Review of Fault Detection Based on Determining Software Inter-Dependency Patterns for Integration Testing Using Machine Learning on Logs Data

Shashikant Ishvarbhai Patel[1*] and Rakesh Kumar Bhujade[2]

[1] Research Scholar Gujarat Technological University, India
[2] Government Polytechnic, Daman Affiliated to GTU, India
*shashi.patel777@gmail.com

Abstract. This study reviews a machine learning model applied to log files that creates entity and relationship patterns to be used in fault detection. Fault detection is an important part of integration testing when software project development is ongoing, and it could be integrated into a continuous testing approach. As the project grows, new features and code are added, and the code becomes more complex. The use of test logs enables the detection of patterns before they become deeply embedded in the code, which might make them difficult to comprehend and understand relationships and entity components. The model presented in this work is helpful in identifying which parts of the code are frequently changed, providing useful information for test case creation. Additionally, the entity relationship models are automatically created, and they may provide relevant patterns to create new tests to avoid or identify new faults in established relationships if they have never been tested.

Keywords: Fault Detection, Software Testing Inter-Dependency Patterns, Integration Testing, Logs Data.

1 Introduction

The design of high-quality large systems includes the verification and validation of the assumptions made when software parts interact with each other before actual deployment. Early detection of software-related issues related to this interaction could significantly reduce the costs and time taken during integration testing, which is expensive. Usually, in practical software projects, multiple deviations are discovered at runtime or are left undetected. Both situations should be treated as the expense of system safety measures. [1] Good design at the beginning of software development helps to reduce the risk and expense of detecting all the possible issues after the implementation and introduction of sophisticated error-preventing mechanisms into the testing regime [2]. This paper provides two contributions: the historical transition of adaptive software based on machine learning from analytical processing of large volumes of software self-monitoring logs data, and a strong, layered, scalable, quick access method of extracting decision information to validate software interaction at runtime. The access method is supported by the use of a medium that ensures the

S. Bhalerao et al. (eds.), *Proceedings of the International Conference on Recent Advancement and Modernization in Sustainable Intelligent Technologies & Applications (RAMSITA-2025),* Advances in Intelligent Systems Research 192,
https://doi.org/10.2991/978-94-6463-716-8_86

availability of runtime interaction logs data. Its model is weighed for industrial datasets that resemble future integration testing scenarios.

1.1 Background and Motivation

Integration testing is the process in the software life cycle where error detection is the main responsibility. A typical modern software system consists of a very large number of executable modules called software components. We assume that each of these components has been previously tested at some level of the testing hierarchy. However, these testing levels, no matter how detailed they are, are still not capable of finding the Achilles' heel of every integration point of these components. [3][4]Because these components interact with so many others, it would be impossible to exhaustively test every possible interaction of every subset of them. This is the problem in integration testing that motivates us: it is impossible to enumerate all possible usages of the software at this phase of development and then generate test cases for each usage. Objective: The aim of this work is to come up with a detection of system failure from using data gathered during the operation of the system. The system in our work is a narrowly defined subset of a software system, a type of software that is very popular and has a long record of enormous production and practical use, usually with large-scale usage. This type is called event-driven systems, and the type we are most interested in this work is the Web-driven systems, also known as Web-centered. That is, systems whose implementation structure can be deduced from web log data.

1.2 Research Objectives

The main guidelines in research as solution strategies are presented in all phases that are directed to the main subject following described objectives. Developing automatic efforts to reduce the search space aimed at increasing the efficiency of the integration process of software components, where the learning approach is able to identify frequently recorded sequences. Investigating the occurrence of less observed software codes during software integration that are usually caused by latent behavior related to poor message scheduling management and design, and the occurrence of unexpected outcomes that may go through different causes such as coding faults, late occurrences of input errors, and little insertion of components in the system subjected to testing. Designing a machine learning process using classification algorithms to work on the log data prepared for machine learning to detect faults, low volatile codes, and unexpected results in integration components, focusing on enhancing the integration space [5].

2 Software Inter-Dependency Patterns in Integration Testing

To detect faults in integration testing from log data, it is important to leverage the states and temporal properties of the processes that the logs imply. Some inter-dependency patterns need to be recognized and modeled in log data, which are fundamental for Windows operating systems as an example. In the case study implementation, log data of processes, states of processes at intervals, and start and end timestamps of processes created in Windows operating systems for integration testing were converted into a format that is suitable for algorithms in a machine learning library, using scripts. The rephrased log data were then trained and tested with different machine learning models to detect anomalous and faulty states [6][7].

Results from the case study were presented, along with how different objectives and factors, such as selectively desirable malfunctioning rates, would affect and influence the different model performances discussed. Such findings can aid automated integration testing in practice, in fine-tuning the settings and parameters employed in the library, to more accurately locate and identify where faults were induced in an application under test.

2.1 Definition and Importance

Integration testing is an important phase of the software development life cycle. It is employed to detect faults in the software system for systematic testing of modules in resolving those faults timely. Lack of any type of testing causes the failure of systems. Sometimes faults occur because of underestimation of integration testing. Insufficient or ineffective integration testing can cause a significant loss to society. Society will be affected adversely during any natural or economic crises if software is not properly tested, especially for police, governmental systems, banking systems, railway systems, airway systems, website systems, and medical systems. [8] Due to this inadequacy, integration testing for modern and complex software is characterized by infinite input/output sequences of interaction between entities and the environment, thus jeopardizing the certainty that it is impracticable, considering all the reasons. Such complexity indicates that not all functioning remnants of the software are exercised by the same profile of tests, and perhaps the defect repair can help others remain spotless.

Integration testing aims to properly allocate, activate, and deactivate entities within a suitable definition of stopping. Each test case is driven by a profile where the test would lead another test entity history to the test termination. The preparation of test data records related to good profiling can be labor-intensive, and some integration testing enterprises include a preliminary phase of profiling support [9]. However, the objective of exercising program components that interact with one another can lead to a significant number of deterministic programs that help generate the test profile for its components. The training of profiles' analytical segments can be reduced if contributions synthesized by a set of established programs have been run subject to the target test entity activation order; the instruction of both performed and non-performed program interactions; and the set of possible joint programs that had been led during the previous computation of sub-profiles, by one or more independent programs through previous testing. Equally for independent programs, whether profile program synthesis is very costly, program exercise or watch has been recognized as a practical alternative.

2.2 Types of Inter-Dependency Patterns

This section describes different types of sub-failure interdependency patterns, originating from software internal subdivisions. During regular operation, a software application manipulates state; for instance, it writes data to logs, outputs information to the user interface, and interacts with other software components. Some components generate information, which is consumed and used by other components. If some of the production-handling components produce irrelevant data, the consumer's accuracy may degrade. Such degradation may be isolated, causing a "sub-failure" effect that prevents data handling applications, components, or functions from processing a particular subset or class of input data or application messages correctly, leading to consequential failures elsewhere in the system. Those "sub-panics" that can cause the system to fail elsewhere over a period of time are referred to as sub-failures. The detection of sub-failures and the prevention of consequential failures are the goals of our research. [10]

3 Machine Learning for Fault Detection in Integration Testing

In the context of integration, we generally consider fault detection techniques that inject faults in the interfaces of components, allowing for the detection of faults around the defined scope. Faults are generally identified because the fault either generates a different result or a result that causes an observable change of output that can be used to set up or verify the test expectation. Other kinds of techniques use specific assertions to check for correct results and identify, for instance, a possible deadlock condition. Determination of code coverage can be used to select parts of code that should be tested more carefully, which increases the chance of detecting susceptible parts and helps in debugging activities [11].

Existing approaches to fault detection determine test cases with explorative strategies that take source code into account and therefore can be applied early in the unit testing phase, but are not very effective in integration testing, as they do not cover the interactions necessary for fault detection. We introduce a data-driven strategy to explore test case patterns from acceptance testing that allows us to identify such patterns and produce tests that are frequently maintained and executed by the software industry, testing how components are integrated with the operational environment.

Let's use machine learning over access logs, generally available in the operational environment, where usage and its outcomes, including transaction results, are recorded. We propose a general scheme for extracting scope-integrating canary-like patterns to be added to the software for supporting integration testing, accelerating the detection of real integration bugs, performing with production data, and discovering its private contractual interfaces.

3.1 Overview of Machine Learning

Machine learning is a subset of data mining that involves building a mathematical model to help in predictions and classifications. The model is built through learning from historical data, after which it is used to predict future data. Machine learning is divided into two primary categories: supervised learning and unsupervised learning. Supervised learning requires a training dataset that includes the input variable used for prediction and the output variable from historical data. On the other hand, unsupervised learning uses input variables only. It is more exploratory than supervised learning since it searches for hidden patterns or intrinsic structures in the input data [12]. The utilized algorithms may be classified as 'generative' or 'discriminative.' Generative models attempt to determine the whole joint probability distribution of the input data, whereas discriminative models directly determine the conditional probability of the target variables given the input data. Due to the need to determine a conditional probability, discriminative algorithms can generally be more flexible and lead to more effective models compared to generative systems. Examples of specific algorithms within these categories include: Naive Bayes, Support Vector Machines, Logistic Regression, Decision Trees, K Nearest Neighbors, and Deep Learning. Various algorithms and tools for machine learning include: RapidMiner, Excel, Octave, WEKA, and Orange.

To make the realization of machine learning possible, a representative set of patterns has to be chosen. Input details for machine learning are collected from input logs based ideally on the assumptions. Examples include the Bayesian belief network algorithm, which utilizes a Bayesian network as a classifier; the estimation of distribution algorithm, which tries to optimize the parameters of a probability distribution model; and random forests, which use bootstrapping and resampling technologies. Furthermore, randomness is typically introduced by bootstrapping and resampling technologies to the learning algorithm and predictions. Unlike other material fields, software fault detection based on the ability to learn patterns from logs has not been done effectively. That is despite the increased use of machine learning for fault prediction in real life. With software bugs and faults detection still remaining the primary quality needs in modern systems, there are only two known algorithms that have achieved success in the area of bug prediction and classification using binary label datasets. [13]

3.2 Applications in Fault Detection

The two applications for this pattern are after each recoding of a log and after the development process of a component. It breaks the logs according to a strategy selected before and tries to map lines with their corresponding recoding procedures that are executed by the test reason of their developments.

One of the machine learning algorithms that has an important role in this pattern is the decision tree algorithm. It is used as an association role. With some decision tree characteristics, it is possible to understand and visualize the possible feature associations for the blocks in the logs. The implementation of this fault detection in association with split logs is programmed so that it could be used with the shell.

The other pattern flaw deals with the quality of software interdependence. Problems related to adequate data entries, validation, transformation functions, and software critical situations are covered by these types of solutions. Both of the solutions have the validation phase for the software development in their tests, helping before its final setting. This kind of interdependence issue is easier to detect regarding the previously addressed problems. The patterns for the short- and long-term solutions have a similar cause. [14]

4 Logs Data in Integration Testing

Surprisingly, for integration testing in safety-critical industries, one type of test data that has remained largely overlooked is logs data. Once a system or software or several subsystems or software are integrated as a composite, testing is performed to ensure that the systems or software operate as expected. The tasks for integration testing include establishing that software failures are discovered quickly, early system integration, early continuous testing, ensuring all levels and types of requirements-based testing are executed and tested, and casting test logs to be used for evaluation. These can be preserved in logs data collected from various tools that are generated within the integration testing period. Nevertheless, logs data have been referred to only in passing. In this chapter, we introduce log statements that can be used on specific system interfaces using software-runtime-testing. [15]

The motivation for artifacts in the area of software-runtime-testing comes from gathering artifact data in safety-critical environments where no real-time process information is available or overloading results into statistics, which then serve for dashboarding. Failure and recovery causes come from a wide range of sources that include human error, data corruption, software problems, and hardware device faults. ADMS help support various life-cycle development activities in the form of system documentation, understanding, analyzing potential risks before and during hardware-in-the-loop testing, and evaluating testing risk in pre-FS(P)E area. Analysis and information about the system are provided, and the objective is to help avoid system failures. [16]

4.1 Role and Importance

Introduction In this digital century, software inter-dependency is increasing even faster than the space that software merges into. This means that it is now essential to position software fault detection as one of the software creation integrated processes that need to be effectively and sustainably repeated from requirements creation until an end-of-life event. Role and importance The focus is integration testing, a process that was passed to the beginning of the 21st century without taking advantage of data analytically rich supporting accesses, which, if not available before, are now needed due to progress imposed on IT by these waves: digital transformation, cloudification, big data, artificial intelligence, and the uncertainty created by these three operational challenges: the rapidly evolving software architecture style, the orchestrator complexity that triggers test cases, and the multitude of deployment options that can be mixed in real situations.

4.2 Challenges and Opportunities

Challenges in our formulation stem from the diverse nature of software involved in a given system, the existing methods of modular addition of software and systems, and the pessimistic strategy of applying fault detection at the level of whole systems. Identifying the key system-level properties that are influenced (indirectly or otherwise) by the presence or absence of faults in the collaboration of particular representative software components is fundamental to the idea of focusing fault detection effort and resources on parts of the system that are not operating correctly and may not be separately tested. A key question in this paper and in practice is the extent to which indirect fault considerations can be used to detect or localize faults at a high level of abstraction, such as system components. This also leads to the interesting question of whether system testing can utilize already existing small component-level test configurations to make its task more efficient in terms of effort and cost. The discussion includes an analysis of how control and data flow properties change in the presence of system faults. These properties are shown to exhibit characteristic changes when a system with fault components is used. Apart from providing guidance to specific software fault generation, this also influences the discussion of test adequacy criteria that detect such changes. Concerning detection as an aid, the control flow-based fault detection method is analyzed in detail. [17]

5 Methodologies for Determining Inter-Dependency Patterns

A correlation matrix is generated by statistical analysis to determine service inter-dependency patterns. Specifically, the time windows of execution logs of dependent services are obtained by the sliding window method, and the sequential log length is determined. By abandoning or retaining some log attribute traversal combinations, weight evaluation models are derived, and the evaluation score of event pair correlation ratio is based on support degree, confidence degree, and significance degree between independent data. If the value is high, the user-specified threshold can be used to determine the relative inter-relationship. To sum up, if all the event pairs in the time window have at least one of two pairs of inter-related services, which can take at least one of the current service names, it indicates that the direct service has a connection. [18]

The correlations between different services for all recorded events at the previous time and the current services in the event pair list at time T are calculated. In this study, the dynamic sliding window method for preprocessing is developed to solve the length problem of the data attributes sequence in time-window based approaches. Firstly, attribute traversal combinations are obtained by analyzing service execution logs. Unused attributes and instances of the two classes, useful attributes and data instances, are removed. Subsequently, the correlation ratio feature optimization approach is conducted to calculate the significance rating of the attribute combination. Finally, the candidate feature is generated according to a user-defined discrete degree, and an event-scaling matrix, with features and data instances, is generated.

5.1 Traditional Approaches

The first general traditional approach to fault detection strategies of integration testing is based on the assumption that an application fault means that the integration of components is not successfully completed. The main goal is to detect this problem when modules are being assimilated, and it belongs to regression testing, where the tests automatically generate the behavior of the composition, compare it with the expected one, and signal a problem in the latter. The state of these strategies is more complex if the software under test requires consumer-provider or master-slave relationships between service components. Service dependency is a critical issue in service-oriented systems. Making services recognize their own dependency is the first step in improving a wide range of software engineering activities, such as identifying integration test objectives, risk assessment, feature testing, etc. The second general traditional approach states that a characteristic of component-based systems is that most of the components are supposed to be black boxes, and the possibly non-trivial interactions with the environment are specified by the interface or connectors, while the underlying infrastructure takes care of composing together the components that are properly assembled. These strategies deal with testing system-level properties; there is no need to test individual components of the system because a significant part of the system-level testing activity is dedicated to verifying the property that the components satisfy the interface required for an implementation context. [19]

5.2 Machine Learning-Based Approaches

Numerous studies and tools have addressed detecting software interaction faults for integration testing. However, determining how to accurately and quickly detect different software inter-dependency patterns, such as many-to-one and many-to-many patterns using learning methods, directly from log data has not been previously researched. In this review, we discuss related works and tools based on the notion of location and examine a variety of machine learning approaches to detect software inter-dependency patterns (Fig. 1).

Machine Learning-Based Approaches Statistical support vector machines compare a new pattern with known patterns and find the best match, which transcripts the model of the diagnostic decisions taken by experts. One-class SVM shows efficient detection of inconsistency-related faults in call configurations. The One-Class Support Vector Machine is used for context-aware anomaly detection, such as checking whether the work items exhibit different deviance patterns that the physical and comment activities have shown during work. Support Vector Machine is used to detect three fatal fault types of highway-pile interaction faults, and the feature subset of SVM is selected according to the sensitivity of the feature dimension used for classification. Four types of random forests are researched for sequence score, sequence features, and their combination to detect the co-evolution of co-authorship and co-citation relationships among institutions at roughly the same pace. [20]

6 Case Studies and Experiments

For real industrial benchmarks such as Jetty client and server implemented in Java, we explore the possibility of using the automatically discovered software inter-dependency clustering information to improve fault detection for integration testing using machine learning on log data. Due to the stability of the learning performance and the absence of subject domain knowledge involved in the software inter-dependency clustering based machine learning model, the semi-CEKiT prototype system has the potential to be employed early in the integration phase without requiring full knowledge of error repair. The machine learning techniques adopted in the prototype system are naive Bayes, which is applied to training instance rejection and precision enhancement, decision tree with the evolved accountable architecture of a program detection strategy, and neural network classification or linear regression for test results assessment. Under a 5-gram language model, the LogTopN patterns are thresholded by the occurrence frequency and selected to early prune off uninteresting log events for noiseless log data, where conservative data compression is elective for less as well as the maintenance of systematic traceability such as bug diagnosis. Based on the reverse accuracy defined on the traditional precision and recall of the training set, the machine learning rate is utilized as a coherence measurement to enforce the filtering of the transactions of hierarchical log phosphors. Furthermore, for the detection of test automation, LogEx input is used to manage the non-functional test mode by representing the status of debug log data as a classification problem, where LogEx output is utilized to discover serious test room maintenance work affected by non-functional test issues.

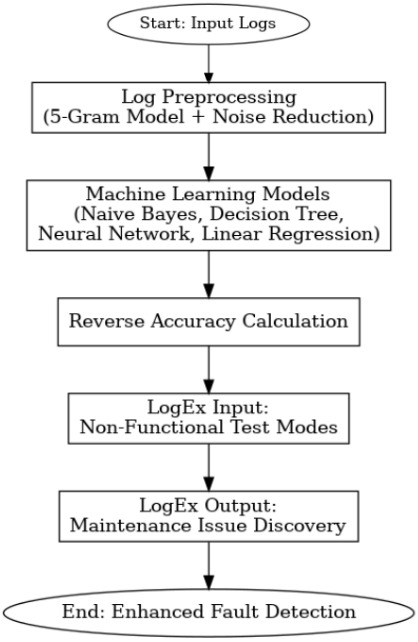
Start: Input Logs
Log Preprocessing (5-Gram Model + Noise Reduction)
Machine Learning Models (Naive Bayes, Decision Tree, Neural Network, Linear Regression)
Reverse Accuracy Calculation
LogEx Input: Non-Functional Test Modes
LogEx Output: Maintenance Issue Discovery
End: Enhanced Fault Detection

Fig. 1. Flow diagram of Fault Detection on Log Data Using ML

6.1 Studies on Real-world Applications

Due to monitoring logcat standard features and special user log API, the user interaction data is stored in the Android OS. Information about a customer's communication with the system is collected through the Customer Communication Software product on customer service channels. The cooperation of CCS and the login function of the Android OS enables the user to log in to the entire user interface and communication logs. We implemented the detection of application faults in similar real-world cases by highlighting faults automatically using a log event that had a connection loss and had to be restarted after the occurrence of log events. The software testing domain claims to serve scenarios and applications to support practitioners by using logs. However, there are some challenges in the implementation of the real-world log-based fault predictor that resist such application.

The dataset and experimental settings, which are used to investigate user interface log-based predictors, might be judged by the quality of users. Optimizing the construction of training data relies on defining reliable ground truths, which is an expensive process that requires significant user engagement. Training data is identified in real-world user interface logs where descriptions are manually checked regarding application behaviors. Moreover, a second expert checks a subsample of the logs to confirm the annotations' quality. Then, the best quality data for building reliable fault prediction models are selected. The data is valuable. The performance of fault prediction in the case of real-world tools is frequently subject to identification and description time, and even an experienced tester needs about 65 days to deliver the ground truth for testing. In the Android user interface, due to a mixture of human-level detailing, complex applications, and a high number of challenges, frequent problems with badly performing error messages are observed. Although the idea is that the logs are closely linked with faults and thus contain information about the problem entity, often, including during the testing phase, that data is not suitable. At the very least, the elements are not well separated in the visual output or have become cryptic because of imperfect visualization for a complex application.

6.2 Experimental Setup and Results

6.2.1 Experimental Environment - For our experiment, we chose the real JUnit test logs for several subjects. We used a commercial off-the-shelf integrated development environment for parsing the Java files and running the JUnit test cases. The optimizer was used in the post-processing steps to merge the multiple per-function patterns into a dense asset. We used Python and a machine learning library for the machine learning model. The library is a popular open-source machine learning library based on Python, providing a convenient and well-integrated interface for various machine learning algorithms. It has clear, simple, and efficient tools for predictive data analysis built upon other scientific computing libraries. We thus used the library from a Python environment for our data preparation and machine learning for failure prediction and compared our built model with a number of traditional and state-of-the-art machine learning models.

6.2.2 Evaluation Metrics - We use a variety of evaluation metrics to assess the performance of our machine learning models and to compare our built models with potential models. We summarize the definitions of those evaluation metrics.

6.2.3 Results - Question 1 RQ1: Are software inter-dependency patterns leveraged from logs for the purpose of failure detection in integration testing? The predictive performance of models has proven useful for both fault-prone module detection and family versus individual fault detection for developers in previous research. Metrics are therefore useful for the purpose of fault detection. Moreover, for integration testing, the inter-module interactions are straightforward. As part of the integration testing, logs of failed test cases can easily be collected, and an inventory of frequently interrelated modules can therefore be constructed based on these logs.

7 Critical Analysis of Existing Approaches

The unsupervised or supervised pairwise determining interdependencies do not resolve the critical but simplifying assumptions. Software interdependencies are inherently complex and inherently multivariate. It requires multiple dependencies to detect the expected failed behavior of the integrated software system. On the other hand, the expected hazards in complex systems originate from a complex causal chain understanding of the interdependencies.

We critically analyzed many software interdependency detector solutions. The problems with certain methods are pointed out. We also offered a new method for the users. We can concur in the frustration to "create scripts for testing" by considering the interaction style. Since this is the prelude to more comprehensive methods, we proved with a real scenario that each existing method is biased without offering correct solutions for multivariable inherent software multi-interdependencies.

7.1 Strengths and Limitations

Strengths- The presented work has several advantages, such as: - Fault detection measure based on the number of established paths per software pattern. - The ability to determine patterns of the same level in relation to the user's needs to assess how much the software interacts at the top level with the system, and in the lower level to investigate the relationships of all the parts of the pattern. - Simple determination of the presence of the above patterns in the project by analyzing the log files of the integration testing. - Extra data that can help to assess the state of verifying the elements of comprehensive automated tests depending on the set goals and criteria of determination. - Using the case study to confirm the need to determine the relationship between the modules from the set features, which will help to increase the capacity for predicting the program's completion time in future research.

Limitations-The proposed decision can also be criticized for having such limitations as: - It is difficult to determine the test when assigning a high level because the system using the test has an infinite number of interaction paths, and the potential to use the structure of the categories is higher. - The proposed method is poorly scalable because software with large functionality takes a long time to check the interaction paths. - The inability to determine the paths for typing. - The absence of a non-standard delay between the input and output of the signals that lead to the fixed path of interactions.

8 Future Directions and Research Opportunities

To optimize the model, an empirical investigation is proposed to analyze direct or indirect correlations between logs and enhance software inspection using static analysis tools to generate richer logs embedded into interconnected software chains. Current time-based approaches for log retention require improvement, and further research on NLP algorithms and deep learning is needed to maximize sequence efficiency during implementation and evaluation. By recording developer activities, such as navigation and keyboard logs, using existing tools in editors, the proposed approach aims to generate and compress logs, improving productivity at no additional cost. This method encourages a smooth start and offers value-added log compression while emphasizing the need to actively test and evaluate these systems to enhance their internal and external value.

8.1 Emerging Trends

For the development of software-intensive systems, many applications and systems depend on the business logic provided by one or more software systems. In today's cloud and web service environments, this may very well be comprised of several commercial systems that are either bought, rented, or accessed as a service. In this paper, we establish a pattern-based technique for fault detection based on determining software inter-dependency patterns. In industries, the trends of DevOps, microservices, and continuous deployment have been emerging, and as a result, the architectural styles and deployment models such as service-oriented architecture and cloud are heavily used. These trends also lead companies from bespoke in-house development of systems towards a mixed strategy, where they build upon copies of existing systems. We propose an integration testing regime that differs from a traditional perspective on software integration, relying on a purely technical integration overview and instead working with a goal-driven way of performing integration testing.

8.2 Potential Impact

Research provides a simple and effective fault detection solution for the Testing-in-the-Small phase of software testing that requires less human effort than traditional methods. This is achieved by determining an inter-dependency pattern of software written in Java that enables detection of faults based on logs generated during a single run of tests. Using change data provided from an actual integration testing of a real-world application, determining inter-dependency patterns was accurate and maintained their effectiveness for the majority of tests. However, it becomes less reliable or ineffective over time when applied to different versions of the same program. This could be due to architectural changes to the software, making any determined inter-dependency pattern irrelevant. Our proposed solution uses a very small sample found within the source code, and the retrained classifier is shown to be effective in filtering out a large proportion of false alarms found in a previous version.

Currently, this research is solely applicable to integration testing of software written in Java. Its future impact could be significant for potentially being reused for similar phases using the same criteria, or by using the proposed feature solvers as a base, but modified to be applicable to other languages and program phases. The main contribution and its implications span not just across the IT industry, but across other industries that need to regularly test programs so that they have reason to be confident that a program they rely on, or wish to make use of, is free of defects. Our research on better integration test fault detection for software enhancement benefits a company that develops financial market predictive models for investment firms.

9 Conclusion and Summary

The review has highlighted the significant potential of machine learning techniques in enhancing fault detection during software integration testing. By analyzing software inter-dependency patterns in logs data, ML models can identify complex faults that are often missed by traditional methods. The review also highlights the effectiveness of various ML techniques, such as neural networks, decision trees, and clustering, in improving fault detection accuracy.

This study presents a novel solution to enhancing software integration testing through machine learning on logs. By automating fault detection and inter-dependency analysis, the proposed method significantly improves the reliability and efficiency of testing processes. This work sets a foundation for further advancements in adaptive, log-driven testing frameworks. The use of logs as a primary data source for integration testing ensures efficient fault detection while minimizing downtime and resource use. Overall, the use of machine learning for fault detection represents a promising direction for advancing software testing practices. The ability of ML models to handle large volumes of logs data and detect intricate fault patterns makes them a valuable tool for ensuring software reliability.

References

1. M. Alphonce, "Enhancing Software Quality through Early-Phase of Software Verification and Validation Techniques," Available at SSRN 4611404, 2024. ssrn.com
2. M. Segovia and J. Garcia-Alfaro, "Design, modeling and implementation of digital twins," Sensors, 2022. mdpi.com
3. Nils Wild and H. Lichter, "Unit Test Based Component Integration Testing" APSEC, 2023. [HTML]

4. D. Marijan, "Comparative study of machine learning test case prioritization for continuous integration testing," Software quality journal, 2022. [HTML]
5. S. U. Jan, Y. D. Lee, and I. S. Koo, "A distributed sensor-fault detection and diagnosis framework using machine learning," Information Sciences, 2021. [HTML]
6. R. Rajaraman, P. K. Kapur, "Determining software inter-dependency patterns for integration testing by applying machine learning on logs and telemetry data," 2020 8th International ..., 2020. [HTML]
7. A. Akbarzadeh and S. Katsikas, "Identifying and analyzing dependencies in and among complex cyber physical systems," Sensors, 2021. mdpi.com
8. C. Qian, X. Cong, C. Yang, W. Chen, and Y. Su, "Communicative agents for software development," arXiv preprint arXiv, 2023. openreview.net
9. Peixun Long and Jianjun Zhao, "Enabling Easy Integration Testing of Microservices in Kubernetes", 2023
10. K. Burns, "Flipping failure in graphic design: A design mindset method to navigate mindset loops and pathways to professional success," 2023. swinburne.edu.au
11. Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting*, 2021. acm.org
12. S. Naeem, A. Ali, and S. Anam, "An unsupervised machine learning algorithms: Comprehensive review," International Journal of ..., 2023. researchgate.net
13. N. Li, M. Shepperd, and Y. Guo, "A systematic review of unsupervised learning techniques for software defect prediction," Information and Software Technology, 2020. [PDF]
14. Z. Chen, Z. O'Neill, J. Wen, O. Pradhan, T. Yang, and X. Lu, "A review of data-driven fault detection and diagnostics for building HVAC systems," *Applied Energy*, 2023. sciencedirect.com
15. M. Landauer, F. Skopik, M. Wurzenberger, and A. Rauber, "System log clustering approaches for cyber security applications: A survey," Computers & Security, 2020. sciencedirect.com
16. E. Kurian, D. Briola, P. Braione, and G. Denaro, "Automatically generating test cases for safety-critical software via symbolic execution," Journal of Systems and Software, 2023. [PDF]
17. R. Xiong, W. Sun, Q. Yu, and F. Sun, "Research progress, challenges and prospects of fault diagnosis on battery system of electric vehicles," Applied Energy, 2020. [HTML]
18. L. Zeng, W. Ren, and L. Shan, "Attention-based bidirectional gated recurrent unit neural networks for well logs prediction and lithology identification," Neurocomputing, 2020. [HTML]
19. I. Cohen and D. Peled, "Integrating distributed component-based systems through deep reinforcement learning," in *International Conference on Bridging the Gap between …*, 2023. aisola.org
20. J. Dong, J. Lee, A. Fuentes, M. Xu, and S. Yoon, "Data-centric annotation analysis for plant disease detection: Strategy, consistency, and performance," Frontiers in Plant, 2022. frontiersin.org