 [160], Barinel [172], and DStar [161]. 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 [173]. The spectra of the SBFL methods are defined by the code coverage of executed ssptatements in the collaboration protocol files of the target system. Based on the spectra of each failed scenario 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 implemented protocol [42]. The PLTBench dataset particularly contains a few multi-statement bugs, where bug fixes span multiple statements [173]. 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 [174, 148] when multiple statements have the same suspiciousness score, then all of the codes are treated as the average rank of the statements.

6.1.2 Benchmark Dataset

As it is described in detail in Section 5.2, we mainly utilized the PLTBench dataset in the experiment. We totally used 16 failure classes generated from about 8,000 randomly generated platooning execution scenarios. The generated logs are verified by the two goal properties: operation_sucess_rate (OSR) and collision_existence (COLL). Figure 6.1 describes the failure mode coverage of CPSoS failures by the OSR and COLL analysis results. The OSR analysis results mainly target the software-software interaction failure cases. Otherwise, COLL analysis results contain the software-environment impact failures and comprehensive failures caused by the software-software-environment interactions.

The PLTBench dataset was built via a systematic process involving the manual investigation of thousands of logs [42]. 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.

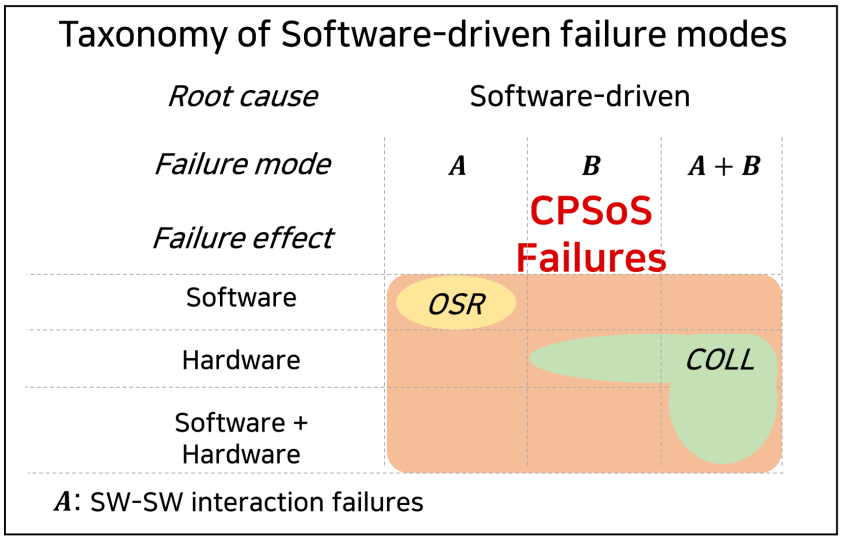


Figure 6.1: Failure mode coverage of CPSoS failures by the OSR and COLL analysis results

Figure 6.2a describes the *ideal pattern* for the failure class 4 in PLTBench[4]. 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 `Split`s 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 `Merge`s 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 Figure 6.2a. The new FII pattern 1 in Figure 6.2b 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`. Our approach could extract the new pattern 1, because LCS algorithm generally converges to the sequence with smaller length. This means that the probability of extracting new pattern 1 is higher than that of the *ideal pattern 4*. This can also be confirmed by the fact that PITW scores of Class 4 are lower than those of New Class 1 in 6.4. In addition, the other reasons affecting to the Class 4 PITW scores are that (1) the pattern of class 4 has the longest length of 29 among the other patterns and (2) the intervals between the first and second `Split` were varied because of the delays.

Figure 6.2c 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 as depicted in lines 3 to 6, 10, and 11. After a few times of omission of the both operations (a.k.a a dead-lock situation), `Leave` is finally executed. This failure has totally different symptoms from *ideal pattern 4*, in that both of the operations are omitted a few times like a dead-lock situation 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 8,000 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

[4]https://sites.google.com/se.kaist.ac.kr/pltbench

overlapping clustering also contributes to the classification and extraction of the unique failure pattern.

6.1.3 Experiment Group

Through **RQ1** to **RQ3**, we evaluated a total of six approaches for OSR analysis results: *CAFCA*, *C-FCM* [1], *TIME*, *BASE* [24], *SPADE*[175], and *LOGLINER*[16]. *CAFCA*, *TIME*, and *BASE* are the approaches proposed by the author. *BASE* is our previously proposed *LCS*-based pattern mining and clustering method that does not fully consider the features of interaction logs. *TIME* is a context mining and localization approach focusing on the software-software interaction failures. *CAFCA* is also a context mining-based fault localization approach, but targeting not only the software-software interaction failures, but also the software-environment impact failures and software-software-environment interaction failures. *SPADE* is one of the most commonly used frequent sequence mining algorithms in various domains [25, 107, 108]. *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.

For the COLL analysis results, we compared the proposed *CAFCA* approach with *C-FCM*, *C-KS2M* [157], *C-MTS* [104] approaches. To the best of our knowledge, we have not found the existing studies that can cover the software-software-environment interaction failures. Therefore, we designed the experiment to compare different clustering approaches based on the same pattern mining and similarity calculation algorithm proposed in this study.

In **RQ4**, we compared the proposed fault localization method, *SeqOverlaps*, with several SBFL methods: Tarantula [158], Ochiai [159], OP2 [160], Barinel [172], and DStar [161]. 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 [173].

6.2 Experiment Results

6.2.1 Qualitative Analysis

Qualitative Analysis on Platooning SoS Results

The PLTBench dataset was built via a systematic process involving the manual investigation of thousands of logs [42]. 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.

Figure 6.2a 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 `Split`s 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 `Merge`s 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 Figure 6.2a. The new FII pattern 1 in Figure 6.2b 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`.

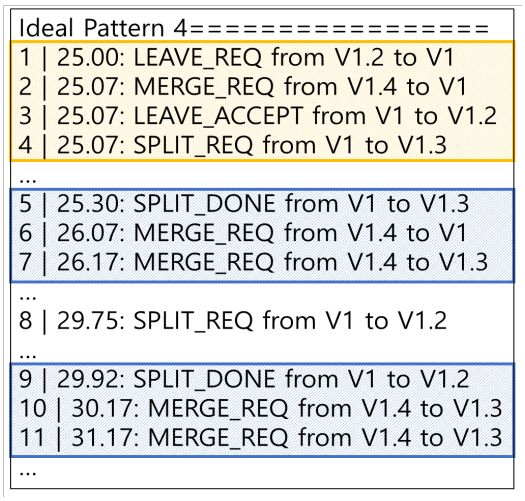
Our approach could extract the new pattern 1, because LCS algorithm generally converges to the sequence with smaller length. This means that the probability of extracting new pattern 1 is higher than that of the *ideal pattern 4*. This can also be confirmed by the fact that PITW scores of Class 4 are lower than those of New Class 1 in 6.4. In addition, the other reasons affecting to the Class 4 PITW scores are that (1) the pattern of class 4 has the longest length of 29 among the other patterns and (2) the intervals between the first and second `Split` were varied because of the delays.

Figure 6.2c 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. After the `Leave` and `Merge` operations are omitted a few times (a.k.a dead-lock situation), `Leave` is finally executed. The illustrative example of this new failure class is described in Figure 1.1. 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 8,000 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.

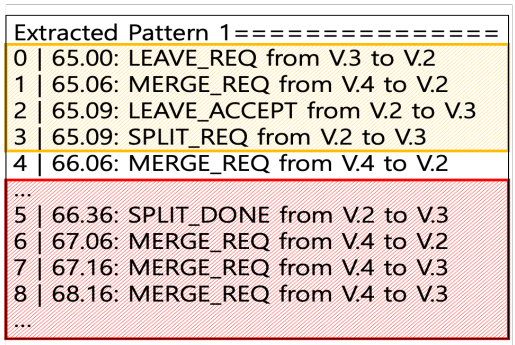
Moreover, we identified a new bug causing the new failure class 2. Figure 6.2d illustrates the bug location and an example patch of the bug. Line 815 of the "*05_PlatoonMg.cc*" file in VENTOS simulator [5], which has a target platooning protocol code, contains one of the buggy codes. The patch example 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.

Qualitative Analysis on Drone Swarming SoS Results

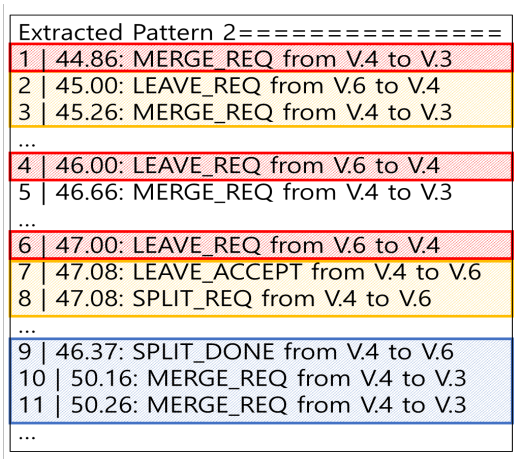
We have found several failure patterns of collision in drone swarming execution. One of the commonly detected failure scenarios regardless of the drone swarming algorithms is a failure caused by the increasing number of maximum neighbor in a swarm. Extracted patterns commonly represent the high number of neighbors in a communication and vector velocity values towards the center of the mass in a swarm. Vasalhelyi et al. [68] specified that as the communication range (i.e., maximum neighbor) of swarm increases, the drone collisions within the swarm tended to decrease. However, the extracted patterns in this study explain that even if the collision-avoidance speed vector, V^{rep}, is large, the speed vector for maintaining the local center of mass, V^{wall}, is too large due to the high number of neighbors; thus, The vector velocity values of the adjacent drones are calculated in the direction in which the drones approach each other. This failure scenario can be detoured by modifying the main algorithm described in Section 5.4 to weight appropriate coefficient to each velocity vector.



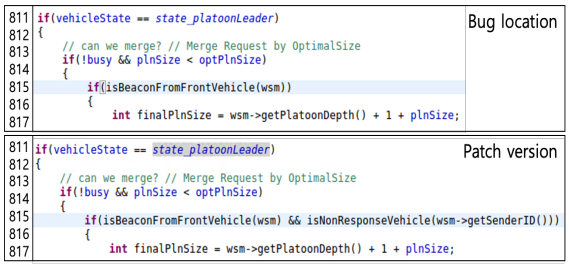
a Ideal pattern for failure class 4 (simultaneous `Leave` & `Merge` requests)



b New FII pattern 1 extracted in experiment



c New FII pattern 2 extracted in experiment

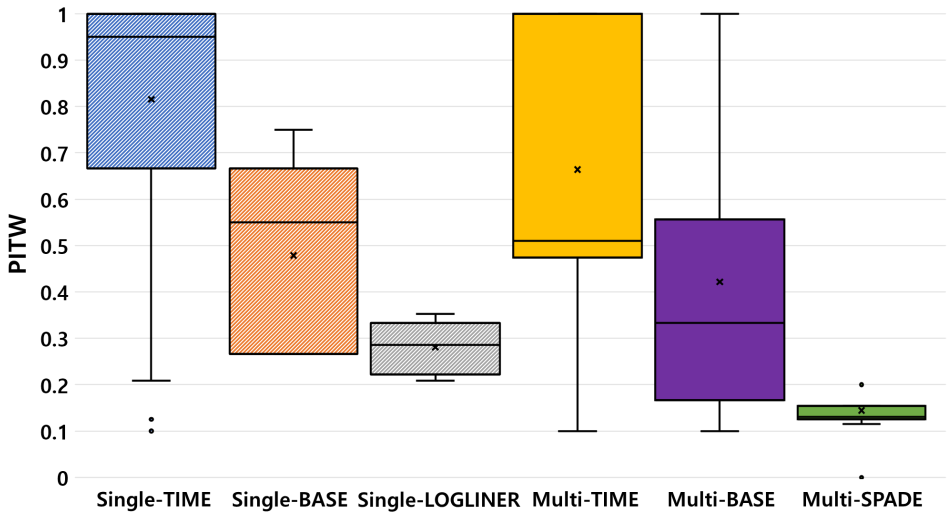


d Bug location and patch example for FII pattern 2 failures

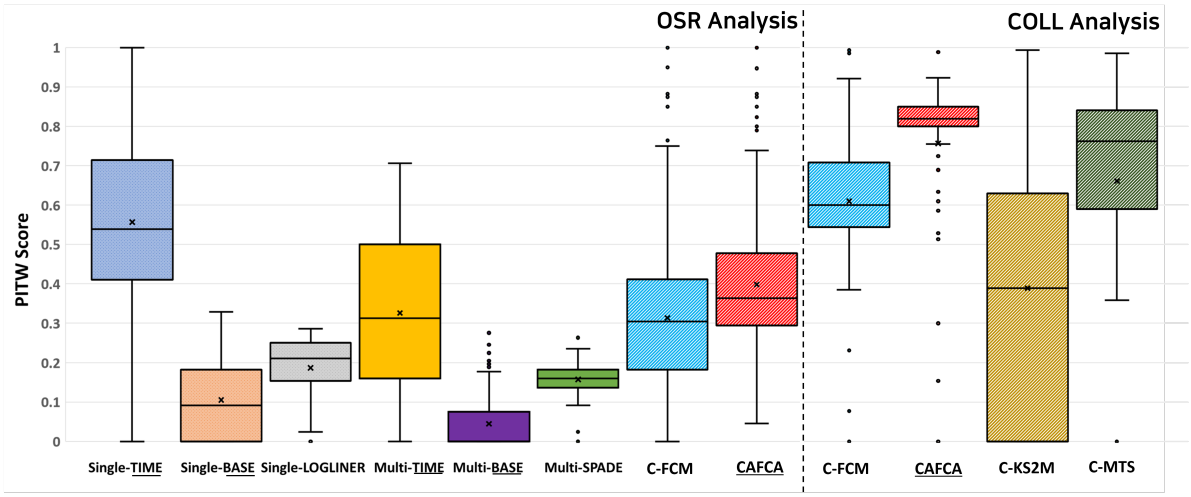
Figure 6.2: Example FII patterns and bug location in code

6.2.2 Quantitative Analysis

RQ1. Failure Context Pattern Mining Accuracy



a PITW evaluation results on the MCI-R SoS data



b PITW evaluation results on OSR and COLL analysis results of platooning SoS data

Figure 6.3: PITW evaluation results

Figure 6.3 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 Figure 6.3 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 of pattern mining 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 Figure 6.3. This demonstrates that the proposed approach is relatively sensitive to the order 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 Figure 6.4. 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 6.2.2.

In the COLL analysis results, *CAFCA* approach achieved the highest average PITW score than other existing techniques. All the experiment group has achieved the maximum values close to 1.0, even considering the classification of the multiple failures. *C-KS2M* approach showed the lowest average PITW score, even the approach is focusing on fuzzy clustering of the sequential data. *CAFCA* presented not only the highest average PITW score, but also achieved the lowest deviation of the accuracy by the random inputs than other studies.

Findings. The *CAFCA* approach presented the highest accuracy on the mining of failure context patterns in both of OSR and COLL analysis results.

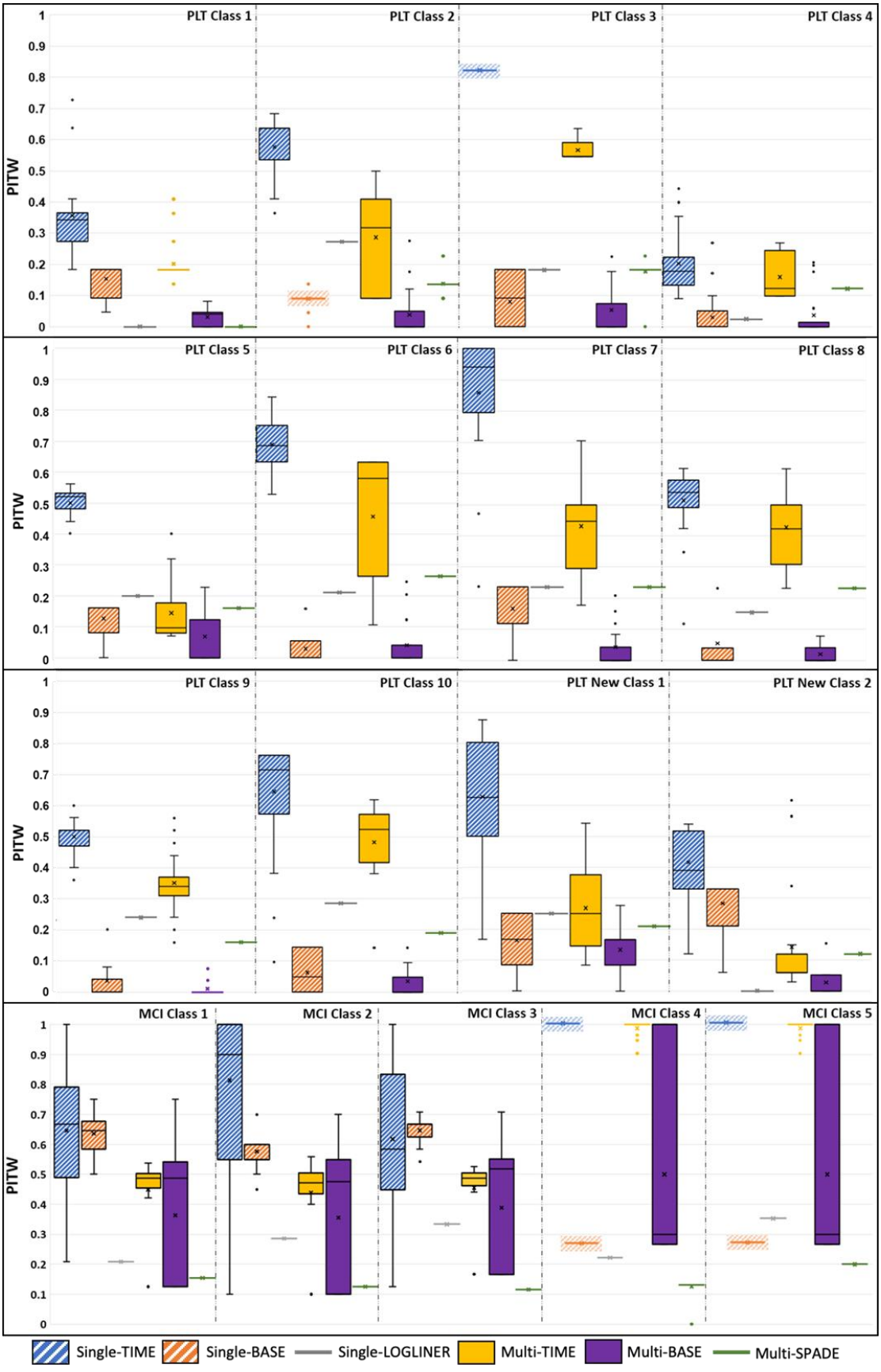


Figure 6.4: PITW evaluation results of all failures scenarios in platooning and MCI-R SoS

RQ2. Overlapping Clustering Precision

We evaluated the efficacy of our clustering approach considering the extraction of multiple failure patterns in a single 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 [170].

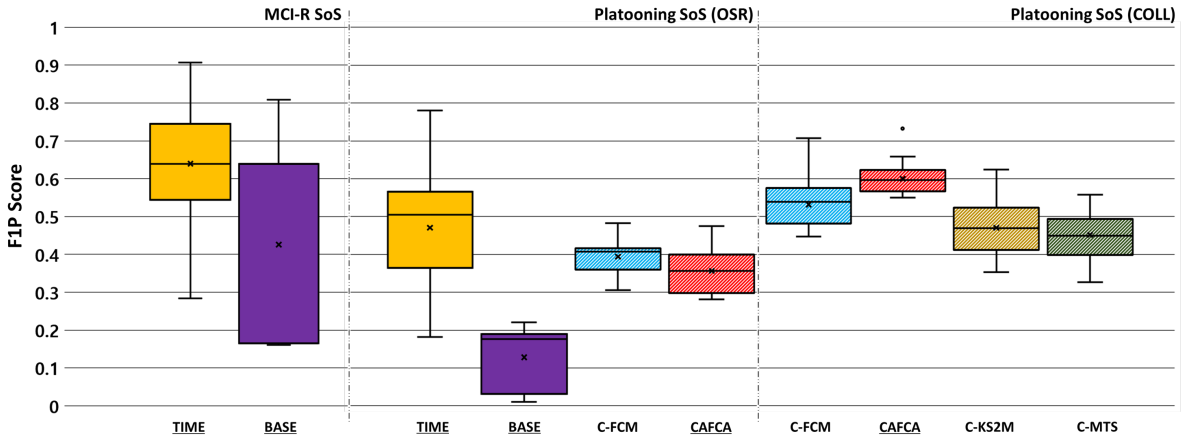


Figure 6.5: Overlapping clustering precision accuracy evaluation results on MCI-R and platooning SoS analysis results

Figure 6.5 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 6.2.1. 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 *TIMESim* and *TIMELen* metrics and the *TIME* clustering process proposed in this study 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 6.2.1.

In COLL analysis results, *CAFCA* depicted the highest F1P score in both of the best and average options of the F1P score. Even *C-KS2M* showed the lowest accuracy in the failure context mining accuracy, *C-KS2M* did not show the lowest overlapping clustering precision than other studies. *C-MTS* showed the lowest F1P scores in both of the best and average options of the F1P score. This indicates that the *C-MTS* and *C-KS2M* approaches have strengths on mining and clustering specific failure classes. Otherwise, *CAFCA* and *C-FCM* presented the higher F1P scores and pattern mining accuracy at the same time and have strengths on mining diverse failure cases than *C-MTS* and *C-KS2M*.

Findings. The proposed *CAFCA* similarity metrics and clustering process showed significantly high overlapping clustering precision than other existing studies.

RQ3. Failure Context Mining Efficiency

Figure 6.6 depicts the log-scale running time for each single execution of the approaches. In OSR analysis results, *TIME* and *BASE* that only utilized the software-software interaction data showed much efficient time on average than *C-FCM* and *CAFCA* approaches that utilized both the software-software interaction data and software-environment interaction data. This trade-off is inevitable for the analysis approaches. However, the *TIME* approach showed highly distributed time efficiency than other studies; thus, by the hyperparameter settings, *TIME* needs much cost than the approaches handling both the software interaction and environment sensor data. Particularly, the worst time efficiency was achieved by the *TIME* approach. This results indicate that for analyzing the failures caused by the software-software interactions, techniques focusing on the interaction data showed the best efficiency on average, but the cost efficiency can be adversely affected by the hyperparameter settings.

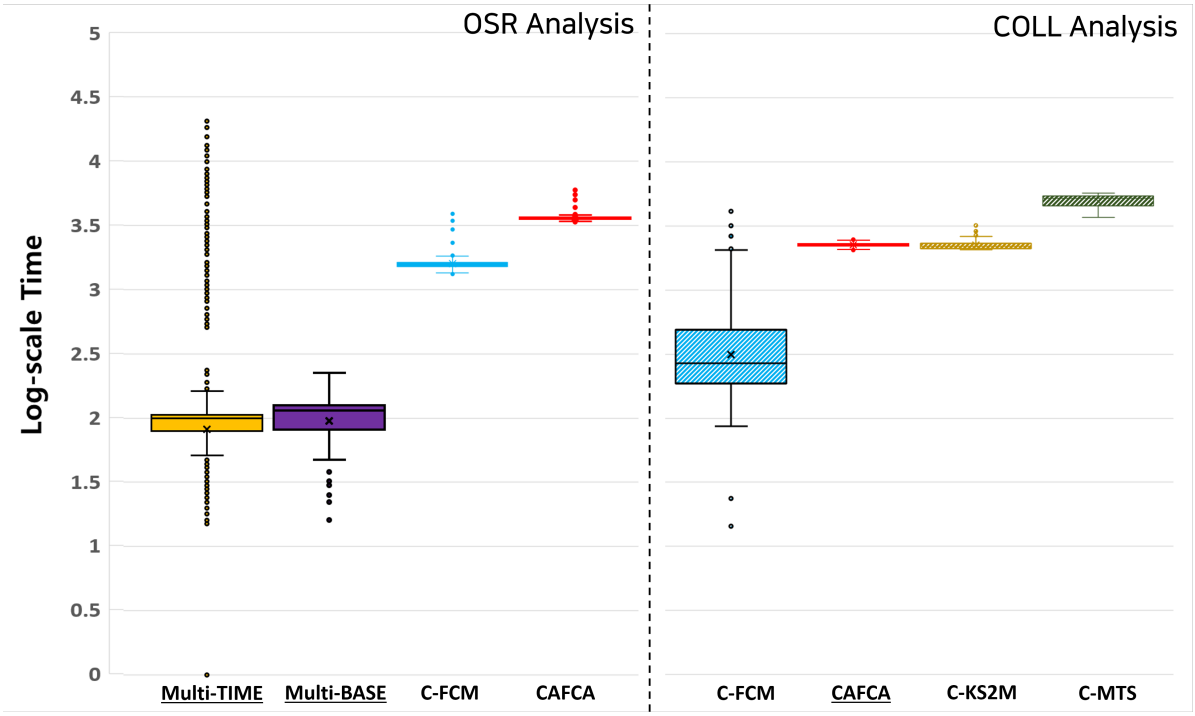


Figure 6.6: Context mining efficiency in log-scale time on OSR and COLL analysis

In COLL analysis, *C-FCM* showed the best efficiency on average than other approaches. This is because the *CAFCA* and *C-KS2M* utilized the pair-wised similarity evaluation in the fuzzy objective function; thus, the *CAFCA* and *C-KS2M* have the time complexity of $O(N^2)$, where N denotes the total number of data element while *C-FCM* has the time complexity of $O(C * N)$, where C denotes the number of clusters. The worst time efficiency was presented by the *C-MTS* approach, because the *MTS* approach compared the all time segment of one data element with the all time segment of the other data element for each similarity calculation.

Findings. The proposed context mining approach in *CAFCA* presented the feasible time efficiency in COLL analysis, but inevitably showed worse efficiency than the techniques focusing on software interaction data in software-software interaction failures.

RQ4. Localization Efficacy

As the last evaluation factor, we checked the feasibility of the pattern-based fault localization by comparing the proposed localization 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.

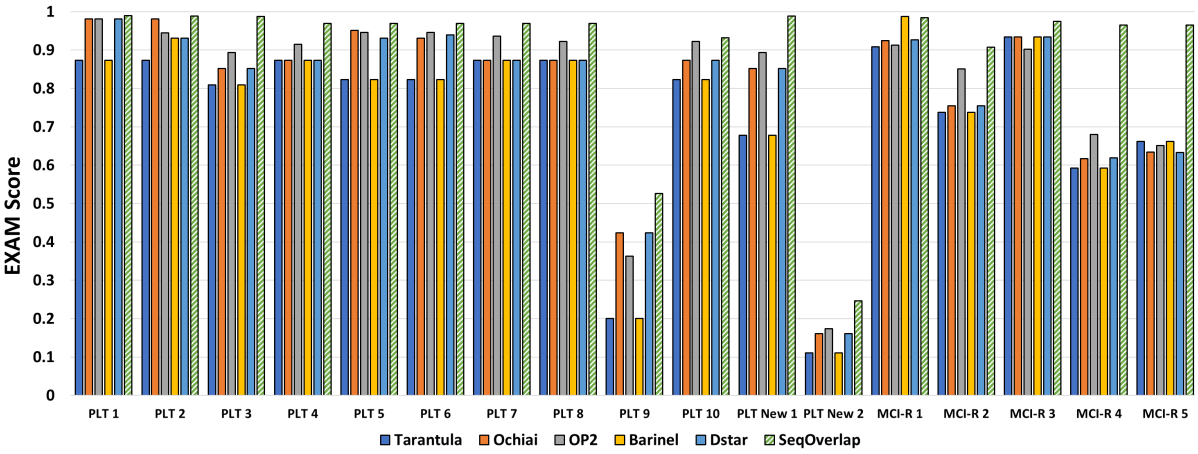


Figure 6.7: EXAM analysis results on bugs causing collaboration failures in platooning and MCI-R SoS

Figure 6.7 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 Figure 6.7, 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 suspiciousness calculation of 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 failure class 9 and new class 2 are lower than other results in Figure 6.7. 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 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

Table 6.2: 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

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 6.2 describes the Top-K results of the SBFL and the proposed localization methods. Top-K scores indicate the accuracy of the localization results by counting the number of faulty statements within K-ranks. Higher values are better for this metric. The proposed *SeqOverlap* method achieved the best Top-K score for every K value in Table 6.2. 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.

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

We explained the primary threats to the validity of the experiment using the internal, construct, and conclusion validity [176]. 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 6.1, 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 6.2. In Figure 6.4, 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 [42]. 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 [10]. We generated thousands of logs for the experiment by injecting the most relevant types of faults and stimuli in the MCI-R collaboration protocol.

We also used the *op_success_rate* property with 80% of the empirical value. Even though the criterion was applied in the recent study of StarPlateS, it is insufficient to prove the justification of the criteria. 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 [139, 146], or verification of a single operation execution [140, 145]. Instead, we attempted to generate cogent properties for the platooning SoS based on international standards, such as ISO26262 [177]. 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 [143, 9, 144]. 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 [17] and modified for application to the simulation logs of the platooning system.

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 [5], using the LCS length and similarity parameters in existing studies [21]. Based on the ranges, there exist 6,560 combinations of the three hyperparameters. As we repeated the experiment thirty times, all the combinations were tested.

However, for the results of RQ1, we inferred that the best hyperparameter option did not return the expected clustering results wherein each pattern contained both occurrence context and failure symptom. Additionally, we show that the cluster patterns 2 and 7 originally depict the same failure scenario. This was caused by the basic characteristics of *LCS*-based algorithms, which are vulnerable to mis-

allocated elements because they extract patterns including all elements, even misallocated ones. One way to improve hyperparameter optimization is to assign individual hyperparameter values to clusters. We expect to improve the clustering results qualitatively and quantitatively by optimizing the hyperparameters for each cluster, but not for the entire cluster.In our study, the difference in precision performance between the best and worst was approximately doubled, and about 6,500 combinations of hyperparameters were considered. Therefore, we tried to suggest a general direction for hyperparameter settings for this proposed technique to increase the practicality of our technique when it is applied to various domains, including the cases where the hyperparameter optimization through ground-truth is impossible, owing to sufficient fault knowledge being not provided. We utilized Functional ANOVA (F-ANOVA) [178], a common hyperparameter importance analysis method used in diverse machine learning techniques [179, 180, 181, 182].

Fig. 6.8 describes the F-ANOVA test results of this study. We found that the *len_threshold*, the threshold for minimum *LCS* pattern length, had the highest importance on the clustering results than the other two hyperparameters. The importance value in the table in Fig. 6.8a shows that the *len_threshold* has significantly larger values than the other hyperparameters, reflecting the influence of the hyperparameters on precision performance. The Fig. 6.8d also shows that *len_threshold* increases along the x-axis, the *F1P* precision values in the y-axis are shown to decrease with an obvious negative correlation.

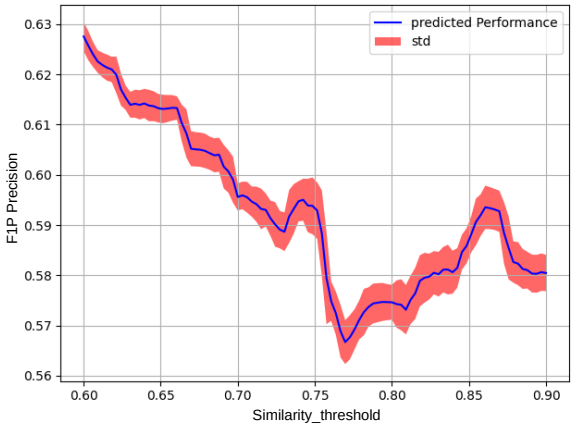
On the other hand, in the case of the other hyperparameter variables in Fig. 6.8b and 6.8c, we confirmed that neither variable showed a consistent trend according to the increase or decrease of the variable values, nor did it have relatively large standard deviation ranges of *F1P* values, which is a disadvantage to the precise estimation of the influence of the variables. We expect that the hyperparameter effect analysis results may serve as a guideline for the reference of hyperparameter settings in further studies.

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 6.2. Because the proposed clustering method is based on the prevalently used subsequent time-series (STS) clustering for time-series data analysis [21, 97, 98], the clustering 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.

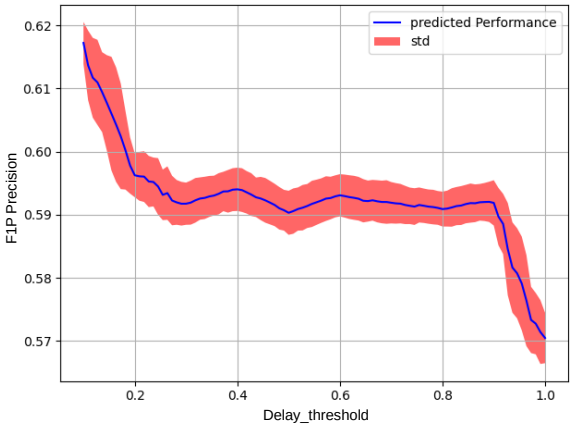
We proposed the fault analysis process that includes faulty interaction pattern mining and code isolation with suspicious rankings to effectively identify the code-level root causes of interaction failures in platooning SoS. Our approach used two inputs: a set of system execution logs and goal property checking results for each log. In this study, we implemented the log parser for the log format of VENTOS and also applied the existing goal property in StarPlateS. We expect that our technique can be applied to general SoS or various complex CPS systems because our technique is not technically domain-specific for platooning SoS and only needs communication logs with *Passed/Failed* tags. To enable the application to general SoS, a log parser will be required to extract interaction message sequences from particular log formats and a goal property checking module is also necessary to attach *Passed/Failed* tags to each log. We plan to apply our technique to other SoS scenarios, such as a smart warehouse system or intelligent transportation system, to show the general utility of the proposed approach.

Hyperparameter	Importance
Similarity_ threshold	0.021
Delay_ threshold	0.039
Len_ threshold	0.867

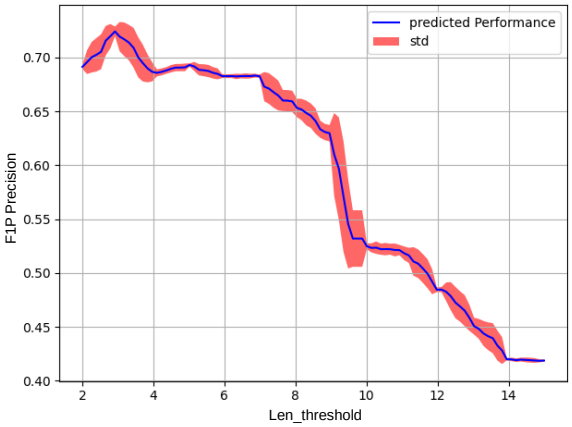
a Importance values of hyperparameters



b Importance graph of *similarity_ threshold*



c Importance graph of *delay_ threshold*



d Importance graph of *len_ threshold*

Figure 6.8: Hyperparameter importance analysis results

Chapter 7. Conclusion

We proposed a context mining-based fault analysis technique for effectively analyzing collaboration failures in CPSoS. We addressed four issues associated with log anomaly detection, time-series data analysis, sequence data analysis [24, 25], and graph mining-based fault localization studies for various systems. Concentrating on the general fault analysis process, including fault detection, understanding on failure occurrence context generation, root cause localization, root cause identification, our investigation of the applicability of existing methods to CPSoS collaboration failures indicated that (1) their data models do not handle both of the discrete and continous data generated in CPSoS execution; (2) they do not cover the major features required to the sequential analysis of the discrete and continuous data; (3) have limitations in terms of identifying multiple failure patterns in a single log; and (4) do not provide an end-to-end solution from failure pattern analysis to the root cause identification.

To address the issues, we proposed a context mining-based fault localization approach composed of *CAFCA* failure context mining and clustering approach, and a pattern-based fault localization method, *SeqOverlap*. In this thesis, we first defined an Interaction and Environment Model (IEM) to handle the discrete message logs and continuous sensor logs in CPSoS. Second, we proposed a Context-Aware Failure pattern-based Clustering Approach (CAFCA) in this study. CAFCA-Longest Common Subsequence (*CAFCA-LCS*) pattern mining algorithm that accurately extracts FII patterns by covering the main features of sequential analysis of the CPSoS logs. Next, the CAFCA contains a Fuzzy-based 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.

In the experiment, the proposed approach achieved the highest pattern mining accuracy compared with existing pattern mining studies and also yielded an overall improvement in clustering precision compared with our previous study. Lastly, the localization method achieved a 24% higher localization accuracy than SBFL methods on average and the highest Top-K score. We expect that the conclusions of this study can enrich the accurate analysis of CPSoS failures. This study is a first attempt at checking the feasibility of pattern-based fault localization for such communication-intensive CPSoS. In the next section, we will describe the remaining works of this dissertation in detail.

Chapter 8. Future Work

8.1 Localization Method for Distributed Multi-Statement Bugs

In the empirical analysis of the platooning system, we found that most of the failures are caused by the logic errors, such as missing logic and incorrect logic in Section 5.1. Faults in integration-level rules and logic are often multi-location as they may span multiple lines of codes [70]. In the evaluation of the localization efficacy, we found that existing fault localization techniques have relatively less effectiveness on distributed multi-statement bugs in Section 6.2.2.

We tried to survey existing studies that focused on localizing the multi-statement bugs. However, existing studies that deal with multi-statement bugs only select the best, average, and worst options in the evaluation process. To the best of our knowledge, we have found a recent study that localizes the multi-statement bugs based on the extended spectrum-based fault localization technique (SBFL) for the faulty paths of the integration rule [70].

To effectively localize the distributed multi-statement bugs in CPSoS, we are going to improve the fault localization metric by extending SBFL that utilizes both of the *Pass/Fail* results. We will extend the spectrum of the fault localization to a sequence of executed code lines from the patterns to accurately localize the distributed multi-statement bugs in CPSoS.

8.2 Localization Method for Implicit Collaboration Code

Even though we proposed a localization method, *SeqOverlap*, for the explicit collaboration protocol code, the localization method for the implicit collaboration is also needed to effectively reduce the cost of identifying the root causes of implicit collaboration failures. The primary difference between the two collaboration implementations is that the implicit collaboration algorithm mostly depends on the environmental sensor data rather than the message-based communication data. For example, in drone swarming SoS, the main factors for calculating the vector velocity of each drone are the current distances with and previous speed vectors of neighbor drones. Because message-based communication is only utilized to exchange the current distances and speed values between the drones, it is infeasible to apply the proposed localization method, *SeqOverlap*, to the implicit collaboration protocol code.

Moreover, unlike the explicit collaboration protocol code that can specify the code execution coverage according to the execution of collaborative operations (e.g., *MERGE_REQ*, *LEAVE_REQ*), in implicit collaboration, code execution coverage of the collaboration algorithm code does not vary greatly depending on what environmental values are given. No matter what environment state is given as inputs, most of the code statement would be executed in the vasalhelyi algorithm described in Section 5.4 because the algorithm calculates V values for deciding the final vector velocities of each drone. Hence, different localization metrics that sufficiently contemplate the characteristics of implicit collaboration are needed.

Bibliography

[1] J. C. Bezdek, R. Ehrlich, W. Full, Fcm: The fuzzy c-means clustering algorithm, Computers & geosciences 10 (2-3) (1984) 191–203.

[2] Z. Shi, Y. Xie, W. Xue, Y. Chen, L. Fu, X. Xu, Smart factory in industry 4.0, Systems Research and Behavioral Science 37 (4) (2020) 607–617.

[3] G. Trencher, Towards the smart city 2.0: Empirical evidence of using smartness as a tool for tackling social challenges, Technological Forecasting and Social Change 142 (2019) 117–128.

[4] A. Zambrano, M. Zambrano, E. Ortiz, X. Calderón, M. Botto-Tobar, An intelligent transportation system: The quito city case study, International Journal on Advanced Science, Engineering and Information Technology 10 (2) (2020) 507–519.

[5] M. Amoozadeh, H. Deng, C.-N. Chuah, H. M. Zhang, D. Ghosal, Platoon Management with Cooperative Adaptive Cruise Control Enabled by VANET, Vehicular communications 2 (2) (2015) 110–123.

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

[7] Z. Lü, Y. Lü, M. Yuan, Z. Wang, 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, 2017, pp. 1–7.

[8] F. Rossi, S. Bandyopadhyay, M. T. Wolf, M. Pavone, Multi-agent algorithms for collective behavior: A structural and application-focused atlas, arXiv preprint arXiv:2103.11067 (2021).

[9] P. E. Group, Enabling safe multi-brand platooning for europe (ensemble), [Online; accessed 26-May-2022].
URL https://platooningensemble.eu/

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

[11] E. Soria, F. Schiano, D. Floreano, Swarmlab: A matlab drone swarm simulator, in: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2020, pp. 8005–8011.

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

[13] M. Augustine, O. P. Yadav, R. Jain, A. Rathore, Cognitive map-based system modeling for identifying interaction failure modes, Research in Engineering Design 23 (2) (2012) 105–124.

[14] M. Landauer, M. Wurzenberger, F. Skopik, G. Settanni, P. Filzmoser, Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection, computers & security 79 (2018) 94–116.

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

[16] A. Amar, P. C. Rigby, 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, 2019, pp. 140–151.

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

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

[19] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, et al., 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, 2019, pp. 807–817.

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

[21] G. Soleimany, M. Abessi, A new similarity measure for time series data mining based on longest common subsequence, American Journal of Data Mining and Knowledge Discovery 4 (1) (2019) 32.

[22] M. Y. Choong, L. Angeline, R. K. Y. Chin, K. B. Yeo, K. T. K. Teo, 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, 2017, pp. 74–79.

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

[24] S. Hyun, J. Song, S. Shin, Y.-M. Baek, D.-H. Bae, 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, 2020, pp. 326–335.

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

[26] M. Gaber Abd El-Wahab, A. E. Aboutabl, W. M. EL Behaidy, Graph mining for software fault localization: An edge ranking based approach, Journal of Communications Software and Systems 13 (4) (2017) 178–188.

[27] S. Parsa, S. A. Naree, N. E. Koopaei, Software fault localization via mining execution graphs, in: International Conference on Computational Science and Its Applications, Springer, 2011, pp. 610–623.

[28] J. Qian, X. Ju, X. Chen, H. Shen, Y. Shen, Agfl: A graph convolutional neural network-based method for fault localization, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2021, pp. 672–680.

[29] H. Zhong, H. Mei, Learning a graph-based classifier for fault localization, Science China Information Sciences 63 (6) (2020) 1–22.

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

[31] J. Chu, T. Yu, J. Huffman Hayes, X. Han, Y. Zhao, Effective fault localization and context-aware debugging for concurrent programs, Software Testing, Verification and Reliability 32 (1) (2022) e1797.

[32] H. A. de Souza, D. Mutti, M. L. Chaim, F. Kon, Contextualizing spectrum-based fault localization, Information and Software Technology 94 (2018) 245–261.

[33] X. Yu, J. Liu, Z. J. Yang, X. Liu, X. Yin, S. Yi, Bayesian network based program dependence graph for fault localization, in: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2016, pp. 181–188.

[34] J. Zhang, R. Xie, W. Ye, Y. Zhang, S. Zhang, Exploiting code knowledge graph for bug localization via bi-directional attention, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 219–229.

[35] H. He, J. Ren, G. Zhao, H. He, Enhancing spectrum-based fault localization using fault influence propagation, IEEE Access 8 (2020) 18497–18513.

[36] H. Gao, B. Cheng, J. Wang, K. Li, J. Zhao, D. Li, Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment, IEEE Transactions on Industrial Informatics 14 (9) (2018) 4224–4231.

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

[38] M. Muhammad, G. A. Safdar, Survey on existing authentication issues for cellular-assisted v2x communication, Vehicular Communications 12 (2018) 50–65.

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

[40] T. J.-M. Meango, M.-S. Ouali, Failure interaction model based on extreme shock and markov processes, Reliability Engineering & System Safety 197 (2020) 106827.

[41] H. Jiang, X. Li, Z. Yang, J. Xuan, 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, 2017, pp. 712–723.

[42] S. Hyun, L. Liu, H. Kim, E. Cho, D.-H. Bae, An empirical study of reliability analysis for platooning system-of-systems, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2021, pp. 506–515.

[43] D. Drozdov, V. Dubinin, S. Patil, V. Vyatkin, A formal model of iec 61499-based industrial automation architecture supporting time-aware computations, IEEE Open Journal of the Industrial Electronics Society 2 (2021) 169–183.

[44] Y. Zhou, X. Gong, B. Li, M. Zhu, A framework for cps modeling and verification based on dl, in: 2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS), IEEE, 2018, pp. 173–179.

[45] K. Halba, E. Griffor, A. Lbath, A. Dahbura, A framework for the composition of iot and cps capabilities, in: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2021, pp. 1265–1272.

[46] D. Calvaresi, Y. Dicente Cid, M. Marinoni, A. F. Dragoni, A. Najjar, M. Schumacher, Real-time multi-agent systems: rationality, formal model, and empirical results, Autonomous Agents and Multi-Agent Systems 35 (1) (2021) 1–37.

[47] Y. Zhao, J. Liu, E. A. Lee, A programming model for time-synchronized distributed real-time systems, in: 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07), IEEE, 2007, pp. 259–268.

[48] C. Sun, L. Zhang, Design and modeling of intelligent home security monitoring system based on cps, in: 2021 IEEE 12th International Conference on Software Engineering and Service Science (ICSESS), IEEE, 2021, pp. 186–189.

[49] K. H. Lee, J. H. Hong, T. G. Kim, System of systems approach to formal modeling of cps for simulation-based analysis, Etri Journal 37 (1) (2015) 175–185.

[50] N. Chen, S. Geng, L. Li, Modeling and verification of cps based on uncertain hybrid timed automaton, in: 2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), IEEE, 2021, pp. 971–978.

[51] A. Bouheroum, D. Benmerzoug, S. M. Hemam, F. Belala, From ca-brs to bpmn: Formal approach for modeling adaptive security in cyber-physical systems., in: TACC, 2021, pp. 149–163.

[52] S. Luna, A. Lopes, H. Y. S. Tao, F. Zapata, R. Pineda, Integration, verification, validation, test, and evaluation (ivvt&e) framework for system of systems (sos), Procedia Computer Science 20 (2013) 298–305.

[53] N. Akhtar, S. Khan, Formal architecture and verification of a smart flood monitoring system-of-systems., Int. Arab J. Inf. Technol. 16 (2) (2019) 211–216.

[54] A. Rehman, N. Akhtar, O. H. Alhazmi, Formal modeling, proving, and model checking of a flood warning, monitoring, and rescue system-of-systems, Scientific Programming 2021 (2021).

[55] J. Fitzgerald, J. Bryans, R. Payne, A formal model-based approach to engineering systems-of-systems, in: Working Conference on Virtual Enterprises, Springer, 2012, pp. 53–62.

[56] R. Payne, J. Bryans, J. Fitzgerald, S. Riddle, Interface specification for system-of-systems architectures, in: 2012 7th International Conference on System of Systems Engineering (SoSE), IEEE, 2012, pp. 567–572.

[57] C. Wiecher, J. Greenyer, C. Wolff, H. Anacker, R. Dumitrescu, Iterative and scenario-based requirements specification in a system of systems context, in: International Working Conference on Requirements Engineering: Foundation for Software Quality, Springer, 2021, pp. 165–181.

[58] J. Bryans, R. Payne, J. Holt, S. Perry, Semi-formal and formal interface specification for system of systems architecture, in: 2013 IEEE International Systems Conference (SysCon), IEEE, 2013, pp. 612–619.

[59] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, D. Ding, Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study, IEEE Transactions on Software Engineering 47 (2) (2018) 243–260.

[60] Y.-M. Baek, J. Song, Y.-J. Shin, S. Park, D.-H. Bae, A meta-model for representing system-of-systems ontologies, in: 2018 IEEE/ACM 6th International Workshop on Software Engineering for Systems-of-Systems (SESoS), IEEE, 2018, pp. 1–7.

[61] A. A. Shchurov, A formal model of distributed systems for test generation missions, arXiv preprint arXiv:1410.1729 (2014).

[62] V. Srivastava, R. S. Pandey, A reward based formal model for distributed software defined networks, Wireless Personal Communications 116 (1) (2021) 691–707.

[63] Y. Kubiuk, K. Kharchenko, Design and implementation of the distributed system using an orchestrator based on the data flow paradigm, Technology audit and production reserves 3 (2) (2020) 53.

[64] L. Bin, W. Xingmin, S. Jun, Reliability evaluation method of dds-based distributed system, in: 2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE), IEEE, 2019, pp. 2028–2033.

[65] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, M. D. Ernst, Visualizing distributed system executions, ACM Transactions on Software Engineering and Methodology (TOSEM) 29 (2) (2020) 1–38.

[66] F. Neves, N. Machado, R. Vilaça, J. Pereira, Horus: Non-intrusive causal analysis of distributed systems logs, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2021, pp. 212–223.

[67] G. Vásárhelyi, C. Virágh, G. Somorjai, N. Tarcai, T. Szörényi, T. Nepusz, T. Vicsek, Outdoor flocking and formation flight with autonomous aerial robots, in: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2014, pp. 3866–3873.

[68] G. Vásárhelyi, C. Virágh, G. Somorjai, T. Nepusz, A. E. Eiben, T. Vicsek, Optimized flocking of autonomous drones in confined environments, Science Robotics 3 (20) (2018) eaat3536.

[69] C. Virágh, G. Vásárhelyi, N. Tarcai, T. Szörényi, G. Somorjai, T. Nepusz, T. Vicsek, Flocking algorithm for autonomous flying robots, Bioinspiration & biomimetics 9 (2) (2014) 025012.

[70] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, T. Stifter, Automated repair of feature interaction failures in automated driving systems, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 88–100.

[71] G. Asaamoning, P. Mendes, D. Rosário, E. Cerqueira, Drone swarms as networked control systems by integration of networking and computing, Sensors 21 (8) (2021) 2642.

[72] M. Schranz, M. Umlauft, M. Sende, W. Elmenreich, Swarm robotic behaviors and current applications, Frontiers in Robotics and AI 7 (2020) 36.

[73] F. Saffre, H. Hildmann, H. Karvonen, The design challenges of drone swarm control, in: International Conference on Human-Computer Interaction, Springer, 2021, pp. 408–426.

[74] L. Marsh, C. Onof, Stigmergic epistemology, stigmergic cognition, Cognitive Systems Research 9 (1-2) (2008) 136–149.

[75] T. G. Lewis, Cognitive stigmergy: A study of emergence in small-group social networks, Cognitive Systems Research 21 (2013) 7–21.

[76] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, et al., Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study, Proceedings of the national academy of sciences 105 (4) (2008) 1232–1237.

[77] J. Dias-Ferreira, Bio-inspired self-organising architecture for cyber-physical manufacturing systems, Ph.D. thesis, Kungliga Tekniska högskolan (2016).

[78] A. Forestiero, G. Spezzano, D. Talia, Swarm-based algorithms for decentralized clustering and resource discovery in grids, Ph.D. thesis (2012).

[79] D. J. Brooks, A human-centric approach to autonomous robot failures, Ph.D. thesis, University of Massachusetts Lowell (2017).

[80] A. Sutcliffe, G. Rugg, A taxonomy of error types for failure analysis and risk assessment, International Journal of Human-Computer Interaction 10 (4) (1998) 381–405.

[81] J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, R. Laemmel, E. Meijer, et al., Wes: Agent-based user interaction simulation on real infrastructure, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020, pp. 276–284.

[82] V. Vijayan, S. K. Chaturvedi, R. Chandra, A failure interaction model for multicomponent repairable systems, Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability 234 (3) (2020) 470–486.

[83] L. Yang, Y. Zhao, X. Ma, Group maintenance scheduling for two-component systems with failure interaction, Applied Mathematical Modelling 71 (2019) 118–137.

[84] L. Li, M. Lu, T. Gu, Extracting interaction-related failure indicators for online detection and prediction of content failures, in: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2018, pp. 278–285.

[85] A. Avizienis, J.-C. Laprie, B. Randell, Fundamental concepts of dependability, Department of Computing Science Technical Report Series (2001).

[86] L. Xu, Y. Chen, F. Briand, F. Zhou, M. Givanni, Reliability measurement for multistate manufacturing systems with failure interaction, Procedia CIRP 63 (2017) 242–247.

[87] C. Parnin, A. Orso, Are automated debugging techniques actually helping programmers?, in: Proceedings of the 2011 international symposium on software testing and analysis, 2011, pp. 199–209.

[88] K.-M. Seo, K.-P. Park, Interface data modeling to detect and diagnose intersystem faults for designing and integrating system of systems, Complexity 2018 (2018).

[89] S. Wadhai, M. Wadekar, S. Junankar, C. Rathod, V. Hingane, A. A. Zade, Detection of power grid synchronization failure by sensing bad voltage and frequency (2017).

[90] T. Fu, Studies on memory consistency and synchronization: failure detection in parallel programs, Ph.D. thesis, Concordia University (1998).

[91] X. Q. Tang, Q. Li, G. Lu, H. Xiong, F. He, An application-level method of arbitrary synchronization failure detection in ttethernet networks, Journal of Circuits, Systems and Computers 29 (07) (2020) 2050102.

[92] D. Deng, W. Zhang, S. Lu, Efficient concurrency-bug detection across inputs, Acm Sigplan Notices 48 (10) (2013) 785–802.

[93] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, C. Zhai, Bug characteristics in open source software, Empirical software engineering 19 (6) (2014) 1665–1705.

[94] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, Concurrency bugs in open source software: a case study, Journal of Internet Services and Applications 8 (1) (2017) 1–15.

[95] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, T. Stifter, Testing autonomous cars for feature interaction failures using many-objective search, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2018, pp. 143–154.

[96] F. Wu, P. Anchuri, Z. Li, Structural event detection from log messages, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 1175–1184.

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

[98] S. Rodpongpun, V. Niennattrakul, C. A. Ratanamahatana, Selective subsequence time series clustering, Knowledge-Based Systems 35 (2012) 361–368.

[99] K. Zhou, S. Yang, Z. Shao, Household monthly electricity consumption pattern mining: A fuzzy clustering-based model and a case study, Journal of cleaner production 141 (2017) 900–908.

[100] D. Zhang, K. Lee, I. Lee, Hierarchical trajectory clustering for spatio-temporal periodic pattern mining, Expert Systems with Applications 92 (2018) 1–11.

[101] D. Hallac, S. Vare, S. Boyd, J. Leskovec, Toeplitz inverse covariance-based clustering of multivariate time series data, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 215–223.

[102] B. G. Sürmeli, M. B. Tümer, Multivariate time series clustering and its application in industrial systems, Cybernetics and Systems 51 (3) (2020) 315–334.

[103] H. Li, Multivariate time series clustering based on common principal component analysis, Neurocomputing 349 (2019) 239–247.

[104] P. D'Urso, L. De Giovanni, R. Massari, Robust fuzzy clustering of multivariate time trajectories, International Journal of Approximate Reasoning 99 (2018) 12–38.

[105] P. D'Urso, L. De Giovanni, R. Massari, Trimmed fuzzy clustering of financial time series based on dynamic time warping, Annals of operations research 299 (1) (2021) 1379–1395.

[106] P. D'Urso, E. A. Maharaj, A. M. Alonso, Fuzzy clustering of time series using extremes, Fuzzy Sets and Systems 318 (2017) 56–79.

[107] B. Huynh, C. Trinh, H. Huynh, T.-T. Van, B. Vo, V. Snasel, An efficient approach for mining sequential patterns using multiple threads on very large databases, Engineering Applications of Artificial Intelligence 74 (2018) 242–251.

[108] D. Maylawati, H. Aulawi, M. Ramdhani, The concept of sequential pattern mining for text, in: IOP Conference Series: Materials Science and Engineering, Vol. 434, IOP Publishing, 2018, p. 012042.

[109] Y. Cai, H. Yun, J. Wang, L. Qiao, J. Palsberg, 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, 2021, pp. 255–267.

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

[111] B. Liu, J. Huang, D4: fast concurrency debugging with parallel differential analysis, ACM SIGPLAN Notices 53 (4) (2018) 359–373.

[112] S. Behere, M. Törngren, Systems engineering and architecting for intelligent autonomous systems, in: Automated Driving, Springer, 2017, pp. 313–351.

[113] A. L. Juarez-Dominguez, N. A. Day, J. J. Joyce, Modelling feature interactions in the automotive domain, in: Proceedings of the 2008 international workshop on Models in software engineering, 2008, pp. 45–50.

[114] M. Weiss, B. Esfandiari, Y. Luo, Towards a classification of web service feature interactions, Computer networks 51 (2) (2007) 359–381.

[115] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, H. Velthuijsen, A feature-interaction benchmark for in and beyond, IEEE Communications Magazine 31 (3) (1993) 64–69.

[116] D. O. Keck, P. J. Kuehn, The feature and service interaction problem in telecommunications systems: A survey, IEEE transactions on software engineering 24 (10) (1998) 779–796.

[117] M. Y. Choong, R. K. Y. Chin, K. B. Yeo, K. T. K. Teo, Trajectory pattern mining via clustering based on similarity function for transportation surveillance, International Journal of Simulation-Systems, Science & Technology 17 (34) (2016) 19–1.

[118] Y. Harada, Y. Yamagata, O. Mizuno, E.-H. Choi, Log-based anomaly detection of cps using a statistical method, in: 2017 8th International Workshop on Empirical Software Engineering in Practice (IWESEP), IEEE, 2017, pp. 1–6.

[119] C. H. Fontes, O. Pereira, Pattern recognition in multivariate time series–a case study applied to fault detection in a gas turbine, Engineering Applications of Artificial Intelligence 49 (2016) 10–18.

[120] H. Huang, S. Yoo, Failure analysis on multivariate time-series data given uncertain labels, Tech. rep., Brookhaven National Lab.(BNL), Upton, NY (United States) (2019).

[121] Y.-J. Lee, D.-Y. Kim, M.-S. Hwang, Y.-S. Cheong, A study on data pre-filtering methods for fault diagnosis, Korean Journal of Computational Design and Engineering 17 (2) (2012) 97–110.

[122] F. Serdio, E. Lughofer, K. Pichler, T. Buchegger, M. Pichler, H. Efendic, Fault detection in multi-sensor networks based on multivariate time-series models and orthogonal transformations, Information Fusion 20 (2014) 272–291.

[123] A. Singhal, D. E. Seborg, Pattern matching in multivariate time series databases using a moving-window approach, Industrial & engineering chemistry research 41 (16) (2002) 3822–3838.

[124] K. Yang, C. Shahabi, A pca-based similarity measure for multivariate time series, in: Proceedings of the 2nd ACM international workshop on Multimedia databases, 2004, pp. 65–74.

[125] S. Li, J. Wen, Application of pattern matching method for detecting faults in air handling unit system, Automation in Construction 43 (2014) 49–58.

[126] W. Krzanowski, Between-groups comparison of principal components, Journal of the American Statistical Association 74 (367) (1979) 703–707.

[127] T. Y. Sing, S. E. B. Siraj, R. Raguraman, P. N. Marimuthu, K. Nithiyananthan, Cosine similarity cluster analysis model based effective power systems fault identification, Int. J. Adv. Appl. Sci 4 (1) (2017) 123–129.

[128] L. G. B. Ruiz, M. Pegalajar, R. Arcucci, M. Molina-Solana, A time-series clustering methodology for knowledge extraction in energy consumption data, Expert Systems with Applications 160 (2020) 113731.

[129] S. Aghabozorgi, A. S. Shirkhorshidi, T. Y. Wah, Time-series clustering–a decade review, Information Systems 53 (2015) 16–38.

[130] V. Hautamaki, P. Nykanen, P. Franti, Time-series clustering by approximate prototypes, in: 2008 19th International conference on pattern recognition, IEEE, 2008, pp. 1–4.

[131] F. Gullo, G. Ponti, A. Tagarelli, G. Tradigo, P. Veltri, A time series approach for clustering mass spectrometry data, Journal of Computational Science 3 (5) (2012) 344–355.

[132] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, E. Keogh, Experimental comparison of representation methods and distance measures for time series data, Data Mining and Knowledge Discovery 26 (2) (2013) 275–309.

[133] C. S. Möller-Levet, F. Klawonn, K.-H. Cho, O. Wolkenhauer, Fuzzy clustering of short time-series and unevenly distributed sampling points, in: International symposium on intelligent data analysis, Springer, 2003, pp. 330–340.

[134] G. Jiang, W. Wang, W. Zhang, A novel distance measure for time series: Maximum shifting correlation distance, Pattern Recognition Letters 117 (2019) 58–65.

[135] Y. Xiong, D.-Y. Yeung, Mixtures of arma models for model-based time series clustering, in: 2002 IEEE International Conference on Data Mining, 2002. Proceedings., IEEE, 2002, pp. 717–720.

[136] S. Salvador, P. Chan, Toward accurate dynamic time warping in linear time and space, Intelligent Data Analysis 11 (5) (2007) 561–580.

[137] P. D'Urso, L. De Giovanni, Temporal self-organizing maps for telecommunications market segmentation, Neurocomputing 71 (13-15) (2008) 2880–2892.

[138] P. D'Urso, L. De Giovanni, R. Massari, Garch-based robust clustering of time series, Fuzzy Sets and Systems 305 (2016) 1–28.

[139] B. Vieira, R. Severino, A. Koubâa, E. Tovar, Towards a Realistic Simulation Framework for Vehicular Platooning Applications, arXiv preprint arXiv:1904.02994 (2019).

[140] M. Kamali, S. Linker, M. Fisher, Modular Verification of Vehicle Platooning with respect to Decisions, Space and Time, in: International Workshop on Formal Techniques for Safety-Critical Systems, Springer, 2018, pp. 18–36.

[141] Aaron Blasdel, Robot Operating System (ROS), [Online; accessed 2-July-2019].
URL https://www.ros.org/

[142] Math Works, Simulink, [Online; accessed 2-July-2019].
URL https://www.mathworks.com/products/simulink.html

[143] M. Elgharbawy, A Big Testing Framework for Automated Truck Driving, Urban transportation and construction 4 (1) (2019) e27–e27.

[144] S. Achrifi, Coverage Verification Framework for ADAS Models (March 2017).

[145] P. Mallozzi, M. Sciancalepore, P. Pelliccione, Formal Verification of the On-the-fly Vehicle Platooning Protocol, in: International Workshop on Software Engineering for Resilient Systems, Springer, 2016, pp. 62–75.

[146] K. Meinke, Learning-based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study, in: European Workshop on Performance Engineering, Springer, 2017, pp. 135–151.

[147] Math Works, VnV Toolbox, [Online; accessed 2-July-2019].
URL https://www.mathworks.com/products/transitioned/simverification.html

[148] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740.

[149] Y.-J. Shin, S. Hyun, Y.-M. Baek, 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), IEEE, 2019, pp. 358–363.

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

[151] N. C. Jones, P. A. Pevzner, P. Pevzner, An introduction to bioinformatics algorithms, MIT press, 2004.

[152] L. A. Zadeh, Fuzzy logic, Computer 21 (4) (1988) 83–93.

[153] X. Wang, Y. Wang, L. Wang, Improving fuzzy c-means clustering based on feature-weight learning, Pattern recognition letters 25 (10) (2004) 1123–1132.

[154] A. M. Anter, A. E. Hassenian, D. Oliva, An improved fast fuzzy c-means using crow search optimization algorithm for crop identification in agricultural, Expert Systems with Applications 118 (2019) 340–354.

[155] Z. He, S. Zhang, F. Gu, J. Wu, Mining conditional discriminative sequential patterns, Information Sciences 478 (2019) 524–539.

[156] Z. He, S. Zhang, J. Wu, Significance-based discriminative sequential pattern mining, Expert Systems with Applications 122 (2019) 54–64.

[157] B. Tripathy, et al., Fuzzy clustering of sequential data, International Journal of Intelligent Systems and Applications 11 (1) (2019) 43.

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

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

[160] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on software engineering and methodology (TOSEM) 20 (3) (2011) 1–32.

[161] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Transactions on Reliability 63 (1) (2013) 290–308.

[162] S. Honig, T. Oron-Gilad, Understanding and resolving failures in human-robot interaction: Literature review and model development, Frontiers in psychology 9 (2018) 861.

[163] P. Schnoebelen, The Complexity of Temporal Logic Model Checking., Advances in modal logic 4 (393-436) (2002) 35.

[164] A. Wald, Sequential Tests of Statistical Hypotheses, The annals of mathematical statistics 16 (2) (1945) 117–186.

[165] M. Y. Arafat, S. Hoque, S. Xu, D. M. Farid, Machine learning for mining imbalanced data, IAENG International Journal of Computer Science 46 (2) (2019) 332–348.

[166] G. Figueroa, Y.-S. Chen, N. Avila, C.-C. Chu, Improved practices in machine learning algorithms for ntl detection with imbalanced data, in: 2017 IEEE Power & Energy Society General Meeting, IEEE, 2017, pp. 1–5.

[167] R. Olfati-Saber, R. M. Murray, Distributed cooperative control of multiple vehicle formations using structural potential functions, IFAC Proceedings Volumes 35 (1) (2002) 495–500, 15th IFAC World Congress. doi:https://doi.org/10.3182/20020721-6-ES-1901.00244.
URL https://www.sciencedirect.com/science/article/pii/S1474667015386651

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

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

[170] A. Lutov, M. Khayati, P. Cudré-Mauroux, 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, 2019, pp. 1–8.

[171] E. Wong, T. Wei, Y. Qi, L. Zhao, A crosstab-based statistical method for effective fault localization, in: 2008 1st international conference on software testing, verification, and validation, IEEE, 2008, pp. 42–51.

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

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

[174] F. Steimann, M. Frenkel, R. Abreu, 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, 2013, pp. 314–324.

[175] M. J. Zaki, Spade: An efficient algorithm for mining frequent sequences, Machine learning 42 (1) (2001) 31–60.

[176] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[177] O. I. de Normalización, ISO 26262: Road Vehicles : Functional Safety, ISO, 2011.
URL https://books.google.co.kr/books?id=3gcAjwEACAAJ

[178] F. Hutter, H. Hoos, K. Leyton-Brown, An efficient approach for assessing hyperparameter importance, in: International conference on machine learning, PMLR, 2014, pp. 754–762.

[179] M. Feurer, F. Hutter, Hyperparameter optimization, in: Automated machine learning, Springer, Cham, 2019, pp. 3–33.

[180] J. Vanschoren, Meta-learning: A survey, arXiv preprint arXiv:1810.03548 (2018).

[181] N. Xie, G. Ras, M. van Gerven, D. Doran, Explainable deep learning: A field guide for the uninitiated, arXiv preprint arXiv:2004.14545 (2020).

[182] N. Reimers, I. Gurevych, Reporting score distributions makes a difference: Performance study of lstm-networks for sequence tagging, arXiv preprint arXiv:1707.09861 (2017).

Acknowledgments in Korean

카이스트에서 어느덧 5번째 겨울을 맞이하고 있습니다. 박사과정이라는 큰 산에 오르게되어 감회가 새롭고, 앞으로 나아가야할 넓은 바다와 다른 산들을 보게되니 가슴이 떨립니다. 먼저, 많이 미흡했던 저희를 석사과정때부터 한 명의 연구원으로써 존중해주시고, 연구의 큰 방향에 대한 참된 조언을 해주신 배두환 지도교수님께 깊은 감사를 드립니다. 배두환 교수님께서 해주셨던 많은 격려와 조언이 없었더라면, 저는 이 자리에 없었을 거라고 생각합니다. 함께 연구를 진행하며 연구의 세부 사항들에 대해 조언해주시고 특히나 논문 작성과정에서 많은 도움을 주셨던 지은경 교수님께도 감사드립니다. 박사 과정으로서의 문제를 정의하는 자질을 기르게 도와주신 고인영 교수님, 기법/실험적인 측면에서 실질적인 조언을 주셨던 유신 교수님, 기업적 측면에서 연구의 가치에 대해 조언해주신 민상윤 대표님께도 다시 한번 감사의 인사 드립니다.

학위과정동안 연구실에서 동고동락해온 SE랩 선후배분들께도 감사드립니다. 마치 부모의 마음으로 초창기에 문장단위까지 논문을 봐주셨던 지영누나, 영민이형, 아무것도 모를때 참된 조언을 해주셨던 동환이형께 감사전합니다. 함께 있을때 즐겁고 믿을 수 있는 동료이자 선배였던 용준이, 수민이형, Zele 감사합니다. 모두가 성공가도를 달려 국내/외에서 좋은 소식 전해주시기를 기원합니다. 석사를 졸업하고 이미 사회에서 멋진 일을 하고있거나, 앞으로 하게 될 유림누나, 승철이, 성진이, May, Lingjun, Anthony, 미현, 한수 모두에게 감사의 인사 전하고 싶습니다. 그리고 박사과정을 하고 있는 든든한 은호에게도 진심어린 응원과 감사를 전합니다. 워낙 꼼꼼하고 잘 하는 친구라 좋은 결과로 졸업할 것을 믿어 의심치 않습니다. 다른 선후배님들도 항상 응원하고 앞으로도 가시는 길에 행운과 좋은 일이 가득하기를 기원합니다.

고된 대학원 생활에서 많은 공감을 해주고 웃음을 주었던 친구들에게도 감사 인사 전합니다. 고등학교 친구 일융이, 태현이, 종언이가 카이스트에 함께 있었기에 더 행복했습니다. 어느덧 18년째 알고지낸 고향친구들 상오, 상우, 승민이, 용원이, 지예가 있어서 평택에 갈때마다 즐겁고 힘이되었습니다. 답십리에서 같이 지내며 아무것도 몰랐던 저를 넓은 마음으로 받아주셨던 원혁이형, 승준이형, 명성이형, 상헌이형에게도 감사드립니다. 서울에서 같이 놀아주며 힘을 주었던 한양대 APEX 동기들 민수, 수종이, 한솔이, 혁진이, 효찬이, 시현이, 준석이, 필무 감사하고 항상 응원합니다. 카이스트에서 대학원 생활을 하며 더욱 친해지고 새로 알게되었던 달구지 친구들 원일이, 재영이, 민경누나에게도 감사의 인사 전합니다. 앞으로도 오랜 시간 동안 함께하고 서로 추억을 나누며 웃을 수 있기를 바랍니다. 그리고, 대학원이라는 인생에서 손꼽힐 고된 시간에서 가장 가까이 있어주었던 나의 연인 영경이에게도 감사의 마음 전합니다. 덕분에 행복할때 웃고, 슬플때 울면서 박사학위과정을 아름다웠던 추억으로 기억할 수 있게 됐습니다. 이제 학교를 떠나 사회에 나가게 된 후에도 더 즐겁고 솔직하게, 행복한 시간 오래도록 함께할 수 있을 것이라 믿어 의심치 않습니다.

마지막으로, 사랑하는 가족에게 감사를 전합니다. 언제나 저의 든든한 버팀목이자 그늘이 되어주셨기에 제가 대학원에 진학하고 학위 공부에 집중할 수 있었습니다. 일체유심조라는 명언을 수십년간 직접 행동으로 보여주시며 지금까지 사회에서 큰 역할을 하고 계시는 아버지, 항상 저희를 걱정하시고 챙겨주시고 재테크에도 큰 재능을 보여주시며 지금도 선생님으로 저희를 뒷바라지 해주시는 어머니 정말 감사합니다. 지금까지 부모님께서 저에게 주셨던 은혜 잊지 않고, 평생 보답하며 살겠습니다. 지금은 공부하느라 고생이 많지만 앞으로는 세계적인 셰프로써 이름을 떨칠 동생 상호에게도 감사와 응원을 전합니다. 매사 진지한 저와는 다른 집안의 분위기 메이커인 동생이 있었기에 참 웃을 일과 추억이 많았습니다. 항상 응원해주시고 좋은 조언 많이 해주셨던 큰이모, 큰 이모부, 작은이모, 작은이모부, 사촌이자 대표님이 된 기홍이형과 충북대 로스쿨 교수님이신 형수님께도 감사의 인사드립니다. 그리고 오랜기간 함께 살았지만 지금은 더 높은 곳으로 떠나계신 할아버지, 할머니께도 감사의 인사 올립니다. 제가 갓난아기일때부터 돌봐주시고, 십수년간 한 지붕아래 함께한 가족이었던 할아버지, 할머니께 그래도 좋은 소식 전해드린것 같아 다행이라고 생각하며 항상 사랑한다고 기도 드립니다. 제가 우리 가족의 일원이라 너무 행복하고 감사하게 생각하며, 일평생 부모님을 위하고 동생 상호와도 서로 존중하며 지내겠습니다. 항상 사랑하고 감사드립니다.

Curriculum Vitae in Korean

이 름: 현 상 원

생 년 월 일: 1994년 12월 10일

학 력

2010. 3. – 2013. 2. 공주 한일고등학교

2013. 3. – 2018. 2. 한양대학교 컴퓨터공학부 소프트웨어전공 (학사)

2018. 3. – 2023. 2. 한국과학기술원 전산학부 (석박통합과정)

연 구 업 적

1. **Sangwon Hyun**, Jiyoung Song, Eunkyoung Jee, and Doo-Hwan Bae, " Timed Pattern-based Analysis of Collaboration Failures in System-of-Systems", Available at SSRN 4197677.
2. Jiyoung Song, Jeehoon Kang, **Sangwon Hyun**, Eunkyoung Jee, and Doo-Hwan Bae, "Continuous verification of system of systems with collaborative MAPE-K pattern and probability model slicing", Information and Software Technology 147 (2022) 106904.
3. Esther Cho, Yong-Jun Shin, **Sangwon Hyun**, Hansu Kim, and Doo-Hwan Bae, "Automatic Generation of Metamorphic Relations for a Cyber-Physical System-of-Systems Using Genetic Algorithm", Asia-Pacific Software Engineering Conference (APSEC), 2022, 29th, Online.
4. **Sangwon Hyun**, Lingjun Liu, Hansu Kim, Esther Cho, and Doo-Hwan Bae, "An Empirical Study of Reliability Analysis for Platooning System-of-Systems", International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2021, pp. 506–515.
5. Yong-Jun Shin, Lingjun Liu, **Sangwon Hyun**, Doo-Hwan Bae, "Platooning LEGOs: An Open Physical Exemplar for Engineering Self-Adaptive Cyber-Physical Systems-of-Systems", 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Online.
6. SungJin Lee, Young-Min Baek, **Sangwon Hyun**, and Doo-Hwan Bae, "Generation of Adaptation Strategies for Dynamic Reconfiguration of a System of Systems", Proceedings of 2021 16th Annual Conference System of Systems Engineering (SoSE) (SoSE2021), Online, Jun 2020.
7. Seungchyul Shin, **Sangwon Hyun**, Yong-jun Shin, Jiyoung Song, Doo-Hwan Bae, "Uncertainty based Fault Type Identification for Fault Knowledge Base Generation in System of Systems", Proceedings of 2021 16th Annual Conference System of Systems Engineering (SoSE) (SoSE2021), Online, Jun 2020.
8. **Sangwon Hyun**, Jiyoung Song, Seungchyul Shin, Young-Min Baek, and Doo-Hwan Bae, "Pattern-based Analysis of Interaction Failures in Systems-of-Systems: a Case Study on Platooning", Asia-Pacific Software Engineering Conference (APSEC), 2020, 27th, Singapore, Singapore.

9. Sumin Park, Yong-jun Shin, **Sangwon Hyun**, Doo-Hwan Bae, "SIMVA-SoS: Simulation-based Verification and Analysis for System-of-Systems", Proceedings of 2020 15th Annual Conference System of Systems Engineering (SoSE) (SoSE2020), Budapest, Hungary, Jun 2020.

10. **Sangwon Hyun**, Jiyoung Song, Seungchyul Shin, and Doo-Hwan Bae, "Statistical Verification Framework for Platooning System of Systems with Uncertainty", Asia-Pacific Software Engineering Conference (APSEC), 2019 26th, Putrajaya, Malaysia.

11. Jiyoung Song, Jacob O. Torring, **Sangwon Hyun**, Eunkyoung Jee, and Doo-Hwan Bae, "Slicing Executable System-of-Systems Models for Efficient Statistical Verification", Proceedings of the 3rd ACM/IEEE International Workshop on Software Engineering for Systems-of-Systems (SESoS 2019).

12. Yong-Jun Shin, **Sangwon Hyun**, Young-Min Baek, and Doo-Hwan Bae, "Spectrum-Based fault localization on a collaboration graph of a System-of-Systems", Proceedings of 2019 14th Annual Conference System of Systems Engineering (SoSE) (SoSE2019), Anchorage, USA, May 2019.

13. 김한수 (Hansu Kim), **현상원 (Sangwon Hyun)**, 배두환 (Doo-Hwan Bae), "다변수 시계열 데이터 군집화 알고리즘의 사이버 물리 시스템 오브 시스템즈 실패 분석을 위한 연구 조사 (A Survey on Multivariate Time-Series Clustering Techniques for Analyzing Cyber-Physical System-of-Systems Failures)" 한국컴퓨터종합학술대회 논문집 [KCC 2022].

14. May Myat Thwe, **현상원 (Sangwon Hyun)**, 배두환 (Doo-Hwan Bae), "엣지 컴퓨팅 기반 지능형 교통 시스템 오브 시스템즈의 품질 속성 평가를 위한 사례 분석 (Towards the Quality Assessment of Intelligent Transportation System of Systems using Edge Computing)" 한국컴퓨터종합학술대회 논문집 [KCC 2021].

15. **현상원 (Sangwon Hyun)**, 신용준 (Yong-Jun Shin), 배두환 (Doo-Hwan Bae), "시스템 오브 시스템즈의 오류 위치 추정을 위한 통계적 검증 결과 활용 기법 분석(Analysis of Utilization Methods of the Statistical Model Checking Results for Localizing Faults on System of Systems)" 정보과학회 논문지 [Journal of KIISE 2020]: 380-386.

16. 신승철 (Seungchyul), **현상원 (Sangwon Hyun)**, 송지영 (Jiyoung Song), 배두환 (Doo-Hwan Bae), "시스템 오브 시스템즈의 특성을 고려한 발현 위치 기반 불확실성 요소 분류 (Manifestation Location-based Classification of Uncertainty Factors Considering Characteristics of System-of-Systems)" 정보과학회 컴퓨팅의 실제 [KIISE Transactions on Computing Practies, Vol.26, No. 10]: 451-457.

17. 신승철 (Seungchyul), **현상원 (Sangwon Hyun)**, 송지영 (Jiyoung Song), 배두환 (Doo-Hwan Bae), "시스템 오브 시스템즈에서의 불확실성 요소 분석: 군집운행 시나리오에서의 사례 연구 (Analysis of Uncertainty Factors in System of Systems: Case Study in Platooning Scenario)" 한국정보과학회 학술발표논문집 [KSC 2019]: 278-280.

18. **현상원 (Sangwon Hyun)**, 신용준 (Yong-Jun Shin), 배두환 (Doo-Hwan Bae), "시스템 오브 시스템즈의 오류 위치 추정을 위한 통계적 검증 결과 활용 기법 분석 (Analysis of Utilization Methods of Statistical Model Checking Results for Localizing Faults on System of Systems)". 한국컴퓨터종합학술대회 논문집 [KCC 2019], 2019. 06.

19. **현상원 (Sangwon Hyun)**, 송지영 (Jiyoung Song), 지은경 (Eunkyoung Jee), 배두환 (Doo-Hwan Bae), "시스템 오브 시스템즈의 효율적 검증을 위한 목표 모델 슬라이싱 (Goal Model Slicing for Efficient Verification of System of Systems)". 한국정보과학회 학술발표논문집 [KSC 2018], 2018.12, 432-434.