Fault Localization in Distributed Adaptive Systems

by

Amit Raj, M-Tech.

Supervisor : Prof. Siobhán Clarke

PhD Thesis

University of Dublin, Trinity College

TRINITY COLLEGE DUBLIN
COLÁISTE NA TRÍONÓIDE
THE UNIVERSITY OF DUBLIN

0

Contents

List of Tables

List of Figures

Abstract

Modern systems that execute in ubiquitous environments must be adaptive in order to maintain an acceptable quality of service. As in traditional distributed systems, faults in such systems are likely to propagate across several components and become manifest as apparently unrelated faults in components other than the responsible one. An efficient and accurate run-time root cause detection mechanism is required for fault recovery, which has been solved for non-adaptive systems. However, it is non-trivial in a distributed adaptive system (DAS) because of the changing nature of the system's structure and behavior, the potential inaccessibility of adapted system components, and the potential for inaccurate diagnosable information about the system.

The key driver of this research is the observation about a debugging expert's wasted effort in finding the root cause within a component where a fault's symptom was observed. However, the root cause may exists in another component. The key idea is to pinpoint the actual faulty component. This thesis presents FaLDAS, a fault localization approach to find actual faulty components responsible to originate propagated faults in DAS. The contribution of the work are: (i) A novel fault propagation graph and its construction using components' names and their run-time input/output values, (ii) Faulty Candidates' detection which returns a sorted list of potentially faulty candidates, and (iii) A fine grained fault analysis which enables to find the potentially faulty corrupted variable, of a faulty component, whose propagation has caused a propagated fault.

The key outcomes of the thesis are (i) improved efficiency over existing mechanisms, (ii) ability to diagnose inaccessible components, and (iii) detection of faulty component and its corrupted output variable. FaLDAS uses run-time values to identify if a component is suspicious to be faulty (suspicious components) or not, by traversing an FPG. Unlike existing mechanisms which use all the components of a system to identify actual faulty components, whereas FaLDAS uses only suspicious components. It reduces the search space to find faulty components; increases the efficiency. The run-time input output values enable the fault diagnosis of components whose internal designs e.g., source code, test cases, data flow, etc., are inaccessible. Unlike existing mechanisms which fail to diagnose inaccessible components, FaLDAS show substantial success in their diagnosis. In addition, detection of corrupted variable of a faulty component enables a debugging expert to find a specific root cause in a specific part of the component, which reduces root cause analysis effort.

FaLDAS's fault localization algorithm finds a set of potentially faulty candidates. A candidate is a node in FPG (FPGNode) which has a reference to a potentially faulty component. In addition, FaLDAS sort the candidates according to their probability of being faulty. The final output of FaLDAS is the sorted list of candidates, which is a recommended order for a debugging expert to prioritize components for root cause analysis.

FaLDAS was evaluated for time efficiency and accuracy on simulated systems and a real-world system TCAS (Traffic Collision Avoidance System) in widespread use by major aircrafts in United States. Simulated systems with a large number of components were used to measure time efficiency. FaLDAS shows higher time-efficiency than that of other related approaches by up to two orders of magnitude. FaLDAS's accuracy was evaluated on both simulated systems and TCAS. In particular, accuracy was evaluated in scenarios when the actual faulty component is inaccessible and also when the required input information is inaccurate. In the former case, FaLDAS achieves higher accuracy but not in the later case. In summary, FaLDAS achieves higher efficiency, without compromising the quality of results, than that of other related approaches.

Chapter 1

Introduction

1.1 Motivation

Modern software systems play a crucial role in our daily life. Common systems in daily use are email, phone calls, traffic management, medical services, air traffic control, security, etc. Users are so dependent on software systems that any fault in such systems may result in loss of money and sometimes may cause death. For example, a technical malfunction in a Chinook helicopter ZD576 killed 29 people on board [108]. A software fault in computerized radiation therapy machine 'Therac-25' results in many deaths and serious injuries [75]. The Edwin I. Hatch nuclear power plant was forced into an emergency shutdown after a software update installed on a single computer [66]. Many software systems are so critical to our life that they must be robust, reliable and adaptive to automatically recover from a fault.

However, maintaining such adaptive systems is a complicated and challenging task. One of the challenging tasks is to efficiently identify a fault and recover from it. Fault analysis is complicated by the potentially large structure and complicated behaviour of the systems. The information required for fault analysis may be either partially available or inaccurate. Components involved in the system are likely to have a complicated dependency structure which restricts a detailed analysis of faults and their effects. Sometimes, a system may have several components that are not even accessible to fault localization mechanisms; restricting fault analysis. Short deadlines and market pressure compel the developers to ship the system without exhaustive testing.

Faults in software systems are unavoidably introduced and to recover from a fault is costly. A NIST report estimates that software faults cost the U.S. economy $59.5 billion annually, approximately 0.6 percent of U.S. gross domestic product (GDP) [95]. A recent Cambridge University Research report states that the global cost of software faults has risen to $312 billion annually [44]. Moreover, this study found that approximately 50% of a software developer's time is spent on finding and fixing faults. An efficient and accurate fault analysis and recovery mechanism is required to deliver a fault tolerant software system. Faults and their effects must be thoroughly analyzed and root cause analysis mechanisms must identify the faults efficiently and accurately.

1.2 Problem

Modern distributed systems are frequently composed of a range of components that collaborate in a potentially complex manner to achieve the overall systems' goals. To maintain an acceptable quality of service for such systems in dynamic environments, individual components may be adapted i.e., upgraded, removed, replaced or reconfigured, at runtime, without interrupting the systems' execution [102] [62]. For example, Popescu et al. [101] describe a *crisis management system* case study where different components are imported to the system when the environment or requirements change. Such systems are referred to as Distributed Adaptive Systems (DAS). In a DAS, several components interact to provide a service, creating dependencies among the components. When a fault occurs, a dependency emerging through component interactions may cause the fault to propagate among inter-connected components [13]. Thus, the original fault becomes manifest as an apparently unrelated fault in components other than the responsible one [68] [70]. Such faults are referred to as *propagated faults* (or *external faults* [13]). One of the important challenges in a DAS is to efficiently determine the root cause of a propagated fault.

A component may have several faults. Some faults are *active*, producing errors during the component's execution, and others are *dormant* [13]. A dormant fault may activate when a component receives an input to start a computation. A fault may transforms into other fault within the same component is called *internal propagation* [13]. Such a fault may be properly handled within the component, say component A, but some may propagate to the component's interface. In such a case, the fault will further propagate to other components, say component B, which receives services from component A. The propagation of an fault from component to component is known as *external propagation* [13]. External propagation emerges in this example because the output of component A has deviated from correct output, causing incorrect input to component B, which results in a propagated fault to B. Propagated faults may further propagate to other components. In short, corrupted inputs and outputs between components cause fault propagation, and this thesis concentrates on supporting the identification of the root cause of such faults.

Beizer highlighted and analysed several different kinds of faults in a software system, in particular functional faults, structural faults, data faults (incorrect input and output), implementation faults, integration faults and architectural faults [16]. In this work's occurrence frequency analysis, the frequency of data faults was estimated at 22.4%, the second most abundant in a software system. Also from Beizer's analysis, a typical software program contains 50% lines of data declarations and assignments. Such program statements have the potential to become major sources of data faults, which in turn may cause propagated faults, and may severely impact the execution of a system. While the data fault frequency of 22.4% does not capture the majority of overall faults in a system, analysis of such faults is nonetheless potentially very time consuming, and is a substantial research field in its own right [41] [10] [8] [14].

1.3 Challenges

When a fault propagates and manifests as an unrelated fault in a component other than the responsible one, a two step fault diagnosis process is carried out as described below:

1. Fault Localization: The process for identifying the component responsible for generating a fault [137] [132] [116] [110] [106].

2. Root Cause Analysis (RCA): The process for identifying the root cause of the fault within an identified faulty component [70] [69].

This thesis presents a fault diagnosis process called FaLDAS, and its application to a real-world system. The primary contribution of FaLDAS is Fault Localization, where the challenges presented by system adaptation are primarily manifest. Although Fault Localization is an extensively studied area, locating faulty components in a DAS remains challenging for reasons as follows:

1.3.1 Inaccessible Faulty Components

The diagnosis information needed about a component can be obtained by having: (i) access to the machine on which the component is hosted or (ii) access to the component's diagnosable information such as logs or (iii) access to remote debugging capabilities for the component or (iv) access to the component's source code or detailed internal design. When an adaptation imports components for which the DAS administrator (automated or not) has none of these access levels (i.e., the components are *inaccessible*), fault localization is more challenging. In practice, such situations occur when Commercial Off-The-Shelf (COTS) components, Component Object Model (COM) components or any other blackbox components are included in a DAS. The designer has limited (or no) control that enables finding the information needed to analyze the root cause of a fault within the components [19]. Dependency analysis of such components is likely to be non-trivial [11]. Their failure semantics may be different from those of the other components in the system, so applying homogeneous failure analysis may not be suitable [109]. Moreover, applying probes such as embedding a sensor or an agent inside such components may not be allowed.

1.3.2 Inaccurate Diagnosable Information

Many information sources are useful in fault diagnosis, such as logs, components' dependencies, faults' causal relationships, stack traces, etc. A DAS system is likely to be large and complicated, and correspondingly challenging to analyze. Notwithstanding several advances in monitoring technology [52] [72] [64] [32] [48] [15] [113], the information captured by monitors may be noisy or unreliable: (i) the system's monitors may not cover all aspects of a system, which leaves the information incomplete, or (ii) a monitor may not be sufficiently sophisticated to precisely identify the system's structure and behaviour, which may result in inaccurate information. Incomplete or inaccurate information is likely to negatively impact the fault localization technique. For example, let us assume a system of 20 components, where it is expected that the failure or success status of all 20 components is logged for a failed execution. In this case, a log has 20 lines, each with a <component_name, success/fail indicator> pair for the execution under consideration. If any of the components either do not log success/failure at all, or log this incorrectly, then this will negatively impact the result.

Gap In Literature

To overcome such challenges in a DAS, the required input data must be dynamically generated and collected as existing data becomes inconsistent with the adapted system. In a DAS, obtaining a new configuration's data at runtime is likely to be inefficient and the results may be incomplete or inaccurate. It has been observed that existing techniques use logs and error messages as preferred information to diagnose a fault [20] [17] [104]. Where logs are used to analyze faulty components, collective analysis of logs from multiple components, while possible, is non-trivial. This is because different logs may have different granularities, syntax and semantics [77]. Advanced logging techniques have been proposed to improve logging, but it remains difficult to correlate an error message with source code using logs [1] [139] [28]. For example, even knowing a component's source code, a propagated fault's error message "Exception in thread main: service endpoint not available" may not help to locate the actual faulty component and root cause within it. Developers also throw custom error messages which may be irrelevant or voluminous [77]. Moreover, a component's log may not outline the execution context relevant to another component, as they may be loosely coupled and correlating different logs is a time consuming process. Overall, using logs is limited for faults' correlation analysis and root cause determination.

Other techniques use a fault's stack trace. However, this is not likely to be useful in a DAS as it traces back only to the fault in the local component and does not establish correlation with faults in other components [56]. Remote debugging of remote components may be blocked and even if possible, the component structure may change while these processes are executing [88] [71]. Several mechanisms apply design-time techniques for Fault Localization in static systems [106] [116] [132] [110] [137], which could also be applied in a DAS where a new system configuration is known *a priori* [69] [21] [76]. However, it is not generally the case that all possible configurations of an adaptive system will be known at design time [111].

1.4 An Overview of Research

A novel approach for fault localization has been designed to cope with the challenges of an adaptive system (Section 1.3). Based on valuable insights from existing approaches, this thesis explores the possibility of constructing a knowledge-base about a system under consideration when the system exhibits the challenges described earlier. A heuristics statistical technique has been designed which reasons over the knowledge-base to locate the fault. The purpose of this approach is to provide an efficient fault localization useful for debugging experts to effectively identify the root cause of a propagated fault in a DAS.

Assumption This thesis makes the following assumptions about the environment in which debugging experts face challenges in analyzing the root cause of propagated faults.

1. The actual faulty component activates the fault within itself and cannot prevent it reaching the component's interface. As a result the fault propagates, in the form of corrupted output data, to other components dependent on the actual faulty component. The other components receive corrupted input from the actual faulty component and may manifest it as a visible fault such as a system failure, outage, inconsistency, service unavailable, webpage not found, etc.

2. Components are loosely coupled in that they are independently developed and a component does not have information (internal structure and behaviour e.g., data flow, source code, test cases) about other components. They interact through well defined interfaces such as ESB communication bus.

3. The system's operating environment is highly dynamic where components are frequently adapting (components' addition and removal). An adaptive system re-configures itself at runtime in such a way that a number of components are added and deleted, and placed in a topology which cannot be foreseen at design-time.

Observation One of the key drivers for this research is the observation that a debugging expert first invests efforts in finding the root cause of a fault in a component where the fault or its symptom appears. However, the debugging expert's efforts are wasted when the fault's root cause exists in another component. When the expert realizes that the root cause exists in another component, the expert finds the actual faulty component by analyzing other components and their diagnosable data. However, the actual faulty component may be inaccessible or the required information may be inaccurate or the system adapts so frequently that the expert has a limited time to analyze the fault. This means it is difficult for a debugging expert to efficiently identify the root cause of a propagated fault at runtime.

Hypothesis This thesis investigates how to identify the components actually responsible to originate a propagated fault. It frames its hypothesis as follows: The use of components' real-time input output values enable to reduce the search space of finding the actual faulty candidate responsible for a propagated fault, to diagnose inaccessible components, and to identify actual faulty component even if its information in the knowledge-base is inaccurate.

Basic Idea This thesis introduces FaLDAS (Fault Localization in DAS), a run-time fault localization approach to determine the component(s) responsible for a propagated fault in a DAS. The novel concept of finding the suspicious components for a fault is based on the components' real-time input and output values. Unlike other existing approaches which generally use run-time logs data which may not be available for inaccessible components, FaLDAS is based on real-time input/output values. A system's behaviour is drawn from input/output values into a directed graph. Graph traversal enables an efficient isolation of suspicious components potentially responsible for originating a propagated fault. Unlike other approaches which apply statistical analysis on all available components in a system, FaLDAS first isolates the suspicious components, which reduces the search space. Motivated by BARINEL's approach [4] [2], a novel statistical approach was developed to support finding the actual faulty component among suspicious components by sorting the suspicious components according to their probability of being faulty. Unlike BARINEL where all components registered in the knowledge-base as having failed for multiple previous executions are included, even if diagnosis is required just for a single failed execution, FaLDAS is designed to find the faulty components for a specific failed execution. This helps debugging experts to precisely identify the root causes for a particular failed run, efficiently. BARINEL's output i.e., a set of faulty components, are not necessarily responsible for all failed execution. Different failed execution may have different sets of faulty components [112].

Requirements A number of requirements analyzed for a fault localization mechanism to support the identification of an actual faulty component responsible for a DAS:

- **Adaptation**: The mechanism should be capable of identifying the fault in an adaptive system where components may adapt (add or remove) at runtime. Otherwise, an adaptation may occurs such that no faults traces, logs, or other evidences remains. Without evidences, fault analysis is difficult and the fault remains unresolved.
- **Inaccessible Components**: A mechanism should produce substantially correct results even if a DAS has several inaccessible components. A major impact of inaccessible components may be the inaccessibility of diagnosable information required for fault localization. As a result of inaccessible information, overall knowledge-base to reason about faults could be inaccurate or incomplete.
- **Distributed Systems**: A mechanism should work in a distributed system which may have several unknown components whose internal structure and behavior may be unknown, which impacts the fault localization. Otherwise, such components, which may be actually faulty one, remains undetected.
- **Root Cause**: In a DAS, a fault localization mechanism should be capable of identifying the actual root cause of a fault, i.e., faulty statements or corrupted variable responsible to cause a fault. Otherwise, fault recovery cannot be planned and applied. However, root cause analysis is not in thesis's scope.
- **Ranking**: A mechanism should produce a sorted list of faulty candidates according to the likelihood of their faultiness. Otherwise, the debugging expert may not efficiently find actual faulty component.

1.5 Thesis Contribution

This thesis investigates how an adaptive system's run-time data can be used to efficiently identify the root cause of a propagated fault in a system. This research provides the following contributions to the knowledge:

Fault Propagation Graph Existing fault localization approaches generally use components' runtime information, made available in logs. However, access to logs may be blocked in COTS components or time consuming to access and analyze. This thesis describes a novel mechanism to construct a fault propagation model by analyzing only the real-time inputs and outputs of the components in the system. Even when the components are inaccessible, their requests and responses can be intercepted to read their inputs and outputs. This thesis describes an extension to directed acyclic graphs to represent an innovative fault propagation graph. An algorithm to construct a fault propagation graph from real-time inputs and outputs is presented, which has not been previously investigated. Generating the model in this manner supports addressing adaptive system challenges and enables an efficient way to isolate suspicious components for a failed execution.

Faulty Candidates' Detection Existing related mechanisms find a common set of faulty candidates as the root cause for multiple faults, which is not the optimal set of candidates responsible for individual faults [112]. This thesis describes an algorithm to efficiently isolate the faulty candidates responsible for a specific failed execution. The algorithm's underlying statistical mechanism sorts the candidates according to their probability of being faulty. This algorithm innovatively finds the faulty candidates for a target failed execution, which has not been discussed by existing mechanisms.

Fine Grained Fault Analysis Existing fault localization mechanisms support finding an actual faulty component. Even when an actual faulty component is identified, a debugging expert invests a substantial effort in finding the actual fault within the component. It is a complicated and time consuming task, in particular, when the component is large and its design is complex. This thesis supports to find the actual faulty component along with its corrupted output variable. It helps a debugging expert to analyze only a part of the component, which has effected the computation of the corrupted output variable; likely to increase root cause analysis efficiency.

To compare the effectiveness of FaLDAS with that of other related approaches, a number of experiments carried out on simulated systems, TCAS [67] and TRANSFoRm [37] systems to investigate its diagnostic quality and time efficiency. The evaluation results show that FaLDAS achieves time efficiency up to two orders of magnitude while maintaining best diagnostic quality[1] in finding inaccessible faulty component. However, FaLDAS does not performs best when information is inaccurate.

As a demonstration of how FaLDAS fits into an overall Root Cause Analysis process, FaLDAS is integrated with an existing RCA technique, which works with individual components in the ranked list. Le et al. [69] present an RCA mechanism which identifies the program statements responsible for corrupting an output variable. This mechanism was suitable to integrate with FaLDAS because FaLDAS identifies a faulty component along with its corrupted output variable, which might have propagated the fault. A debugging expert, using Le et al.'s approach, can identify the program statements, in the component, responsible for the corrupted output variable.

1.6 Thesis Structure

State of the art Chapter 2 analyses how existing mechanisms carry out run-time fault localization in adaptive system. In particular, the study explores the underlying data models and the reasoning algorithms used to find the faulty candidates. The chapter also identifies some mechanisms that can be applied to adaptive systems e.g., Spectra-based Fault Localization (SFL).

Design Chapter 3 discusses the basic terminologies used and describes the scope of this research. It then illustrates the problem statement with a detailed example and

[1]The percentage of components, out of all components in a system, analyzed before the actual faulty component(s) is identified.

analyses a number of existing mechanisms to solve the problem such as BARINEL [4] [2], Pinpoint [27] and Bellur et al.'s approach [17]. It then returns to the challenges of fault localization in adaptive system outlined in section 1.3 and describes the design objectives and design decision of this thesis which underlie the fundamental structure of the desired approach.

Approach Chapter 4 describes a detailed overview of the FaLDAS approach. It includes the monitoring of an adaptive system, construction of a knowledge-base (i.e., fault propagation graph), a statistical mechanism and algorithm to localize the faulty components and finally ends with the description of FaLDAS's integration with an existing root cause analysis mechanism.

Evaluation Chapter 6 evaluates FaLDAS's time efficiency and diagnostic quality by comparing with that of other related approaches. It first describes the experimental set-up, in particular simulation-based system and the TCAS system. The time efficiency evaluation illustrates that FaLDAS achieves the significant gain (up to two orders of magnitude) over other approaches. However, the diagnostic quality evaluation illustrates that FaLDAS achieves best accuracy to diagnose inaccessible components but not when information is inaccurate.

Discussion Chapter 7 summarizes FaLDAS and its achievements. It highlights a number of scenarios where this research can be useful and not. A number of limitations are discussed targeting the potential areas for future work.

1.7 Chapter Summary

Software systems play crucial role throughout our daily life. The software systems have become so integral to our life that any disruption in their services may result in loss of time, money and sometime causes death. Faults in software systems are one of the main reasons for services disruption. To maintain the quality of services, efficient and effective fault localization mechanisms are required to analyze the faults.

A fault propagates among inter-connected components and becomes manifest as an apparently unrelated fault in components other than the responsible one. Analyzing the root cause of a propagated fault is a complicated problem. It becomes even more complicated in Distributed Adaptive System (DAS) where components adapt at runtime. Root cause analysis of a propagated fault is complicated in DAS because (i) inaccessible components may be adapted at runtime whose access to required information may be blocked, and (ii) the complicated and dynamic configuration of a system limits the monitoring mechanisms to monitor accurate information of the system.

This thesis develops an approach to efficiently and effectively localize the faulty components responsible to originate a propagated fault. It presents a novel mechanism to construct a fault propagation graph using component's real-time inputs and outputs. In addition, this thesis develops an algorithm which reasons the graph to analyze actual faulty component(s). The approach is evaluated for its diagnostic quality and time efficiency to compare with other related approaches. The results illustrate that the FaLDAS mechanism is two orders of magnitude efficient and achieves best quality to diagnose inaccessible faulty components.

This thesis assumes a system environment where a fault propagates among components through the corrupted output data. Moreover, the components are loosely coupled that prevents an upfront causal relationship analysis. It is also assumed that the systems under consideration are adaptive in that the components adapt at runtime that poses a number of challenges against existing fault localization mechanisms. A key observation that led to this research is the debugging expert's wasted effort in locating the fault. The expert invests his efforts in finding the root cause of a propagated fault in the same component where the fault has appeared, whereas the root cause exists in a different component.

Chapter 2

Related Work

Fault localization is an established field for non-adaptive systems, with considerable successes reported. A number of such techniques are included here, and while many of them were not designed for adaptive systems, both their capabilities in this regard are considered, and also the challenges they face even within non-adaptive systems. A number of more closely-related techniques for adaptive systems are also emerging, which have provided this thesis with useful insights.

Fault Localization approaches use a system's data to construct fault propagation model which is analyzed to localize a fault. A Directed Acyclic Graph (DAG) is one of the popular techniques to represent the causal relationships between faults. Section 2.1 investigates approaches where a DAG is used to represent causal relationship between faults, dependencies between components, propagation of faults, etc., for fault localization. A variant of DAG i.e., Bayesian Belief Network (BBN) provides the probabilities of causality between software entities e.g., faults. Section 2.2 investigates BBN-based approaches which use probabilistic inferences to conclude the actual cause of a fault.

A number of techniques use program source code to create program slices. Section 2.3 investigates how debugging experts utilize program slices and their dependencies to analyze the root cause of a fault. Program source code also used by researchers to construct the ontology of a system. Section 2.4 describes the ontologies-based fault localization approaches. Where the source code is not available, researchers use components' activities e.g., component's involvement in an execution, to identify potential faulty components. Section 2.5 investigates statistical models and techniques which use a component's activities to conclude a list of potential faulty components. However, not all components in the list are actually faulty, they are sorted according to their probability of being faultiness. Section 2.6 analyzes such ranking mechanisms which sort the faulty candidates according to their faultiness for a failed execution. Lastly, section 2.7 presents existing fault analysis techniques currently used in the software industry.

2.1 Causal Graphs Models

A causal graph model represents the causal relationship between entities of a software system. This section analyzes graph-based fault localization techniques to localize faults in a DAS.

Candea et al. [21] describe an automatic failure-path generation technique that automatically generates fault propagation information of a system at design-time. They inject faults in one component and analyze their effects in other components. A fault is manually injected into a component. A monitoring mechanism detects the fault propagation across several other components. The system need to reboot for each fault injection and its propagation analysis. The mechanism does not require prior information about the components, but the process takes considerable time to run, and requires several reboots which is not possible in adaptive systems. Moreover, fault propagations analyzed at design-time becomes invalid after an adaptation as new component may be added and an existing one may have removed. In addition, it is difficult to inject faults into components at runtime, when the components are inaccessible, to manipulate the source code.

Andrews et al. [10] use a directed graph of a components' variables to construct a fault tree, where two faults are connected with a logical operator. This approach assumes a static set of components, which is not a safe assumption in a DAS. An adaptation is likely to make a pre-defined fault tree inconsistent with the newly adapted system. Apart from the fault tree, other techniques describe several data structures to store the faults' causal relationships such as a fault propagation graph, a dependency matrix, etc. Liu et al. [81] describe the use of a fault dependency relationship matrix for fault diagnosis. They generate a matrix for each component, which shows the relationship between each component's faults and other components' faults. However, retrieving the faults from an inaccessible component is non-trivial. The identified relationships between faults may be incorrect which results in incorrect fault localization. Moreover, in a DAS, such matrices would have to be re-generated after each adaptation, which becomes a performance bottleneck.

Yemini et al. [135] describe an event correlation technique in real-time. They consider a set of symptoms as a code that is decoded to analyze the problem. The technique uses the topology of the system to create a bipartite graph that is required to generate the code. In a DAS, the topology of the system changes, thus bipartite graph changes and hence the code is required to be regenerated. It limits its applicability in adaptive systems.

Le et al. [69] discuss a fault correlation technique based on the fault propagation paths. The premise of the approach is that when two faults occur in a single execution path, they are correlated. However, several faults are latent faults whose error or evidences do not appear or are difficult to detect. Where the root fault is a latent one, this mechanism is not likely to find the root fault and its root cause.

Several other graphical models exist to visualize large complicated systems. For example, Feng-wu et al. [131] describe a diagnostic tree, which effectively is an extended version of a fault tree, to analyze system-level faults. Another version of a DAG is Bond graphs in which arcs are bidirectional. These graphs are used to identify the bi-causality between two faults [83] [18]. These graphs, when compared to a DAG, provide more detailed information, particularly quantitative information. On the other hand, a DAG is mainly concerned with the qualitative characteristics of a system such as dependencies, fault propagations, data flow etc., which also can be easily visualized. Thus, a DAG is more widely used research and industry [133].

2.2 Bayesian Networks

A Bayesian Belief Network (BBN) is a probabilistic directed acyclic graph where nodes denote random variables and edges represent the conditional probabilities. It is a casual graph except that the nodes and edges support conclusions about the posterior probabilities. Let us assume a BBN where a node f has a directed edge to node s where f and s represent a fault and a symptom, respectively. The conditional probability is represented as $p(s|f)$ which represents the probability of f causing s.

Bellur et al. [17] construct a BBN of faults. They use Jex tool [107] to discover and analyze a system's call traces to generate components' dependencies and to analyze fault occurrence evidences. Correspondingly, BBN nodes (<component, fault>) are constructed. Their assumption, if a component A uses component B then A's exception E_1 triggers B's exception E_2, is used to create edges between BBN nodes. However, call traces, to construct BBN, are not always available for inaccessible components in a DAS. Motivated by this approach, FaLDAS constructed and used FPG where a node in FPG represents (<component, output variable>).

Joshi et al. [59] identify a fault using probabilistic inferences. At design-time, they determine a set of faults that can occur in a system. Such sets are characterized by a fault hypothesis e.g., a fault F occurrence corresponds to fault hypothesis 'Server has crashed' [59]. Bayesian probabilistic inferences are used to determine the responsible fault for each fault hypothesis. These pre-determined inferences are used to identify the actual cause of a system's failure evidence i.e., a fault hypothesis, at runtime. However, such pre-determined probabilistic inferences are invalid after adaptation. Moreover, analyzing faults and their failures in an adapted system *a priori* is difficult as a DAS may adapt unknown or inaccessible components. Similarly, Lo et al. [82] create a Bayesian network where nodes represent components and edges represent the causal connection of components' faults. When a fault evidence is detected, the Bayesian network inferences are used to locate the actual faulty component.

A major drawback of Bayesian networks is that it should be a directed acyclic graph. It is useful where processes do not have cycles. However, a dynamic adaptive system consists of numerous components, and executions may have cycles of processes. Another challenge with Bayesian network approaches is their execution time; they are computationally expensive or sometimes inferences are NP-hard [30]. A number of linear-time algorithms exist for probabilistic inference of BBNs where nodes are singly linked. However, the inference time for multiply-connected large BBNs may be exponential [30]. In addition, incomplete information may cause several variables in a BBN to be uninstantiated, which makes probabilistic inference NP-hard [30].

2.3 Program Slicing Techniques

Program slicing is a well known technique to analyze program statements, particularly in the field of fault localization [114] [73] [9] [74]. Program slicing divides a program into a group of statements responsible for a specific computation. These groups of statements are known as program slices. A software developer analyzes the data and control flow between slices to effectively pinpoint a slice which effects a program's output, in particular a corrupted output value.

A number of specialized program slicing techniques exist such as static slicing, dynamic slicing, intra-procedural slicing, backward slicing, etc., [114]. A static program slice corresponding to a variable contains all the execution statements that may effect the variable's value. The fault domain search is reduced to the slices which contain that variable [6]. Since, the program slices are constructed statically, they may not be the most optimal slices in a dynamic environment because the identified slices responsible for a corrupted variables may also include slices which should not be included [128]. Moreover, fault localization using static slicing is limited to the available test cases which may not cover all possible inputs to a DAS at runtime [128].

To construct an effective set of program slices, researchers have used an advanced technique with dynamic program slicing [73] [9] [74]. Lei et al. [74] present their approximate dynamic backward slicing technique which manages the trade-off between the size and accuracy of a slice to create an effective set of slices. The approach presents an algorithm that automatically analyzes the statements which directly or indirectly effects the output value of a variable. The set of slices is fed as input to their slice-based statistical fault localization tool which assigns different probabilities to slices based on their suspiciousness of being faulty.

Alves et al. [9] present an improvement over the Tarantula technique [58] using dynamic program slicing. Using the output of Tarantula, Alves et al. remove the statements from a slice which do not appear in the dynamically created program slice corresponding to a corrupted output variable. They argue that such statements will not have any impact or do not participate in the computation of a corrupted output variable. Further, they remove those statements which are part of the older version of same system. They argue that this results in the loss of the residual error which may have been introduced by the legacy code of an older version.

Mariani et al. [84] produce code fixes for known problems, requiring the program's bug-raising lines of code as input. This fault analysis approach is based on pass/fail tests, which needs a diverse set of test cases and a significant amount of execution time. However, obtaining the bug-raising lines of code is one of the biggest challenges for inaccessible components. Where the components are accessible, researchers have generated a knowledge-base of faults and apply expert systems for fault localization. Expert system approaches involve rule-based reasoning [35], model-based reasoning [85] or case-based reasoning [115]. These techniques work well for either small or static systems, though must be customized to a specific system. Also, a noisy or incorrect knowledge-base is likely to negatively impact the fault localization results.

In the program slicing literature, program slicing-based fault localization techniques generally depend on the test cases coverage [114] [73] [9] [74]. However, in a DAS that was composed dynamically at runtime, running test cases and obtaining their coverage is non-trivial. Another major problem with the program slicing techniques is access to the source code which may not be available for DAS components because several components may not be accessible, or if accessible, source code access may be blocked. Even if the source code of a components is accessible, the discussed mechanism do not specify how to use program slicing at the system level that includes several components. Moreover, in large-scale systems with a large code-base, there may be a large number of slices that need to be examined, which is not efficient [128].

2.4 Ontology-based Models

The world is evolving to where people are working on multi-disciplinary systems. In such systems, the output of a service, running by a specific set of engineers, is transformed and consumed as input to other services running by another set of engineers. The semantics of one community is not easily understandable to other communities. As a result, standard data conversion mechanisms are required. To standardize the data conversion, the World Wide Web Consortium (W3C) [121] developed an ontology architecture that includes a set of languages such as OWL, XML, RDF and RDFS. EXtensible Markup Language (XML) is one of the widely used languages to represent the data in a human readable format, especially used to represent web services outputs. Resource Description Framework (RDF) is a framework to model the information described in web resources. This information can be represented in a variety of formats. To describe an RDF document, RDFS (RDF Schema) presents an XML vocabulary to define RDF elements and relationship between those elements.

An ontology represents the fundamental objects and their relationships in a particular domain. Objects are nouns and their relationships are verbs. For example, 'bank has ATM' represents 'bank' and 'ATM' as nouns and 'has' is a verb which represents the relationship between the two nouns. Web Ontology Language (OWL) is used to represent the formal semantics of an ontology. OWL is built upon RDFS objects. OWL is more powerful than XML in that it represents the semantics for encoding and exchanging ontologies between different user groups, machines, languages etc. The OWL data model consists of several triplets having *subject*, *predicate* and *object*. A triplet is represented in XML syntax. A *subject* represents a particular resource of a domain which has a property denoted by *predicate* where *object* is the instance value of that property.

Zhou et al. [142] describe a fault propagation analysis in networked control systems. Zhou et al. describe an approach to capture the entire structural and behavioral information of a system in ontology-based models. They describe two types of ontology of a system: object-centered ontology and system-centered ontology. The object-centered ontology represents the semantics of fundamental components of a system whereas system-centered ontology represents the semantics of the interactions between the components. When a fault occurs, fault and effect traces are used to reason about the ontology model. The outcome of such reasoning indicates the fault propagation paths. However, adaptive system challenges − inaccessible components and inaccurate information − restricts the construction of such ontology models in entirety. Inaccurate models negatively impact the outcome of the approach. Moreover, constructing ontologies after every adaptation is a time consuming and complicated process.

Minsoo et al. [63] presents an ontology-based mechanism to identify a faulty member in a computing system. They define the ontology of each member i.e., a component. They illustrate a mechanism through which the system maintains the replicas of each component. Replica means components ontology. Each component sends its activity (fault) information to central system administrator. Accordingly, the administrator update the corresponding ontology based on the received input to maintain the members status. When the administrator identifies a component's failure, it performs a recovery action instead of finding the actual cause of the fault. The ontology of that component is sent across all the members so that a new member (added in place of a failed member) can understand this ontology and perform collaborative actions to tolerate the fault.

In ontology-based fault analysis, fault models are constructed using system ontology over which inferences are conducted. In such mechanisms, a system's knowledge-base is represented in the form of an OWL ontology. Several tools exist to construct and store such ontologies such as protege, MediaWiki, etc. As these ontologies are constructed as an RDF data model, effectively in XML syntax, and queries can be formulated to conduct inferences. SPARQL, RDQL Versa and several other query languages exist to query an OWL ontology. SPARQL is one of the predominantly used languages and recommended by W3C.

In a DAS, a major challenge is the unavailability of the internal design of a component, and so constructing a detailed ontology of a component is non-trivial. Moreover, no ontology-based fault localization mechanism exist which works on real-time data. Most of the techniques work on structural information of a system and identify the structural faults.

2.5 Structural and Statistical Equation Models

Several approaches use Structural Equation Modeling (SEM) in that statistical techniques are used to analyze and identify the root cause of a fault [100] [27] [4]. For example, Casanova et al. [23] use a system's real-time program spectra. They rely on the availability of instrumentation inside the code, which may not be possible for inaccessible components. Abreu et al. [4] [2] have developed an algorithm called BARINEL to diagnose a set of faulty components using program spectra. The spectra is described as a matrix that captures the involvement of individual components for both failed and success executions, which is then used to analyze the faulty components. A drawback of this approach is the level of granularity at which it returns potentially fault components. All components registered in the matrix as having failed for multiple previous executions are included, even if diagnosis is required just for a single failed execution; results are not optimal (Section 1.5). Santelices et al. [112] state that faults in different failed execution are better found by different spectra and ranking mechanisms, Abreu et al.'s [4] mechanism is limited. Whereas FaLDAS can be used to find the faulty candidates for a specific failed execution. In addition, the evaluations shows that BARINEL is not as efficient as FaLDAS.

Similarly, in the Pinpoint algorithm, Chen et al. [27] create a program spectra which captures the failed or success status obtained from the logs of individual components, for a set of executions. Chen et al. track request/responses relating to individual components for a particular execution. They apply the Unweighted Pair-Group Method using Arithmetic Averages (UPGMA) algorithm on the spectra to identify the faulty components. This is likely to work for a DAS because diagnosable information is collected at runtime. However, the evaluations show it is computationally expensive and more in-efficient than FaLDAS.

Statistical equation models are also used to represent the causal relationship between two variables. For example, equation $z = x + y$ represents causal relationship of z with x and y. Baah et al. [14] present a probabilistic graphical model to represent the causal relationship between the variables of a program. The probabilistic reasoning is used to rank the program statements identified as faulty. A limitation of such statistical techniques is that the program variables should be identified in advance to construct the hypothesis. However, where a DAS brings unknown and inaccessible components in the system, identification of such variables is not possible.

Such equation models require a detailed quantitative information about a system to effectively carry out Fault Localization. In a real-world large complicated system, it is a challenging task. It becomes more challenging when the system structure and behavior adapts at runtime. As discussed, other qualitative models exist such as directed graphs, dependency graph, topologies, etc., whose construction is easier as they represent high level qualitative information of a system.

2.6 Probabilistic Ranking

Whether fault localization is SEM-based or graph-based or any other technique, false-positive results are inevitable. Along with the actual fault, several other faults are detected which effectively increase the time and efforts to identify the actual fault and its root cause. Probabilistic ranking of faults aims to prioritize the faulty candidates. A sorted list of candidates helps developers to efficiently identify the actual fault in that candidates with higher probability are analyzed prior to the ones with lower probability. To rank candidates, techniques generally use the concept of similarity (i.e., similarity coefficient) between a software entity behavior and error occurrences.

Similarity Coefficient

Fault localization techniques generally use systems information such as execution traces, logs, dependencies, etc. Such information e.g., execution traces, is used to identify the faulty candidates through analyzing the differences between correct execution information and failed execution information. A mechanism is required that can differentiate correct executions's information from failed executions's information. In other words, a mechanism is required to assess the similarity between two sets of information.

A statistical measure used to compare the similarity or diversity between two sets is called similarity coefficients. In data clustering area, similarity coefficient is measured on binary, nominally scaled data sets such as {0, 1, 0, 0} and {0, 1, 1, 1}. Given any two such sets, a similarity coefficient calculates the overlap between the sets. To express the similarity, each combination of values in the two sets can be represented in four notations as described in Table 2.1.

Notation	**Set$_1$**	**Set$_2$**
a_{11}	1	1
a_{10}	1	0
a_{01}	0	1
a_{00}	0	0

Table 2.1: The notations used in the computation of Similarity coefficient.

The similarity coefficients are mostly used in spectrum-based fault localization techniques [138]. Several similarity coefficients exist out of which Jaccard [54] [27], Tarantula [58] [112] and Ochiai [87] are most widely used in spectrum-based techniques [138]. A detailed description of these coefficients is given below (s_j is the value of similarity coefficient):

- Jaccard Coefficient: $s_j = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$
- Tarantula Coefficient: $s_j = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$

- Ochiai Coefficient: $s_j = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})*(a_{11}+a_{10})}}$

As an example of how a similarity coefficient helps fault localization, a sample hit spectra is used [3] where '1' and '0' represents whether a software entity is used or not, respectively (Table 2.2). This spectra illustrates that software entity 4 is the mostly likely to be faulty because its information set has maximum similarity with the error set. In other words, a fault occurs only when software entity 4 was involved in the execution. The following section analyzes a number of spectrum-based fault localization techniques which rank the faulty candidates through a similarity coefficient.

	Software Entities						
input	0	1	2	3	4	5	error
S_1	1	1	1	1	0	1	0
S_2	1	1	1	1	1	1	1

Table 2.2: A sample hit spectra.

Ranking Faulty Candidates

Several static approaches use the pass/fail status of test cases to identify and rank faults [46] [130] [136] [4]. Existing techniques instrument a monitoring agent inside the code to obtain the runtime information about test suites [58] [79]. The instrumented monitor is responsible for tracking the executing entities such as suspicious program statements, suspicious components, inputs and outputs, pass/fail status of test cases, etc. The collected information is used in statistical techniques to compute the heuristic measure of suspicious entities. The heuristic measure indicates the probabilities of being responsible for test case failures. The suspicious entities are sorted according to their probabilities to indicate the level of likelihood for being faulty.

Jones et al. [58] presents an evaluation to analyze the effectiveness of the Tarantula technique − a fault localization and ranking technique. To evaluate its effectiveness, they compared Tarantula with Set union, Set intersection, Nearest Neighbour and Cause Transition techniques [58]. The techniques were compared on the basis of a programmer's effort in identifying the faulty statements and the actual fault. Their study illustrates that the Tarantula approach constantly outperforms other approaches.

In addition, they performed experiments to evaluate its efficiency in which Tarantula was found six orders of magnitude faster than other techniques.

Abreu et al. [4] present an algorithm BARINEL and compared it with Tarantula. BARINEL is a fault localization algorithm that produces a sorted list of faulty candidates. It takes the hit spectra [49] of a system as an input to the algorithm. The hit spectra captures the involvement of individual components in an execution through flags. The flag value '1' and '0' represents that the components was used and not used in an execution, respectively. An algorithm STACCATO [2], as a part of BARINEL, isolates the faulty components. The isolated components are further sorted according to components Bayesian probability of being faulty. Abreu et al. states that Bayesian probability is the foundation for the derivation of diagnostic candidates in any reasoning approach. However, Bayesian probability calculation may become an NP-hard problem (Section 2.2).

Abreu et al. [4] found, in their experiments, that BARINEL outperforms the Tarantula technique. BARINEL find 60% of the faults by examining less than 10% of the source code. With the same effort, the Tarantula technique finds only 46% of the faults. Moreover, in terms of effort required to identify the actual fault, BARINEL was found better than Tarantula. Overall, Abreu et al. concluded that BARINEL is an improvement over Tarantula approach for fault localization.

Gopinath et al. [45] illustrate a fault localization technique that includes spectra-based and specification-based analysis. It uses SAT technology to assess the satisfiability of correctness specifications to identify the faulty statements and to generate test cases for spectra-based fault localization. Correctness specifications are effectively first-order relational logic on the object of a program. Such specifications are written in the Alloy specification language [53]. In addition, it uses traces of a failed execution to determine the list of statements which violated the correctness specifications. Such statements are further used in spectra-based localization to obtain their faultiness ranking.

In spectra-based fault localization/ranking techniques where test coverages are used as input, it was observed that the quality of fault localization strongly correlates with the quality of test suites. There is an obligation on developers to build thorough test cases that cover almost all of the possible scenarios, which is not possible in real-world systems [141]. Zeller [141] states that having all possible test cases for a system is a contradiction to the reality of how large-scale software systems are developed. Moreover, constructing thorough test cases becomes more difficult in adaptive systems where the system's structure and behavior is decided at runtime.

2.7 Fault Localization in Practice

The global software development community still see fault analysis as a significant challenge [38]. Organizations use customized fault analysis techniques sufficient to meet their current quality of service requirements. Still, failures effect their services' availability. Fiondella et al. [38] report outage incidents for large companies such as Google, Rackspace, Microsoft, etc., where Rackspace has encountered 80, Google - 45 and Twitter - 33 outages. To prevent such outages, Naksinehaboon et al. [94] suggest rejuvenation of systems to improve availability. Many industrial practices mimic the whole system as a scaled-down version, to speed the faults analysis process. However, such scaled systems may not sufficiently exhibit the size and complexity properties of the original system [78].

Large companies, e.g., Google, have a significant dependency on the analysis of execution traces in logs to identify faults in the servers [104] [105]. Where logs are used to analyze faulty components, collective analysis of logs from multiple components, while possible, is non-trivial. This is because different logs may have different granularities, syntax and semantics [77]. Advanced logging techniques have been proposed to improve logging, but it remains difficult to correlate an error message with source code using logs [1] [139] [28]. For example, even knowing a component's source code, a propagated fault's error message "Exception in thread main: java.lang.NullPointerException" may not help to locate the actual faulty component and root cause within it. Developers also throw custom error messages which may be irrelevant or voluminous [77]. Moreover, a component's log may not outline the execution context relevant to another component, as they may be loosely coupled and correlating different logs is a time consuming process. Overall, using logs is limited for faults' correlation analysis and root cause determination.

Other techniques use a fault's stack trace. However, this is not likely to be useful in a DAS as it traces back only to the fault in the local component and does not establish correlation with faults in other components [56]. Remote debugging of remote components may be blocked and even if possible, the component structure may change while these processes are executing [88] [71]. Several mechanisms apply design-time techniques for Fault Localization in static systems [106] [116] [132] [110] [137], which could also be applied in a DAS where a new system configuration is known *a priori* [69] [21] [76]. However, it is not generally the case that all possible configurations of an adaptive system will be known at design time [111]. Nagappan et al. [93] use execution traces to create operational profiles for diagnostic tools. Nagappan et al. use regular expressions to extract log messages of interest. However, analyzing logs/traces in this manner requires an extensive set of regular expressions, which may not be feasible and may be difficult to maintain.

2.8 Summary

The Causal Graphs Models section investigates a number of causal graph-based fault localization techniques. Researchers have used graphs to analyze causal relationships between several types of software entities. For example, a fault propagation graph represents causal relationship between faults, dependency graph represents data or control relationship between components, petri-nets were used to identify the data propagation, fault-testcase dependency matrix was used to analyze the faults that may caused test case failure. Another form of graph, i.e., Bond graph, was discussed to analyze the bi-directional relationship between faults. It was found that graphs are predominantly used in fault localization techniques because of its ability to represent the relationship between any type of entity. Moreover, it is simple and efficient in that several graph traversal techniques exist which are highly efficient. A system's design e.g., fault propagations, dependencies between components, etc., can be represented in a graph and an offending entity can be identified through a graph traversal technique.

The Bayesian Networks section investigates a number of Bayesian network construction mechanism and their use in Fault Localization. Different techniques use different inputs such as logs, call traces, components dependency, etc., to construct a Bayesian network. When an execution fails, the corresponding symptom is used to conclude the Bayesian inferences which identify responsible faulty entity. Where an entity is multiply-connected with other entities, i.e., high cardinality, inferences are computationally expensive and inference time may be exponential. In a DAS where system information is partially available, which is also a case with many real-world systems, several BBN nodes remain uninstantiated and BBN inferences become an NP-hard problem.

The Program Slicing Techniques section investigates a number of program slicing techniques in the context of fault localization. These techniques divide a program's statements into a group of statements corresponding to a variable, called slices. When a corrupted variable's value cause an execution to fail, slices corresponding to the variable are identified as faulty. A major drawback to such technique is the requirement for access to source code, which may not be possible in a DAS because of inaccessible components. Moreover, the related literature indicates that such techniques generally work on test case coverage. However, running test cases at runtime in a dynamically assembled DAS is non-trivial. The number of program slices could be extremely large when a large-scale system has a large code-base. As a result, numerous slices need to be examined.

The Ontology-based Models section investigates a number of ontology-based techniques for fault localization. An ontology is the representation of the semantics of business terms and their relationships. OWL is an XML-based language to define the ontologies. Researchers have used detailed information about the structure and behavior of systems to define ontologies. In addition, they use a reasoning engine which reasons over the corresponding ontology to analyze the fault for a given symptom. In dynamic environments where a system changes at runtime, the ontology needs to be regenerated, which is cumbersome and time-consuming. Moreover, it was found that where detailed information about a system is not available (of inaccessible components), the corresponding ontology is inaccurate and the reasoning engine produces incorrect results, i.e., the wrong root causes of faults.

The Statistical Equation Models section describes a set of statistical techniques for fault localization. Such techniques require input data such as test case coverage, component's failure status for an execution, or the causal relationships between the pairs of software entities. These techniques require detailed quantitative information about involved software entities. Obtaining such detailed data in a DAS is non-trivial because of the dynamically changing structure and behavior of the system. Moreover, in a large-scale system with large number of software entities, statistical techniques may be time consuming.

The Probabilistic Ranking section describes the fault localization techniques where faulty candidates are sorted according to their probability of being faulty. Spectrum-based fault localization (SFL) is a popular technique (Section 2.6). It employs a statistical technique to which a number of inputs can be applied such as test case coverage, component;s involvement status, components failure/success status for a request, etc. Where SFL uses test case coverage, such techniques are impractical for a DAS as running test cases in a dynamically assembled system is non-trivial. Moreover, a number of techniques indicate that they read the required input data from logs which could be extremely large for large-scale systems and impacts efficiency. To sort the candidates, probabilities are calculated based on the similarity of a component's behavior to error occurrence behavior. The Ochiai similarity coefficient performs better than others such as Tarantula and Jaccard.

Fault Localization in Practice section describes the current practices of fault localization in industry. Software developers rely on debugging the code through putting checkpoints in the source code and analyzing the variables' real-time values at those checkpoints. Advanced logging and log analysis technologies are not generally used. Developers manually analyze the logs and call traces to find the root cause of a fault. Where components are distributed, developers communicate over emails or attend physical meetings to analyze the root cause of a fault.

Wrap-up A number of existing fault localization approaches were presented. Existing mechanisms are evaluated against the above five identified criteria. Figure 2.1 shows the evaluation in a KIVIAT diagram where each criteria has three levels which increase in applicability for fault localization in a DAS, the further they are away from center. The diagram shows how existing approaches address the five reference criteria of fault localization in distributed adaptive systems. A few approaches exist that attempts to address all five criteria discussed above. These criteria are used throughout the thesis to carefully design FaLDAS and to compare with other existing approaches. The diagram highlights the following findings:

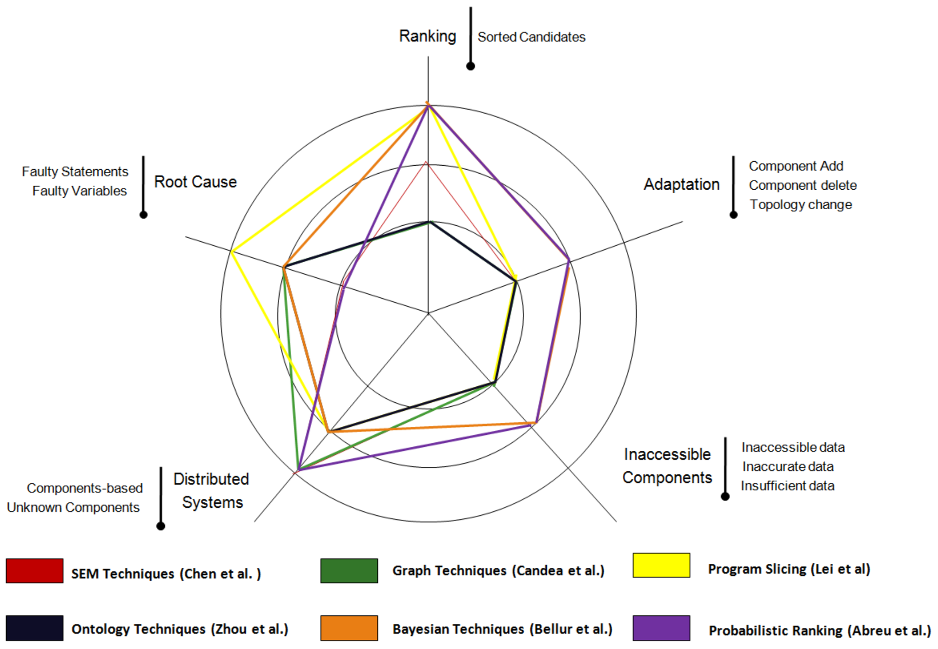


Figure 2.1: State of the art evaluation.

1. Existing techniques are inadequate to work when the components adapt at run-time. Techniques specify the collection of required information for fault localization assuming a static system. However, it is not mentioned that how to cope when such an information becomes invalid after an adaptation.

2. To the best of our knowledge, no mechanism exist that address the problem of inaccessible components. Existing mechanisms assume that the required information about all components is available, which is not the case in real-world systems. Existing fault localization mechanisms are effected when information is unavailable for actual faulty component.

3. A few techniques with some modifications can be applied, to an extent, for fault localization in DAS, although the author of such work do not claim for DAS. However, such techniques are not able to indicate the root cause, in particular faulty program statements or faulty variables. As a result, developer has to

invest a substantial amount of time in finding the actual root cause in a faulty component.

In summary, existing techniques are limited for Fault Localization at runtime in a DAS, because either they cannot cope with the adaptations, or are time-consuming and computationally expensive. FaLDAS is designed to cope with challenges presented by adaptations (section 1.3), and to efficiently locate the component(s) and corresponding corrupted variable responsible for generating a propagated fault. This thesis presents a graph-based fault localization mechanism which uses the components' runtime input output values. Even when a component's access is denied, input and output values can be gathered from a central coordinating agent such as Enterprise Service Bus, Message-oriented Middleware, etc. Graph-based search allows to identify the faulty candidates efficiently. The identified faulty candidates are fed to a statistical approach to sort the candidates according to their probability of being faulty. Integration of an existing root cause analysis mechanism with FaLDAS is demonstrated.

Chapter 3

System Model and Fault Management Scope

This chapter introduces the basic terminologies, problem, and design objectives and decisions of FaLDAS outlined in this thesis. Firstly, it introduces the terminologies related to distributed adaptive systems, dynamic adaptation and faults in Section 3.1. Section 3.2 describes the problem with an illustrative example of a DAS. Section 3.3 assesses a number of existing solutions to solve the problem. Useful insights from existing techniques analysis guided the formulation of the design objectives (Section 3.4). This chapter ends with the description of the design decisions made to fulfill the design objectives (Section 3.5).

3.1 Terminology and Scope

This thesis is developed from two key concepts: *propagated faults* and *distributed adaptive systems (DAS).* This section presents the terminology related to these two concepts and presents the scope of FaLDAS for DAS. While many of the concepts used in this thesis are familiar to distributed systems engineers and researchers, this set of definitions is included to clarify the FaLDAS assumptions, and indicate the scope of its operation.

3.1.1 Distributed Adaptive System

FaLDAS assumes a distributed, component-based system model, where components are subject to change at runtime, and faults may propagate across the system as a whole.

Definition 3.1.1. A *Component* is the building block of a DAS, and encapsulates a unit of functionality (either business or technical). A component is a unit of distribution and configuration, and can be independently developed and reused [65].

Definition 3.1.2. A *Distributed Adaptive System (DAS)* is a distributed system consisting of a set of interacting components, to which new components can be added, or any participant components can be deleted, or updated or their relationships with other components changed, at runtime. Any of these actions means that the DAS as a whole has been adapted.

A DAS may be composed of several software and hardware components such as a middleware, remote objects, databases, Java beans, application servers, etc. To maintain an acceptable quality of service, such components must be adapted. For example, a Tomcat application server is removed and IBM websphere is added or an existing Java bean is removed and a new Java bean is deployed in the software. The next section describes the considered adaptations, with examples.

3.1.2 Dynamic Adaptation

Dynamic adaptation is the process of changing the structure or behavior of a system at runtime. There are a number of adaptation patterns exist to formalize the adaptation process [43] [102]. However, this thesis concentrates on *component insertion*, *component removal* and *server reconfiguration* adaptation patterns. Such adaptation patterns can be performed using an adaptation framework, e.g., RPLUSEE [43]. An adaptation framework dynamically changes a target system from an existing configuration to a new configuration [43]. The following is a detailed description of these adaptation patterns.

Component Insertion Adaptation Pattern

The component insertion pattern describes the process of adding a new component into an existing target system [102]. An application of this pattern on a sample system is illustrated in Figure 3.1. This sample system is an access controller system on which a user enters their login details and the system authenticates the provided details using the Shibboleth authentication mechanism [90]. The system initially consists of three components: (i) A Login component which allows users to enter their login details, (ii) The Shibboleth Authentication component, which authenticates the login details, and (iii) A User Detail component which shows the logged in user's details.

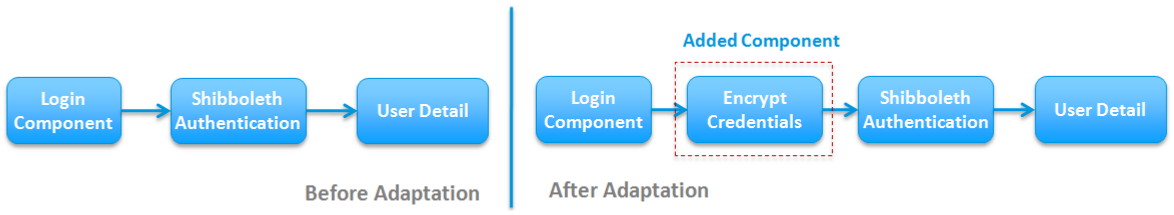


Figure 3.1: Example of component insertion adaptation pattern.

When a user enters his login details, they are sent to the Shibboleth Authentication component for authentication. However, the transportation of login details from the Login component to the Shibboleth Authentication component is not secure. The system is dynamically adapted to introduce security usign the Encrypt Credentials component, which encrypts the data transported between the Login component and the Shibboleth component. This component inserted using the component insertion pattern, as shown in the right hand side of Figure 3.1. To insert the component, the predecessors and successors components are first identified. Then, the connections between predecessors and successors are modified to deploy the target component [43] [102]. Specifically, the predecessor and successor components are put into a passive state so that no new computation starts, to prevent inconsistencies. The new component is safely inserted and initiated to its default state. Then, the concerned components are marked active to enable new computations.

Component Removal Adaptation Pattern

The *Component removal* pattern describes the process of removing a component from an existing target system. In real-world systems, an adaptation framework generally use state-based decision making to remove a component [43] [102]. To remove a component from a system, the framework checks if the component is in *active* state, i.e., the component is doing a computation. When the component completes its computation, it is in *passive* state. The adaptation framework performs the reconfiguration in which a passive state component is removed and dependencies are re-configured.

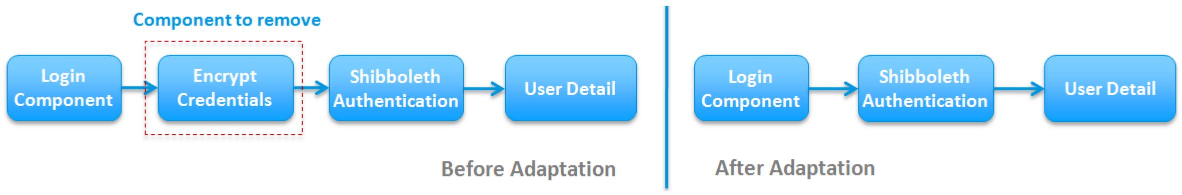


Figure 3.2: Example of component removal adaptation pattern.

As an example, let us re-consider the sample system where login details were transported in encrypted form. The system is required to adapt to allow unencrypted data transportation between Login and Shibboleth components. The adaptation framework waits for these components to complete their current computations after which the Encrypt Credentials component is safely removed as shown in Figure 3.2.

Server Reconfiguration Adaptation Pattern

The purpose of the *server reconfiguration* adaptation pattern is to add, remove and reconfigure the components. This adaptation pattern in turn is a combination of component insertion and component removal patterns [43] [102]. To reconfigure the components safely, the framework [42] wait for all the concerned components to complete their current computations and the new requests are redirect to a temporary buffer, to not loose them. The components are added, removed and reconfigured at runtime, and restarted to process requests.

An example of server reconfiguration pattern is shown in Figure 3.3. The figure illustrates the assumed sample system which is adapted through server reconfiguration pattern at runtime. The sample system initially consist of three components as dis-

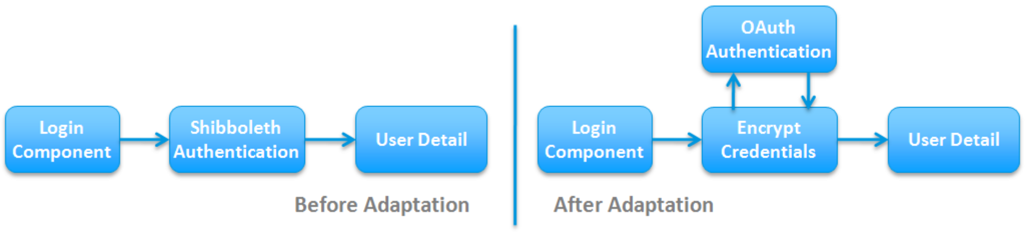


Figure 3.3: Example of server reconfiguration adaptation pattern.

cussed above. To the system reconfiguration adaptation pattern was applied which in turn applied insertion pattern to add 'Encrypt Credentials' and 'OAuth Authentication' components, and applied removal pattern to remove 'Shibboleth Authentication' component. In addition, the dependencies among components are reconfigured as shown in Figure 3.3.

3.1.3 Propagated Faults

A *fault* is a system state where a required system constraint is violated [13]. Any system is likely to have vulnerabilities, which may activate and cause a fault such as buffer overflow, null pointer exception, etc. A fault prevents a component to produce the desirable output. The corrupted output in turn causes another fault in a different component dependent (directly or indirectly) on the actual faulty component. It causes a fault to propagate from one component to another component.

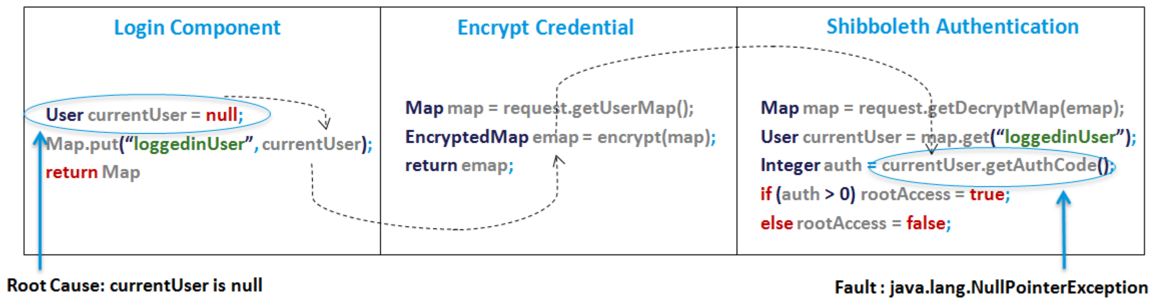
Definition 3.1.3. A *propagated fault* is a fault in a component where the root cause of the fault is in a different component. A propagated fault may have originated from one or many components, in which case it may have one or many root causes.

Definition 3.1.4. A *faulty component* is a component which has originated a propagated fault. In particular, the component corrupted an output, which further propagates and causes a propagaed fault.

Definition 3.1.5. The *root cause* of a fault is the erroneous program statement(s) in the system where a system constraint was first violated, further causing faults in connected component(s).

Figure 3.4 shows an example of a propagated fault in the running sample DAS. The DAS consists of three components 'Login Component', 'Encrypt Credential' and

‘Shibboleth Credential’, which are responsible for taking user credentials as input, encrypting details and checking access, respectively. The dashed arrows from one component to another show the data flow between components. In ‘Login Component’, the variable ‘`currentUser`’ is set to ‘`null`’ which is sent to ‘Encrypt Credential’ component as a part of the request. ‘Encrypt Credential’ encrypts the user details and puts the encrypted data in variable ‘`emap`’ which is securely sent to ‘Shibboleth Authentication’ to assess user access. The user details are retrieved from the request and stored in variable ‘`currentUser`’. This variable is used to obtain the corresponding authentication code from an authentication store. However, this task encounters a fault ‘`java.lang.NullPointerException`’ because ‘`currentUser`’ is ‘`null`’ as shown in Figure 3.4. This fault is a propagated fault because its root cause is in the ‘Login Component’. The root cause is the ‘`null`’ assigned to ‘`currentUser`’ in ‘Login components’. In this case, ‘Login Component’ is the *faulty component* and statement ‘`User currentUser = null`’ is the *root cause*. The component ‘Encrypt Credential’ is not the faulty component because it does not modify the user details being transported, rather it just encrypt whatever it receives.


Login Component
Encrypt Credential
Shibboleth Authentication
Root Cause: currentUser is null
Fault : java.lang.NullPointerException

```
User currentUser = null;
Map.put("loggedinUser", currentUser);
return Map
```

```
Map map = request.getUserMap();
EncryptedMap emap = encrypt(map);
return emap;
```

```
Map map = request.getDecryptMap(emap);
User currentUser = map.get("loggedinUser");
Integer auth = currentUser.getAuthCode();
if (auth > 0) rootAccess = true;
else rootAccess = false;
```

Figure 3.4: An example of propagated fault.

In the above discussed example, only one faulty component was responsible for originating the propagated fault. However, there may be situations where a propagated fault may have originated from one or many components, in which case it may have one or many root causes, respectively.

Definition 3.1.6. *Single Component Fault.* Where a vulnerability in only one component causes the fault to propagate, resulting in a failed execution [27].

Definition 3.1.7. *Multiple Components Faults.* Where the vulnerabilities in more than one component causes the faults to propagate, resulting in a failed execution [27].

3.2 Problem Statement

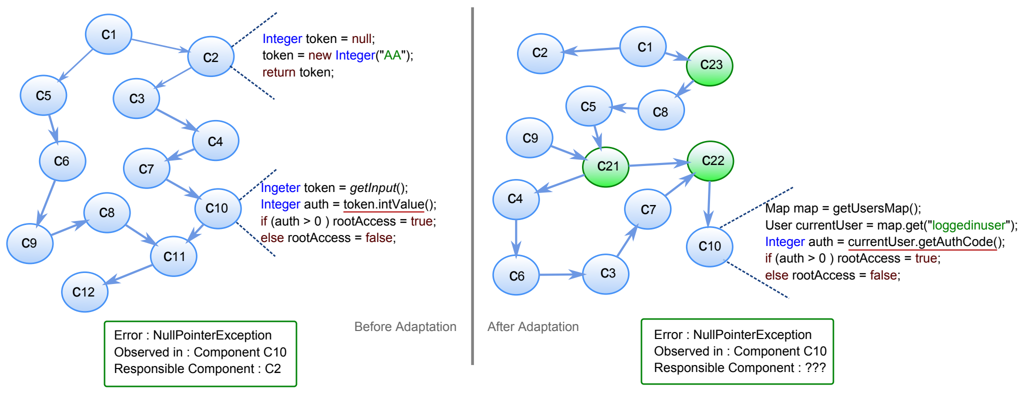
Faults are inevitable in a system. This is partly due to increasing business pressure to release a software component to hard deadlines [92]. To achieve the deadlines, developers write a limited number of test cases, which may miss the testing of a number of potential faults. A component developed under such conditions is likely to cause a fault in a system. State-of-the-art fault localization techniques report a number of reasons for a fault (potential faulty components) and require substantial manual effort to correctly diagnose the actual faulty component and root cause within it [4].

Fault localization is even more complex because the root cause may not exist in the same component where the fault occurred. Fault activation while a system is running is likely to cause an error that may propagate and cause the failure of a service [13]. A component's fault first causes an error within itself, which does not propagate to other interacting components until it leaves the boundary of the component. In fact, several faults may be handled within the component successfully. For example, the Java programming language provide constructs to catch Java exceptions in a `try-catch` block. However, this is not always the case [57]. It is not always possible to provide a catching mechanism for all possible faults in a system because of the system's complicated structure and behavior [57]. Even if all exceptions are caught, it is unlikely that the component still produces the desired output. When a fault is not handled successfully within a component, the resulting errors propagate to the component's interface. As a result, the service provided to other dependent components fails. The dependent components may receive corrupted input data that results in another fault unrelated to the original fault [13]. Effectively, a fault is propagated from one component to another component.

Without proper fault handling mechanisms installed in the system, such propagated faults may result in a failed execution. Ideally, a fault diagnosis and management mechanism should find and resolve the fault before it causes serious consequences. The challenges (Section 1.3) presented by adaptation makes the fault localization of propagated faults even more complex. The key problem is to efficiently identify the actual faulty component and potential root cause of a propagated fault in a DAS even when the DAS has inaccessible components or the system's information is inaccurate.

To explain the problem in detail, Figure 3.5 illustrates the problem using an example fault propagation in a sample DAS. Figure 3.5 shows an extended version of the sample DAS (discussed in Section 3.1.1) to introduce the complexities of a real-world system. This DAS is a distributed authentication framework that provides functionality to assess if a user is allowed root access to a secured resource. The example is motivated from real-world authentication systems such as Shibboleth [90] and OpenID [103], which provides federated authentication to several organizations such as the IEEE Institutional Login.

The left hand side of Figure 3.5 shows the sample DAS, before an adaptation, and is composed of a set of components (C_1...C_{12}). Two code snippets are illustrated: a fault activates in component C2 because of program statement '`new Integer(''AA'')`', which results in output variable '`token`' having a '`null`' value because '`''AA''`' is a word which cannot be converted into an integer by the Java compiler. This causes a '`java.lang.NumberFormatException`', which prevents the update of the '`token`' variable. As a result, '`token`' retains the original value i.e., '`null`'. Ideally, a proper fault handling mechanism should be employed here, in that the exception should have been caught. However, the source code does not reflect any such mechanism.



```
Integer token = null;
token = new Integer("AA");
return token;
```

```
Ingeter token = getInput();
Integer auth = token.intValue();
if (auth > 0 ) rootAccess = true;
else rootAccess = false;
```

```
Error : NullPointerException
Observed in : Component C10
Responsible Component : C2
```

```
Map map = getUsersMap();
User currentUser = map.get("loggedinuser");
Integer auth = currentUser.getAuthCode();
if (auth > 0 ) rootAccess = true;
else rootAccess = false;
```

```
Error : NullPointerException
Observed in : Component C10
Responsible Component : ???
```

Figure 3.5: Problem description in a sample DAS before and after adaptation

As a result, the '`null`' value further propagates through C_3, C_4, C_7 to C_{10}. The components data-dependent on C_2 receive the output computed in C_2. The component configuration in Figure 3.5 illustrates that C_3 is dependent on C_2. Therefore, the computed value of variable '`token`', which is '`null`', is passed on from C_2 to C_3. Similarly, the data propagates through C_4 and C_7, and reaches C_{10} as input. In this case, components C_3, C_4 and C_7 are not manipulating the data during transportation as they only save data to different databases. Finally, the '`null`' value computed in C_2 is passed through C_3, C_4 and C_7, and appears as an input to C_{10}, because the '`null`' value has not been updated during the transportation.

The component C_{10} receives the input and assigns it to one of its variables '`token`' which ultimately becomes `null` as shown in the source code snippet corresponding to C_{10} (left of Figure 3.5). The second line of the source code uses the variable '`token`' to invoke the method '`intValue()`'. However, this statement encounters a fault[1] as 'java.lang.NullPointerException' because the method's invoking object, i.e., '`token`', is '`null`'. This illustrates the propagation of fault '`java.lang.NumberFormatException`' from C_2 as an apparently unrelated fault '`java.lang.NullPointerException`' to component C_{10}.

As shown, component C_2 originates and propagates the fault to component C_{10}. Before any adaptation, or in a non-adaptive system, a developer could determine this using existing mechanisms [17] [6] [81] [21]. For example, Candea et al.'s [21] fault injection technique may help as they inject a fault in components and analyze effects in other components.

However, say the system is adapted to use a different mechanism to check root access authority. Components 'C_{11}' and 'C_{12}' are removed, 'C_{10}' is updated, 'C_{21}', 'C_{22}', 'C_{23}' are added, and the component topology is changed as shown in Figure 3.5 (after adaptation). In this case, assume that the adapted component 'C_{21}' causes a fault, which is propagated and manifests as an apparently different unrelated fault '`java.lang.NullPointerException`' in component 'C_{10}'. The existing fault propagation graph (pre-adaptation graph) indicates that 'C_2' is responsible, which is wrong because 'C_{10}' is no longer dependent on 'C_2' in the post-adaptation configuration. Clearly, the existing graph is now invalid because it is inconsistent with the new configuration e.g., the dependency of 'C_{10}' on 'C_2' is no longer valid. An important question arises as to how to identify the actual faulty component 'C_{21}' responsible for fault in 'C_{10}' at runtime. The following section analyzes a number of other mechanisms to carry out

[1]In the Java programming language, no method can be invoked using null as an object. Such statements throw java.lang.NullPointerException.

fault localization in the post-adaptation system.

3.3 Existing Solutions

When a fault occurs in a system causing a failed execution, the debugging experts usually use logs to analyze the evidence of a fault [53] [104] [105][28]. Occasionally, they use remote debugging to identify the problem in a remote component [88] [71]. When remote access in unavailable, the team members usually have physical meeting and conference calls to discuss issues. However, it is time consuming to contact team members and discuss issues. In large-scale systems, this involves a lot of time and cost to localize such faults. Moreover, a component's design may be too complex for a development team to identify the faults within it. In such complicated and demanding environments, developers use existing fault localization mechanisms (discussed in Section 2) to minimize time, effort and cost. The following sections discuss some existing mechanisms to solve the state problem in Section 3.2. The existing techniques are categorized as (i) Design-time techniques which use design-time generated information, and (ii) Run-time techniques, which use dynamically generated information for fault localization.

3.3.1 Design-time Techniques

A design-time fault localization technique analyzes a system at design-time and generates a knowledge-base about possible faults and their propagation. When a fault occurs at runtime, these techniques refer to that knowledge-base to reason about the possible causes of the fault. For example, Candea et al. [21] inject faults in a component and analyze the effects in other components. Such faults and information about their effects is saved in their knowledge-base at design-time and used at run-time. Such design-time analysis using fault injection result in a fault propagation map as shown in Figure 3.6. However, when a fault occurs in component 'C_{20}' at runtime, this map is invalid for fault localization as it does not show component 'C_{20}' and related fault propagation paths. It shows that the design-time knowledge-base becomes invalid after an adaptation because new components and their faults are added to the system, and existing components may have been removed. It can be argued that this knowledge-

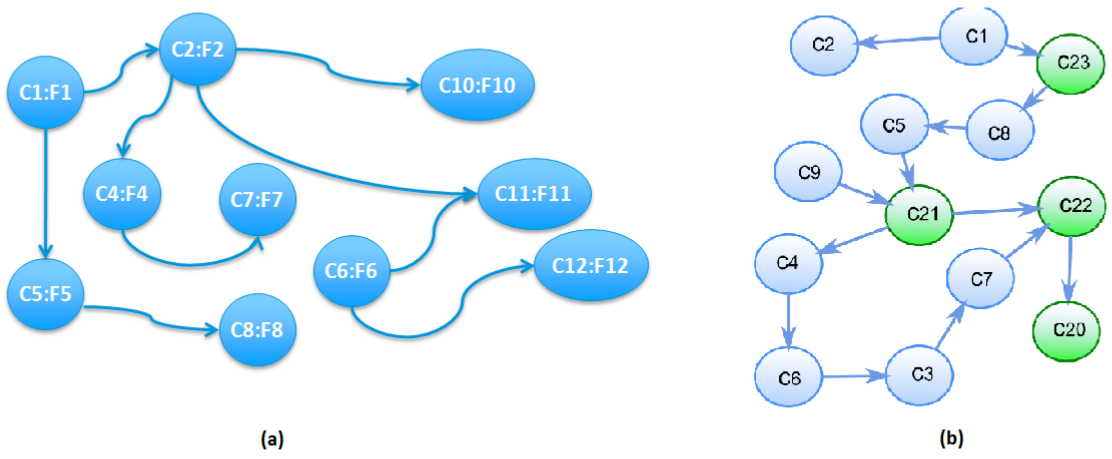


Figure 3.6: (a) Fault Propagation Map resulted from fault injection (b) Current system after adaptation.

base could be re-generated at runtime. However, this is not feasible as components may not allowed fault injection and even if allow, it is a time consuming process and requires several reboots that may negatively impact the availability of the system.

Wang et al [122] create a dependency matrix representing the dependencies between a fault and a test case. The dependency represents whether a test case supports the identification of a fault. The matrix is generated at design-time. When a fault occurs at runtime, the algorithm reasons over the knowledge-base, analyzing the test cases with a dependency on the fault. As discussed earlier, design-time generated matrix may no longer valid after an adaptation. Moreover, generating such a matrix at runtime is not always possible because the system may have several inaccessible third-party COTS components, e.g., C_{21}, C_{22}, whose access to test cases may not be available.

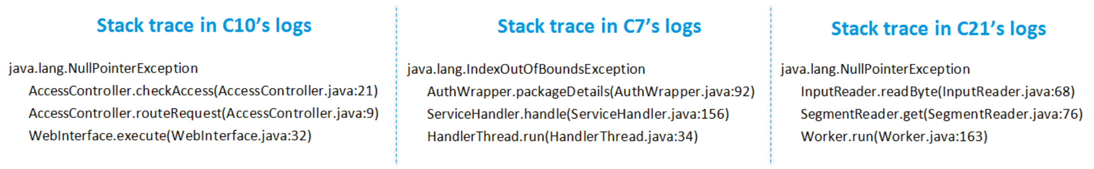
There are a number of program slicing techniques available which generate a number of slices for a particular variable to analyze the slice where it might get corrupted (discussed in Section 2.3). For example, Lei et al. [74] identify a number of slices from different components and apply a statistical mechanism on those slices to identify the corrupted slices responsible for generating the fault. However, design-time generated slices are not consistent with the adapted system. In addition, identifying program slices from components' source code at runtime after an adaptation is not possible where components do not allow access to their source code.

In summary, the mechanisms that use a design-time generated knowledge-base are not suitable because the knowledge-base become invalid for the adapted system. Moreover, such techniques require detailed knowledge about components, such as source code, test cases, etc., which is not always possible to obtain at runtime from an adapted system. Even if possible, generating such data is a time consuming process which may impact the availability of the system. Overall, such mechanisms are not suitable for fault localization in a DAS.

3.3.2 Run-time Techniques

Some mechanisms handle system adaptation using real-time data to dynamically construct their knowledge-bases [27] [23] [17]. The most frequently used used sources of information that may help ascertain the potential root cause are stack-traces or error messages in components' logs.

As an example, stack traces found in the logs for components C_{10}, C_7 and C_{21} are shown in Figure 3.7. The stack traces or error messages in a component's log may not outline the execution context related to other, potential faulty, components. C_{10}'s stack trace indicates that `java.lang.NullPointerException` occurred at line 21 of `'AccessController.java'` located in C_{10}. However, there is no reference to either C_{21} or C_7. In this example, C_{10}'s developer needs other components' logs, in this case C_7 and C_{21}, where access may be blocked because they are COTS components with security concerns (Section 1.3.1). Even if the logs are accessible, correlating them comprehensively is complicated because they may have different schema and represent different semantics as evident in Figure 3.7, where C_{10}'s and C_{21}'s stack traces are not correlated (Chapter 2). Moreover, the logs may be massive, and their analysis correspondingly computationally inefficient.


Stack trace in C10's logs

```
java.lang.NullPointerException
    AccessController.checkAccess(AccessController.java:21)
    AccessController.routeRequest(AccessController.java:9)
    WebInterface.execute(WebInterface.java:32)
```

Stack trace in C7's logs

```
java.lang.IndexOutOfBoundsException
    AuthWrapper.packageDetails(AuthWrapper.java:92)
    ServiceHandler.handle(ServiceHandler.java:156)
    HandlerThread.run(HandlerThread.java:34)
```

Stack trace in C21's logs

```
java.lang.NullPointerException
    InputReader.readByte(InputReader.java:68)
    SegmentReader.get(SegmentReader.java:76)
    Worker.run(Worker.java:163)
```

Figure 3.7: Stack traces in the logs of components 'C_{20}', 'C_7' and 'C_{21}' - After Adaptation.

A number of techniques use causal graphs for fault localization [17] [81] [10] [21]. A causal graph, in general, is a directed acyclic graph where nodes represent software entities and edges represent causal relationship between those software entities. For example, Bellur et al. [17] use Bayesian Belief Network (BBN) graph whose nodes represent the faults and edges represent probabilities of a fault;s propagation from one component to another. It was not clear in Bellur et al.'s work whether they create BBN at design-time or at runtime. So, both the cases are analyzed where the BBN is constructed at design-time and at runtime.

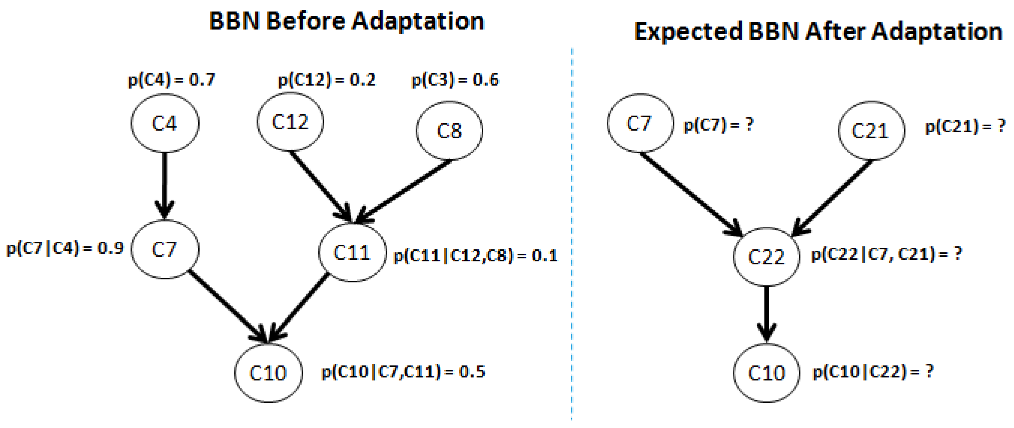


Figure 3.8: An excerpt of a Bayesian Belief Network for the sample DAS. The symbol p(X|Y) represents the posterior probability of hypothesis variable X for the evidence Y.

When a BBN is constructed at design-time, it will be invalid at runtime if an adaptation has occurred as discussed in Section 3.3.1. Figure 3.8 shows that BBNs before an adaptation and expected after an adaptation are different. The C10 node has two parents C7 and C11, but after an adaptation C10 has only one parent, C22. This shows that BBNs (or other types of causal graphs) created before an adaptation may not be valid after an adaptation. To keep the BBN synchronized with the system's configuration, a BBN may be re-generated at runtime.

Logs are predominantly used to observe the real-time diagnosable information to locate a fault. However, logs analysis faces several challenges which reduces their usefulness in effective fault localization [96]. For example, the required entries may not be present in the logs if the developer does not implement a mechanism to log specific events. Moreover, improper log schema or overlapping of event messages may make it difficult to distinguish the required logs from the rest of the logs. Oliner et al. [96]

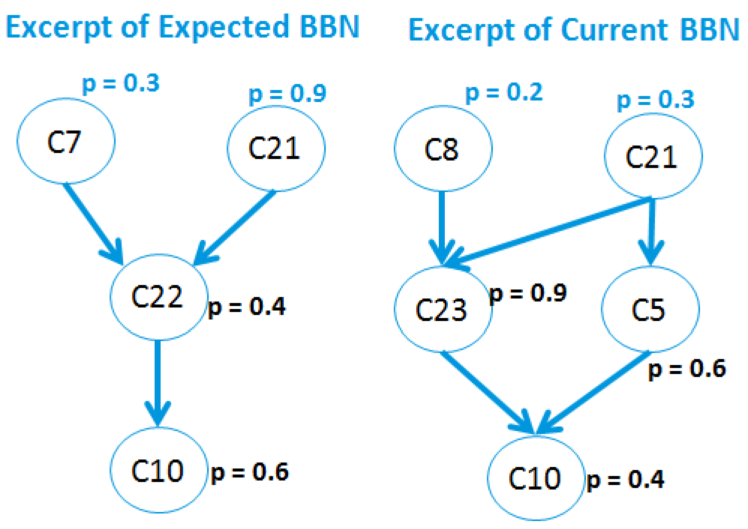
present such an example using a log message from BlueGene/L supercomputer:

```
YY-MM-DD-HH:MM:SS NULL RAS BGLMASTER FAILURE ciodb exited normally with
exit code 0
```

This message may wrongly indicate a failure event (through the `FAILURE` word) which may have been generated in a successful execution because the programmer may have mistakenly included `FAILURE` is a successful log message [96]. In addition, logs are generated using streams of text. Where multiple sources are logging in the same file, there is a chance that the log messages may be interleaved. Without proper identifiers in the logs, interleaved messages are complicated to assess the failure of a particular entity. Another major challenge is the volume of logs, which could be extremely large for a large-scale system. It is because developers tend to log every event such as polling a device, events in a timed recurring task, acquiring and releasing thread locks, etc. Heterogeneous components are likely to produce heterogeneous logs which are difficult to analyze straightaway. Overall, logs are useful in real-time fault localization but their analysis is complicated and time consuming, which limits its applicability in DAS.

Incorrect logs result in incorrect fault localization. For example, Figure 3.9 shows (a) an expected (correct) BBN, and (b) the current (incorrect) BBN in that the 'C_{10}' node should have only one parent 'C_{22}' but has two parents 'C_{23}' and 'C_5'. The BBN shown is incorrectly constructed through Bellur et al.'s approach because of interleaved call traces in the logs. A parent is expected to log its call trace prior to the child to indicate the dependency or data flow path. Since call traces are written as streams of characters in the logs, the streams were interleaved [96] i.e., C_{22}'s call trace stream was not written prior to C_{10}'s, C_{23}'s and C_5's were found prior to that of C_{10}. Such interleaved traces do not illustrate the correct parent-child relationships: here, C_{10} has two parents C_{23} and C_5, but it should have only one parent C_{22}. An incorrect BBN may result in incorrect posterior probabilities e.g., the BBN inferences wrongly indicate the actual faulty component as C_{23} because its posterior probability was highest for the given evidence [17] whereas C_{21}'s probability should be highest as shown in the expected BBN (Figure 3.9). Moreover, when components deny access to their logs, such techniques are not feasible. Even if the BBN is generated at runtime, it must wait for a sufficient executions to train the BBN, calculate the nodes' Bayesian probabilities, and conclude inferences.

Similarly, spectra-based fault localization techniques (e.g., [4] [2]) may produce an



Actual Faulty Component:	C21
Identified Faulty Component:	C23

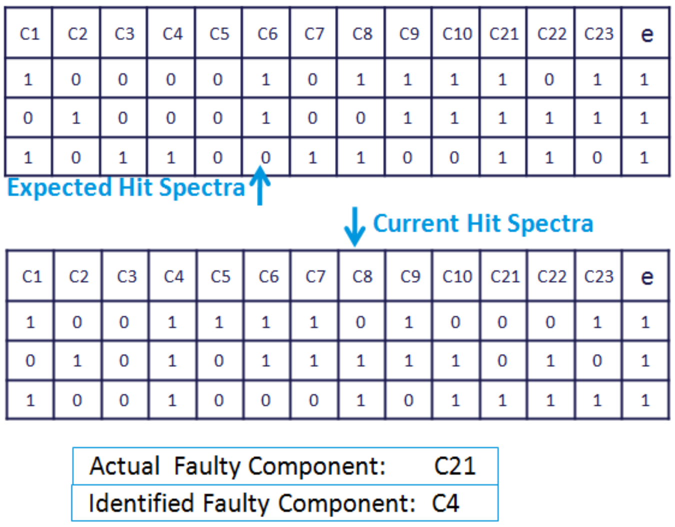
Figure 3.9: Example of incorrect BBN

incorrect result as shown in Figure 3.10. The hit spectra was generated from logs with interleaved log entries, causing incorrect status values [96], e.g., component C_{21} must have $< 1, 1, 1 >$ but has $< 0, 0, 1 >$. In this case, Abreu et al.'s technique [4] [2] wrongly identifies C_4 as the most likely faulty component because C_4's entries are all 1s, indicating the highest similarity with the error vector, whereas C_{21} should have the highest similarity.

In summary, responsible faulty component localization for propagated faults remains a non-trivial problem in a DAS. A number of Fault Localization techniques [2] [4] [27] [17] attempt to address this issue. This thesis compares FaLDAS's efficiency and accuracy results against that of BARINEL [4], Pinpoint [27] and Bellur et al. [17] approaches.

3.4 Design Objectives

This thesis's aim is to develop a mechanism to support root cause analysis of a propagated fault in a DAS, focussing identifying the component that originates the prop-


Expected Hit Spectra
Current Hit Spectra

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C21	C22	C23	e
1	0	0	0	0	1	0	1	1	1	1	0	1	1
0	1	0	0	0	1	0	0	1	1	1	1	1	1
1	0	1	1	0	0	1	1	0	0	1	1	0	1

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C21	C22	C23	e
1	0	0	1	1	1	1	0	1	0	0	0	1	1
0	1	0	1	0	1	1	1	1	1	0	1	0	1
1	0	0	1	0	0	0	1	0	1	1	1	1	1

Actual Faulty Component:	C21
Identified Faulty Component:	C4

Figure 3.10: Example of inaccurate spectra

agated fault. This research developed designed to address the challenges described in Section 1.3. In a DAS, a debugging expert's goal is to identify the root cause of a fault efficiently, accurately and with minimum manual efforts. Existing mechanisms do not fulfill this goal because mechanisms can not cope with adaptations in a DAS. The mechanism described in this thesis, called FaLDAS, helps the expert to efficiently identify the root cause of a fault in a DAS even when the DAS includes inaccessible components or the monitored system's information is inaccurate. To address the challenges, a fault localization mechanism must:

- **Design Objective 1: Gather real-time data**
  The mechanism should have, or be capable of integrating, a monitoring mechanism which can deploy its monitoring agent across a DAS to gather the required real-time data to cope with the evolving configuration of the system.

- **Design Objective 2: Synchronize knowledge-base**
  In an evolving system, a fault localization mechanism should constantly update its knowledge-base to reflect information about current faults and their propagation behavior. It should make sure that the knowledge-base is not out-of-date and is valid after an adaptation.

- **Design Objective 3: Diagnose inaccessible components**

An evolving system may have several third-party COTS components, and access to their internal structure and behavior may be unavailable. A fault localization mechanism should be capable of diagnosing such components for any faulty behavior.

- **Design Objective 4: Cope with inaccurate knowledge-base**
  It is likely that monitored data from a real-world system may be inaccurate because of the system's complicated structure and monitoring technologies' limitations. As a result, the generated knowledge-base may be inaccurate. A fault localization mechanism must be able to assess a component's faultiness even if its information is inaccurate in a mechanism's knowledge-base.

- **Design Objective 5: Sort the diagnosed candidates**
  A fault localization mechanism may returns a number of faulty component(s), all of which may not be actually faulty. Time spent in analyzing a component that is not actually faulty, is time wasted. A fault localization mechanism should produce a sorted list of diagnosed faulty components, in order of their probability of being faulty, to minimize root cause search effort.

- **Design Objective 6: Integrate with root cause analysis**
  The ultimate goal of any fault analysis mechanism is to identify the root cause of a fault so that corrective measures can be taken. Therefore, it is importance that a fault localization mechanism should demonstrate its integration with a root cause analysis, otherwise fault localization results may be of little help.

Figure 3.11 shows how these design objectives address the target system characteristics described in section 1.3 and 1.4. To cope with an adaptive system, FaLDAS generates and gathers real-time data about a DAS. For example, instead of using design-time components dependencies, it generates the dependency information at runtime. Moreover, the gathered information support to update the knowledge-base. The technique must be able to work in the context of an adaptive system otherwise gathering of real-time data after an adaptation may not be possible. When an adaptation introduces inaccessible components into the system, the technique must be able to diagnose such components. In such cases, it may happen that the collected information may be inaccurate. Therefore, the technique must be able to cope with such inaccuracies in the required data. A reasonable technique in the state-of-the-art is to provide a ranked list of diagnosed candidates according to their probability of being faulty. Therefore, the technique must be able to sort the diagnosed candidates. The next section demonstrates the characteristics of FaLDAS which address these objectives.

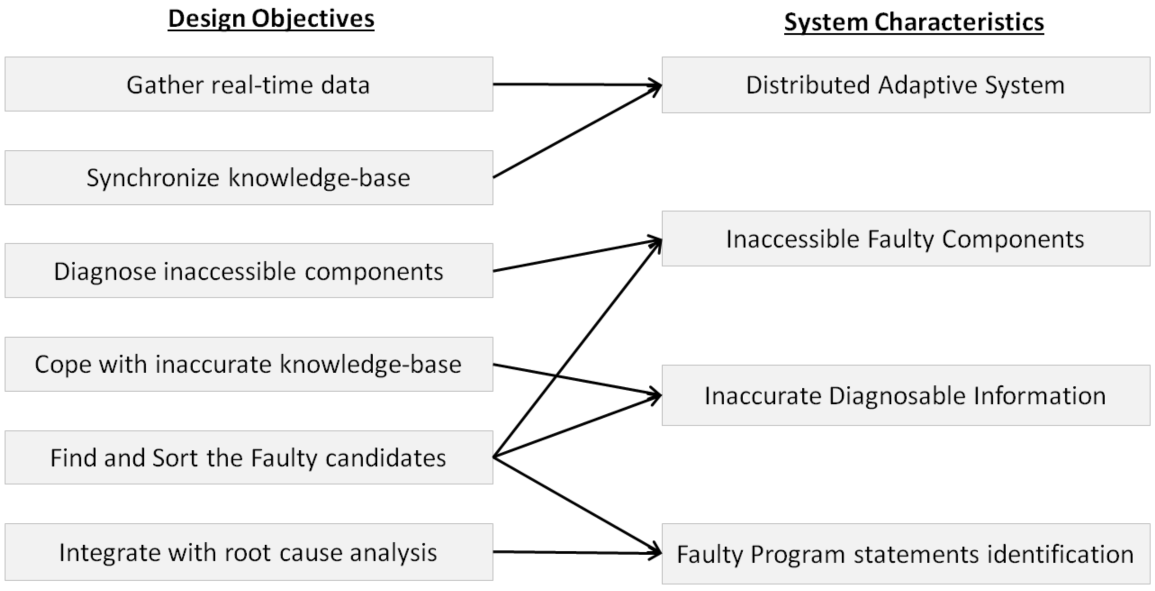

Design Objectives
System Characteristics
Gather real-time data
Synchronize knowledge-base
Diagnose inaccessible components
Cope with inaccurate knowledge-base
Find and Sort the Faulty candidates
Integrate with root cause analysis
Distributed Adaptive System
Inaccessible Faulty Components
Inaccurate Diagnosable Information
Faulty Program statements identification

Figure 3.11: Design Objectives to address the target system characteristics

3.5 Design Decisions

This section illustrates the FaLDAS design decisions that fulfill these design objectives.

3.5.1 Fault Localization in Adaptive Context

A number of existing mechanisms have the potential to carry out fault localization in adaptive systems [25] [123] [9] [2] [24] [27] [17], though none explicitly claims that they work on adaptive systems. A major requirement is to integrate with the adaptive systems, which has not been described by their authors. For example, Abreu et al. [2] use hit spectra but do not specify how and when it is generated in the overall context of adaptive systems management. Bellur et al. [17] use execution traces to generate a BBN but it is unclear how to monitor the execution traces of adapted components.

The Monitor, Analyze, Plan and Execute (MAPE) loop [62] is a widely used model for adaptive systems management, and this section describes the integration of the Fault Localization in Distributed Adaptive System (FaLDAS) mechanism within the context of MAPE. MAPE has clear distinction between different management phases, which supports analysis of how and where to place FaLDAS. The MAPE loop has four phases: (1) Monitor - which monitors the characteristics of an executing system such as real-time request response data, performance parameters, fault symptoms, environment variables, etc., (2) Analyze - which aggregates the monitored data to draw conclusions, e.g., an error occurred, an SLA is missed, performance is degraded, etc., (3) Plan - which identifies the most suitable adaptation in terms of addition or removal of components, reconfiguration of topology, etc., and (4) Execute - which implements the planned adaptation on the target system.

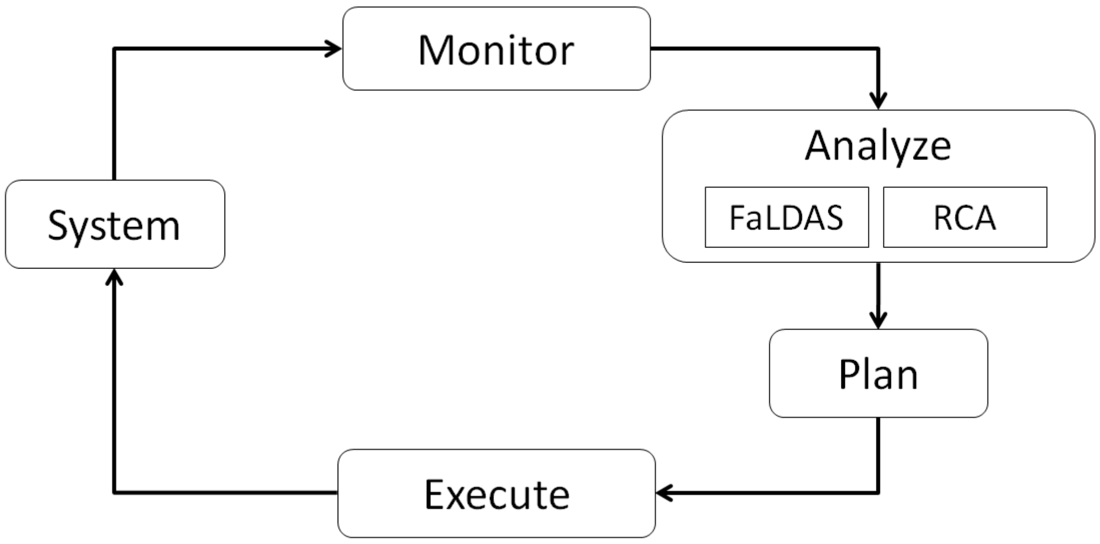


Figure 3.12: Overview of fault localization and root cause analysis in the context of adaptive system management.

Design Decision 1 - FaLDAS in the Analyze phase of the MAPE loop: FaLDAS fits inside the Analyze phase of the MAPE loop because this phase receives monitored data which guide further improvements in the system, e.g., recovery of the system in case of a fault (Figure 3.12). FaLDAS uses monitored data to identify a faulty component responsible for a propagated fault, which further supports root cause analysis to find actual cause within the faulty component. The Analyze phase's output i.e., root cause can be further used in the next phase - Plan phase - to identify an adaptation plan and execute the necessary recovery actions. This addresses the design objectives 1 and 2.

3.5.2 System Monitor

A business analyst or domain expert specifies a system's goals. Suitable monitors are then developed to analyze violations of the defined goals and placed in the system [123]. For example, Wang et al. [123] analyze goal violations using the truth values of corresponding goals' conditions. Monitors equipped with such logic are used to identify the goals' violations, which represent the occurrence of a fault. When a fault occurs, a system administrator (probably automated) triggers FaLDAS to identify the faulty candidates i.e., faulty components.

There are several techniques for monitoring the run-time data of a DAS [52] [72] [64] [32] [48] [15] [113]. Ilarri et al. [52] describe a decentralized technique using instrumentation within mobile agents' components, monitoring distributed systems running in a dynamic environment. Legrand et al. [72] developed a framework 'Monalisa' to monitor large-scale distributed systems where monitoring agents are tightly integrated within the components. However, instrumentation and integrating monitoring agents with components is not always possible and permissible. Moreover, agents may not be allowed to access the relevant data for fault diagnosis, e.g., source code, test cases, stack traces, etc. FaLDAS requires information only about requests to, and responses from, a component, which does not require accessibility to the internal design of a component [80]. As such, an interceptor-based monitoring mechanism that intercepts such information is sufficient for FaLDAS, as shown in Figure 3.13.

Such an interception mechanism has been used in a number of fields such as Quality of Service (QoS) management, load balancing, system audit and also in fault management [12] [80] [40]. Generally, debugging experts implement an intercept mechanism to monitor a DAS. The interceptor must be capable of intercepting a range of data types because heterogeneous components will be under consideration. Otherwise, com-

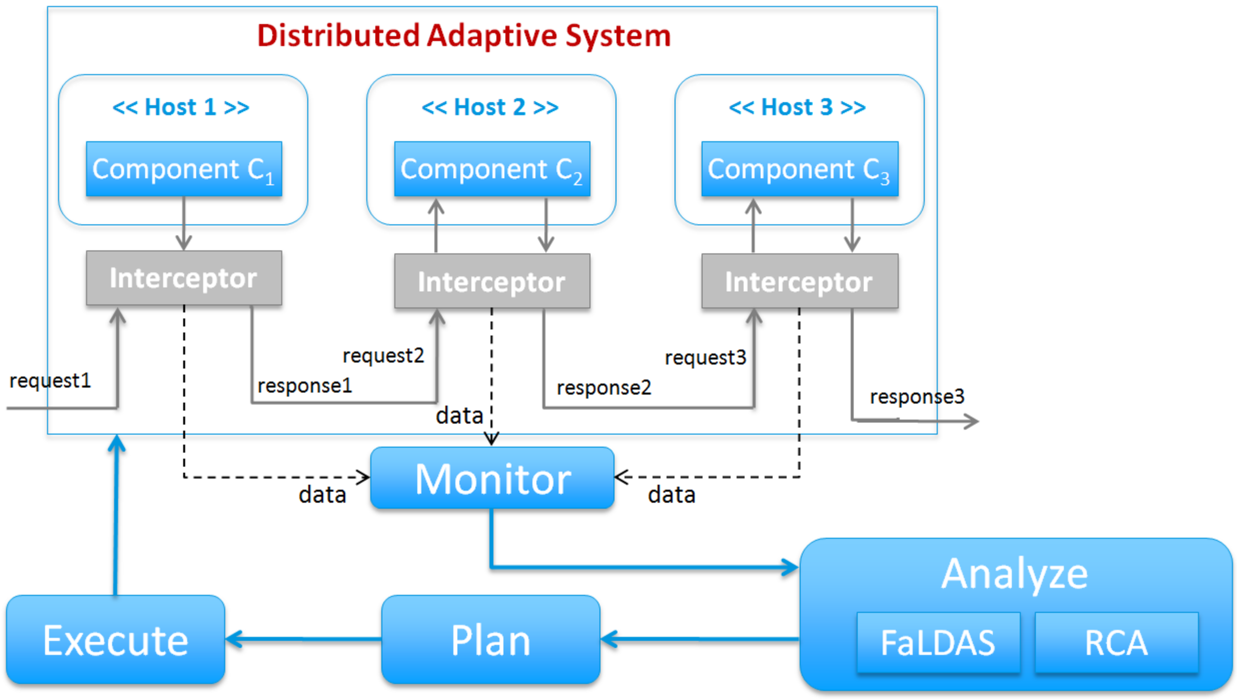


Figure 3.13: Interceptor-based Monitoring Mechanism

patibility issues may emerge, such as, for example, a JSON request attribute may not be intercepted by an XML interceptor. Lin et al. [80] illustrate the use of an interceptor based on abstract data types, which can intercept a wide range of data types parameters. Where a target component's host machine is accessible, an interceptor is placed on this host. For example, where Apache Tomcat is the host to a Java-based component, the interceptor module will be plugged-in to the Apache Tomcat. The host engine loads and initializes the interceptor to track the target requests and responses. Where a host machine is inaccessible, the interceptor should be placed in a location that has closest access to the requests to and responses from the target component. The interceptor must provide an application programming interface (API) through which a central monitoring mechanism may obtain the intercepted data as shown in Figure 3.13.

Design Decision 2 - Interceptor-based Monitoring: This thesis proposes the use interceptor-based monitoring of individual components' requests and responses during an execution. FaLDAS uses this data to construct its knowledge-base. Using such monitoring, FaLDAS can gather information about inaccessible components and diagnose it for its faultiness. This fulfills design objectives 1 and 3 in that the gathered information is real-time and inaccessible components can be diagnosed.

3.5.3 Knowledge-base

A fault localization algorithm reasons over a knowledge-base to identify the faulty candidates. A number of mechanisms have been discussed in Chapter 2 and their corresponding knowledge-bases, e.g., components' dependencies, fault propagation graphs, Bayesian Belief Networks, program slices, faults' ontology, program spectra and statistical equations. These knowledge-bases can be categorized as (i) graph-based, e.g., fault propagation graph or Bayesian Belief Network , and (ii) non-graph based, e.g., ontology or program spectra.

Non-graph based knowledge-bases (i) can not be easily visualized, e.g., complicated equations or matrix cannot be understood easily and (ii) require a complicated statistical mechanism to reason over the knowledge-base, which may be time consuming. On the other hand, graph-based knowledge-bases can be (i) visually analyzed in a graphical editor, (ii) reasoned with a simple BFS or DFS search, and (iii) traversed more efficiently than non-graph-based ones. Graph-based knowledge-bases appear suitable for FaLDAS as graph construction and searching is likely to be efficient even for large-scale systems.

Most existing mechanisms that use a number of graph-based knowledge-base represent detailed knowledge about components in their graphs, e.g., Candea et al. [21] and Liu et al. [81] represent a component's faults as nodes in the graphs, Le et al. [69] represent program code as nodes in the graphs, Lei et al. [74] represent program slices as nodes in the graph to indicate dependency between slices, etc. These authors assume complete access to the required information (source code, tests, faults, etc.,) about the components, which is not a safe assumption in a DAS. When an inaccessible third-party COTS component is added to the system, obtaining faults, test cases, program slices, etc., is non-trivial. To deal with the inaccessibility of components, FaLDAS constructs its knowledge-base using components' external information i.e., inputs and outputs. This has the benefit of not requiring internal access to a component.

The FaLDAS knowledge-base is a collection of Fault Propagation Graphs (FPG). An FPG is a set of FPGNodes and edges where an FPGNode is combination of a component name and its output parameter. Edges of the graph represent fault propagation behavior between components. Existing mechanisms either assume that such a graph already exists or they construct it manually which is limited to the designer's knowledge of the system. FaLDAS automatically constructs FPGs from components' name and output variables, at runtime. This addresses the design objective 2 and 3 in that it always maintains the knowledge-base synchronized with the current system's configuration and is able to generate data for inaccessible components.

> **Design Decision** 3 **- Automatic, run-time construction of FPGs:** When an execution completes, FaLDAS obtains monitored data i.e., the components names and their real-time output parameters. FaLDAS uses these to construct the FPGs at runtime to synchronize the knowledge-base. This fulfill design objective 2. Moreover, information about inaccessible components is generated, which enables their diagnosis; fulfills design objective 3.

3.5.4 Suspicious Components Isolation

When a fault occurs in a DAS, the first step is to identify suspicious components. Suspicious components are those which are likely to be faulty but may not be actually responsible for originating the fault. In a small scale system, such components can be visually identified by investigating burned or broken components, or the designer may pinpoint such components using fault symptoms. In large-scale complicated systems, an automatic suspicious components isolation technique is required. A number of such techniques exist which are discussed in Chapter 2.

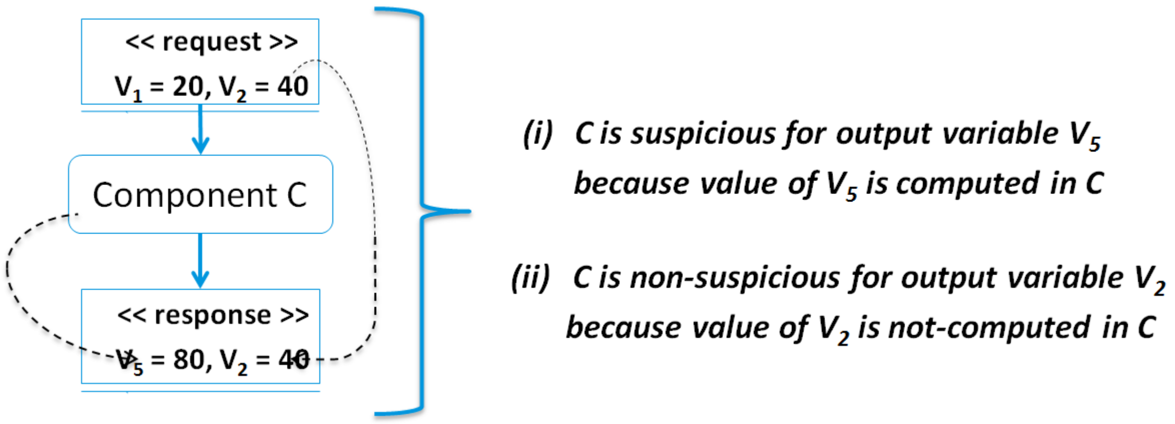
A promising technique is to sort components according to their probability of being faulty [17] [24] [4]. A major drawback of such mechanisms is that they include all the components involved in an execution, which increases the search space. However, the search space can be reduced by ignoring a number of obviously not faulty components; producing a a more precise set of potentially faulty components. For example, assume a system with three components (C_1, C_2 and C_3), which computes the value of expression

$(x + y) \times z$ as shown in Figure 3.13. The component C$_1$ computes $(x + y)$ and sends result to C_2, which forwards the computed sum to C$_3$. The component C$_3$ multiplies the received sum by z and produces the final output. When the final output is not desirable, existing fault localization mechanisms include all the three components in their candidate set. However, it is clear that C$_2$ cannot be responsible for faulty output because it did not take part in the computation rather it just forwarded the output of C$_1$ to C$_3$. Excluding C$_2$ from fault localization may result in a more precise candidate set. Moreover, it reduces the time it takes a developer to pinpoint the actual faulty component in the candidate set.

Unlike existing mechanisms, FaLDAS first finds the suspicious components and then sorts them to produce a precise candidate set. Existing mechanisms use binary decisions ($true$ or $false$) to indicate a component's suspiciousness [17] [4]. However, FaLDAS uses a fine-grained approach to identify a component's suspiciousness i.e., output parameter-specific. A component as a whole is not suspicious rather it is suspicious if it is possible that it corrupted only a set of output variables. A component is marked suspicious for its corrupted output variables as shown in Figure 3.14. When an input parameter is exactly the same as an output parameter (e.g., V$_2$ in Figure 3.14), the component has not computed (or not corrupted) that output parameter i.e., the component is non-suspicious for the undesirable value of that output. Otherwise, the output parameter is computed (or corrupted) in the component i.e., the component is suspicious for the undesirable value of that output. A brief summary of all possible related cases is presented in Section 6.1.1.

The identification of the suspiciousness of a component per output parameter-specific supports FaLDAS managing inaccurate information. Using a binary decision model, inaccurate information inf wrongly indicates the suspiciousness of a component, as a whole. However, in FaLDAS, inaccurate information only indicates suspiciousness of a part of the component. In terms of impact, inaccurate information impacts only a part of the component. In other mechanisms, it impacts the whole component. It enables FaLDAS to produce suitable results even with inaccurate information.

> **Design Decision** 4 **- Per parameter-specific suspiciousness detection:** FaLDAS uses per parameter-specific suspiciousness identification of a component to manage with inaccurate information. In-



<< request >>
V_1 = 20, V_2 = 40
Component C
<< response >>
V_5 = 80, V_2 = 40
(i) C is suspicious for output variable V_5 because value of V_5 is computed in C
(ii) C is non-suspicious for output variable V_2 because value of V_2 is not-computed in C

Figure 3.14: Description of the suspiciousness of a component in the context of different output variables

accurate information is not likely to impact the whole component, rather it may affect the analysis of a part of a component. This minimizes the negative impact of inaccurate information, which ultimately minimizes the impact on the fault localization results. This fulfills the design objective 4 in that parameter-specific suspiciousness identification supports FaLDAS to work with inaccurate information.

The next step is to find the actual faulty component out of all the potentially suspicious components identified, which is discussed in the next section.

3.5.5 Faulty Candidates Localization

A suspicious candidate, alone or along with other suspicious candidates, may precipitate a failed execution. Such suspicious candidates is referred to as faulty candidates. The process of finding faulty candidates and sorting them is shown in Figure 3.15.

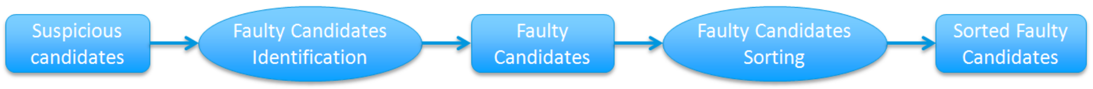


Figure 3.15: Process flow to identify and sort the faulty candidates.

Faulty Candidates Identification

A number of mechanisms for faulty candidates identification were discussed in Chapter 2. For example, statistical equation model techniques define the system model, which requires components' internal design (e.g., dataflow among program slices), which may be restricted. Program slicing and ontology-based techniques require a component's internal design and structure. Bayesian networks techniques have a learning curve which negatively effects candidate's identification.

Every technique has its benefits and drawbacks in different situations. Therefore, FaLDAS is based on a hybrid approach, which integrates Causal graph-based and Spectra-based Fault Localization (SFl) techniques. Causal graphs, used as FPGs, represent the causal relationship between any two software entities, e.g., program statements, variables, services, components, machines, etc. Traversing such a graph to find an entity is relatively faster than that of other data structures. In addition, causal graphs can be investigated visually, which may help a developer to find a trivial fault quickly, e.g., a missing component, incorrect topology, etc.

The SFL technique is used because it considers both failed and successful executions, unlike other mechanisms which make decisions based on failed executions only (discussed in section 2.6) [50]. This helps FaLDAS to identify more accurate faulty candidates [5].

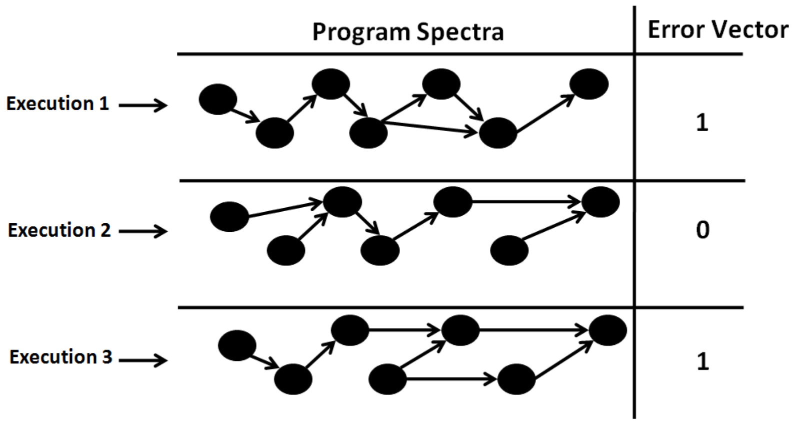


Figure 3.16: A sample program spectra in the form of Fault Propagation Graphs for three different executions

Although SFL-based techniques have been shown to be substantially effective [5], these have a number of drawbacks. SFL techniques based on low granularity entities (e.g., statements, variables, program slices) may show better results than that of coarser grained entities [50]. However, to obtain information about low granularity entities, access to the component may be required, but denied. To deal with this problem, FaLDAS uses FPG nodes as entities in SFL as shown in Figure 3.16.

> **Design Decision** 5 **- Hybrid approach using FPGs in SFL:** FaLDAS uses FPGs as program spectra in SFL, unlike the components' failure status [27] or components' involvement in an execution as program spectra [2]. FPG nodes are more fine grained entities than these [27] [2], and the result of this hybrid approach is likely to incur less effort for root cause analysis. Moreover, this hybrid approach is likely to be more efficient than others as it discards a number of entities (FPGNodes) which are non-suspicious, reducing the search space. The use of FPGs with SFL fulfills design objective 5 in that it supports an efficient faulty candidates identification.

Faulty Candidates Sorting

A debugging expert uses a set of faulty candidates to assess if a candidate is the actual faulty one responsible for generating the fault. The expert may choose a faulty candidate sequentially or randomly. In the worst case, the expert may find the actual fault in the last candidate assessed, in which case all the effort in analyzing the candidates prior to the actual faulty one, is wasted. To minimize this effort, the expert must be provided with a recommended order to analyze the candidates.

Existing mechanisms use data clustering algorithms to sort the candidates [5] [27]. Such algorithms calculate *similarity coefficients* i.e., the similarity between two data sets (e.g., two binary vectors) [54]. A number of existing mechanisms use such coefficients to find the similarity between a component's execution behaviour and error occurrence pattern. The error occurrence pattern is usually a binary vector in which each member corresponds to an execution, and represents whether that execution failed or was successful (see Figure 3.16). Chen et al. [27] compare similarity between components' failure pattern and error occurrence pattern. Abreu et al. [2] compare the similarity between components' usage patterns and error occurrence patterns.

FaLDAS calculates the faultiness of an FPGNode by measuring the similarity between the FPGNode's suspiciousness pattern and error occurrence pattern. Abreu et al. [5] evaluated Ochiai, Jaccard and Tarantula coefficients and found Ochiai's performance best. So, FaLDAS uses the Ochiai coefficient for similarity comparison. Hofer et al. [50] state that the more a candidate's pattern is similar to an error occurrence pattern, the higher the candidate's probability of being faulty. So, FaLDAS sorts the candidates according to their probability of being faulty.

> **Design Decision** 6 **- Similarity Coefficient-based candidates sorting:** FaLDAS uses a similarity coefficients-based calculation of a candidate's probability of being faulty. Further, the candidates are sorted according to their probabilities and presented to a debugging expert. This is a recommended order for the expert to carry out root cause analysis in an efficient manner. This fulfills the design objective 5, in that the candidates are sorted for further root cause analysis.

3.5.6 Root Cause Analysis Integration

A debugging expert carries out a root cause analysis on the sorted list of faulty candidates, to find the actual root cause of a fault. This thesis demonstrates the integration of an existing root cause analysis mechanism with FaLDAS to show how the output of FaLDAS can be further used to pinpoint the actual program statements responsible for originating the fault.

> **Design Decision** 7 **- Path-based Root Cause Analysis Integration:** FaLDAS integrates with Le et al.'s path-based root cause analysis mechanism [69]. This was chosen because Le et al.'s mechanism require a component and its corrupted output variable. Let et al. use a component's data propagation paths to pinpoint a path where a variable might have been corrupted. This fulfills design objective 6 in that FaLDAS's output can be further used with an existing root cause analysis mechanism.

3.5.7 Design Decisions Summary

Six design decisions are presented to address the design objectives as shown in Figure 3.17. These design decisions guide the construction of FaLDAS. For example, FaLDAS was placed inside the Analyze phase of the MAPE loop to enable it working for DAS. Intercept-based monitoring is used to determine the inputs and outputs of components, which becomes the knowledge-base and is represented as a set of FPGs. This enables knowledge-base synchronization with the system, at runtime. Moreover, it enables the construction of a knowledge-base even when the internal design of a component is unavailable or inaccessible. FaLDAS uses a hybrid approach in that FPGs are used as spectra in SFL. Because FPGs are dynamically constructed, fault localization in adaptive systems is supported. The SFL technique is used to sort the identified candidates, according to their probability of being faulty. The sorted order is a recommended order for a debugging expert to identify the actual fault efficiently. Unlike existing SFL-based mechanisms, FaLDAS uses a more fine grained entity, i.e., an FPGNode, which is a combination of the component's name and one of its output variable. This helps FaLDAS to work even with inaccuracies in the monitored data. Overall, this section presented the basic building blocks of FaLDAS approach.

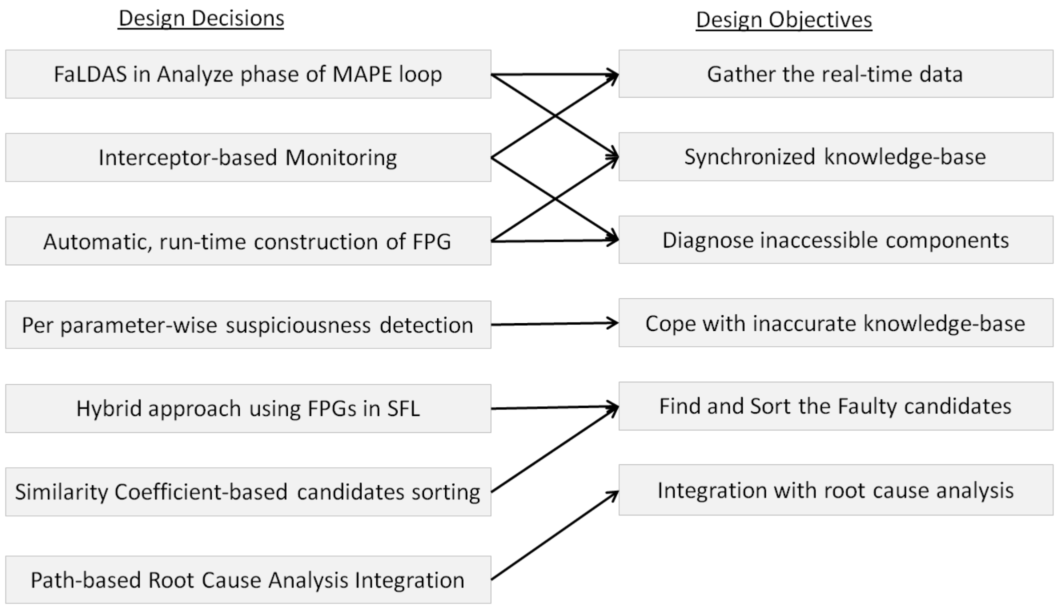

Design Decisions
Design Objectives
FaLDAS in Analyze phase of MAPE loop
Interceptor-based Monitoring
Automatic, run-time construction of FPG
Per parameter-wise suspiciousness detection
Hybrid approach using FPGs in SFL
Similarity Coefficient-based candidates sorting
Path-based Root Cause Analysis Integration
Gather the real-time data
Synchronized knowledge-base
Diagnose inaccessible components
Cope with inaccurate knowledge-base
Find and Sort the Faulty candidates
Integration with root cause analysis

Figure 3.17: An overview of design decisions to address the design objectives

3.6 Chapter Summary

This chapter describes the terminology used throughout this thesis, presents the scope of the work, explains the problem and assesses the deficiencies in existing mechanisms in solving the problem. In addition, the chapter formulates the core design objectives to address the challenges. Lastly, the design decisions made to address the design objectives are discussed.

Terminology and Scope: This section describes a Distributed Adaptive System (DAS) with examples of the three adaptation patterns considered in this thesis: *component insertion, component removal* and *server reconfiguration.* In a DAS, a *propagated fault* encountered in a component has its *root cause* (original fault) in a different component. There may exist more than one root cause of a *propagated fault*: *Single component fault* and *Multiple component faults.* FaLDAS helps debugging expert in identifying the root cause(s) of a propagated fault.

Problem Statement: Identifying the root cause of a propagated fault is a challenging problem. This problem becomes more challenging in a DAS because of the changing structure and behavior of the DAS. Ideally, the root cause(s) needs to be identified before it causes serious consequences. It is important to maintain the reliability and availability of a DAS.

Existing Solutions: Existing fault localization mechanisms are categorized into design-time and runtime techniques. Design-time techniques use static data about a system, e.g., inter- or intra-components data flow, to localize a fault, which becomes invalid after an adaptation. The difficulty in foreseeing all possible run-time configurations of a DAS limits the applicability of design-time approaches. Run-time techniques use runtime generated data e.g., logs, obtained from monitors. The run-time data generally used by existing mechanisms is either enormous whose analysis is inefficient or its generation requires internal design of a component which may not be accessible. For example, accessing program slices from a component is not possible when the access to source code is denied.

Design Objectives: Six design objectives are identified to address the challenges of adaptation in fault localization. To cope with adaptations, real-time data must be used as the knowledge-base for fault localization. The required data must be carefully chosen so that it is accessible even if the internal structure and behavior of a component is inaccessible. The technique must be capable of attaining substantial accuracy even with inaccurate data. To deal with inaccurate results, the diagnosed candidates must be sorted according to their likelihood of being faulty. This helps the debugging expert to locate the actual root cause in an efficient manner.

Design Decisions: Six design decisions are presented to address the design objectives. To enable FaLDAS for DAS, the MAPE loop is the most suitable methodology, and FaLDAS is placed in the Analyze phase. This allows FaLDAS to obtain monitored data, i.e., the inputs and outputs of components, to construct the knowledge-base and to synchronize it with the current system. A variant of a causal graph, i.e., Fault Propagation Graph (FPG) is proposed as the data structure for the knowledge-base. Instead of using existing causal graph-based or SFL mechanisms, a hybrid approach that uses both causal graphs (FPG) and SFL is proposed for locating and sorting of the faulty candidates, which enables FaLDAS to meet the challenges of adaptations.

Chapter 4

FaLDAS Fault Localization

This chapter describes the FaLDAS approach to fault localization. Figure 4.1 shows an overview of FaLDAS's sub-processes and overlays an outline for the rest of the chapter. This chapter starts with a description of a system's execution, with particular focus on data transformation (Section 4.1). During an execution, an interceptor-based monitoring mechanism is responsible for monitoring the run-time input and output values of involved components (Section 4.2). The monitored data is provided to FaLDAS to construct its knowledge-base. Section 4.3 describes an extended version of directed graphs, i.e., Fault Propagation Graph (FPG), its construction from monitored data and its significance as the central knowledge-based for FaLDAS. FaLDAS constructs an FPG for each execution and stores them in a centralized repository. When a fault occurrence is identified, FaLDAS triggers its Fault Localization algorithm to localize the faulty candidates for the failed execution. Section 4.4 describes FaLDAS's fault localization algorithm which uses a number of recently constructed FPGs (retrieved from the FPG repository) and reasons over them to identify the faulty candidates. In addition, the candidates are sorted in a ranked list[1]. The resulting ranked list of faulty candidates is a recommended order to carry out RCA on the individual components, to find the root cause as efficiently as possible. Section 4.5 describes how FaLDAS can be integrated with an existing RCA mechanism to pinpoint the program statements, in an actual faulty component, responsible for originating a propagated fault.

[1]A sorted list according to components' probability of being faulty for a failed execution under consideration.

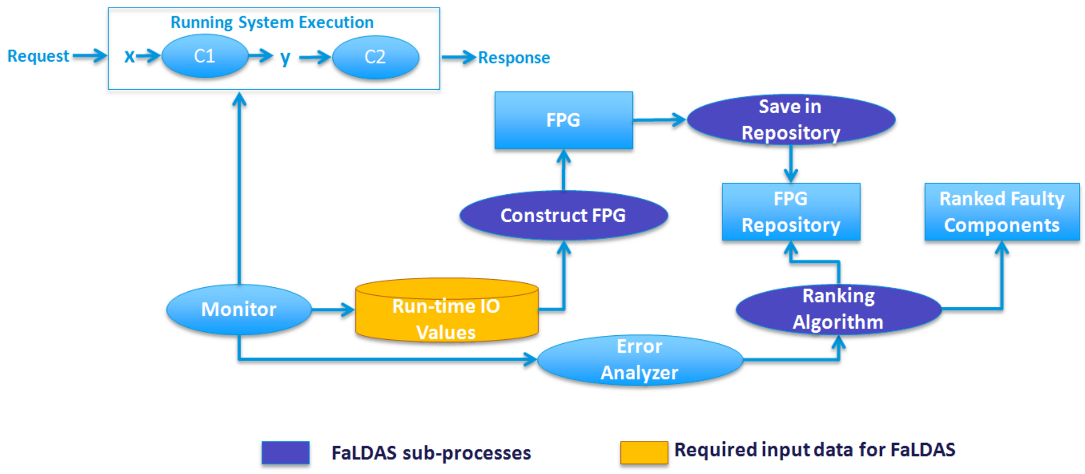


Figure 4.1: An overview of the FaLDAS Fault Localization Approach

4.1 System Execution

A software system is a collection of components to provide a set of services. Figure 4.2 describes a sample system with five components processing a particular request to produce a response. A request is received by a component, which computes an output and provides it to another component for further processing. After a number of computations by different components, a final response is produced.

Definition 4.1.1. *Execution.* An execution is a transformation of data through a system's components, which starts when a new request arrives at the system interface and ends when a response leaves the system interface.

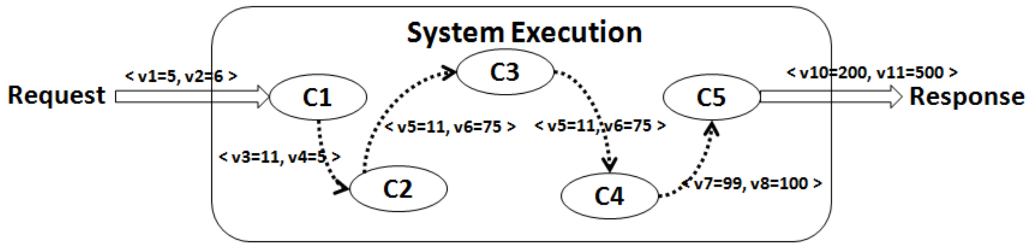


Figure 4.2: An example system and execution

The sample execution (Figure 4.2) starts when the request (data in the request is <v1=5,v2=6>) arrives at the system. The execution consists of the computations carried out by components C1 to C5. When component C5 completes its execution and produces the output <v10=200,v11=500>, the execution completes. Further, this section describes the significance of individual components' inputs and outputs and how they form the basis for fault localization of a fault responsible for a failed execution.

4.1.1 Data Transformation

An execution is a series of data transformations through several involved components of a system as shown in Figure 4.3 [126] [47] [86] [61]. The data at an execution point consists of all the *observable* data variables, i.e., the set of variables that are input to, or output from, the currently executing component.

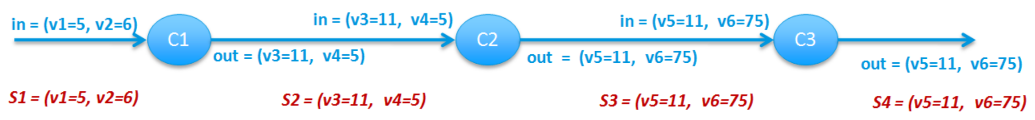


Figure 4.3: State transformation of a sample system.

After a component's execution completes, the data is considered to have changed when it is not the same as before the component's execution. Figure 4.3 presents an example of data transformations through components. The data represented by S2 and S3 are not the same because data variables and their values are different. This is because component C_2 has computed a new output (other possibilities related to component's faultiness are discussed in Section 6.1.1). However, the data represented by S3 and S4 are the same because data variables and their values are same. This is because component C_3 has not computed a new output (discussed in Section 6.1.1).

It can be concluded that when a component's output is different from its input, the component performs a computation. A computation could be correct without any fault or a fault occurrence could have made it faulty. This presents the following two cases:

i. When a component's output is different from its input, the output is correct because the component's computation did not encounter any fault.

ii. When a component's output is different from its input, the output is corrupted because the component's computation has encountered a fault.

Assuming inaccessible components in a DAS, it is difficult to know whether a component's computation has encountered a fault or not. In that case, this thesis makes an assumption that whenever a component's output is different from its input, the output is corrupted and the component has encountered a fault. This assumption ensured that FaLDAS does not miss any potential faulty candidates. It can be argued that several components identified as faulty will not be actual faulty, which is a valid argument. This is a limitation of FaLDAS, and also of other approaches. Evaluations in Chapter 5 describes that FaLDAS still achieves efficient results with similar quality in a number of scenarios.

4.1.2 Failed Execution

A fault causes an error which further propagates and may cause a component's failure resulting in a failed execution [13].

Definition 4.1.2. *Failed Execution.* A failed execution is an execution whose response, for a given request, differs from the expected response.

Definition 4.1.3. *Faulty Candidate.* A faulty candidate is a set of FPGNodes whose components or a subset of components are collectively responsible to originate a propagated fault.

Assuming that the execution shown in Figure 4.2 is a failed execution, i.e., the final response <v10=200,v11=500> is wrong and that component C2 is the actual component responsible for the wrong response, a debugging expert who is unaware of the actual faulty component, can upfront decide that C3 can not be faulty (or least likely to be faulty) because C3's input and output are same as discussed in Section 4.1.1 (see Figure 4.3). Such components may be involved in logging execution traces, storing data in a database, routing a request, etc., which does not effect the system's data.

In addition, the debugging expert can upfront decide that the component C2 could be faulty because its output is different from its input. C2 may have wrongly computed its output V5 as 11 or V6 as 75 which was further used as input (in fact corrupted input) to another component, say C3. Component C3 has routed this data to component C4. C4 may have produced incorrect output because its input was incorrect, which is not C4's fault. It is unlikely to compute a correct output using an incorrect input. Similarly, component C5 may have produced incorrect output and finally results in the failed execution. The described chain of inputs and outputs illustrate that C2 is the actual faulty component, and it is one of the components whose output is different from its input. This example strengthens the stated assumption that components whose output is different from its input has the potential to be faulty.

It could be argued that component C3 might be faulty; invalidating the assumption. It is possible that C3 activated a fault due to which a corrupted output was computed, but coincidentally it was same as the input. Here, C3's incorrect output <v5= 11,v6=75>, not C2's output, should be considered responsible for the failing the execution. However, it is the thesis's limitation that it cannot firmly identify if a component's same input output really indicates that there is no fault in it.

4.2 Data Monitoring

A number of fault localization mechanisms rely on detailed data about components. For example, Gopinath et al. [45] use program statements and test cases, Jones et al. [58] use test suites, Mariani et al. [84] use source code, Lei et al. [74] use program slices, etc. Such mechanisms face a major challenge when a system contains inaccessible components where inaccessibility means access is denied to a component's internal design.

Definition 4.2.1. *Inaccessible Components.* A component is considered inaccessible when access to its host, logs, source code and test cases is blocked for a debugging expert.

Figure 4.4 shows an illustration of an inaccessible component where a debugging expert has no access to the component's host, logs, source code for debugging purposes. In such a situation, it is difficult for existing mechanisms to generate their knowledge-bases. To cope with this problem, this thesis depends on monitoring of real-time inputs and outputs of the components to generate its knowledge-base. Monitoring inputs and outputs does not require access to the components because these are external information.

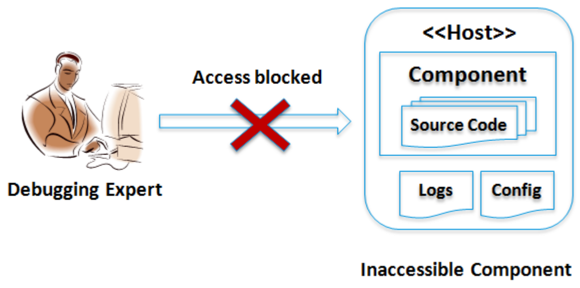


Figure 4.4: Description of Inaccessible Component

4.2.1 Data Monitoring Tool

A preferred technique to monitor a system is to use mobile monitoring agents [52] [72]. Such agents are deployed within the components. Agents monitor a system's data and save it at a preferred location. However, in a DAS, it cannot be ensured that all the adapted components have a monitoring agent within them. Even if a component has a monitoring agent, it may not monitor the required data for debugging purpose. Moreover, it is not always possible to inject a monitoring agent within the adapted components, as it may not be allowed.

To deal with such problems, this thesis envisions an enterprise service bus (ESB) based software architecture. An ESB is a messaging software to coordinate the messages between a set of components. It also provides an ideal infrastructure to implement an adaptive system, as components can be plugged- in or out to adapt the system. The thesis assumes an interceptor-based monitor located within the ESB bus to monitor the inputs and outputs of components as shown in Figure 4.5. In such an infrastructure, the components put their output in the bus which is transferred to other components. The interceptor, located within the bus, accesses the data flowing through the bus without requiring access to the component. The interceptor provides a service through which a fault localization mechanism can access the monitored data.

An illustrative example of interceptor-based monitoring in the Apache Camel-based system [51] is shown in Figure 4.6. Apache Camel is a messaging-based integration framework, which allows integration of components in the form of a workflow. For example, five different components C1 to C5 have been coordinated through Camel's messaging bus as shown in Figure 4.6a. The system receives a request from a Java

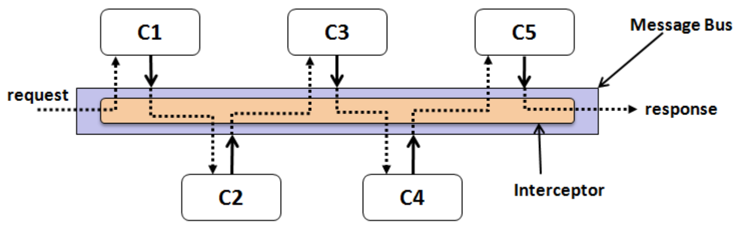
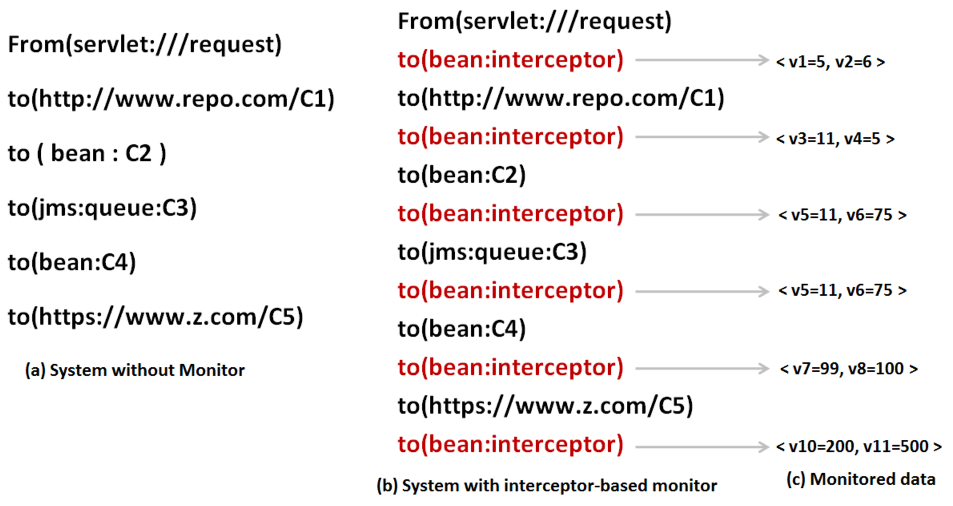


Figure 4.5: Interceptor- and messaging bus-based software architecture

Servlet which is forwarded to C1, which is a web component. The output of C1 is passed to C2, which is a Java bean. The output from C2 is passed on to C3, which is Java Messaging Service Queue, from where is it consumed by C4, which a Java bean. Finally, C4's output is passed to C5, which is a web component that computes and produces the final response.



```
From(servlet:///request)
to(http://www.repo.com/C1)
to ( bean : C2 )
to(jms:queue:C3)
to(bean:C4)
to(https://www.z.com/C5)
```

(a) System without Monitor

```
From(servlet:///request)
to(bean:interceptor)      ——→  < v1=5, v2=6 >
to(http://www.repo.com/C1)
to(bean:interceptor)      ——→  < v3=11, v4=5 >
to(bean:C2)
to(bean:interceptor)      ——→  < v5=11, v6=75 >
to(jms:queue:C3)
to(bean:interceptor)      ——→  < v5=11, v6=75 >
to(bean:C4)
to(bean:interceptor)      ——→  < v7=99, v8=100 >
to(https://www.z.com/C5)
to(bean:interceptor)      ——→  < v10=200, v11=500 >
```

(b) System with interceptor-based monitor

(c) Monitored data

Figure 4.6: An interceptor-based monitoring example with Apache Camel Messaging Infrastructure

To monitor the individual component's inputs and outputs, an interceptor was implemented as a Java bean and integrated with the message bus as shown in Figure 4.6b. The interceptor must be generic enough to handle different data types e.g., a web component produces a web response whereas a Java bean component may produce a Java object. The intercepted inputs and outputs are shown in Figure 4.6c. It shows that the components are monitored even when access to their internal designs is not accessible.

Data Model

A number of models exist to represent a component's monitored information [120] [134] [118]. These models capture a vast amount of information about a component e.g., interfaces, package, connectors, component type, etc., which are not required in FaLDAS. An input-process-output (IPO) model [31] [127] is the closest to FaLDAS's requirements (see Figure 4.7a). This model is widely used in industry to describe a component's processing [140]. It represents the inputs received, computation unit i.e., a component, and outputs of the computations.

In order to use the IPO model to represent the overall processing of a system and to represent components' run-time values, the IPO model is modified as shown in Figure 4.7b. The modified model is referred to as the Input Output Values (IOV) model. When an interceptor-based monitor intercepts a request to, or response from a component, FaLDAS instantiates an instance of the IOV's component along with InputVariable or OutputVariable and their run-time values. FaLDAS uses these instances to further construct the FPGs.

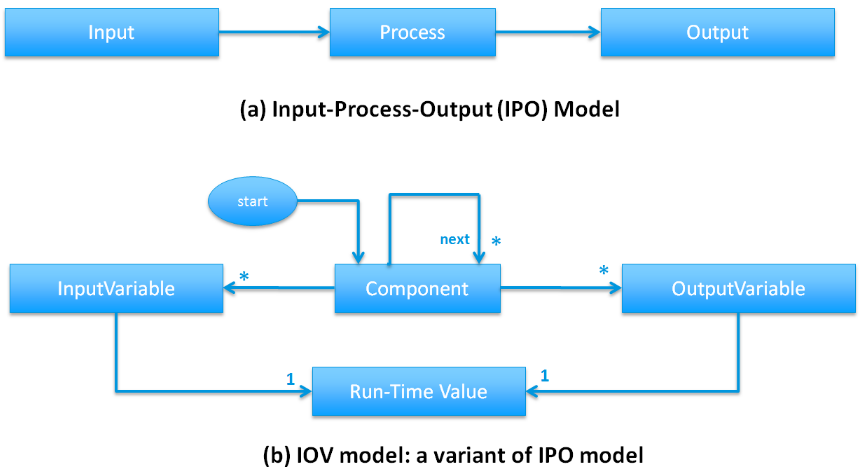


Figure 4.7: UML Metamodel of Components' Run-time I/O values (IOV)

4.3 Fault Propagation Graph (FPG)

A set of Fault Propagation Graph (FPG)s [99] is the central knowledge-base over which FaLDAS reasons to determine the components responsible for a propagated fault. An FPG is an abstract representation of the data flow between components in an execution. This set of graphs exposes sufficient details about components and their variables to analyze data modification locations responsible for fault propagation.

4.3.1 FPG Model

A number of existing approaches use graph-based models to represent fault propagation e.g., Bellur et al.'s graph [17] represent the tuple <component, fault> as nodes and edges show propagation of faults, Candea et al.'s graph [20] represent only components as nodes and edges show fault propagation, etc. Such graphs do not indicate the possible cause within the component (see design decision 4 in Section 3.5). A debugging expert has to manually identify the corrupted data variable and find the problematic program statements responsible for data corruption.

To deal with this problem, an FPG was designed to represent the component and its potentially corrupted output variable. A FaLDAS FPG is a directed graph $G = (V, E)$, where V is a set of FPGNodes and E is the set of edges. An FPGNode is a tuple $< C, V >$ where C is a component name (mandatory), and V is an output variable of C (optional). To represent if a component has corrupted an output variable or not, the FPGNodes are tagged with a symbol T (optional) and FPGNode form then becomes $< C, V, T >$. Two tag symbols, 'CLEAR' and 'SUSPICIOUS', indicate the fault status of a component, based on sameness information about the IOV (discussed in more details later). An edge between two FPGNodes represents the propagation path of a fault between the components specified in those FPGNodes. An FPGNode indicates the faultiness of a component using the following rules:

1. $< C, V, T >$: When T is 'SUSPICIOUS', C may have corrupted output V and may be responsible for a fault caused by the corrupted V. When T is 'CLEAR', C is not responsible for corrupting V and cannot be responsible for a fault caused by a corrupted V.

2. $< C, -, T >$: When T is 'SUSPICIOUS', C may have corrupted all of its output variables and may be responsible for a fault caused by any of these corrupted output variables. When T is 'CLEAR', C is not responsible for any faults.
3. $< C, V, - >$: The faultiness of C for its output variable V cannot be assessed.
4. $< C, -, - >$: The faultiness of C cannot be assessed.

In an FPG, there are several FPGNodes whose tags are SUSPICIOUS. However, an actual faulty FPGNode represents the faulty component.

Definition 4.3.1. An *actual faulty FPGNode* is an FPGNode which refers to the component and its output variables which actually are responsible to generate a propagated fault. The component performs faulty computations to corrupt the output variable which further propagates to other components; causing a propagated fault.

4.3.2 FPG Construction

When an execution completes, a corresponding FPG is constructed (irrespective of whether the execution failed or not), and stored in an FPG repository. To construct an FPG when an execution completes, FaLDAS obtains the instances of IOV corresponding to the current execution from the monitor. Algorithm 1 illustrates how an FPG is constructed for an execution, iterating for each subsequent execution.

Figure 4.8 outlines a sample adaptive system used for step-by-step explanation of this algorithm, where there are three components C1, C2 and C3, with output variables (V3, V4), (V5, V6) and (V7, V8), respectively. As each component has two output variables, six FPGNodes are created. Each node contains the component's name, an output variable and corresponding tag populated using IOV (line 5 - 26). In Figure 4.8, the left-hand side illustrates a sample execution with inputs and outputs for each component, and the right-hand side illustrates the corresponding FPG.

Where a component's output becomes an input to another component, the former is a *source component* and the later is a *sink component*. From this, the following rules are used to create edges between FPGNodes (Algorithm 1 line 17 - 23):

1. Where a sink component's FPGNode refers to the same output variable as any of the source component's FPGNodes, an edge between the two matching FPGNodes is created (Algorithm 1 line 18).

Algorithm 1: Construct Fault Propagation Graph (FPG)

```
Input: Components Run-time IO values (IOV)
Output: Fault Propagation Graph (FPG)
 1: FPG ← $\phi$
 2: FPGNode currNode ← startnode
 3: FPG.add(currNode)
 4: List<FPGNode> prevset ← prevset ∪ startnode
 5: Iterator it ← IOV.getStart().depthFirstSearchIterator()
 6: while it.hasNext() do
 7:    Component comp ← it.next()
 8:    List<FPGNode> thisset ← $\phi$
 9:    for each $outputVar$ in comp.getOutputVariables()
10:       FPGNode $< comp, outputVar, - >$ ← constructFPGNode($comp$, $outputVar$)
11:       if ($comp$.inputVariablesContain($outputVar$) && their values are same) then
12:          FPGNode currNode ← ($< comp, outputVar, CLEAR >$)
13:       else
14:          FPGNode currNode ←($< comp, outputVar, SUSPICIOUS >$)
15:       end if
16:       FPG.add(currNode)
17:       if prevset contains a node such that node.V == currNode.V then
18:          createEdge(node, currNode)
19:       else
20:          for each node in prevset do to
21:             createEdge(node, currNode)
22:          end for
23:       end if
24:       thisset ← thisset ∪ currNode
25:    end for
26:    prevset ← thisset
   end while create edges from startnode to nodes having no incoming edge
```

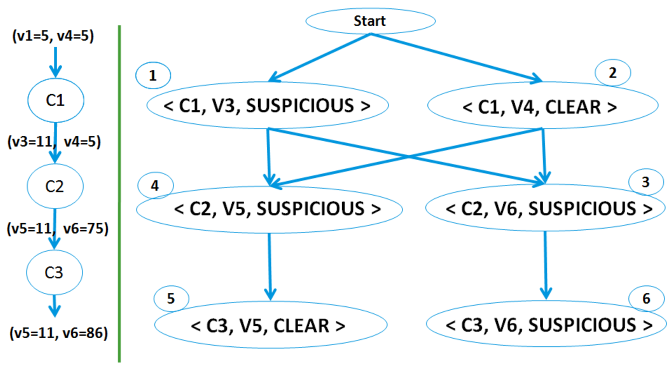


Figure 4.8: Step by step execution of Algorithm 1. The left hand side represents an execution and the right hand side represents the corresponding FPG.

2. Where any sink component's FPGNode has no matching variable in the source component's FPGNodes, incoming edges from all the source's FPGNodes are created, to ensure that no data flow between variables is missed (Algorithm 1 lines 20 - 22).

From Figure 4.8, `C2` (as sink component) gets input from `C1` (as source component). The corresponding FPGNode 4's output variable `V5` is not the same as the output variable of any of C1's FPGNodes (node 1 and 2), so directed edges are created from node 1 and 2 to node 4. Recall that when it is unclear that which inputs (which are outputs of preceding components) are used to compute an output, all inputs are assumed to be used in the computation of the output. Similarly, incoming edges are created for FPGNode 3. For node 5, `C2` is the source component and `C3` is the sink component. As the names of the output variables of `C3`'s FPGNode 5 and `C2`'s FPGNode 4 are the same, i.e., `V5`, a directed edge is created from FPGNode 4 to FPGNode 5. Similarly, an edge is created between FPGNode 3 and 6.

4.3.3 FPG Nodes Tagging

As stated previously, IOV is used to determine whether a component's input and output are the same, from which a conclusion is drawn as to whether the component may have played a role in transforming the data i.e., potential to corrupt a data variable. In Figure 4.8, component `C1` produces output (`V3` and `V4`). The IOV corresponding to `C1` shows that `C1` has two output variables, and so two FPGNodes are created <`C1`, `V3`, - > and <`C1`, `V4`, - > labeled as 1 and 2 in Figure 4.8. As `C1`'s output `V3` is not same as any of its input, the corresponding FPGNode 1 is tagged as SUSPICIOUS (Algorithm 1 lines 11 - 15). `C1`'s output `V4` is the same as its input `V4`, and so the corresponding FPGNode 2 is tagged as CLEAR (Algorithm 1 lines 11 - 15). Similarly, the other FPGNodes are constructed and tagged.

4.3.4 FPG Analysis

FaLDAS constructs an FPG for each execution, irrespective of whether it failed or was successful. In the FPG of a failed execution (definition 4.2.1), the FPGNodes with SUSPICIOUS tag are considered faulty nodes. This in turn means that the components with such nodes are considered faulty, i.e., the component may have wrongly computed or corrupted the corresponding output and causes a propagated fault. If the sample execution illustrated in Figure 4.8 has failed, nodes 1, 4, 3 and 6 are considered faulty because their tags are SUSPICIOUS. This says that component `C1` might be faulty as it corrupted output `V3`; `C2` might be faulty because it corrupted output `V5` or `V6`; or `C3` might be faulty because it corrupted output `V6`. This information enables developers to employ a specific and targeted root cause analysis mechanism. For example, the developer will employ RCA in `C1` only to identify the root cause for data corruption of `V3` but not for `V4`.

Still, a developer is likely to waste effort in analyzing the root fault in a component which is not actually faulty. Let's say `C1` was actually faulty. It is nonetheless possible that the developer will analyze `C2` and `C3` before `C1`. To minimize such wasted effort, FaLDAS ranks components according to their likelihood of being faulty for a failed execution.

4.4 Fault Localization

A traversal of an FPG identifies more than one component as potentially faulty, or 'suspicious', and each of these must be analysed to determine which component(s) contains the root cause of the fault. This is potentially time-consuming where a large number of components have been identified, and so to reduce un-necessary analysis effort, FaLDAS ranks the components according to their likelihood (probability) of being the actual faulty one for a failed execution. This enables a debugging expert to prioritize their investigations, overall reducing effort.

4.4.1 Existing Mechanisms

Chapter 2 discusses a number of fault localization mechanisms. However, this section discusses the closest related mechanism - BARINEL [4]. The BARINEL ranking algorithm takes a set of hit spectra [49] as input and produces a ranked list of faulty components [4] [2]. However, two limitations in this algorithm are of interest to the FaLDAS work: (i) components not involved in data computations are also marked as faulty, and (ii) the algorithm finds faulty components only for a set of failed executions, and not for a particular one. For example, consider a system with five components (C_0,C_1,C_2,C_3,C_4) where a number of components participate in each execution. Component C_2 is a router, and only routes data but does not modify it. This means that whenever a data fault occurs due to corrupted data, component C_2 cannot be responsible. Figure 4.9 and Table 4.1 illustrate the hit spectra and an error vector for five executions. The error vector has a flag for each execution where values '0' or '1' represent a successful or failed execution, respectively. When this information is analyzed using the BARINEL algorithm, component C_2 is determined to have the highest probability of being faulty, while, in fact, it could not be. This is because the usage pattern C_2, i.e., $<0, 1, 0, 1, 1>$ has the highest similarity with error vector e, i.e., $<0, 1, 0, 1, 1>$. In general, a component's involvement pattern is not sufficient to determine faultiness because a component might be involved in an execution but not in changing the data state. The actual root cause will still be found eventually, but these rankings result in unnecessary effort and time in the process.

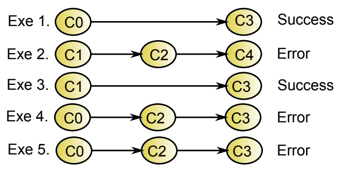


Figure 4.9: Five executions of the sample system.

C_0	C_1	C_2	C_3	C_4	e
1	0	0	1	0	0
0	1	1	0	1	1
0	1	0	1	0	0
1	0	1	1	0	1
1	0	1	1	0	1

Table 4.1: Hit Spectra of the sample system

It is also likely that BARINEL's single ranked list of faulty components may not be the most optimal[2] for finding the root cause of any particular failed execution. Where different failed executions have different root causes, BARINEL's algorithm is unlikely to return an optimal ranking for all failed executions. For example, a ranked list may be optimal to find the root cause of fault in a particular failed execution (say Exe. 2) but may not be optimal to find root cause of fault in Exe. 4. Nonetheless, BARINEL is suitable for analysing faults where the same root cause(s) are responsible for all considered failed executions.

4.4.2 FaLDAS Ranking Algorithm

Inspired by the BARINEL approach, the FaLDAS ranking mechanism, illustrated in Algorithm 2, instead ranks faulty components for any given failed execution using the FPGs constructed during system execution (see section 4.3). For example, Figure 4.10a illustrates three different executions from the system described in Figure 4.8.

[2]The more optimal, the less effort required to find root cause.

Executions 1 and 3 have failed and 2 is successful. The corresponding spectra and error vector e are shown in Figure 4.10b. In the error vector e, values '0' and '1' represent that the execution did not fail, or failed, respectively. Here, ranking algorithm 2 finds and ranks the faulty candidates for failed execution no 1.

Algorithm 2: FaLDAS Ranking Algorithm

```
Input: A set of recently constructed FPGs A, Error Vector E, Failed execution number n
Output: Ranked List of Nodes
 1: int k ← 0
 2: Set D ← empty
 3: fpg ← A[n] //the FPG from which components need to be diagnosed
 4: for each node in fpg such that node.tag == SUSPICIOUS do
 5:    for each immediate child of node do
 6:    Set $d_k$ ← empty
 7:    $d_k$ ← $d_k$ ∪ node
 8:    Iterator it ← child.getDepthFirstTraversalIterator()
 9:    while it.hasNextNode() do
10:       node ← it.nextnode()
11:      if node.tag == SUSPICIOUS then
12:          List ignoredFPGs ← new List()
13:          ignoredFPGs ← ignoredFPGs.add(fpg.id)
14:          $d_k$ ← $d_k$ ∪ isNextNodeFaulty(node,ignoredFPGs)
15:       end if
16:    end while
17:     D.push($d_k$)
18:     $k$ = $k$+1
19:     end for
20: end for
21: return calculateRank(D)

22: function isNextNodeFaulty(node, ignoredFPGs)
23:     Set local ← empty
24:     for each fpg f in A such that f ∉ ignoredFPGs
25:        if node ∈ f && node.tag == SUSPICIOUS && e[f] == 1 then
26:           ignoredpaths ← ignoredpaths.add(path)
27:           local ← local ∪ node
28:        end if
29:     end for
30:     return local
31: end function
```

For execution no. 1, the FPG used has nodes labeled as $i*$ (the first set illustrated in Figure 4.10b). For this failed execution, there are 5 different paths that will be examined, starting from each of the SUSPICIOUS nodes, denoted S for space convenience, as shown in Figure 4.10c. For example, path 1 includes $i1$, $i4$, and $i6$. A candidate set, d_k, is created representing FPGNodes that may be responsible for a fault, populated with those FPGNodes marked as SUSPICIOUS (line 7). In this case, $i1$ is added to d_1, as would its parent be, if it had one. Child node components have received corrupted input, however, and therefore cannot be assessed for their own faultiness in this execution. Further consideration of child nodes happens by investigation of whether the component and corresponding output variable is referenced in any other FPGNode tagged as SUSPICIOUS in any other failed execution. In this case, the potential faultiness of child node $i4$ (child of $i1$) is outlined in the FPG for failed execution 3, i.e., the FPGNode $<C_2$, `V6`, - $>$ has SUSPICIOUS tag in execution 3 (line 14), and so $i4$ is added to the current candidate set, i.e., $d_1 = \{i1, i4\}$. There are no more FPGs representing failed executions left to test the next child node $i6$, so the candidate set is complete. The algorithm then switches to the next path starting from $i3$, iterating until it has completed all 5 paths in this FPG.

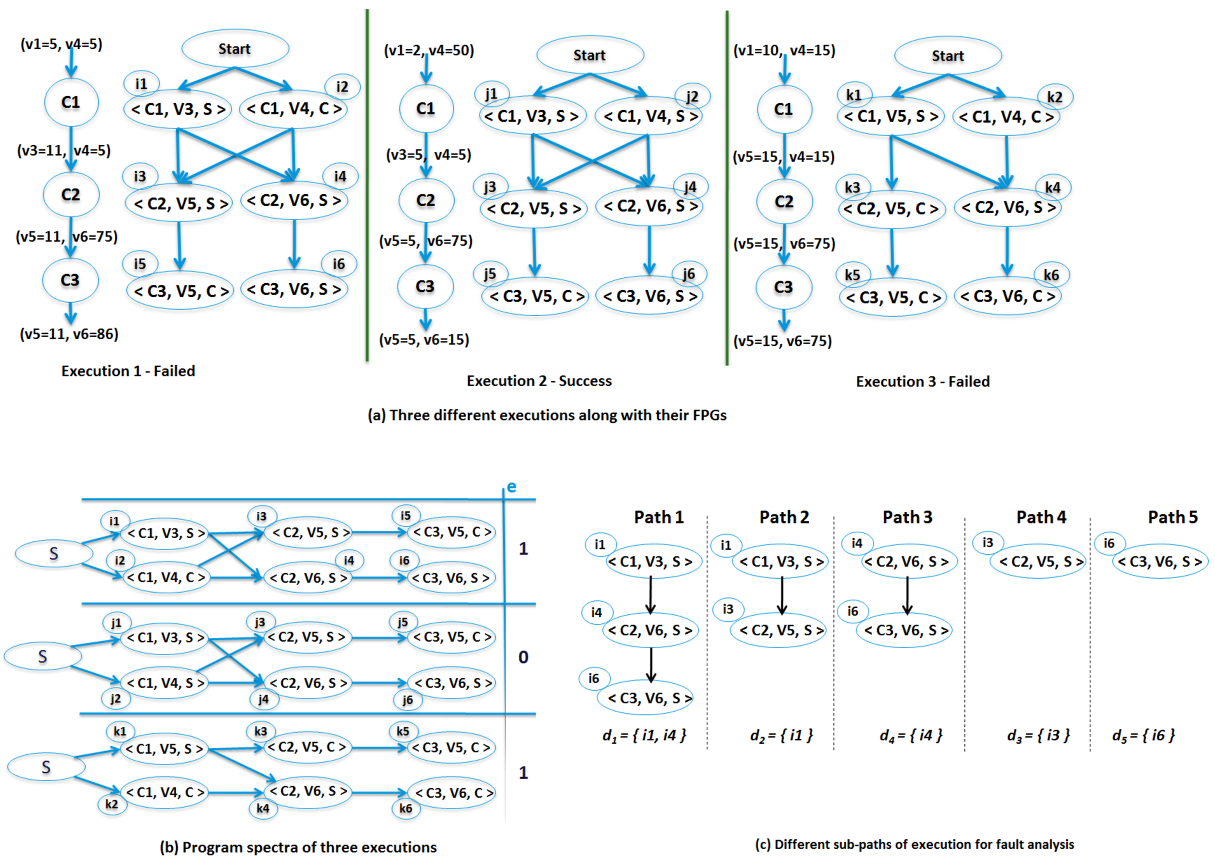


Figure 4.10: Three different traversed paths for three different execution along with their spectra.

Algorithm 2 returns five candidate sets as $d_1 = \{i1, i4\}$, $d_2 = \{i1\}$, $d_3 = \{i3\}$,

$d_4 = \{i4\}$ and $d_5 = \{i6\}$. The candidate set d_1, from reading the FPG for Execution 1 in Figure 4.10a, indicates that `C1` has corrupted `V3` and `C2` has corrupted `V6` to collectively fail execution 1. The single fault candidate d_2 indicates that only `C1` has corrupted `V3` to fail execution 1, and so on. FPGNodes which are not part of any candidate set are not considered responsible for the fault because their components have not modified the corresponding output variables.

4.4.3 FPGNode Health Determination

Ranking these candidates involves stepping back and examining the FPGNodes in all the systems' executions to assess the corresponding component's success/failure behaviour across all executions together. This is the *health* of each FPGNode, and is a probability value indicating the likelihood of its corresponding component being at fault. An FPGNode's health value of 0 or 1 indicates that its corresponding component persistently produces corrupted output or the desired output, respectively. When a component is not functioning correctly, it is likely to manifest faults irregularly, i.e., with different inputs, an output may or may not be corrupted. So, the health calculation will be more precise when more inputs/output observations (FPGs) are available. When available information shows that a node is frequently faulty, the health value tends towards 0, else it tends towards 1. In Algorithm 3, a tendency towards faultiness is captured when a node has a SUSPICIOUS tag in a failed execution (line 6), i.e., n_{11} is incremented, and a tendency towards success is captured when a node has SUSPICIOUS tag in successful execution (line 9), i.e., n_{10} is incremented.

For example, the tag for FPGNode $i4$, i.e., <`C2`, `V6`, `->` is SUSPICIOUS in two failed executions 1 and 3, and so n_{11} is set to 2 (Algorithm 3 line 6). Its tag is also SUSPICIOUS in one successful execution 2, and so n_{10} is set to 1 (Algorithm 3 line 9). FPGNode $i4$'s health value is consequently calculated as 0.34, using the formula given at line 13 of Algorithm 3. Similarly, the health values of FPGNodes $i1$, $i3$ and $i6$ are calculated as 0.34, 0.5 and 0.5, respectively.

4.4.4 FPGNodes Sorting

Finally, the five candidate sets are ranked according to their overall probability. To calculate this rank, the probability calculation illustrated in Algorithm 4 is used (de-

Algorithm 3: Calculate Health

```
Input: Collection of paths A, node n
Output: Health of c
1: median ← 0.5
2: $\epsilon$ ← median/(number of paths in A)
3: $n_{11}$ ← 0
4: $n_{10}$ ← 0
5: for each path in A
6: if e[path] == 1 && node n is SUSPICIOUS in the path then
7: $n_{11}$ ← $n_{11}$ + 1
8: end if
9: if e[path] == 0 && node n is SUSPICIOUS in the paththen
10: $n_{10}$ ← $n_{10}$ + 1
11: end if
12: end for
13: health ← median + ($\epsilon\times$ $n_{10}$) - ($\epsilon\times n_{11}$)
14: return health
```

signed by Abreu et al. [4] [2]). This algorithm uses the health of the FPGNodes in the candidate set. The probabilities of the 5 individual candidate sets are $d_1 = 0.88$, $d_2 = 0.34$, $d_3 = 0.5$, $d_4 = 0.34$ and $d_5 = 0.5$ (Algorithm 4), and so the sorted order for the candidate sets is $\{d_1, (d_3, d_5), (d_2, d_4)\}$. This ranked list of candidates is a recommended order to carry out root cause analysis for a fault.

Algorithm 4: Calculate Rank

```
Input: Set of Diagnosed candidates D, path of diagnosis p
Output: Ranked List of Diagnosed candidates $D_R$
1: for each $d_k$ in D
2: for each node in $d_k$
3: m ← m * health(node)
4: end for
5: Pr($d_k$) ← 1 - m
6: PUSH($D_R$, Pr($d_k$))
7: end for
8: return SORT($D_R$, Pr)
```

4.5 Root Cause Analysis

4.5.1 Existing Mechanisms

Several techniques exist to identify a root cause within a component [60] [124] [69] [129]. These techniques identify a number of statements in a program's source code which appear suspicious for either failed tests or runs. The output of FaLDAS can be used as input to such techniques to identify the actual root cause within a component. However, a technique that is based on dependencies between program elements, is likely to diagnose a more precise and efficient root cause because it includes a broad code coverage. Le et al.'s path-based fault correlation mechanism [69] is used in conjunction with FaLDAS because it correlates program statements to identify the actual faulty statements responsible for corrupting a particular variable. Faulty statements corresponding to data faults such as *integer signedness fault*, *null pointer exception*, etc. are identified, which is substantially helpful to developers.

4.5.2 FaLDAS RCA Integration

The process of root cause analysis using the ranked list of FPGNodes is shown in Figure 4.11. The FPGNodes are retrieved from the diagnosed candidates in order. For each FPGNode, root cause analysis is carried out on the FPGNode's component to find the root cause of the corrupted value of FPGNode's output variable. The identified program statements in the code include the root cause which needs to be validated by a developer or an automated mechanism. When the exact root cause is identified, the system can be patched appropriately to recover from the fault.

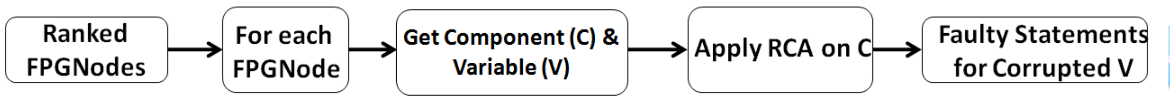

Ranked FPGNodes
For each FPGNode
Get Component (C) & Variable (V)
Apply RCA on C
Faulty Statements for Corrupted V

Figure 4.11: Root Cause Analysis Flow

An example from [69], mapped to an FPGNode <`foo`, `q`, SUSPICIOUS>, illustrates this process. The root cause for corrupted value of `q` in component `foo` must be identified. Le et al.'s path-based RCA is carried out on component `foo` to identify the path, in particular actual program statement, responsible to corrupt the value of `q` as shown in Figure 4.12.

In this example, the output variable `q` is last updated at node 11. Le et al. create a constraint $[Value(x) > 0]$ which is propagated backward to identify its violation. At node 7, it is always true as $1024 > 0$. However, along the branch <8,6> the constraint is always false. Therefore, the path segment <6,8,10,11> is reported as faulty for the corrupted value of `q`. The identified responsible paths indicate the program statements

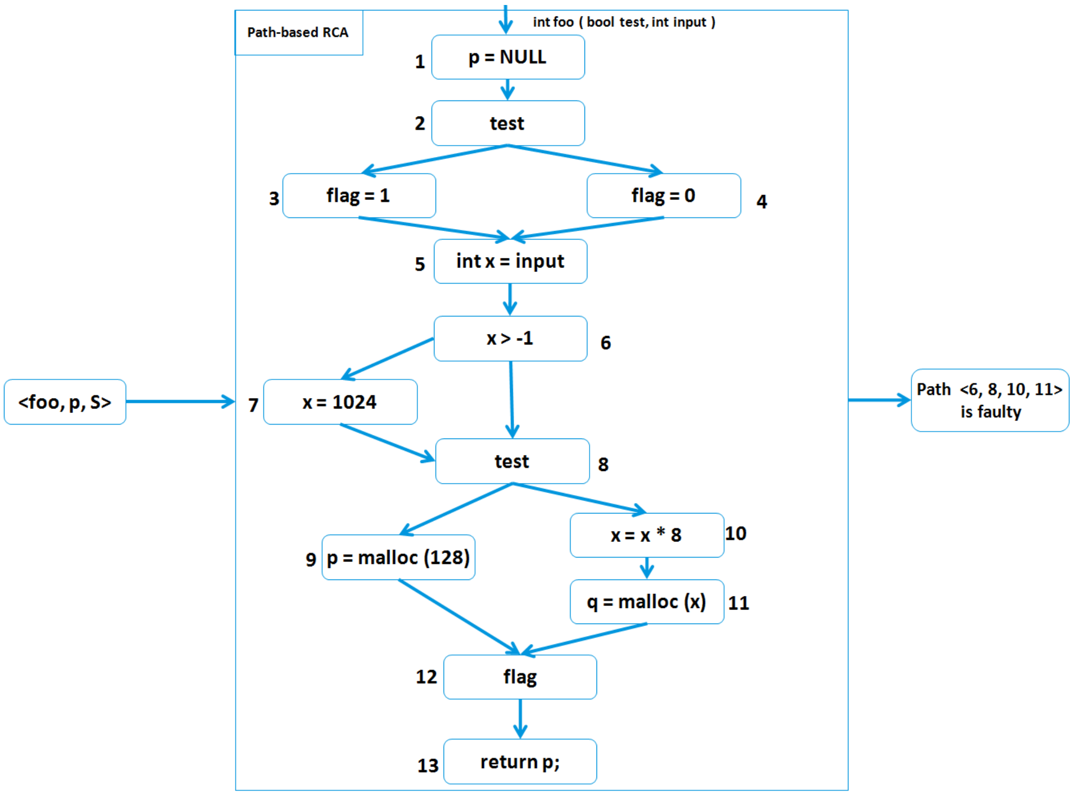


Figure 4.12: Path-based root cause analysis. Image source: [69]

responsible for the corruption of variable `q`. This identified root cause can be used to plan for a recovery, either through an automated recovery mechanism, or as information to developers.

4.6 Chapter Summary

This chapter presents FaLDAS, a fault localization approach to find faulty components and corresponding corrupted output causing a propagated fault in a DAS. FaLDAS uses component's name and its real-time input output values to construct its knowledge-base i.e., a set of Fault Propagation Graphs (FPGs). In an FPG, a node represents an FPGNode <Component, Output> and edges represent fault propagation between the components referenced in FPGNodes. Real-time input output values are used to tag (SUSPICIOUS or CLEAR) an FPGNode. It was described that where a component's output is not the same its input, the corresponding FPGNode is tagged SUSPICIOUS;

representing the component likely to corrupt its output. The corrupted output may have further propagated to cause a propagated fault. Where a component's output is the same its input, the corresponding FPGNode is tagged CLEAR; representing the component has not corrupted the output.

The chapter describes an existing interceptor-based monitoring mechanism to monitor FaLDAS's required input data. It was illustrated on a system whose components interact through a message bus e.g., Apache Came integration framework. FaLDAS uses the monitored data to construct a set of FPGs. FaLDAS traverses the FPGs to analyze the possible faulty candidates responsible to originate a propagated fault. It was observed that not all the candidates are responsible. Thus, the faulty candidates are sorted according to their probability of being faulty. FaLDAS calculates and uses the health of individual FPGNodes for sorting. Higher is an FPGNode's SUSPICIOUS tag frequency for failed executions, lesser is its health. FaLDAS's final output is a sorted list of candidates (a set of FPGNodes) which is a recommended order for a debugging expert to carry output root cause analysis on individual components referenced in FPGNodes. As a demonstration of how FaLDAS fits into an overall context of fault diagnosis, it presents FaLDAS's integration with an existing RCA mechanism.

Chapter 5

Evaluation

FaLDAS was evaluated to find the number of executions required to generate the knowledge-base, the time it takes for fault localization and the effort required to pinpoint the actual faulty candidate. Section 5.1 defines the parameters used for the evaluation. Section 5.2 describes baseline approaches used for comparison with FaLDAS. Section 5.3 describes the experimental setup and the systems used to perform the experiments. Section 5.4 presents an experiment to assess how much execution data is required to bootstrap the knowledge-base used by the FaLDAS algorithm. Section 5.5 evaluates the algorithm's time efficiency. Section 5.6 and 5.7 evaluate the quality of FaLDAS's results when the actual faulty component is inaccessible and when its information is inaccurate, respectively.

5.1 Evaluation Parameters

Where a system is adapting at runtime, a fault diagnosis technique must be both efficient and accurate at identifying a faulty component if it is to be effective. If the technique is accurate but inefficient, i.e., the time it takes to identify the accurate results compromises availability service-level agreements, then the technique may not be suitable. When the technique is efficient but does not give accurate results, the debugging expert will require more time, again compromising availability service-level agreements. In particular, wasted effort and time for evaluation are measured here using the following metrics:

Definition 5.1.1. *Minimum Executions (E)* is the number of past executions required to produce sufficiently accurate results.

Definition 5.1.2. *Construction Time (T_c)* is the time taken by an approach to construct its knowledge-base.

Definition 5.1.3. *Algorithm Time (T_a)* is the time taken to produce the candidates sets in sorted order, assuming the algorithm has the execution data it needs.

Definition 5.1.4. *Wasted Effort (W)* is the number of candidates analyzed prior to locating the actual faulty candidate (C_a), out of all ranked candidates (C_r). This is calculated using W = C_a / C_r [4].

5.2 Related Algorithms

The BARINEL [4], Pinpoint [27] and Bellur et al. [17] approaches are compared against FaLDAS because they (i) localize actual faulty candidates for a fault, (ii) produce a sorted list of faulty candidates, and (iii) can be applied in adaptive system environments.

Definition 5.2.1. A *Faulty Candidate* is a potentially faulty candidate that represents the location of originating a propagated fault. It is represented as an FPGNode in FaLDAS, a component in BARINEL and Pinpoint and a BBN node in Bellur et al's approach.

The BARINEL algorithm was implemented from descriptions in published work [4] [2], which was verified using a number of sample input matrices (available at [22]) to ensure the same candidate sets previously reported [22] were returned. The implementation requires two inputs: (1) An error vector (Definition 5.2.2), and (2) A hit spectra (a matrix) [49] where each row corresponds to an execution and contains a binary value 1 or 0 for each component. The value 1 or 0 represents whether the corresponding component was involved or not involved in an execution, respectively (see section 4.4).

Definition 5.2.2. An *Error Vector* is a list of binary flags `1` or `0` representing an execution's failed or success status, respectively.

The Pinpoint approach uses an Unweighted Pair-group Method using Arithmetic averages (UPGMA) algorithm, and an existing UPGMA implementation [119] was used in these experiments. The Pinpoint implementation requires two inputs: (1) An error vector (Definition 5.2.2), and (2) A pass/fail spectra (a matrix) [27] where each row corresponds to an execution and contains 1 or 0 for each component. The value 1 or 0 represents whether the corresponding component failed or succeeded while it was involved in an execution.

Bellur et al.'s approach uses a Bayesian Belief Network (BBN) graph where a BBN node represents the tuple <component, fault> and an edge between nodes represents the propagation of a fault between the components referenced in the BBN nodes. An existing implementation of BBN was used [55]. The truth and false probability of each node was calculated by their published formula (Eq. 5.1 [17]).

$$P(E_1\ triggers\ E_2) = P(B\ uses\ A) * P(E_1\ triggers\ E_2 \mid B\ uses\ A) \tag{5.1}$$

5.3 System and Experimental Setup

The base component systems used to evaluate FaLDAS and related baselines were the Siemens Traffic Collision Avoidance System in Java (*jtcas* obtained from SIR [33]), simulated systems and the TRANSFoRm system [37]. Simulated systems were used to analyze efficiency for large-scale systems. Using these systems, the experimental setup of four experiments is described in Section 5.3.1. Required input data generation for these experiments is discussed in Section 5.3.2.

5.3.1 Experiments

Table 5.1 outlines the experimental setup for four different experiments. The input data bootstrap experiment analyzes the Minimum Executions (E) required to bootstrap FaLDAS's knowledge-base to produce sufficiently accurate results (Section 5.4). The fault localization efficiency experiment measures the total time (T_c + T_a) it takes for each approach to localize a fault. The last two experiments analyze the Wasted

Effort (W) of algorithms where monitored information is inaccurate or components are inaccessible (Section 5.7 and 5.6).

Experiment	**System**	**Components**	**Executions**	**Faulty component**	Metric
Bootstrapping input data	TCAS	169	5...100	Fixed	E
Fault localization efficiency	Simulated	100, 500, 1000	10	Random	T_c, T_a
Inaccessible components in Adaptive System	TRANSFoRm	6	10	Fixed	W
Inaccurate monitored information	Simulated	200	10	Random	W
	TCAS	169	10	Fixed	W

Table 5.1: Experimental Setup

5.3.2 Data Generation

TCAS Data Generation

TCAS was selected because it is a real-world software system that exhibits real-world software characteristics. Moreover, TCAS is used in a number of related works (e.g., Abreu et al.) supporting comparative analysis. The SIR repository provides the complete source code in addition with 1608 test cases, one correct version and 41 faulty versions, which were used to produce the required input data [33]. Abreu et al. mapped the TCAS source code to 169 components, and based on this, 169 FPGNodes were created and tagged by input output values to/from these components, to further construct FPGs. The components' activities were also recorded to construct hit spectra and pass/fail spectra for the Barinel and Pinpoint algorithms. 169 BBN nodes were constructed and edges between nodes whose components have a data dependency were also constructed for the Bellur et al. algorithm. A faulty version's execution, corresponding to a test case, was considered failed when its output differed from the output when the same test case was applied to the correct version.

Simulated System Data Generation

A simulated system was constructed to contain a number of components, arranged in a random topology representing data dependencies between components. For each execution simulation, a component has a probability of 0.6 of being involved, which follows Abreu et al.'s work [4] [2]. Components' involvement information was used to construct the hit spectra for BARINEL.

For FaLDAS, FPGs were constructed for each execution, as input to the creation of FPG spectra. When an execution completed, FPGNodes were created for each component and edges constructed based on these nodes' data dependencies. Only 70% of all involved components' FPGNodes are tagged SUSPICIOUS. This was sampled from a Bernoulli distribution of the data observed from TCAS FPGNodes. A faulty component was randomly selected using java.util.Random library of Java Development Kit. It was assumed that a component causing a fault also changes the data state (see Section 4.3), so the corresponding FPGNode was always tagged as SUSPICIOUS in failed runs.

To construct pass/fail spectra for Pinpoint, a random 50% of all involved components are considered to have failed in an execution (Chen et al. show 50% failed components in their sample spectra [27]). To construct a BBN, a BBN node was constructed for each component and nodes were connected where corresponding components have a data dependency. To construct an error vector, 30% of the simulated system's executions were deemed to have failed. This percentage is the same as the average percentage of failed executions in sample spectra (available at [22]) and used by Abreu et al., for testing their algorithm.

5.4 Bootstrapping input data

All considered approaches depend on knowledge from a series of executions of the system to build a useful knowledge base. Bootstrapping this knowledge base is required, and for the purposes of this thesis, it was necessary to estimate how many executions are required by a fault localization mechanism to accurately diagnose the candidate sets.

This experiment analyzes the number of executions required to bootstrap a sufficient knowledge-base (a set of FPGs) to enable FaLDAS diagnose sufficiently accurate candidate sets. To achieve realistic results, the TCAS real-world system was used for the experiments. As discussed earlier, the SIR repository provides 41 faulty versions of TCAS, and each TCAS's faulty version was executed 100 times with 100 different test cases. Algorithm 1 constructed 100 corresponding FPGs. Algorithm 2 was used to carry out fault localization on TCAS using 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 FPGs, separately.

Figure 5.1 shows W with different numbers of FPGs. These measurements are averaged over 100 different fault localization runs. The results indicate that 10 executions are sufficient, as the wasted effort W is almost stabilized after 10 executions. It is likely that the more components there are in the system, the more executions would be useful. Therefore, given the scope of experimentation here, 10 is assumed to be a fair number, as where the experiments use more components than TCAS, any influence related to scale is likely to make the algorithms' fault localization task more difficult, and therefore more interesting to measure

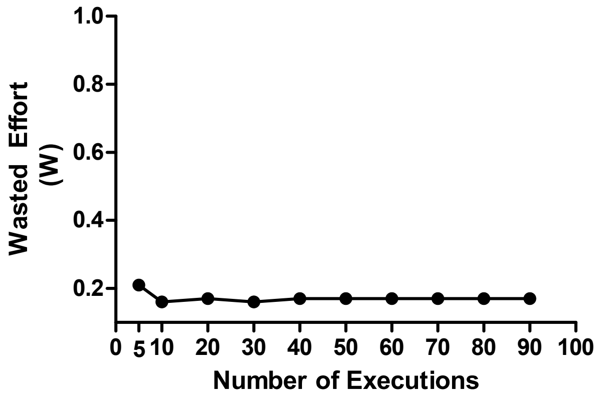


Figure 5.1: Executions required to produce accurate results.

5.5 Time Efficiency

This experiment's aim is to compare FaLDAS's time efficiency with that of other approaches i.e., BARINEL, Pinpoint and Bellur et al. The measured time is the total time incurred in knowledge-base construction (T_c) and in fault localization (T_a). This experiment measures the total time (T_c+T_a) taken by FaLDAS and other considered algorithms The time efficiencies are compared in terms of actual times recorded in the experiments and in terms of algorithms' time complexities.

Scale is the key differentiator between the algorithms, and so the experiments were carried out on simulated systems with 100, 500 and 1000 components. The TCAS system was not considered because it has a fixed, low number of components (169 components identified by Abreu et al.), which does not match the scale of today's large-scale systems with 1000s of components [26] [89] [125]. The times reported were averaged over 20 different fault localization runs. Section 5.5.1 and 5.5.2 describe the time measurements for knowledge-base construction and fault localization, respectively. Section 5.5.3 presents and compares the results.

5.5.1 Knowledge-base Construction

Knowledge-base construction time (T_c) was measured as the time it took an algorithm to read execution data and construct its knowledge-base. The simulated systems considered for the experiments, were developed in the Java programming language. The system performs a mathematical computation where each component (which is a Java object) performs a specific operation of addition or multiplication or division of the component's input. The result calculated is the component's output. The components were coordinated through the Apache Camel integration framework as described in Section 4.2.1.

FaLDAS

FaLDAS uses a set of FPGs as its knowledge-base. To construct an FPG, inputs and outputs of individual components were collected as shown in Figure 5.2. A component's output is intercepted by an interceptor implemented as a Java bean. The intercepted data is immediately placed into a message queue (ActiveMQ). The queue implements a First-In-First-Out (FIFO) order which ensures the data propagation (or dependencies) between components at FaLDAS. To retrieve data, FaLDAS set up a connection with the queue and consumes the data. The consumed data is converted into the IOV model which is further used by Algorithm 1 (see Section 4.3.2) to construct FPGs. The calculated time is the time taken by FaLDAS to consume the data, convert it into an IOV model and construct an FPG.

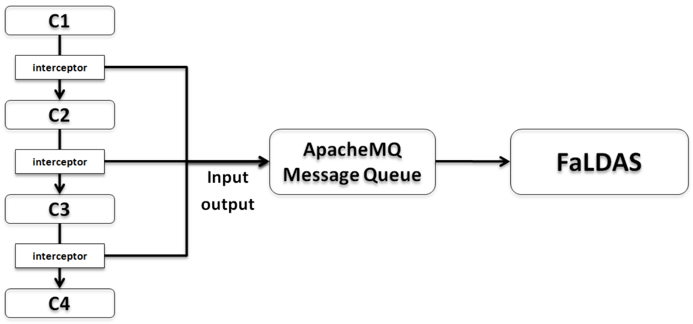


Figure 5.2: Data collection for FaLDAS

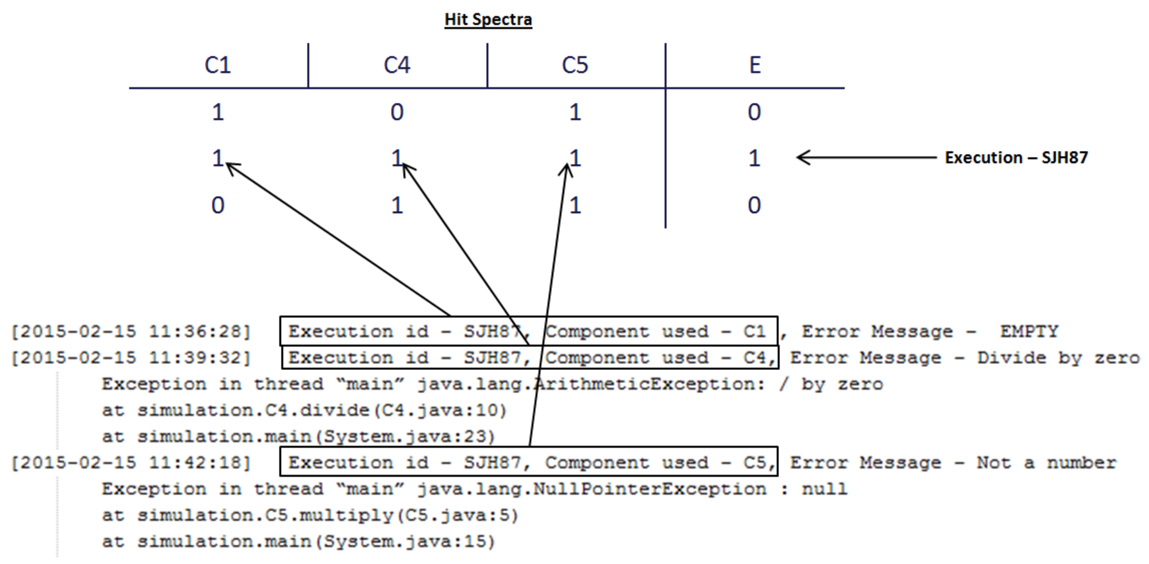
BARINEL, Pinpoint and Bellur et al.

To construct the knowledge-bases for the baseline approaches, logs files were found as the best source, which is also indicated by respective literature. Each component was implemented with a logging facility to log their activities in their respective log files. The average size of each component's log file was 189KB (which is the average size observed for a number of Apache Tomcat server's catalina log files).

Messages logged by components and the corresponding construction of hit spectra is shown in Figure 5.3. Each component logs its activity corresponding to an execution id as shown in black boxes in Figure 5.3. When such a message exists in logs, a parser reads and parse it to create the corresponding entry in the hit spectra. The entries corresponding to the components not logging such messages are 0, representing that the component was not used in the corresponding execution.

The use of logs to construct the pass/fail spectra for the Pinpoint approach is shown in Figure 5.4. The stack traces of an exception in logs are assumed as an indication of the corresponding component's failure, because the authors of Pinpoint did not specify how to detect a component's failure. However, not all components may log the stack traces; a situation where the parsing fails to construct the correct spectra. When an exception is logged by a component (black box in Figure 5.4), the corresponding entry in pass/fail spectra is 1 otherwise 0.

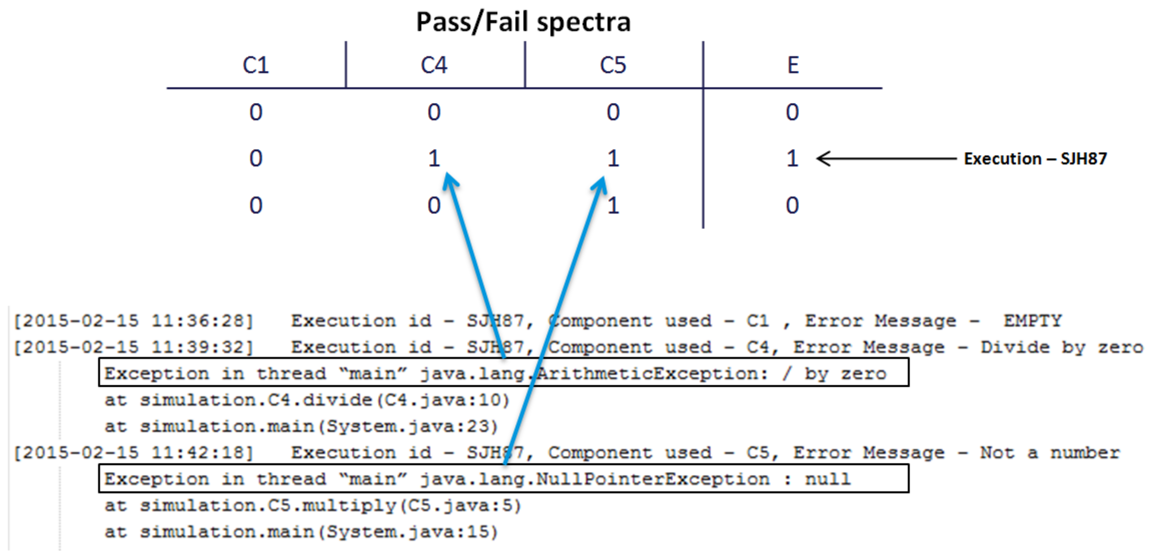
The use of logs to construct the BBN for Bellur et al.'s approach is shown in Figure 5.5. When a component logs its exception's stack trace, the corresponding error message is considered as the exception's identifier and a corresponding BBN node is constructed, as shown in Figure 5.5. When two subsequent components log execution



C1	C4	C5	E
1	0	1	0
1	1	1	1
0	1	1	0

```
[2015-02-15 11:36:28]  Execution id - SJH87, Component used - C1 , Error Message -  EMPTY
[2015-02-15 11:39:32]  Execution id - SJH87, Component used - C4, Error Message - Divide by zero
        Exception in thread "main" java.lang.ArithmeticException: / by zero
        at simulation.C4.divide(C4.java:10)
        at simulation.main(System.java:23)
[2015-02-15 11:42:18]  Execution id - SJH87, Component used - C5, Error Message - Not a number
        Exception in thread "main" java.lang.NullPointerException : null
        at simulation.C5.multiply(C5.java:5)
        at simulation.main(System.java:15)
```

Figure 5.3: Example of hit spectra construction from logs.

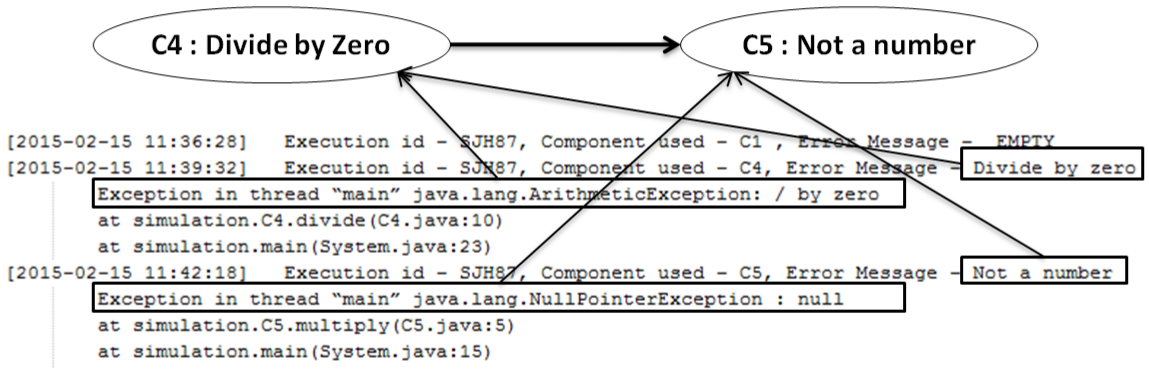


C1	C4	C5	E
0	0	0	0
0	1	1	1
0	0	1	0

```
[2015-02-15 11:36:28]  Execution id - SJH87, Component used - C1 , Error Message -  EMPTY
[2015-02-15 11:39:32]  Execution id - SJH87, Component used - C4, Error Message - Divide by zero
        Exception in thread "main" java.lang.ArithmeticException: / by zero
        at simulation.C4.divide(C4.java:10)
        at simulation.main(System.java:23)
[2015-02-15 11:42:18]  Execution id - SJH87, Component used - C5, Error Message - Not a number
        Exception in thread "main" java.lang.NullPointerException : null
        at simulation.C5.multiply(C5.java:5)
        at simulation.main(System.java:15)
```

Figure 5.4: Example of pass/fail spectra construction from logs.

stack traces for same execution id, corresponding BBN nodes are connected with an edge.



```
[2015-02-15 11:36:28]   Execution id - SJH87, Component used - C1 , Error Message - EMPTY
[2015-02-15 11:39:32]   Execution id - SJH87, Component used - C4, Error Message - Divide by zero
      Exception in thread "main" java.lang.ArithmeticException: / by zero
      at simulation.C4.divide(C4.java:10)
      at simulation.main(System.java:23)
[2015-02-15 11:42:18]   Execution id - SJH87, Component used - C5, Error Message - Not a number
      Exception in thread "main" java.lang.NullPointerException : null
      at simulation.C5.multiply(C5.java:5)
      at simulation.main(System.java:15)
```

Figure 5.5: Example of BBN nodes and edges construction from logs.

Results

This experiment measured the time taken for individual knowledge-base constructions. The corresponding results are shown in Figure 5.6. FaLDAS has the poorest performance mainly because streaming data from a message queue requires establishing a connection, and opening/closing streams. By comparison, reading log files produces more favorable results.

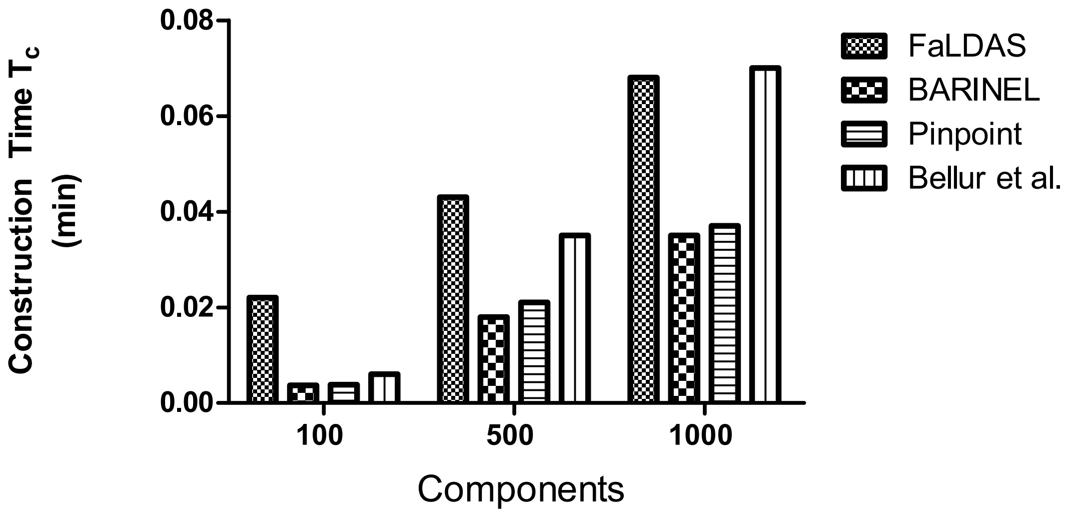


Figure 5.6: Knowledge-base construction time.

5.5.2 Fault Localization

This section describes the experiments to compare the time taken for fault localization by FaLDAS and other considered approaches. In this experiment, 10 different executions were carried out on the simulated systems. Thirty percent of these execution were failed executions. Among the failed executions, a random execution was selected to identify the root cause of the failure.

FaLDAS uses an IOV model to execute Algorithm 2 (see Section 4.4.2) for fault localization. This algorithm in turn calls Algorithm 3 (see Section 4.4.3) to calculate the health of individual FPGNodes and Algorithm 4 (see Section 4.4.4) to sort the faulty candidates. Figure 5.7 shows the time measurements. It shows that FaLDAS's fault localization efficiency (T_a) is better than that of the others because of its fundamental premise that one component can only be assessed in only one failed execution. For example, if there are 3 failed execution in a 10 executions knowledge-base, any faulty candidate set cannot contain more than 3 candidates (e.g., $\{C_1,C_2,C_3\}$ is possible but not $\{C_1,C_2,C_3,C_3\}$). In this case, FaLDAS only assess a maximum of 3 candidates (or 3 root causes) which collectively causing a failure. Components' faultiness are only assessed in failed executions; successful executions are not analyzed, which drastically reduces the analysis time. section 5.7 discusses the results of experiments to compare results' quality.

On the other hand, other approaches do not show high efficiency because of their recursive nature. BARINEL recursively calls itself C times where C is the highest cardinality of the diagnosed sets (e.g., for diagnosed sets $\{3\}$ and $\{4, 5\}$, C is 2). Pinpoint recursively invokes UPGMA's clustering mechanism $n-1$ times for n components in the system. Bellur et al. use posterior probability of BBN nodes to assess faultiness. The algorithm calculates $n \cdot 2^k$ probabilities for n nodes where each node may have a maximum of k parents. On the other hand, FaLDAS traverses the FPG to find suspicious FPGNodes and assess their faultiness only in failed runs, which does not requires recursion. For a fixed number of executions (see Section 5.4), the FaDLAS mechanism scales linearly with the number of suspicious FPGNodes in executions, which gains a substantial efficiency over the others.

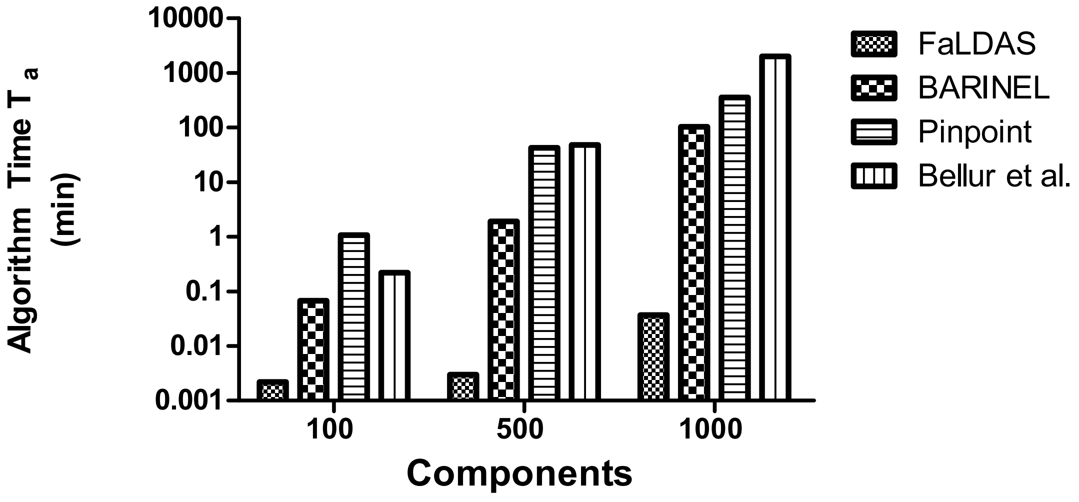


Figure 5.7: Time Measurements of fault localization

5.5.3 Results

The overall time taken, i.e., T$_c$ + T$_a$, is illustrated in Figure 5.8, which shows that FaLDAS is still the most efficient approach. FaLDAS is one order of magnitude more efficient than BARINEL and two orders of magnitude more efficient than the Pinpoint and Bellur et al.'s approaches.

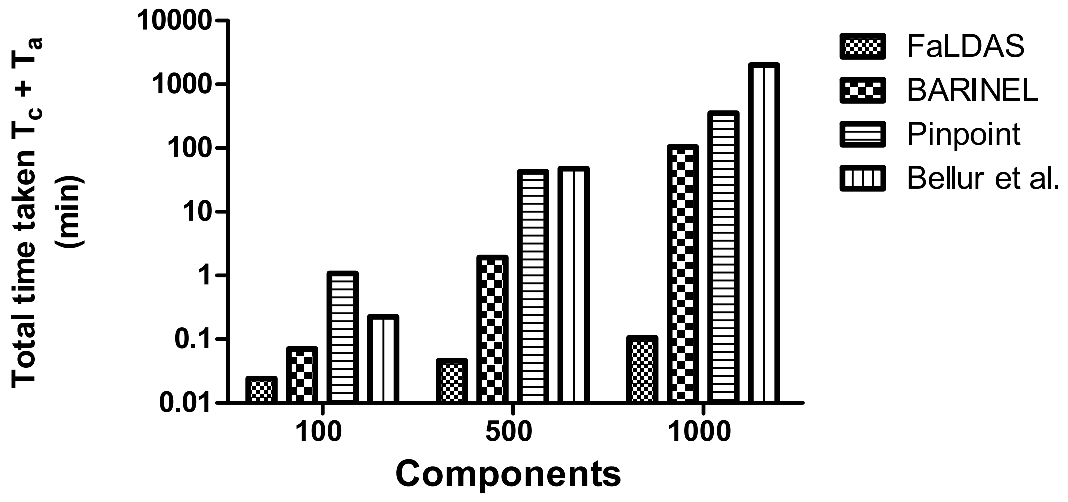


Figure 5.8: Overall time efficiency of approaches.

5.5.4 Time Complexity

The FaLDAS fault localization process consists of three parts: (i) Algorithm 2 (see Section 4.4.2) to find faulty candidates ($\mathrm{O}\big(C_f \cdot N_f \cdot M_f\big)$), (ii) Algorithm 3 (see Section 4.4.3) to determine FPGNodes' health ($\mathrm{O}\big(N_f \cdot M_f\big)$), and (iii) Algorithm 4 (see Section 4.4.4) to sort the candidates ($\mathrm{O}\big(D \cdot logD\big)$), where N_f is the number of suspicious FPGNodes in an execution, M_f is the number of failed executions, C_f is the max cardinality of the diagnosed sets (for set {C1, C2}, cardinality is 2) and D is number of diagnosed candidate sets. Consequently, the overall time complexity is $\mathrm{O}\big(C_f \cdot N_f \cdot M_f + D \cdot logD\big)$.

C_f cannot be more than M_f because each candidate is diagnosed once for each failed execution. In addition, $M_f < 10$ was sufficient for experimentation purposes (Section 5.4). $D = 100$ was also sufficient as no faulty candidate was missed at this setting. Effectively, FaLDAS's time complexity becomes $\mathrm{O}\big(N_f\big)$.

When compared against BARINEL's time complexity $\mathrm{O}\big(n \cdot m\big)$ where m is the number of executions, FaLDAS can be placed in the same class of linear algorithms as BARINEL. For constant m, FaLDAS is efficient when $N_f < n$. Moreover, the FaLDAS time complexity is linear, which outperforms higher order complexities algorithms such as Pinpoint ($\mathrm{O}\big(n^2\big)$) and Bellur et al.'s approach ($\mathrm{O}\big(n \cdot 2^k\big)$). A detailed step by step execution of BARINEL, Pinpoint and Bayesian inferences is shown in Appendix B, C and D, respectively.

5.5.5 Time Efficiency Significance

An adaptation interval is the time between two consecutive adaptations. An adaptation may be opportunistic (at random intervals), continuous (at fixed intervals), or on-demand (intervals decided by designer) [98]. To the best of our knowledge, no study exists that measures adaptation intervals for different types of systems. Therefore, for the purposes of experimentation, a range of adaptation intervals were used between very small, and very large (1ms, 1 sec, 1 min, 1 hour, 12 hour and 1 day). Each algorithm was assessed as to the proportion of times they successfully localize a faulty component before the next adaptation of the system, with the results shown in Table 5.2. The higher the success percentage, the lower the chance of a fault remaining unresolved once a new adaptation occurs. The results indicate that no approach is suitable where the adaptation interval is 1 sec or less. For intervals more than 1 sec, FaLDAS always performed best as it consistently found the faulty candidates within the specified adaptation intervals. More interestingly, as the size of the system increases, FaLDAS is likely to return a ranked list of faulty candidates in a more timely manner than the other approaches.

Scale →	100 components				500 components				1000 components			
Approaches →	F	B	P	Be	F	B	P	Be	F	B	P	Be
Adaptation Interval ↓	Success (%)				Success (%)				Success (%)			
1ms	0	0	0	0	0	0	0	0	0	0	0	0
1 sec	0	0	0	0	0	0	0	0	0	0	0	0
1 min	100	100	55	78	100	45	0	0	100	0	0	0
1 hour	100	100	100	100	100	100	100	68	100	45	0	0
12 hour	100	100	100	100	100	100	100	100	100	100	100	0
1 day	100	100	100	100	100	100	100	100	100	100	100	0

F- FaLDAS, B-BARINEL, P-Pinpoint, Be- Bellur et al.'s approach

Table 5.2: Time efficiency significance in the context of adaptive systems

5.6 Inaccessible Components

This section presents the experiments to evaluate FaLDAS's quality in terms of Wasted Effort (W), compared with that of the other considered approaches. The experiments are carried out on the TRANSFoRm system, where a number of components are inaccessible to fault localization mechanisms. The aim of the experiment is to analyze FaLDAS's W when the actual faulty component is inaccessible.

When components are inaccessible in an adaptive system, this may have the effect of either decreasing or increasing effort. If the inaccessible component is not faulty, and its inaccessibility means it is not included in any analysis, then no time is wasted analysing that particular component. However, if the inaccessible component is actually the faulty one, but again its inaccessibility means it is not included for analysis, then all time spent analysing components in the list that is returned is wasted. For example, assume that a candidate list identified by a fault localization mechanism is <C_1, C_2, C_3, C_4 >, with components ranked as $C_1>C_2>C_3>C_4$. If the actual faulty component is C_3, then the wasted effort is 2/4. If C_2 is inaccessible, it does not appear on the candidates list (non-appearance (or explicit removal) is our assumption because other approaches do not specify the effect of inaccessible components on their approach). In that case, the candidate list is <C_1, C_3, C_4> which is sorted as C_1>C_3>C_4. Here, the wasted effort is 1/3 which is less than 2/4. It shows how inaccessible components have helped to decrease Wasted Effort.

However, when the actual faulty component is inaccessible, wasted effort is likely to be increased. In the example, the candidate list from BARINEL, Pinpoint and Bellur et al., is <C_1, C_2, C_4 > when C_3 is actually faulty, but inaccessible. A debugging expert will analyze all three components but will not find the actual faulty component. In that case, the wasted effort will be 3/3. However, as FaLDAS is not effected by inaccessibility of components (shown in Figure 5.9), the candidate list will be <C_1, C_2, C_3, C_4 > and wasted effort will be 2/4 as earlier which is better than 1.

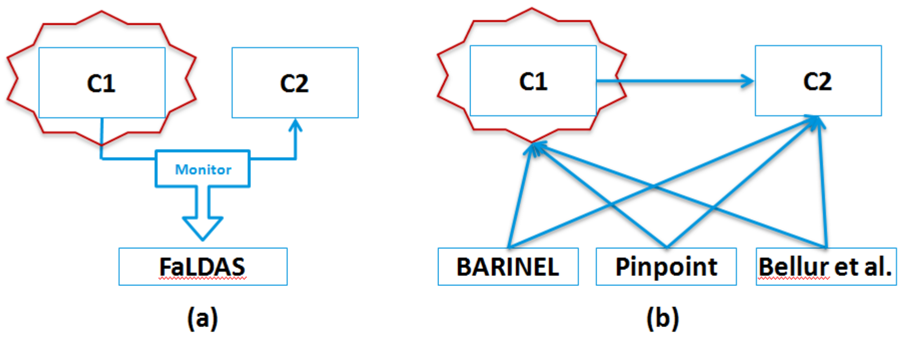
Figure 5.9 illustrates an example of how FaLDAS copes with inaccessible components. In this figure, there are two components, C1 and C2, and C1 is inaccessible, i.e., the logs for this component are inaccessible. Figure 5.9a shows that an interceptor-based monitor (Section 3.5) intercepts the requests from C1 to C2 and provides the request data (variable names and their values) to FaLDAS. Even though the component C1 is inaccessible, FaLDAS generates the required FPG spectra from the intercepted data, as shown in Figure 5.9c. Using this FPG spectra, FaLDAS can identify that C1 is a faulty candidate.

However, BARINEL, Pinpoint and Bellur et al. attempt to access C1's logs, but access is denied. As a result, the corresponding entries in the hit spectra of BARINEL and pass/fail spectra of Pinpoint are missing as shown in Figure 5.9d. Nor will Bellur et al.'s BBN have any BBN node corresponding to C1. The resulting candidate set is ({C2}), which misses component C1. The debugging expert analyzes all candidates in the candidate set, i.e., C2 but will not find the actual fault, making the Wasted Effort=1.

To evaluate FaLDAS for inaccessible actual faulty components, a real-world systems TRANSFoRm was used. Following is a description of experiments on this systems.

5.6.1 Experiment

TRANSFoRm is an FP7 research project [37], which developed a system that enables medical researchers to create queries against a patient database to access relevant



		e
C1-V1-S	C2-V2-S	1
C1-V1-C	C2-V2-S	0

(c)

C1	C2	e
-	1	1
-	1	0

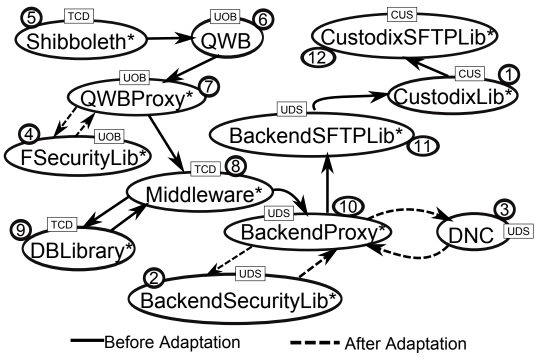
(d)

Figure 5.9: Example and effect of an inaccessible component. Component C1 is the actual faulty component but has inaccessible logs. (a) shows how FaLDAS can access information even when component is inaccessible. (b) shows that other approaches are not able to access logs of an inaccessible component. (c) shows the FPG Spectra for FaLDAS. (d) shows spectra for BARINEL and Pinpoint with missing entry.

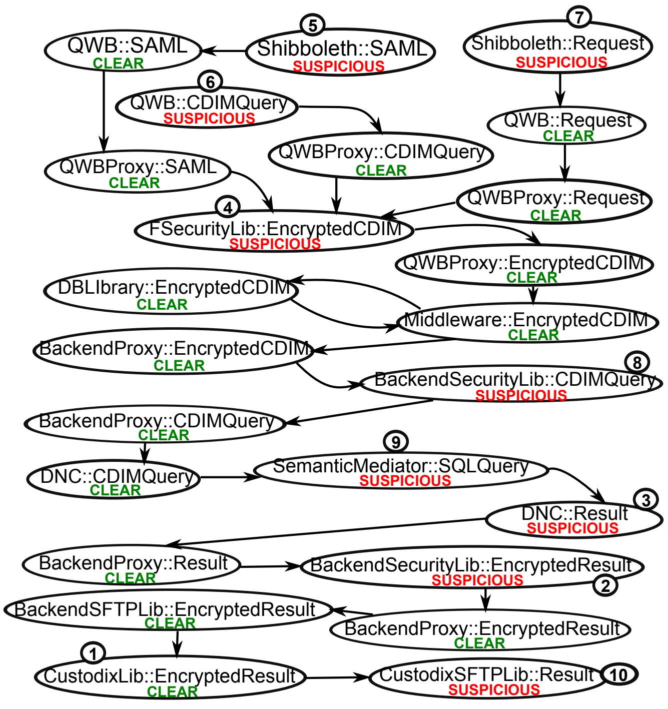
information about patients. The project components are distributed across several geographic locations where different consortium partners are constantly updating their components to cater for new features; a suitable usecase for FaLDAS's evaluation.

Adaptation Scenario

Figure 5.10a describes the data dependencies between the TRANSFoRm system components. Initially, the project was developed in such a way that a researcher can create a simple SQL Query which executes on a database and the results were returned back to the researcher. Due to policy changes in handling patients' secure data, it was required that the queries and results should be transported in a secure and encrypted way. Thus, the project was adapted in such a way that researchers have to create a query in CDIM format [37] and that query should be encrypted before leaving the QWB location (Birmingham). The components FSecurityLib and DNC (numbered 4 and 3 in Figure 5.10a) were added to the system to encrypt and process the query, respectively. The results from DNC to Custodix(Belgium) should be transported in an encrypted way over a secure transport layer. After the adaptation, the system had two components inaccessible to TCD's fault localization mechanisms i.e., QWB, DNC.



(a) TRANSFoRm system. Components deployed at TCD-Trinity College Dublin, UOB - University Of Birmingham, UDS - University of Dundee Scotland,CUS - Custodix Belgium, *components developer by TCD team.



(b) FPG for Transform System

Figure 5.10: Transform System

Fault Analysis

A number of successful and failed executions were carried out. An execution is the construction of a CDIM query, retrieving the corresponding data from a database and returning it back to the researcher. This section presents a scenario of a selected failed execution where data obtained corresponding to a query is invalid. While a researcher was retrieving results through the CustodixSFTPLib component, the system encountered a fault in the CustodixLib (1 in Figure 5.10a) component due to the corrupted input variable `EncryptedResult`. To identify the actual fault, FaLDAS use the FPGs. Figure 5.10b shows one of the FPGs, which is corresponding to the failed execution.

Algorithm 2 (see Section 4.4.2) identified the faulty candidates from these FPGs. It identified eighteen potentially faulty candidates as shown in Table 5.3. These faulty candidates were sorted according to their probability of being faulty. Candidate set 7 was found highest in the ranking because FPGNodes 6 and 3 were always tagged SUSPICIOUS in failed runs. This indicates that either all members of the set or its subset are most likely to originate the fault. Applying root cause analysis on each components, referred in the FPGNodes, identified that the DNC component was actually faulty because its query parsing logic was incorrectly implemented. As a result, the incorrect data set was returned from the database. Effort was wasted in analyzing four FPGNodes i.e., 6, 4, 8 and 9. Consequently, the wasted effort W through FaLDAS is $4/18 = 0.22$. Ten such fault localizations were carried out to estimate the wasted effort for inaccessible components. The average of which is shown in Figure 5.11.

1 - $\{ 5 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 2 \rightarrow 10 \}$	10 - $\{2\}$
2 - $\{ 4 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 2 \rightarrow 10 \}$	11 - $\{3\}$
3 - $\{ 8 \rightarrow 9 \rightarrow 3 \rightarrow 2 \rightarrow 10 \}$	12 - $\{4\}$
4 - $\{ 9 \rightarrow 3 \rightarrow 2 \rightarrow 10 \}$	13 - $\{5\}$
5 - $\{ 3 \rightarrow 2 \rightarrow 10 \}$	14 - $\{6\}$
6 - $\{ 2 \rightarrow 10 \}$	15 - $\{7\}$
7 - $\{ 6 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 2 \rightarrow 10 \}$	16 - $\{8\}$
8 - $\{ 7 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 2 \rightarrow 10 \}$	17 - $\{9\}$
9 - $\{1\}$	18 - $\{10\}$

Table 5.3: TRANSFoRm System Faulty Candidates

To compare FaLDAS's effort with that of BARINEL, Pinpoint and Bellur et al's algorithms, their knowledge-bases were constructed. To construct the hit spectra for BARINEL, the components involved in an execution must be known. Ideally, components log their activities in their logs. Section 2.7 discusses that logs are the preferred way to identify the components involvement. However, the TRANSFoRm system includes a number of components (6 and 3 in Figure 5.10a) where access to the logs was blocked. Using the logs of only accessible components, the generated hit spectra is shown in Table 5.4a. The spectra miss the entries corresponding to components 6 and 3 because their logs were not available. Moreover, all accessible components were found involved in all execution, so the spectra has entries '1'. In such a case, all components are equally likely to be faulty. Moreover, the actual faulty component i.e., 3 can not be included in any faulty candidate set. Thus, a debugging expert analyzes all the components specified in the spectra for root cause analysis making the wasted effort $W = 1$ as shown in Figure 5.11a.

It can be argued that the IOV model could be used to figure out whether components are involved in an execution. For example, a component specified in an IOV model instance must have been used in an execution to produce an output. The IOV model was found helpful in identifying the involvement of inaccessible components in an execution. Using this, the corresponding hit spectra generated is shown in Table 5.4b. This spectra includes the entries corresponding to inaccessible components 3 and 6. However, all the components were used in all executions (all entries are '1'), and so BARINEL found all equally likely to be faulty. The actual faulty component 3 will be included in the faulty candidate set. However, BARINEL can not identify if component 3's likelihood of being faulty is higher or lower than any others.

To calculate the wasted effort using Pinpoint, pass/fail spectra were created. In this case, it was difficult to identify a suitable way to assess the failure status of any individual component for an execution, without having access to the logs, as indicated by Chen et al. [27]. The corresponding spectra generated is shown in Table 5.5, which misses the entries corresponding to inaccessible components 3 and 6. Therefore, the faulty candidate sets identified do not include actual faulty component 3. The debugging expert analyzes all the components specified in the spectra, making the wasted effort $W = 1$, as shown in Figure 5.11a.

To calculate the wasted effort incurred in Bellur et al.'s approach, components' faults were modeled in a Bayesian Belief Network (BBN). A BBN's node specifies a

1	2	4	5	7	8	9	10	11	12	e
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1

(a) Hit Matrix generated using logs

1	2	3	4	5	6	7	8	9	10	11	12	e
1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1

(b) Hit Matrix generated using IOV Model

Table 5.4: TRANSFoRm's Hit Spectra

1	2	4	5	7	8	9	10	11	12	e
0	0	0	0	0	0	0	0	1	0	1
0	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1

Table 5.5: TRANSFoRm pass/fail spectra

component and its fault that may propagate. Bellur et al. obtain components' details and their potential faults from execution traces. However, the TRANSFoRm system does not allow access to actual faulty component 3's execution traces. So, the BBN misses the corresponding nodes. BBN's Probabilistic inferences sort the components according to their probability of being faulty. However, the sorted list does not include the actual faulty component 3. The debugging expert must analyze all components in the sorted list, which makes the wasted effort $W = 1$ as shown in Figure 5.11a.

5.6.2 Results

The experimental results are described in Figure 5.11. As expected, these results show that FaLDAS performs best when the actual faulty component is inaccessible. In

TRANSFoRm system experiments, FaLDAS's W is 0.29. On the other hand, in the BARINEL, Pinpoint and Bellur et al.'s approaches, W is consistently 1. This is because the actual faulty component's information is unavailable in all three cases. The three mechanisms do not assess the component's activities e.g., component's usage, failure and faults of the actual faulty components in an execution. As a result, the actual faulty component is not included in their resulted candidate sets. The debugging expert has to analyze all the candidates, which makes the Wasted Effort W = 1. FaLDAS works as usual and returns candidate sets which include the faulty component.

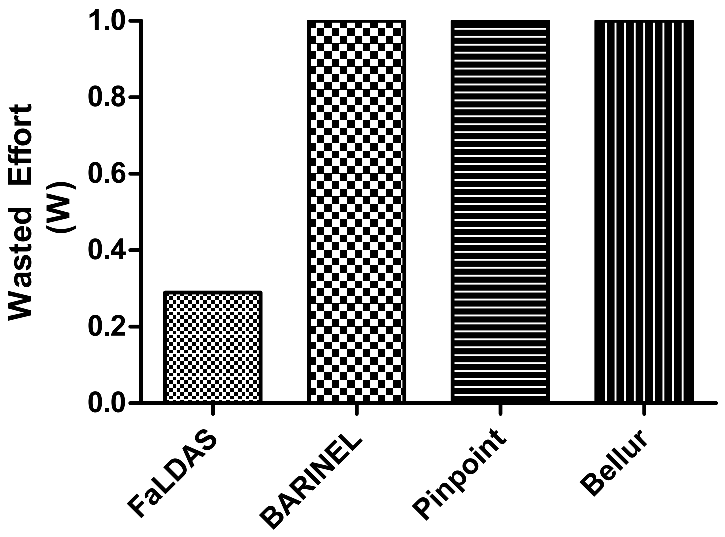


Figure 5.11: Wasted effort when actual faulty component is inaccessible in TRANSFoRm System.

5.7 Inaccurate Monitored Information

This experiment analyzes FaLDAS's Wasted Effort (W) where its knowledge-base has inaccurate data. The aim of the experiment to is assess FaLDAS's W and to compare it with related approaches to find if FaLDAS is an improvement over existing mechanisms or not.

5.7.1 Inaccurate Knowledge-bases

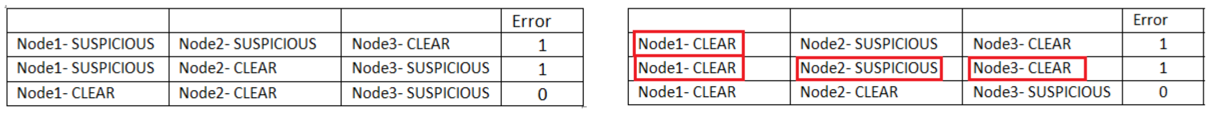
It is possible that the information collected at runtime may be inaccurate due to limited technology or too complex design of the system. This section defines the inaccurate knowledge-bases for the considered approaches.

FaLDAS Knowledge-base

FaLDAS uses a set of FPGs as its knowledge-base, constructed from IOV models. Several factors may cause inaccuracies in an IOV model, such as an inaccurate component name, inaccurate input output values, or inaccurate data flow between components, gathered by FaLDAS. Input output values can be inaccurate when a monitor wrongly records input of one component and output of another, and collectively sends it to FaLDAS. FaLDAS assumes inputs and outputs of the same component, which create inaccurate IOV model, and subsequently inaccurate knowledge-base. Although all factors for inaccuracy are important, this experiment considers inaccuracy in input output values. FaLDAS gathers input output values from a system as described in Section 5.5.1. Input and output values are used to tag the FPGNodes as described in Section 4.3.3. Therefore, inaccurate input output values may result in inaccurate tagging of FPGNodes.

An inaccurate tag of an FPGNode means the current tag is x but it should be tagged with y. For example, an FPGNode <C,V,CLEAR> is wrongly tagged because it should be <C,V,SUSPICIOUS>. The results show that inaccurate tagging (current is CLEAR, should be SUSPICIOUS) of the actual faulty FPGNode increases the Wasted Effort. On the other hand, an incorrect tag in any other FPGNode may increase or decrease the Wasted Effort.

Assume an FPG spectra as shown in Figure 5.12. In the correct FPG spectra (Figure 5.12a), the actual faulty FPGNode for the failed execution no. 1, is Node2 (pre-determined). FaDLAS produces the sorted candidate list as { Node1 > Node2 > Node3}. The wasted effort is 1/3 because the debugging expert wastes effort only in one candidate i.e., Node1. However, when the spectra contains inaccurate tags (see Figure 5.12b), FaLDAS produces a sorted candidate list as { Node2 > Node3 > Node1} because Node1's tag is always CLEAR. In this case, the wasted effort is zero because the debugging expert found the first faulty candidate as the actual faulty one. Here, incorrect tags actually helped to decrease the Wasted Effort.



			Error
Node1- SUSPICIOUS	Node2- SUSPICIOUS	Node3- CLEAR	1
Node1- SUSPICIOUS	Node2- CLEAR	Node3- SUSPICIOUS	1
Node1- CLEAR	Node2- CLEAR	Node3- SUSPICIOUS	0

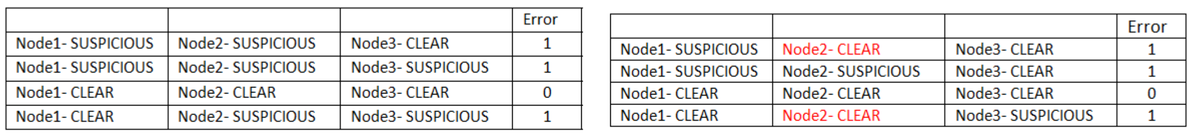
(a) FPG Spectra with FPGNodes having correct tags

			Error
Node1- CLEAR	Node2- SUSPICIOUS	Node3- CLEAR	1
Node1- CLEAR	Node2- SUSPICIOUS	Node3- CLEAR	1
Node1- CLEAR	Node2- CLEAR	Node3- SUSPICIOUS	0

(b) FPG Spectra with FPGNodes having incorrect tags

Figure 5.12: Examples of FPG Spectra with correct and incorrect tags

However, the results show that when the actual faulty FPGNode is wrongly tagged, the Wasted Effort increases. To explain this, let us consider the FPG spectra shown in Figure 5.13. For the spectra shown in Figure 5.13a, FaLDAS produces the sorted candidate list as { Node2 > Node1 > Node3 }. However, when Node2 is tagged CLEAR more often in the failed runs as shown in Figure 5.13b, the sorted list of candidates produced is { Node1 > Node2 > Node3 }. In the earlier case, the wasted effort is zero but in the later case, it is 1/3. This illustrates that SUSPICIOUS → CLEAR tagging of actual faulty node in failed runs is likely to increase debugging effort. This shows that the more the actual faulty FPGNodes are tagged as CLEAR (incorrectly) in failed executions, the less faulty FPGNodes, which overall increases the Wasted Effort.



			Error
Node1- SUSPICIOUS	Node2- SUSPICIOUS	Node3- CLEAR	1
Node1- SUSPICIOUS	Node2- SUSPICIOUS	Node3- SUSPICIOUS	1
Node1- CLEAR	Node2- CLEAR	Node3- CLEAR	0
Node1- CLEAR	Node2- SUSPICIOUS	Node3- SUSPICIOUS	1

(a) Actual faulty node Node2 has more SUSPICIOUS tags in failed runs

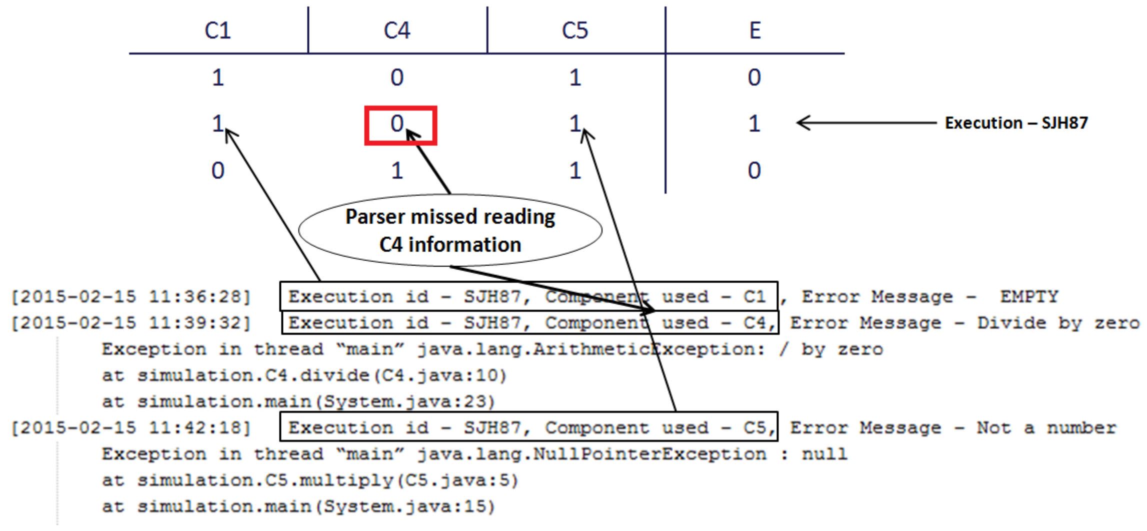
			Error
Node1- SUSPICIOUS	Node2- CLEAR	Node3- CLEAR	1
Node1- SUSPICIOUS	Node2- SUSPICIOUS	Node3- CLEAR	1
Node1- CLEAR	Node2- CLEAR	Node3- CLEAR	0
Node1- CLEAR	Node2- CLEAR	Node3- SUSPICIOUS	1

(b) Actual faulty node Node2 has less SUSPICIOUS tags in failed runs

Figure 5.13: FPG Spectra representing the wrong tags of actual faulty node

BARINEL Knowledge-base

The BARINEL algorithm uses hit spectra as its knowledge-base. These are constructed from the information about a component's involvement in an execution (see Section 5.5.1). When a component's involvement is wrongly identified, the corresponding entry in the hit spectra is incorrect. An example for inaccurate hit spectra is shown in Figure 5.14. A parser reads a component's involvement in the logs and correspondingly populates the hit spectra entries. In this example, the parser missed the log entry for component C4. As a result, the corresponding entry in the hit spectra is 0, whereas it should be 1. Such incorrect spectra are used in this experiment.



C1	C4	C5	E
1	0	1	0
1	0	1	1
0	1	1	0

```
[2015-02-15 11:36:28]  Execution id - SJH87, Component used - C1 , Error Message -  EMPTY
[2015-02-15 11:39:32]  Execution id - SJH87, Component used - C4, Error Message - Divide by zero
            Exception in thread "main" java.lang.ArithmeticException: / by zero
            at simulation.C4.divide(C4.java:10)
            at simulation.main(System.java:23)
[2015-02-15 11:42:18]  Execution id - SJH87, Component used - C5, Error Message - Not a number
            Exception in thread "main" java.lang.NullPointerException : null
            at simulation.C5.multiply(C5.java:5)
            at simulation.main(System.java:15)
```

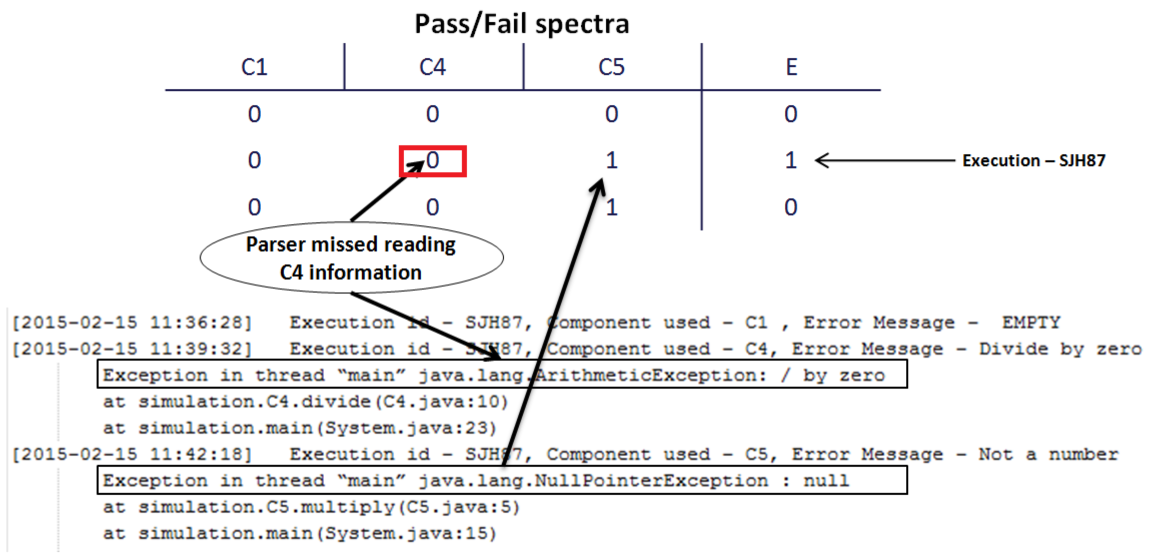
Figure 5.14: Inaccurate hit spectra.

Pinpoint Knowledge-base

The Pinpoint algorithm uses pass/fail spectra as its knowledge-base. These are constructed from the information about a components' failure status in an execution (see Section 5.5.1). When a component's failure status is wrongly identified, the corresponding entry in the pass/fail spectra is incorrect. An example of inaccurate pass/fail spectra is shown in Figure 5.15. A parser reads a component's involvement in the logs and correspondingly populates the spectra entries. In this example, the parser missed the log entry for component C4' failure i.e., exception's stack trace. As a result, the corresponding entry in the spectra is 0, whereas it should be 1.

Bayesian Belief Network

Bellur et al. use a BBN as their knowledge-base, in which a node is of the form <component, fault>. A potential inaccuracy in BBN arises when a component's or its fault's information is not available from a parser. In situations where the parser missed a component and its fault's information, the resulting BBN is inaccurate. An example

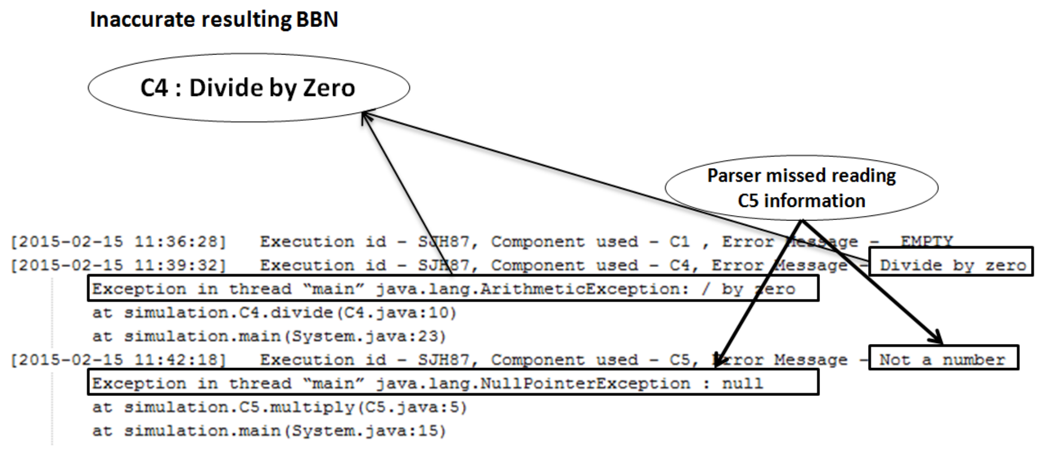


C1	C4	C5	E
0	0	0	0
0	0	1	1
0	0	1	0

```
[2015-02-15 11:36:28]   Execution id - SJH87, Component used - C1 , Error Message -  EMPTY
[2015-02-15 11:39:32]   Execution id - SJH87, Component used - C4, Error Message - Divide by zero
        Exception in thread "main" java.lang.ArithmeticException: / by zero
        at simulation.C4.divide(C4.java:10)
        at simulation.main(System.java:23)
[2015-02-15 11:42:18]   Execution id - SJH87, Component used - C5, Error Message - Not a number
        Exception in thread "main" java.lang.NullPointerException : null
        at simulation.C5.multiply(C5.java:5)
        at simulation.main(System.java:15)
```

Figure 5.15: Inaccurate pass/fail spectra.

of an inaccurate BBN is shown in Figure 5.16. The BBN should have two nodes but has only one node because the parser missed the information about component C5 and its fault's stack trace.



```
[2015-02-15 11:36:28]   Execution id - SJH87, Component used - C1 , Error Message -  EMPTY
[2015-02-15 11:39:32]   Execution id - SJH87, Component used - C4, Error Message - Divide by zero
        Exception in thread "main" java.lang.ArithmeticException: / by zero
        at simulation.C4.divide(C4.java:10)
        at simulation.main(System.java:23)
[2015-02-15 11:42:18]   Execution id - SJH87, Component used - C5, Error Message - Not a number
        Exception in thread "main" java.lang.NullPointerException : null
        at simulation.C5.multiply(C5.java:5)
        at simulation.main(System.java:15)
```

Figure 5.16: Inaccurate BBN.

5.7.2 Experiment

This experiment measures the impact on wasted effort when information input to the algorithms is inaccurate. Specifically, it compares the Wasted Effort of FaLDAS with that of other approaches, when their knowledge-bases are inaccurate. Each entry in the knowledge-bases is termed as a data point e.g., an FPGNode, a component's involvement (1 or 0), a component's failure status (1 or 0) and a BBN node are the data points for FPG spectra, hit spectra, pass/fail spectra and BBN, respectively. Three different experiments were carried out with 0%, 20% and 40% inaccurate data points.

To evaluate rigorously and realistically, the experiments were carried out on simulated systems and on the TCAS system. The knowledge-bases, generated from both the systems, were injected with inaccurate data points. In FPG spectra, an actual faulty FPGNode was wrongly tagged CLEAR (should be tagged SUSPICIOUS) in a number of FPGs corresponding to the failed executions. In hit spectra and pass/fail spectra, data points corresponding to the actual faulty component were flipped from 1 to 0 for failed execution rows. In hit spectra, such flipping means that a component was used in a failed execution but the spectra indicates that it was not used. In pass/fail spectra, the flipping means that an actually failed component is considered as not failed in a failed execution. To introduce inaccuracy in BBN, a number nodes corresponding to actual faulty components were removed from the BBN. Note that removing a node is our assumption to introduce the inaccuracy in BBN because Bellur et al. do not specify what is the effect of inaccurate information. Moreover, removing a node from BBN is a case similar to when a component is inaccessible for Bellur et al.'s approach.

5.7.3 Results

The Wasted Effort (W) incurred was analyzed to evaluate diagnosis quality. Challenge 2, i.e., inaccurate information (discussed in Section 1.3) is the primary factor that increases W. All measurements are an average of 100 diagnostic runs. The results are represented in Figure 5.17 and 5.18 and corresponding to the average values are presented in Table 5.6.

The results show that FaLDAS's diagnostic quality is in line with that of the other baseline approaches in simulated systems experiments. For example, FaLDAS's average

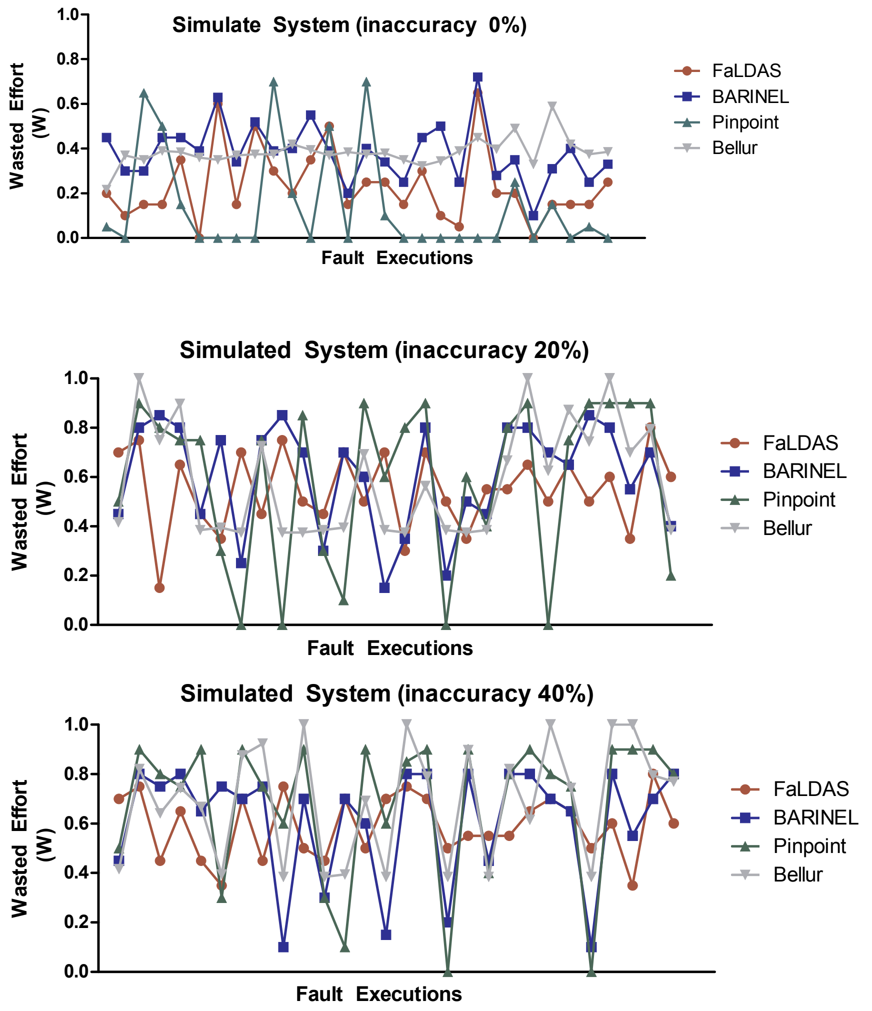


Figure 5.17: Wasted Effort analysis on simulated systems for inaccurate knowledge-base

	Simulated System			**TCAS System**		
Inaccuracy →	0%	20%	40%	0%	20%	40%
FaDLAS (W)	0.35	0.55	0.59	0.1	0.147	0.217
BARINEL (W)	0.41	0.6	0.61	0.1	0.142	0.193
Pinpoint (W)	0.31	0.58	0.67	0.09	0.13	0.17
Bellur et al. (W)	0.54	0.58	0.68	0.23	0.23	0.24

Table 5.6: Average Wasted effort for inaccurate input data

W is 0.35 which is better than BARINEL's 0.41 and Bellur et al.'s 0.54 but more than Pinpoint's 0.31. The results of simulated system experiments with inaccurate data shows that FaLDAS mostly performs better than the other approaches (see Table 5.6).

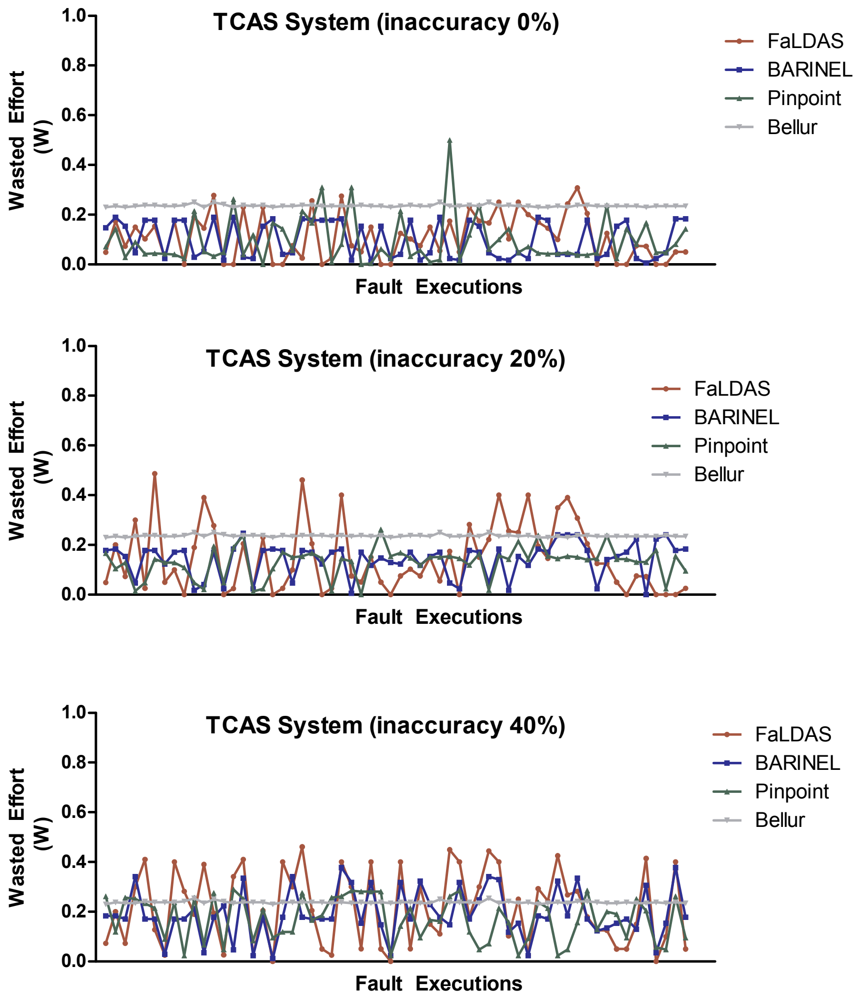


Figure 5.18: Wasted Effort analysis on TCAS systems for inaccurate knowledge-base

However, the TCAS system results are different, showing that Pinpoint performs best (W= 0.09/0.13/0.17) and FaLDAS ranks third (W= 0.1/0.147/0.217). In general, the data shows that the approaches are not significantly different, and so accuracy is not a differentiator.In conclusion, the FaLDAS diagnostic quality is not the best not the worst among all considered approaches when input information is incorrect. As is intuitive, inaccurate input negatively effects the chances for accurate output, for all approaches.

The Wasted Effort varies according to the actual faulty candidate's behavior. More often is the actual faulty FPGNode tagged SUSPCIOUS in the FPGs corresponding to the failed executions, higher it is in the sorted list of candidates by FaLDAS. It is because the health of such an FPGNode is reduced which FaLDAS uses to indicate the FPGNode as more likely to be faulty. On the other hand, where FPGNode is less often tagged SUSPICIOUS in the FPGs corresponding to failed executions, higher is its health and lower it is in the sorted list. Lower in the sorted list means the debugging expert has to analyze more number of candidates before finding it (the actual faulty candidate). It is the reason why there are some red peaks of FaLDAS in Figure 5.18. The health of actual faulty FPGNode was higher for the cases of red peaks. The actual faulty component shows its fault very rarely. The knowledge-bases corresponding to such fault localizations does not show substantial faulty behavior of actual faulty FPGNode.

Although FaLDAS does not performs the best for inaccurate information, it shows to cope with inaccurate information. Assume a component A having three output variables X, Y and Z as shown in Figure 5.19. Also, assume that A's fault causing Y to be corrupted, which further propagates to cause a failed execution. Here, variable Z's real-time values recorded are inaccurate, which makes the corresponding FPGNode <A, Z, CLEAR> inaccurate (wrong tag). In such case, FaLDAS uses other FPGNodes i.e., <A, X, SUSPICIOUS> and <A, Y, SUSPICIOUS> to accurately identify a faulty candidate <A, Y> which indicates that A is potentially faulty and may be causing a fault which corrupts output variable Y as shown in Figure 5.19a. Where A's information is inaccurate in BARINEL's hit spectra and Pinpoints pass/fail spectra, they also consider A as a potential faulty candidate but could not detect corrupted variable. FaLDAS results about corrupted variable is nonetheless helps more a debugging expert than that of BARINEL and Pinpoint helps. It is because a debugging expert has to analyze a specific part of the component (only which effects Y) not the whole one, which is very useful if the component is very big and complicated; a feature which is not provided by BARINEL, Pinpoint and Bellur et al.'s approach.

5.8 Chapter Summary

This chapter presents a set of experiments to evaluate FaLDAS for its time efficiency and quality (Wasted Effort W - see Definition 5.2.1). An additional experiment is presented to assess the number of executions required to generate a sufficient knowledge-

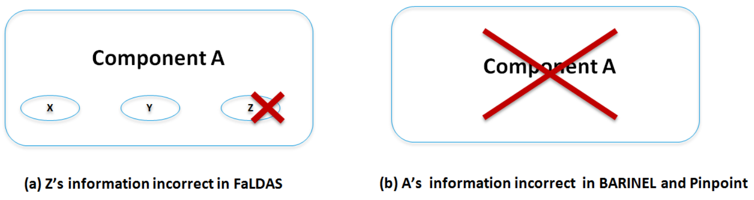

Component A
X
Y
Z
Component A
(a) Z's information incorrect in FaLDAS
(b) A's information incorrect in BARINEL and Pinpoint

Figure 5.19: An example to show the impact of inaccurate information on fault localization

base. The baseline systems used for the evaluation are simulated system, Traffic Collision Avoidance System (TCAS) and TRANSFoRm system. FaLDAS results are compared with that of BARINEL [4], Pinpoint [27] and Bellur et al. [17] approaches. The results illustrate that FaLDAS's time efficiency is the best among all considered approaches. Moreover, FaLDAS performs best when actual faulty component is inaccessible to fault localization mechanisms. However, FaLDAS is not the best where required information information is inaccurate.

Chapter 6

Discussion and Future Work

This chapter presents a discussion about the aspects of FaLDAS, which are not discussed earlier. Section 6.1 discusses the thesis's assumptions and when they do not hold. It discusses cases where FaLDAS successfully identifies the faulty component and where it fails. Section 6.2 discusses FaLDAS's limitations and possible future works to enhance FaLDAS. Section 6.3 ends the chapter with a summary.

6.1 Discussion

This section discusses an extended set of use-cases to assess FaLDAS feasibility when thesis's assumptions hold and when do not hold. A discussion about the knowledge-base, experiments and fault localization is presented.

6.1.1 Input Output Values

FaLDAS constructs its knowledge-base from components' input output values in real-time. These values are the basic building blocks of FaLDAS. So, it is important to assess where these values help FaLDAS finding actual faulty component and where not. Table 6.1 presents four possible combinations of a component's fault status and its input output values. The top row shows two cases where a component is (i) actually faulty and (ii) not faulty. The first column shows sameness between the component's input and output.

Component is → Sameness↓	Faulty	Not faulty
Output ≠ Input	FaLDAS successful (Case 1)	FaLDAS fails (Case 2)
Output = Input	FaLDAS fails (Case 3)	FaLDAS successful (Case 4)

Table 6.1: Four cases related to input output values and faulty components

Case 1: Component is Faulty and Its Output ≠ Input

In this case, FaLDAS successfully identifies a faulty component as a potentially faulty one. It is because its output is not the same as its input, so the corresponding FPGNode is tagged SUSPICIOUS. Consequently, FaLDAS includes the component in the candidate sets which indicates that it could be responsible to originate the fault. A debugging expert finds this faulty component in the corresponding candidate set.

Case 2: Component is Not Faulty and Its Output ≠ Input

In this case, FaLDAS cannot identify a non-faulty component as non-faulty rather considers it as a potentially faulty one. Where the component is not faulty (i.e., output is correct) and output is not the same as its the input, FaLDAS tags the corresponding FPGNode as SUSPICIOUS which is wrong (FaLDAS's limitations). As a result, the corresponding component is included in the candidate set indicating that it is a potential faulty component, which actually is not.

It may also happen that the component may have activated a fault, but also caught it successfully and provided the correct desired output. For example, several Java exceptions are caught. Although such faults may not have implications on the overall system (probably because get caught), the detection and resolution of such faults is nonetheless important. In that case, FaLDAS successfully identify such a component as potentially faulty because the corresponding FPGNode is tagged SUSPICOUS and the component is included in the candidate sets.

Case 3: Component is Faulty and Its Output = Input

In this case, FaLDAS cannot identify a faulty component as a potentially faulty one. Where the component is faulty (output is incorrect) and coincidentally the incorrect output is the same as input to the component, FaLDAS tags the corresponding FPGNode as CLEAR. The CLEAR tag indicates that the component has not modified the data, so considered as non-faulty and not included in the faulty candidate sets. To illustrate it, let us consider an example where a Java object encounters a divide by zero fault but caught by a try-catch block and as a fail-safe strategy, the component returned an output same as input. In such a case, FaLDAS fails to identify the actual faulty component.

Case 4: Component is Not Faulty and Its Output = Input

In this case, FaLDAS successfully identifies a non-faulty component as non-faulty. Where the component is not faulty (correct output) and the output is same as input, FaLDAS tags the corresponding FPGNode as CLEAR to indicate that the component effect the data, which actually is correct. As a result, the component is not included in the candidate set and considered as non-faulty component.

However, it may happen that the component may have activated a fault but the component managed to compute the desired correct output which coincidentally the same as input. Here, the fault need to be resolved, however, FaLDAS fails to indicate the component as a potentially faulty one because of the same input output values which causes FaLDAS to tag the corresponding FPGNode as CLEAR. As a result, the faulty component is not included in the faulty candidate set and considered as non-faulty, which is wrong.

6.1.2 Experiments Data Generation

Chapter 5 describes a number of experiments to assess the effectiveness of FaLDAS. To carry out the experiments, a number of assumptions are considered. Section 5.3.2 describes the knowledge-base generation from simulated systems. The FPGs are assumed to have 70% FPGNodes tagged as SUSPICIOUS, a percentage identified by the Bernoulli distribution data of TCAS FPGNodes. The SUSPICIOUS tag indicates that the component (referenced in the FPGNode) is effecting the system's response by computing an output, which is a sub-computation in the overall computation of the response. More the components are doing such computations, more is the percentage of SUSPICIOUS tagged FPGNodes. It means FaLDAS has to analyze more FPGNodes to find potentially faulty candidates; decreases its time efficiency. On the other hand, lower is the percentage of SUSPICIOUS tagged FPGNodes, lesser the FPGNodes to analyze; increases the time efficiency.

In addition, 30% of the executions required to generate FaLDAS's knowledge-base, are considered to have failed exeuctions. As discussed in Section 5.5.4 that more the failed executions in sample, lesser is its time efficiency as the number of failed executions to assess components' faultiness is increased. Where the number of executions to generate the FaLDAS is less, say 10, the efficiency is likely to be effected by a small factor as compared to when more number of exeuction are used.

6.1.3 Fault Localization

A significant challenge in fault diagnosis is the time difference between a fault occurrence and the detection of its consequence as an error. If an adaptation occurs after a fault happens, but before its consequences in the form of an observable error, FaLDAS (and indeed, other approaches as far as we know) cannot localize the faulty component because of no evidence of a fault. This remains an interesting open research question for all approaches.

When the adaptation occurs in such a way that it leaves no trace of the original configuration or the context of faults that have been previously observed, it is an interesting research problem for fault localization mechanism to perform in such situations.

An execution (Definition 4.1.1) starts when a request arrives at the system. FaLDAS assumes the request arrived is correct and provides correct input to the system to start execution. When a component encounters a fault because of corrupted data in the request, FaLDAS is likely to identify the component that accepted the request as faulty one. In such a case, it is the responsibility of the programmer to assess whether that component has sufficient input validations.

It is possible that a DAS may include a number of proprietary components, whose analysis may be costly. When a component belongs to another company, then even if it is high up the ranked list, it is probable that the debugging expert may still go ahead and check all the components lower in the ranking because there may be a charge to go to the external provider. More the components to analyze for RCA, more is the Wasted Effort. In such situations, the Wasted Effort is likely reach 1.

It is possible that the corrupted values of more than one variable may have collectively caused an execution to fail. For example, a boolean computation (*a or b*) produces undesirable results when one or both the variables are corrupted. Where multiple corrupted variables cause a failed execution, FaLDAS identifies candidate sets containing multiple candidates. Such a candidate set shows that all or a sub-set of the candidates are collectively responsible for a failed execution. However, FaLDAS cannot pinpoint which actual faulty candidates, among potentially faulty candidates, causing the execution to fail. For example, components c_1 and c_3 are collectively causing a failed execution for which FaLDAS identified a candidate set as $\{c_1, c_2, c_3, c_4\}$. FaLDAS can indicate that this set contains faulty components but cannot pinpoint the exact faulty components i.e., c_1 and c_3; a limitation of FaLDAS. It is recommended that a debugging expert run root cause analysis on all the candidates in a candidate set to find multiple root causes. However, it is likely to increase the Wasted Effort. To deal with such case, a direction of future research is described in Section 6.2.6.

6.1.4 Multi-tenant Systems

It is likely that a component may participate in more than one system at a given point in time, as shown in Figure 6.1. This component may not be faulty for all the systems, in which it participates. This is a situation where FaLDAS is more useful than other related approaches. Let us assume that the component's output *b* is corrupted which causes a fault in system Y, whereas *a* and *c* are correct output. Effectively, the component is faulty in system Y but not for X and Z. FalDAS specifically identifies that the component is faulty for output variable *b* but not faulty for outputs *a* and *c*. This can not be identified by other existing approaches. For example, assume that BARINEL found the component as faulty. It informs the debugging experts to analyze the whole component which is a cumbersome task. On the other hand, FaLDAS specifically finds out which output variable is corrupted. It helps debugging experts to analyze only a part of the component, not the whole, thereby reducing root cause analysis efforts.

FaLDAS appears to be more suitable where the components interact with each other through a centralized message bus such as ESB. The TRANSFoRm system experiments validate our intution that FaLDAS is suitable for ESB-based DAS. In such a geographically distributed system as TRANSFoRm, fault localization is non-trivial

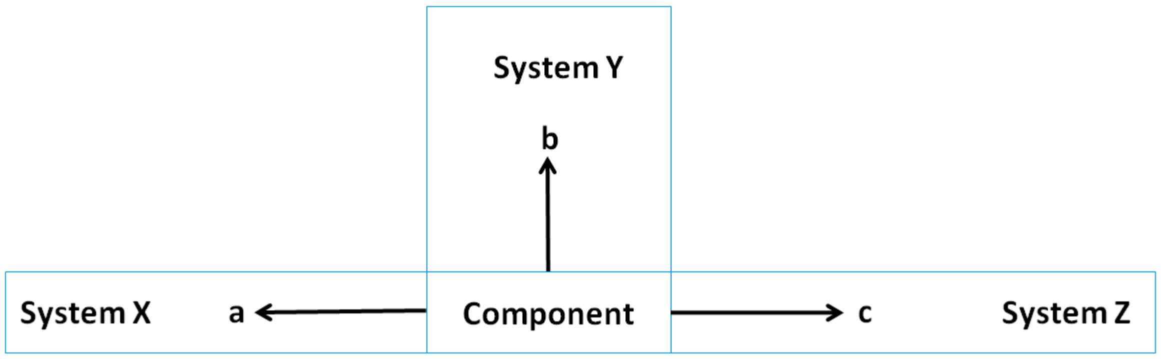


Figure 6.1: A component participating in more than one system.

because collecting data from such components is a complicated and challenging task. Sometimes, the required data from a number of components may not be available because of component's inaccessibility. In such cases, message bus-based communication allows FaLDAS to intercept inputs and outputs to generate its knowledge-base. Moreover, it finds the actual faulty component and corrupted data variable for propagated faults that emerged, without accessing the components.

6.2 Future Work

The discussion section provides useful insights of the approach. These insights helped to find the possible limitations, and provided directions for future work to enhance FaLDAS. This section discusses the possible future works in FaLDAS.

6.2.1 Inaccurate Information

Knowledge-base in an important building block for any fault localization. However, where the monitored information is inaccurate, incorrect knowledge-base generation cannot be prevented. FaLDAS is negatively effected with incorrect knowledge-base. For example, an actual faulty FPGNode's tag CLEAR (which should be SUSPICIOUS) prevents FaLDAS to consider the FPGNode as a faulty one as described in Section 5.7. Inaccuracies in the monitored information is an open research question. Future work aims to enhance FaLDAS tolerance to incorrect knowledge-base.

6.2.2 Accurate FPGNodes Isolation

A fault propagates to a component (local component) where it manifests as an observable error (or a symptom) and causes the failure of an execution. Where the symptoms indicate the corrupted variable, a more precise set of FPGNodes could be analyzed which overall reduces the Wasted Effort.

A component's corrupted output variable is either the input variable, or derived from the input variable, or an independent variable. This information can be determined using a taint analysis mechanism (Appendix A). When a component's corrupted output variable is derived from an input variable, the FPGNodes corresponding to the components which computed (directly or indirectly) the previously discussed input variable, should be included in fault localization. As an example, Figure 6.2b illustrates that variable V4 is composed using variables V3 and (indirectly) V2. When the V4 is corrupted, V3 or V2 could be potentially responsible. Thus, corresponding FPGNodes 2, 3 and 4 should be analyzed (see Figure 6.2b), which is not the case in current FaLDAS. FaLDAS includes FPGNodes 1, 2, 3 and 4 in the fault localization. This example shows that a precise set of FPGNodes can reduce the search space from 4 FPGNodes to 3 FPGNodes. However, when the corrupted variable is independent of any input variable, the local component has independently computed the corrupted variable. Thus, the local component is the faulty component. A future work is to find the variable's dependencies (Figure 6.2b) and use it in FaLDAS to find a more precise candidate sets.

6.2.3 FPG Update

One of the challenges in fault diagnosis is to handle the space and time difference between the occurrence of a fault and the detection of its consequence in the form of an error. Added to this difference, there is also the timing required for the construction of knowledge-base and the analysis to localize the fault. Therefore, during the diagnosis, the system can undergo to an adaptation that does not leave any traces (or evidence) of the fault that occurred. In such cases, the fault remains unresolved because of unavailability of evidence. Thus, the knowledge-base construction and fault localization must be as efficient as possible to complete fault analysis before next adaptation occurs. In future, FaLDAS's knowledge-base construction time can be enhanced such that a

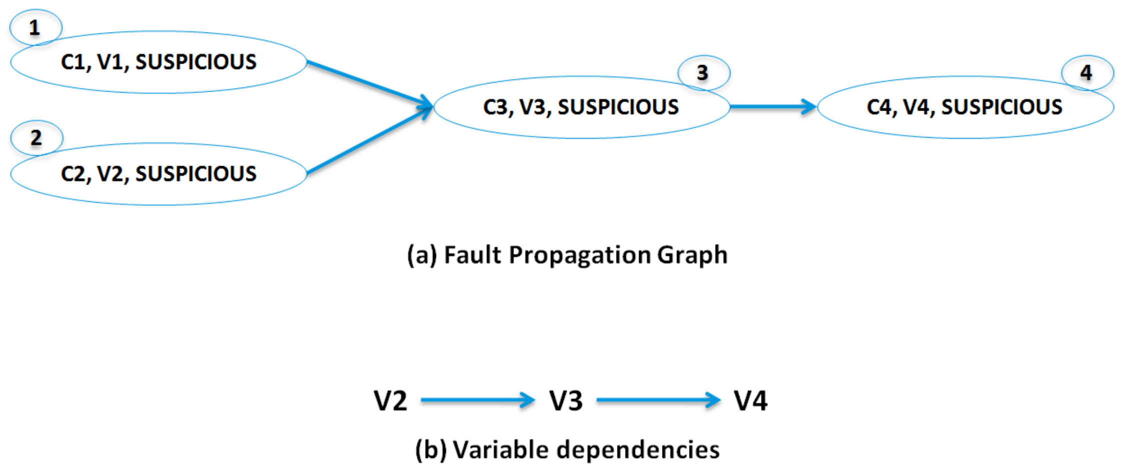


Figure 6.2: An example of FPG with variable dependencies information

new FPG can be constructed from the structural changes that occurred due to an adaptation. Figure 6.3 shows two FPGs, which differ by only one FPGNode i.e., FPGNode labelled as 4. Currently, FaLDAS constructs the whole new FPG after adaptation, however, it should only replace FPGNode 2 by FPGNode 4. Knowledge of structural changes will allow FaLDAS to analyze adaptation changes to update an existing FPG. A number of existing mechanisms [36] [7] exist to find the structural changes, required to updated the FPG.

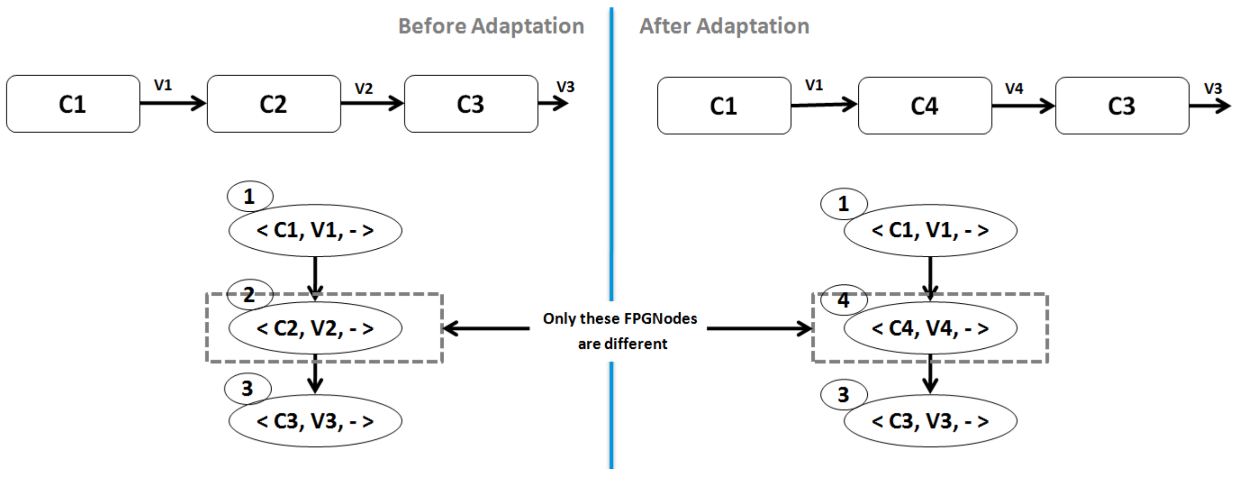


Figure 6.3: An example of difference between FPGs before and after an adaptation

6.2.4 Failure Classification

This research finds the faulty candidates which corrupt their output variables and propagate it to another component to cause a propagated fault. Therefore, this thesis is limited to the cases where a fault occurs due to corrupted data variable only. However, several other faults exist such as implementation faults, integration faults, and architectural faults [13]. An important future work would be to enable FaLDAS to find the faulty components when such faults cause a failed execution.

6.2.5 Distinguish Propagated Faults

It is possible that a fault in a local component (defined in Section 6.2.2) may not have propagated from other components. Instead, it may have generated within itself. Even in such cases, FaLDAS fault localization assume the root cause in a different component and correspondingly find the faulty candidate sets. To distinguish the cases where FaLDAS is suitable, it is necessary to identify if a fault's root cause exist within the local component or in an another component. A future work is to develop a mechanism which assess whether a fault's root cause exist in an external component i.e., the fault is a propagated fault.

6.2.6 Multiple Root Causes

A debugging expert, unaware of multiple root causes collectively responsible for a failed execution, may not find all the root causes. For example, where c_1 and c_3 are faulty, a debugging expert after finding c_1 in a candidate set $\{c_1, c_2, c_3, c_4\}$ may not look further candidates. The expert may assume that the actual faulty component is found, so no need to look further. As a result, other faulty component c_3 remains undiagnosed. Therefore, it is important to find how many root causes possibly causing a fault. It enables the expert to carry out root cause analysis until all faulty candidates are found. A future work is to develop a mechanism that can assess the number of possible root causes responsible for a fault.

6.3 Chapter Summary

This chapter presents a discussion about the possible cases where FaLDAS is suitable and where not, and possible future research directions to enhance FaLDAS. It discusses a number of cases for input output values and analyzes where FaLDAS is successful and failed. It was analyzed that FaLDAS successfully finds a faulty component when its output is not the same as input. Moreover, FaLDAS successfully ignores the non-faulty components when its output is the same as its input.

It was analyzed that existing approaches face challenges when an adaptation occurs before a fault's consequence is observed as an error. Moreover, adaptation may leave no traces of a fault and the original configuration of the system, which makes the fault localization a difficult task. FaLDAS's limitations were discussed such as it can not pinpoint an exact faulty component in a candidate set of possible faulty components, which is also a limitation of other related approaches. To deal with limitations, a number of future research directions are presented to enhance FaLDAS.

Future work describes six possible enhancements in FaLDAS. Some enhancements require algorithmic modifications e.g., to deal with inaccurate information, accurate FPGNodes isolation and FPG update from architectural changes of an adaptation. However, some enhancements open a possible new research problems such as enabling FaLDAS for a wide variety of faults, distinguish if a fault is a propagated fault or not, and identifying if a fault has resulted from a single root cause or multiple root causes.

Chapter 7

Conclusion

This chapter summaries the thesis in terms of its key achievements, findings and evaluation results. This chapter ends with a remark on usefulness of this work and a direction for possible related research.

Introduction Chapter 1 motivated this work and argued that faults in software systems present substantial challenges to computing society, and finding offending components responsible for faults, in particular propagated faults, is important. Fault localization is a well studied area, still it faces several challenges in adaptive systems such as diagnosing inaccessible components and coping with inaccurate monitored information about system. The chapter hypothesized that using the components' real-time input output values, as the knowledge-base, could enable an efficient diagnosis of inaccessible components and tolerance to inaccurate information.

Related Work Chapter 2 reviewed the fault localization domain in detail. Mechanisms based on graphs, bayesian networks, program slices, ontologies, statistical equations and industrial practices, were examined. It examined the data used as knowledge-base in existing mechanisms, and how these mechanisms find the potential faulty components. Moreover, their capabilities to work in adaptive systems, was analyzed. The analysis highlighted the gap for fault localization of propagated faults in adaptive system as existing mechanisms assume the target systems − static.

System Model and Fault Management Scope Chapter 3 first describes the terminologies and scope of this thesis. In addition, the problem was described and analyzed using a running example. In addition, existing mechanisms' capabilities were assessed to solve the problem. Some mechanisms found useful to an extent in adaptive system, which are compared with the thesis's work. To devise a solution, a set of design objectives were derived and argued that a desired fault localization mechanism (i) must gather required input data at real-time, (ii) must synchronize the knowledge-base, (iii) must be able to diagnose inaccessible components, (iv) cope with inaccurate information, (v) output an ordered list of faulty candidates and (vi) must be able to integrate with a root cause analysis mechanism. The discussion of design objectives then led to the design decisions that forms the fundamentals of thesis contribution.

FaLDAS Fault Localization Chapter 4 presented FaLDAS, Fault Localization in Distributed Adaptive Systems. FaLDAS uses components' names and real-time input output values to construct its knowledge-base in the form of Fault Propagation Graphs (FPGs). This chapter discussed FaLDAS's algorithms to construct its knowledge-base and fault localization algorithm that reasons over FPGs to produce a sorted list of potential faulty candidates for a fault. In addition, integration of FaLDAS with an existing root cause analysis mechanism is described.

Evaluation Chapter 5 evaluated how well FaLDAS achieved the overall goal of efficient diagnosis of inaccessible faulty components and tolerance to inaccurate information. Simulated systems, Traffic Collision Avoidance System (TCAS) and European FP7 research project TRANSFoRm was used for experiments. BARINEL [4], Pinpoint [27] and Bellur et al.'s [17] approaches were used for comparison. The results showed that FaLDAS's efficiency was the best. In addition, FaLDAS is the best for fault localization for adaptive systems. Also, FaLDAS's results quality was best to diagnose inaccessible components. FaLDAS did not perform better than other when the required input information was inaccurate. However, it shows better support for further root cause analysis by indicating the corrupted output variable of a faulty component, which enables a debugging expert to analyze a part of the component.

Final Remark Fault Localization in Distributed Adaptive Systems (FaLDAS) mechanism is suitable when a fault is propagated across different components and manifests itself as an apparently unrelated fault causing an observable error (subsequently a failure) in a component other than the one where the fault originated. The results showed that FaLDAS performs most efficiently, compared to considered approaches, to cope with adaptive system that adapts often. Moreover, the results showed that FaLDAS is suitable to diagnose faulty components whose internal design (e.g., source code, test cases, logs) is inaccessible to fault localization mechanisms. It is hoped that findings of this thesis offer a new advancement for fault diagnosis in adaptive systems and encourage further research in emerging areas such as smart cities.

Appendix A

Taint Analysis

Taint Analysis is the process of identifying if a particular variable has been composed of other variables or not. Moreover, which variables have been used to compose a variable. A number of research and tools exist to facilitate the Taint analysis [29] [117] [34]. The Taint analysis is generally used in the field of application security. The purpose of Taint analysis is to analyze the explicit or implicit involvement of an input variable in the computation of another variable. For example, in Figure A.1a, assuming that the variables `a` and `b` are tainted at line 1 and 2 with taint marking $\{t_a\}$ and $\{t_b\}$, respectively. Taint analysis propagates the marking and inform the marking of `y` and `z` as $\{t_a\}$ and $\{t_a, t_b\}$, respectively. The output variable `x` remains untainted. Using this marking information, it can be inferred that the output variables `x`, `b` and `z` is NEW, SAME or COMPOSED, respectively as described in Figure A.1b.

```
1. a = 10
2. b = 10
3. Procedure (a, b)
4. w = a * 10
5. x = 100
6. y = w + 50
7. z = b + y
8. return x, b, z
```

(a)

InputVariable	OutputVariable	Status
-	x	NEW
b	b	SAME
a, b	Z	COMPOSED

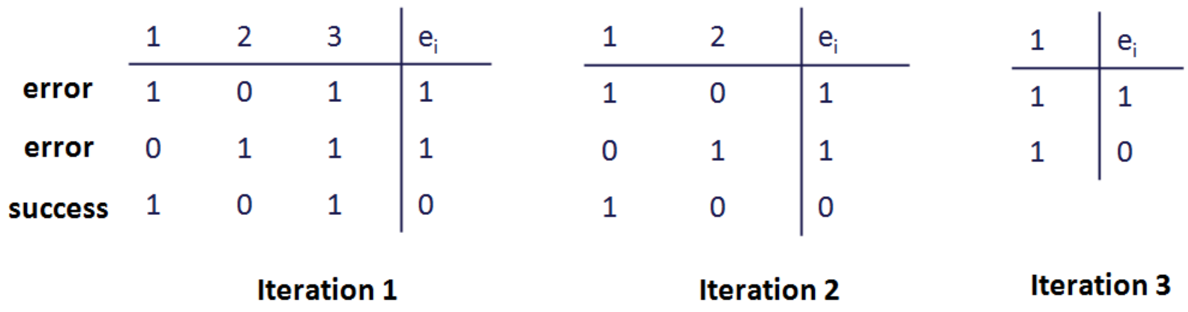
(b)

Figure A.1: Taint Analysis: (a) Sample code for Taint analysis from [29], (b) Variables composition

Appendix B

BARINEL Methodology

BARINEL algorithm is applied on a hit spectra. A step by step BARNIEL's execution using a sample hit spectra from their work [2] is shown in Figure B.1. Here, component 3 is used in all execution, so it is assumed as a faulty candidate $d_1 = \{3\}$. Component 3's information is removed from spectra to construct a new hit spectra. In the new spectra, component 2 is used in only one failed execution. Entries corresponding to component 2 and corresponding failed execution are removed. In the next round, component 1 and corresponding failed execution's information are removed. As a result, no more components left in the spectra, union of component 2 and 1 forms a faulty candidate $d_2 = \{1, 2\}$.



	1	2	3	e_i
error	1	0	1	1
error	0	1	1	1
success	1	0	1	0

Iteration 1

1	2	e_i
1	0	1
0	1	1
1	0	0

Iteration 2

1	e_i
1	1
1	0

Iteration 3

Figure B.1: BARINEL's step by step execution.

The likelihood of a faulty candidate is calculated through posterior probability as shown in Table B.1. In the given example, the probability of error for candidate set d_2 is calculated. Since, d_2 has components 1 and 2, the probability is calculated using their health i.e., h_1 and h_2 as described in the last row of Table B.1.

$\Pr(e_1\|d_2)$	$=$	$1 - h_1$
$\Pr(e_2\|d_2)$	$=$	$1 - h_2$
$\Pr(e_3\|d_2)$	$=$	h_1
$\Pr(e\|d_2)$	$=$	$(1 - h_1) \cdot 1 - h_2 \cdot h_1$

Table B.1: Probability of error for faulty candidate d_2.

Appendix C

Pinpoint Methodology

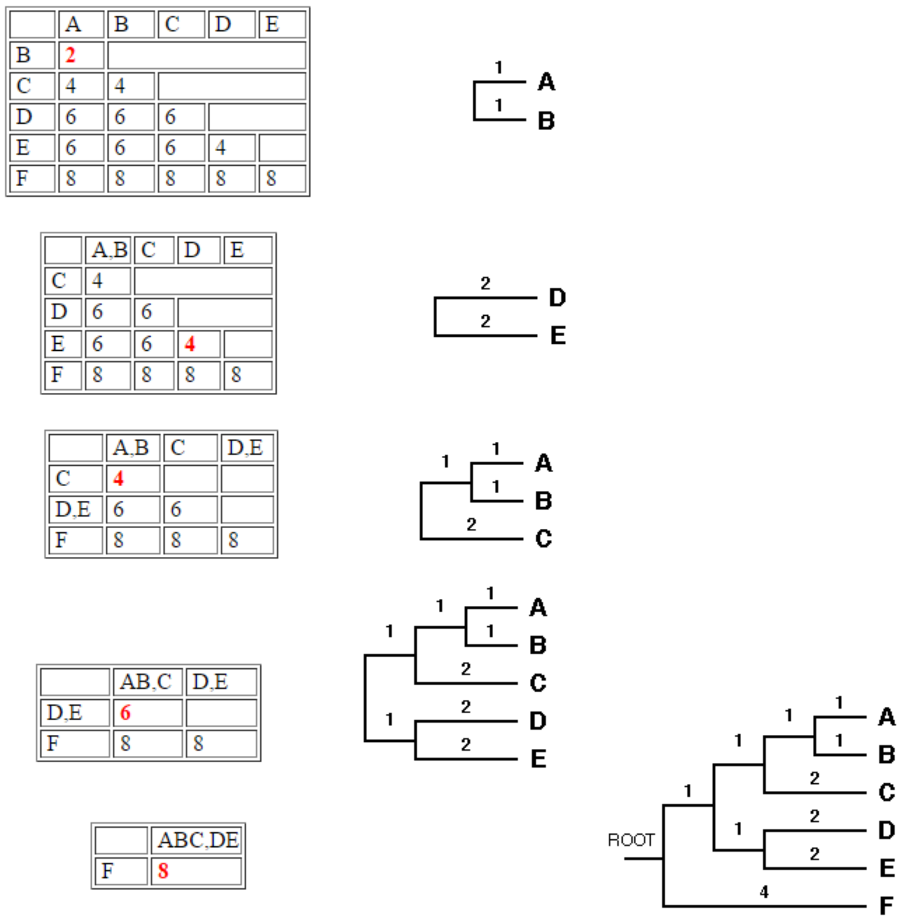
Pinpoint [27] collects individual component's failure (1 - failure and 0 - success) information in a pass/fail spectra. A sample pass/fail spectra is shown shown in Table C.1. This data is fed to Unweighted Pair-Group Method using Arithmetic Averages (UPGMA) algorithm which groups the data based on similarity. UPGMA uses a clustering algorithm, in which similarity between entities (represented as distance matrix) are measured and phylogenetic tree [39] is build in a stepwise manner. The time complexity of UPGMA algorihtm is $O(n^2)$ where n is the number of components in the system [91].

Error E	Component A	Component B	Component C	Component D
0	1	0	0	1
1	1	1	0	0
1	0	1	0	1
0	0	0	1	0

Table C.1: Sample pass/fail spectra

Pinpoint uses the pass/fail spectra to calculate a distance matrix and fed to UPGMA. In UPGMA, initially, two entities are identified which are most similar and are clustered together to form a new entity as shown in Figure C.1. In the first cycle, A and B are clustered because distance between them is the least. Using this new entity, the similarities are re-calculated and clustering re-occurs. In second cycle, the distance matrix is re-calculated where D and E are clustered because distance between them is least. The cycle continues until two elements are left and then the tree is completed.

The final tree shows that Component D behaviour is the most similar to Error E's behaviour.



	A	B	C	D	E
B	2				
C	4	4			
D	6	6	6		
E	6	6	6	4	
F	8	8	8	8	8

	A,B	C	D	E
C	4			
D	6	6		
E	6	6	4	
F	8	8	8	8

	A,B	C	D,E
C	4		
D,E	6	6	
F	8	8	8

	AB,C	D,E
D,E	6	
F	8	8

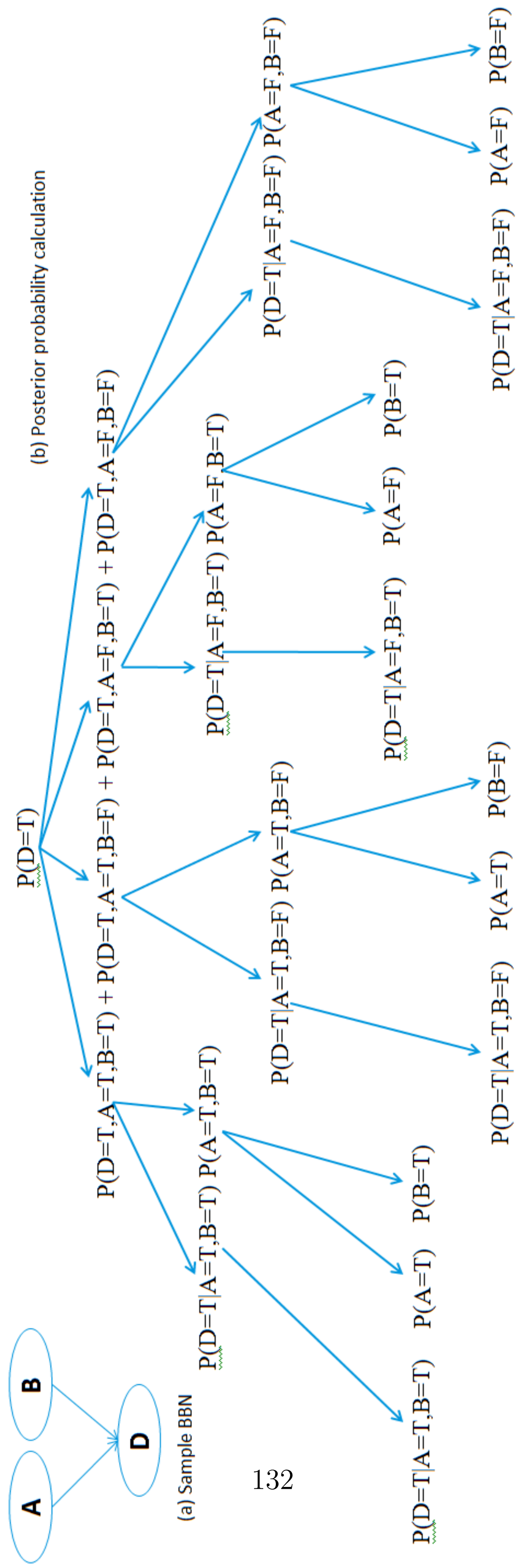
	ABC,DE
F	8

Figure C.1: Step by step exeuction of UPGMA [97].

Appendix D

Bayesian Belief Networks Methodology

Bellur et al. [17] state that the faulty candidates are sorted according to their posterior probability in BBN. The posterior probability function complexity is directly proportional to the number of predecessors. When a node has a large number of predecessors, the more number of probabilities need to be calculated. In a BBN having n nodes where each node may have a maximum of k parents, the total number of probabilities required is $\mathrm{O}\big(n \cdot 2^k\big)$. Moreover, k is likely is increase with the increase in system's scale because larger the scale more the components are likely to be inter-connected. The posterior probability calculation time is exponential. For example, let us assume a BBN as shown in Figure D.1. BBN is a graphical model which represents a set of random variables connected with edges. According to Bellur et al. failure event i.e., the component that encountered a visible fault is fed to the BBN as evidence. Inferencing algorithms propagate this evidence to other nodes in the network. It involves re-calculation of posterior probability of each effected node which is exponential in time.



A
B
D
(a) Sample BBN
(b) Posterior probability calculation
P(D=T)
P(D=T,A=T,B=T) + P(D=T,A=T,B=F) + P(D=T,A=F,B=T) + P(D=T,A=F,B=F)
P(D=T|A=T,B=T) P(A=T,B=T)
P(D=T|A=T,B=F) P(A=T,B=F)
P(D=T|A=F,B=T) P(A=F,B=T)
P(D=T|A=F,B=F) P(A=F,B=F)
P(D=T|A=T,B=T)
P(A=T)
P(B=T)
P(D=T|A=T,B=F)
P(A=T)
P(B=F)
P(D=T|A=F,B=T)
P(A=F)
P(B=T)
P(D=T|A=F,B=F)
P(A=F)
P(B=F)

Figure D.1: A sample BBN

Bibliography

[1] A xml-based workflow event logging mechanism for workflow mining. In H. Shen, J. Li, M. Li, J. Ni, and W. Wang, editors, *Advanced Web and Network Technologies, and Applications*, volume 3842 of *Lecture Notes in Computer Science*, pages 132–136. Springer Berlin Heidelberg, 2006.

[2] R. Abreu and A. J. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. *Abstraction, Reformulation, and Approximation (SARA)*, 2009.

[3] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[4] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[5] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009.

[6] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 13:1–13:2. ACM, 2012.

[7] A. Ahmad, P. Jamshidi, M. Arshad, and C. Pahl. Graph-based implicit knowledge discovery from architecture change logs. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 116–123, New York, NY, USA, 2012. ACM.

[8] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM.

[9] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 520–523, Nov 2011.

[10] J. Andrews and G. Brennan. Application of the digraph method of fault tree construction to a complex control configuration. *Reliability Engineering & System Safety*, 28(3):357 – 384, 1990.

[11] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles. Dependability of cots microkernel-based systems. *IEEE Trans. Comput.*, 51(2):138–163, Feb. 2002.

[12] P. Arpaia, M. L. Bernardi, G. Di Lucca, V. Inglese, and G. Spiezia. An aspect-oriented programming-based approach to software development for fault detection in measurement systems. *Comput. Stand. Interfaces*, 32(3):141–152, 2010.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[14] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 189–200, New York, NY, USA, 2008. ACM.

[15] L. Baresi and S. Guinea. Self-supervising bpel processes. *Software Engineering, IEEE Transactions on*, 37(2):247–263, March 2011.

[16] B. Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[17] U. Bellur and A. Agrawal. Root cause isolation for self healing in j2ee environments. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '07, pages 324–327, Washington, DC, USA, 2007. IEEE Computer Society.

[18] S. Benmoussa, B. Bouamama, and R. Merzouki. Bond graph approach for plant fault detection and isolation: Application to intelligent autonomous vehicle. *Automation Science and Engineering, IEEE Transactions on*, 11(2):585–593, April 2014.

[19] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano. Effective fault treatment for improving the dependability of cots and legacy-based applications. *IEEE Trans. Dependable Secur. Comput.*, 1(4):223–237, Oct. 2004.

[20] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proceedings of the The Third IEEE Workshop on Internet Applications*, WIAPP '03, pages 132–, Washington, DC, USA, 2003. IEEE Computer Society.

[21] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(2):175–190, Apr. 2006.

[22] N. Cardoso. Mhs2. https://github.com/npcardoso/MHS2, 2014.

[23] P. Casanova, D. Garlan, B. Schmerl, and R. Abreu. Diagnosing unobserved components in self-adaptive systems. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 75–84, New York, NY, USA, 2014. ACM.

[24] P. Casanova, B. Schmerl, D. Garlan, and R. Abreu. Architecture-based runtime fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 261–277, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] A. Cavalcante and M. Grajzer. Fault propagation model for ad hoc networks. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1–5, June 2011.

[26] R. N. Charette. This Car Runs on Code. *IEEE Spectrum*, Feb. 2009.

[27] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Y.-H. Chiang, M. Keller, R. Lim, P. Huang, and J. Beutel. Poster abstract: Light-weight network health monitoring. In *Information Processing in Sensor Networks (IPSN), 2012 ACM/IEEE 11th International Conference on*, pages 109–110, April 2012.

[29] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[30] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks (research note). *Artif. Intell.*, 42(2-3):393–405, Mar. 1990.

[31] A. Curry, P. Flett, and I. Hollingsworth. *Managing Information & Systems: The Business Perspective.* Taylor & Francis, 2006.

[32] S. De Chaves, R. Uriarte, and C. Westphall. Toward an architecture for monitoring private clouds. *Communications Magazine, IEEE*, 49(12):130–137, December 2011.

[33] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[34] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proceedings of the 12th International Conference on Tools*

and Algorithms for the Construction and Analysis of Systems, TACAS'06, pages 73–89. Springer-Verlag, Berlin, Heidelberg, 2006.

[35] J. Ehlers and W. Hasselbring. A self-adaptive monitoring framework for component-based software systems. In I. Crnkovic, V. Gruhn, and M. Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 278–286. Springer Berlin Heidelberg, 2011.

[36] C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proceedings of Conference on Organizational Computing Systems*, COCS '95, pages 10–21, New York, NY, USA, 1995. ACM.

[37] TRANSFoRm Project. http://www.transformproject.eu/, 2013.

[38] L. Fiondella, S. Gokhale, and V. Mendiratta. Cloud incident data: An empirical analysis. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 241–249, March 2013.

[39] W. M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155(3760):279–284, 1967.

[40] G. Fuxiang, W. Yanyan, S. Wenjun, and Y. Lan. Design and implementation of the web access monitoring system based on url analysis. In *Information Technology and Applications (IFITA), 2010 International Forum on*, volume 1, pages 425–428, July 2010.

[41] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 15:1–15:12, New York, NY, USA, 2007. ACM.

[42] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[43] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 79–88, June 2004.

[44] P. Goodliffe. *Becoming a Better Programmer: A Handbook for People Who Care About Code.* O'Reilly Media, 2014.

[45] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 40–49, New York, NY, USA, 2012. ACM.

[46] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, June 2006.

[47] A. Gupta and A. Raj. Strengthening method contracts for objects. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 233–242, Dec 2006.

[48] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.

[49] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *SIGPLAN Not.*, 33(7):83–90, 1998.

[50] B. Hofer, F. Wotawa, and R. Abreu. Ai for the win: Improving spectrum-based fault localization. *SIGSOFT Softw. Eng. Notes*, 37(6):1–8, 2012.

[51] C. Ibsen and J. Anstey. *Camel in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[52] S. Ilarri, E. Mena, and A. Illarramendi. Using cooperative mobile agents to monitor distributed and dynamic environments. *Inf. Sci.*, 178(9):2105–2127, May 2008.

[53] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[54] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[55] JavaBayes. Carnegie Mellon University, http://www.cs.cmu.edu/ javabayes/Home/, 2001.

[56] S. Jiang, W. Li, H. Li, Y. Zhang, H. Zhang, and Y. Liu. Fault localization for null pointer exception based on stack trace and program slicing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 9–12, Aug 2012.

[57] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe. An uncaught exception analysis for java. *Journal of Systems and Software*, 72(1):59 – 69, 2004.

[58] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[59] K. Joshi, M. Hiltunen, W. Sanders, and R. Schlichting. Probabilistic model-driven recovery in distributed systems. *Dependable and Secure Computing, IEEE Transactions on*, 8(6):913–928, Nov 2011.

[60] M. Kalinowski, H. Teixeira, and P. van Oppen. Abat: An approach for building maintainable automated functional software tests. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the*, pages 83–91, Nov 2007.

[61] G. Kalyon, T. Le Gall, H. Marchand, and T. Massart. Symbolic supervisory control of distributed systems with communications. *Automatic Control, IEEE Transactions on*, 59(2):396–408, Feb 2014.

[62] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[63] M. Kim and M. Kim. Member failure recovery mechanism using context ontology for community computing. In *Multimedia and Ubiquitous Engineering, 2007. MUE '07. International Conference on*, pages 831–836, April 2007.

[64] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *Cluster, Cloud*

and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, pages 398–407, May 2010.

[65] V. Kozaczynski and J. Q. Ning. Component-based software engineering (cbse). In *Proceedings of the 4th International Conference on Software Reuse*, ICSR '96, Washington, DC, USA, 1996. IEEE Computer Society.

[66] B. Krebs. Cyber incident blamed for nuclear power plant shutdown. In *Washington Ppost*, Jun 2008.

[67] J. K. Kuchar and A. C. Drumm. The traffic alert and collision avoidance system (tcas). *MIT Lincoln Laboratory Journal*, 16(2), 2007.

[68] J. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[69] W. Le and M. L. Soffa. Path-based fault correlations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 307–316, New York, NY, USA, 2010. ACM.

[70] W. Le and M. L. Soffa. Marple: Detecting faults in path segments using automatically generated analyses. *ACM Trans. Softw. Eng. Methodol.*, 22(3):18:1–18:38, July 2013.

[71] K. Lee, C. Lim, K. Kong, and H.-N. Kim. A design and implementation of a remote debugging environment for embedded internet software. In J. Davidson and S. Min, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1985 of *Lecture Notes in Computer Science*, pages 199–203. Springer Berlin Heidelberg, 2001.

[72] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, and C. Stratan. Monalisa: An agent based, dynamic service system to monitor, control and optimize distributed systems. *Computer Physics Communications*, 180(12):2472 – 2498, 2009. 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.

[73] Y. Lei, X. Mao, and T. Y. Chen. Backward-slice-based statistical fault localization without test oracles. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 212–221, July 2013.

[74] Y. Lei, X. Mao, Z. Dai, and C. Wang. Effective statistical fault localization using program slices. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 1–10, July 2012.

[75] N. Leveson and C. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[76] B. Li. Managing dependencies in component-based systems based on matrix model. In *Proc. Of Net.Object.Days 2003*, pages 22–25, 2003.

[77] J. Li, S. He, L. Zhu, X. Xu, M. Fu, L. Bass, A. Liu, and A. B. Tran. Challenges to error diagnosis in hadoop ecosystems. In *Proceedings of the 27th International Conference on Large Installation System Administration*, LISA'13, pages 145–154, Berkeley, CA, USA, 2013. USENIX Association.

[78] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 425–434, June 2006.

[79] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, June 2005.

[80] C.-C. Lin, C.-L. Fang, and D. Liang. A portable interceptor mechanism for soap frameworks. *Comput. Stand. Interfaces*, 36(1):209–218, 2013.

[81] Y. Liu, L. Ma, and S. Huang. Construct fault diagnosis model based on fault dependency relationship matrix. In *Proceedings of the 2009 Pacific-Asia Conference on Circuits, Communications and Systems*, PACCS '09, pages 318–321, Washington, DC, USA, 2009. IEEE Computer Society.

[82] C. H. Lo, Y. K. Wong, and A. B. Rad. Bond graph based bayesian network for fault diagnosis. *Appl. Soft Comput.*, 11(1):1208–1212, Jan. 2011.

[83] C. B. Low, D. Wang, S. Arogeti, and M. Luo. Quantitative hybrid bond graph-based fault detection and isolation. *Automation Science and Engineering, IEEE Transactions on*, 7(3):558–569, July 2010.

[84] L. Mariani, F. Pastore, and M. Pezze. Dynamic analysis for diagnosing integration faults. *Software Engineering, IEEE Transactions on*, 37(4):486–508, July 2011.

[85] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 128–137, Sept 2008.

[86] S. Merseguer, Josand Bernardi. Dependability analysis of des based on marte and uml state machines models. *Discrete Event Dynamic Systems*, 22(2):163–178, 2012.

[87] A. A. d. S. Meyer, A. A. F. Garcia, A. P. d. Souza, and C. A. L. d. Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, 27:83 – 91, 00 2004.

[88] J. Mickens. Rivet: Browser-agnostic remote debugging for web applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 30–30, Berkeley, CA, USA, 2012. USENIX Association.

[89] L. Moonen. Towards evidence-based recommendations to guide the evolution of component-based product families. *Science of Computer Programming*, 97, Part 1(0):105 – 112, 2015. Special Issue on New Ideas and Emerging Results in Understanding Software.

[90] R. L. Morgan, S. Cantor, S. Carmody, W. Hoehn, and K. Klingenstein. Federated Security: The Shibboleth Approach. *EDUCAUSE Quarterly*, 27(4):12–17, 2004.

[91] F. Murtagh. Complexities of hierarchical clustering algorithms: State of the art. *Computational Statistics Quarterly*, 1:101–113, 1984.

[92] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[93] M. Nagappan, K. Wu, and M. A. Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 41–50. IEEE Computer Society, 2009.

[94] N. Naksinehaboon, N. Taerat, C. Leangsuksun, C. Chandler, and S. Scott. Benefits of software rejuvenation on hpc systems. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 499–506, Sept 2010.

[95] Software errors cost u.s. economy $59.5 billion annually. News Release: National Institute of Standards and Technology, Department of Commerce, U.S., 2002.

[96] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, 2012.

[97] F. Opperdoes. UPGMA example, 1997. [Online; accessed 25-April-2015].

[98] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, May 1999.

[99] S. Padalkar, G. Karsai, C. Biegl, J. Sztipanovits, K. Okuda, and N. Miyasaka. Real-time fault diagnostics. *IEEE Expert*, 6(3):75–85, June 1991.

[100] J. Pearl. *Causality: Models, Reasoning, and Inference.* Cambridge University Press, 2000.

[101] R. Popescu, A. Staikopoulos, A. Brogi, P. Liu, and S. Clarke. A formalized, taxonomy-driven approach to cross-layer application adaptation. *ACM Trans. Auton. Adapt. Syst.*, 7(1):7:1–7:30, May 2012.

[102] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 49–58, New York, NY, USA, 2010. ACM.

[103] D. Recordon and D. Reed. Openid 2.0: A platform for user-centric identity management. In *Proceedings of the Second ACM Workshop on Digital Identity Management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[104] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[105] C. Reiss and J. Wilkes. Google cluster-usage traces: Format + schema. Google White Paper, 2011.

[106] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39, Oct 2003.

[107] M. Robillard. Jex - a tool for analyzing exception flow in Java programs, 2007. [Online; accessed 25-April-2015].

[108] S. Rogerson. ETHIcol - the chinook helicopter disaster. *IMIS*, 12(2), Apr 2002.

[109] L. Romano, A. Bondavalli, S. Chiaradonna, and D. Cotroneo. Implementation of threshold-based diagnostic mechanisms for cots-based applications. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 296–303, 2002.

[110] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. *SIGPLAN Not.*, 48(4):139–152, Mar. 2013.

[111] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, 2009.

[112] R. Santelices, J. Jones, Y. Yu, and M. Harrold. Lightweight fault-localization using multiple coverage types. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 56–66, May 2009.

[113] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google Technical Report*, April 2010.

[114] S. Singh and L. Singh. Study of current program slicing techniques. In *Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference -*, pages 810–814, Sept 2014.

[115] S. Slade. Case-based reasoning: A research paradigm. *AI MAGAZINE*, 12(1):42–55, 1991.

[116] M. I. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165 – 194, 2004.

[117] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, 2009.

[118] UML 2.0 Infrastructure Specification. Object Management Group, www.omg.org, 2013.

[119] UPGMA implementation in Java. Google Source Code, https://code.google.com/p/upgma/, 2013.

[120] T. Vogel and H. Giese. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 39–48, New York, NY, USA, 2010. ACM.

[121] The world wide web consortium (w3c). http://www.w3.org/, 2003.

[122] L. Wang, J. Shi, and X. Lin. An extend dependency matrix generation method using structure information. In *Prognostics and System Health Management (PHM), 2012 IEEE Conference on*, pages 1–6, May 2012.

[123] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 257–268, Nov 2009.

[124] W. Wen. Software fault localization based on program slicing spectrum. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1511–1514, June 2012.

[125] J. White, B. Dougherty, R. Schantz, D. Schmidt, A. Porter, and A. Corsaro. R&d challenges and solutions for highly complex distributed systems: a middleware perspective. *Journal of Internet Services and Applications*, 3(1):5–13, 2012.

[126] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 314–323, New York, NY, USA, 2000. ACM.

[127] Wikipedia. Ipo model — wikipedia, the free encyclopedia, 2015. [Online; accessed 25-April-2015].

[128] W. E. Wong and V. Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, The University of Texas at Dallas, November 2009.

[129] W. E. Wong and Y. Qi. Effective program debugging based on execution slices and inter-block data dependency. *J. Syst. Softw.*, 79(7):891–903, July 2006.

[130] F. Wotawa, M. Stumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In *Developments in Applied Artificial Intelligence*, volume 2358 of *Lecture Notes in Computer Science*, pages 746–757. Springer Berlin Heidelberg, 2002.

[131] F. wu Wang, J. you Shi, and L. Wang. Method of diagnostic tree design for system-level faults based on dependency matrix and fault tree. In *Industrial Engineering and Engineering Management (IE EM), 2011 IEEE 18Th International Conference on*, volume Part 2, pages 1113–1117, Sept 2011.

[132] J. Xu, W. K. Chan, Z. Zhang, T. H. Tse, and S. Li. A dynamic fault localization technique with noise reduction for java programs. In *Proceedings of the 2011 11th International Conference on Quality Software*, QSIC '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[133] F. Yang, S. Shah, and D. Xiao. Signed directed graph based modeling and its validation from process knowledge and process data. *International Journal of Applied Mathematics and Computer Science*, 22(1):41–53, 2012.

[134] A. Yazdanshenas and L. Moonen. Tracking and visualizing information flow in component-based systems. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 143–152, June 2012.

[135] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *Communications Magazine, IEEE*, 34(5):82–90, 1996.

[136] C. Yilmaz and C. Williams. An automated model-based debugging approach. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 174–183, New York, NY, USA, 2007. ACM.

[137] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19:1–19:29, July 2013.

[138] Y.-S. You, C.-Y. Huang, K.-L. Peng, and C.-J. Hsu. Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 180–189, July 2013.

[139] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *in 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 293–306, 2012.

[140] J. Zelle. *Python Programming: An Introduction to Computer Science 2Nd Edition*. Franklin, Beedle & Associates Inc., Wilsonville, OR, USA, 2010.

[141] A. Zeller. Comment in an open discussion at the 2nd international workshop on the future of debugging, 2013.

[142] C. Zhou, X. Huang, X. Naixue, Y. Qin, and S. Huang. A class of general transient faults propagation analysis for networked control systems. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, PP(99):1–1, 2015.