RESEARCH ARTICLE

International Journal of Computer Communication and Informatics

DOI: 10.34256/ijcci1916

Mining Techniques For Invariants In Cloud Computing

K. Sadhika [a*]

[a] UG Scholar, Department of Computer Science and Engineering, GMR Institute of Technology, Razam.

*Corresponding Author
sadhikakarri2@gmail.com
(K. Sadhika)

Received : 12-04-2019
Accepted : 20-05-2019

ABSTRACT: The increasing popularity of Software as a Service (SaaS) stresses the need of solutions to predict failures and avoid service interruptions, which invariably result in SLA violations and severe loss of revenue. A promising approach to continuously monitor the correct functioning of the system is to check the execution conformance to a set of invariants, i.e., properties that must hold when the system is deemed to run correctly. This paper proposes a technique to spot a true anomalies by the use of various data mining techniques like clustering, association rule and decision tree algorithms help in finding the hidden and previously unknown information from the database. We assess the techniques in two invariants' applications, namely executions characterization and anomaly detection, using the metrics of coverage, recall and precision. In this work two real-world datasets have been used - the publicly available Google datacenter dataset and a dataset of a commercial SaaS utility computing platform - for detecting the anomalies.

Keywords: Anomaly detection, Invariants, SaaS, Cloud

1. Introduction

Dynamic invariants are properties of a program that holds at a certain point or points in a program and this dynamic invariant detection runs a program, observes the values, and then reports properties over the observed executions. So system invariants are attractive for modelling runtime behaviour of data centres and cloud based utility computing system from a service operation viewpoint. Due to the size and complexity of such systems, it is very hard for human operators to detect problems in real time like timing issues, exceptions, system crash etc., The violations of system invariants are considered as symptoms of execution malfunctions and mining invariants include activities like capacity planning, detecting anomalous behaviours and violations of service level agreements. But practitioner faces several problems to select a proper technique for their analysis goals and this can be analysed by analysing and comparing techniques to mine invariants. By empirically analysing and comparing techniques to mine invariants, we contribute to gain quantitative insights into advantages and limits of such techniques, providing operation engineers with practical usage implications and a heuristic to select a set of invariants from a dataset.

There are three techniques namely clustering, association rules and decision list. They are applied to two independent datasets collected in real world systems - Google and SAAS platform for finding correct and anomalous executions. We assess this technique in two invariants namely executions characterization and anomaly detection based on coverage and precision. So by using these mined invariants, it was possible to provide a valuable result, spotting for anomalies for a number of transactions. The study focuses on three techniques: two unsupervised, namely clustering and association rules, and one supervised, decision list. They are applied to two independent datasets collected in real-world systems: a cluster operated by Google, whose traces from about 12,500 machines are publicly available, and a SaaS platform in use by various medium- to large-scale consumer packaged goods (CPG) companies worldwide. The datasets comprise 679,984 executions (correct and anomalous) of batch units of work, namely jobs and transactions.

The considered techniques provide a valuable support for characterizing executions and detecting anomalies in an automated way. For the SaaS cloud platform in particular, using the mined invariants it was possible to provide a valuable result to the service operation team of the IT company, spotting true anomalies for a number of transactions out of the seven month's of operation data, which were indeed missing and went unnoticed. A relatively small number of invariants hold in a majority of system executions. For example, in the Google dataset less than 10 invariants cover more than the 80% of job executions (using association rules - Apriori algorithm). Using further invariants does not increase coverage significantly. Invariants are very sensitive to the coverage: small variations of the coverage impact significantly recall and precision. In spite of the best coverage, association rules are not well suited for anomaly detection; notwithstanding the smaller coverage, invariants mined by decision list achieve higher recall/precision for anomaly detection. We propose a general heuristic for selecting a set of likely invariants from a dataset.

2. Literature Survey

Dynamically program invariant detection technology is used to detect invariants in the data and we should have lack of accuracy and efficiency for understanding the detected program. In this paper, we divide the invariants into two kinds –functional and non-functional invariants. First it focuses on the functional invariants and later it detects the existent invariants which solves the problem of blind detection to improve the efficiency but also reduces the possibility of missing important functional invariants. To detect the invariants, we have to insert some probes in the detection points without destroying the logic integrity of the program and next we have to select test cases and run program over test suites and analyse the data trace and report likely invariants in the form of relational table. Then we have to deduce functional dependence set from trace relationship and consider each function form from set and deduce the parameters from current data trace file. This approach resolved the problem of how to detect the forms of functional invariants which can improve efficiency of the traditional hypothesis verification approach of detecting invariants [1].

The increasing popularity of software as a service stresses the need of solutions to predict failures and avoid service interruptions, which result in SLA violations and loss of revenue. In this paper we propose a framework and a tool to automatically discover invariants from Saas application logs. Invariants are the properties of a program that are hold for all executions of the program. If these properties are found to be violated while monitoring, it is possible to raise an alarm for immediate action. In this, they consider a log and apply framework and tool for 9 months, it detects 12 invariants with stringent goodness of fit criteria of 0.7 from a possibility of 528 relationships. It is implemented in java as icirrus toolset both for identification of invariants among the relationships from application logs. This approach reduce the quality of data to be analysed for understanding the system behaviour in case of error and detect the error itself [2].

Invariants represent properties of a system that are expected to hold when everything goes well. Thus, the violation of an invariant most likely corresponds to the occurrence of an anomaly in the system. In this paper, we discuss the accuracy and the completeness of an anomaly detection system based on invariants. Invariants represent properties of a program that are guaranteed to hold during its execution. Thus, their violation during the program execution likely represents a symptom of an anomalous behaviour by using invariant detection technology. Here we compare the results of a detection mechanism based on invariant violation with the actual violations present in the logs accurately. Also, we studied how much the time to mine invariants and the time to detect anomalies depend on the sampling time. The accuracy of the approach stays in the range 50-74% depending on (i) used invariants and (ii) sampling time. Thus, a completeness of 100% is found, thus, all the anomalies reported in the application logs are detected through the invariant-based approach [3].

This paper presents an instance based approach for recognizing the failures in computing system. There are some repeated failures in the system. So, our method takes advantage of past experiences by storing historical failures in a database and retrieving similar instances in the occurrence of failure. We extract the system 'invariants' by modelling consistent dependencies between system attributes during the operation. We use a high dimensional binary vector to store those failure evidences, and develop a novel algorithm to efficiently retrieve failure signatures from the database. A template based failure retrieval algorithm has also been developed to gain retrieval efficiencies. This can be applicable to large computing systems. We have proposed our unique representation of failure signature, and the metric for comparing different failures. Experimental results have demonstrated that our method can achieve accurate and fast retrieval of historical failures, in which it leads to save the time. But this cannot be applicable to system undergoes significant updates, such as the structure change, we do not know whether the failure signature will still hold or not. As our future work, we will perform extensive experiments to further verify the current approach [4].

Explicitly stated program invariants can help programmers by identifying program properties that must be preserved when modifying code. In practice, however, these invariants are usually implicit. An alternative to expecting pro- grammars to fully annotate code with invariants is to automatically infer invariants from the program itself. This research focuses on dynamic techniques for discovering in- variants from execution traces. This paper reports two results. First, it describes techniques for dynamically discovering invariants, along with an instrumented and an inference engine that embody these techniques. Second, it reports on the application of the engine to two sets of target programs. This paper documents the feasibility and effectiveness of dis- covering program invariants based on execution traces. The techniques we have developed, along with the prototype implementation, are adequately fast when applied to programs of several hundred lines [5].

3. Design

3.1 Data Sets

In this work two real-world datasets have been used - the publicly available Google datacenter dataset and a dataset of a commercial SaaS utility computing platform - for detecting the anomalies.

3.1.1 Google cluster

The workload consists of tasks, each running on a single machine. Every task belongs to one job; a job may have multiple tasks (e.g., mappers and reducers). There are six tables in the dataset: Machine_events, Machine_attributes, Job_events, Task_events, Task_constraints and the Resource_Usage. Every job and every machine is assigned a unique 64-bit identifier. Tasks are identified by means of the ID of their job and an index; most resource utilization measurements are normalized.

Machines are described by two tables. Machine_events reports addition, removal or update of a machine to the cluster, along with its CPU and memory capacity. Machine_attribute lists key value pairs of attributes representing properties such as kernel version, clock speed, and presence of an external IP address. The Job_events and Task_events tables describe jobs/tasks and their lifecycle. The Resource_Usage table reports resource usage of the tasks.

3.1.2 SaaS platform

The SaaS platform we consider provides cloud-based data processing and analysis capability to several consumer packaged good (CPG) companies. The platform accepts and transforms data files provided by customers through FTP servers or email attachments. The platform accepts and transforms data files provided by customers through FTP servers or email attachments.When a data file accepted by this platform, then it go through processing stages such as validation, data extraction and transformations. A processing stage within a transaction can result in a success or a failure. If success, moves to the next stage otherwise the platform generates an exception then the transaction is aborted. Management modules are responsible for handling the transactions and monitoring the progression of stages.

The platform relies on databases containing the configuration and business rules. The staging database maintains intermediate results of the work item and audit logs contains execution information and error events. This logs tables contains outcome of processing stage, such as id of work item and start/end times.

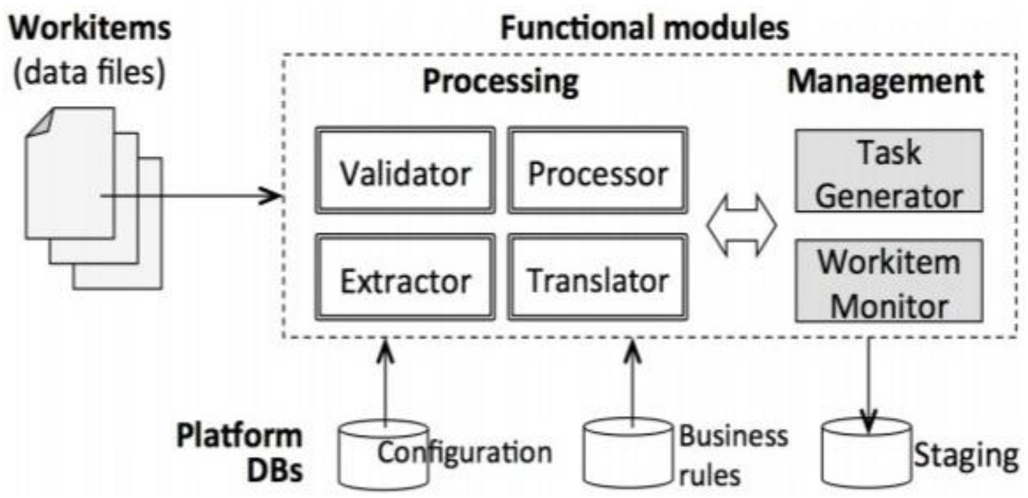

fig. 3.1.2 High-level architecture of the SaaS platform

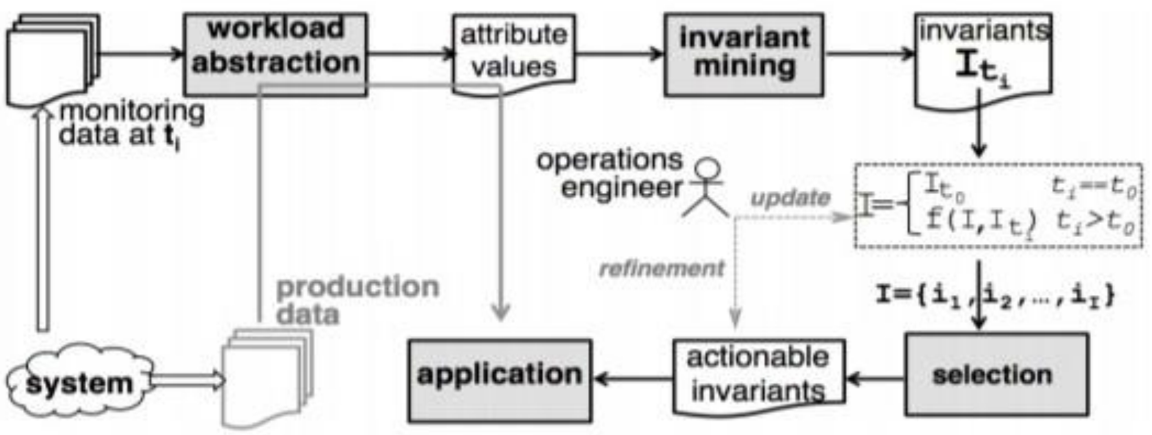

fig. 3.2 Framework to mine invariants and feedback mechanisms.

3.2 Invariant Mining

A workload unit W (i.e., a job in the data centre or a processing stage of a transaction in the SaaS platform) is abstracted by a set of N attributes A1,A2,...,AN. These attributes represent the computing resources used or parameters such as duration, priority and return codes, being collected during the execution of W. The attributes that characterize the execution of a workload unit assume a value in the Cartesian product {VA1 ×VA2 ···×VAN}, where VAj denotes the set of the possible values of Aj(1 ≤ j ≤ N). The values of the attributes are extracted from the input dataset to form an M×N attributes matrix, where M denotes the total workload units Wi (1≤i≤M).

It uses a framework and steps that underlie invariant mining. Among many invariants, they will select a subset of invariants for a specific application. We classify a workload unit to be correct, when it is correctly executed by the system, anomalous otherwise.

Given the input monitoring data at a given time ti, (i) workload abstraction infers the M workload units Wi and the values of the attributes for each Wi; (ii) invariant mining infers the set of recurring relationships among the values of the attributes from the data collected until ti, i.e., invariants Iti in Fig. 3. At ti==t0 (where t0 denotes the time of the first ever mining), the set of invariants available to operations engineers is I=It0, which is mined from the data at t0.Moreover,engineers will select a subset of invariants in I, i.e., actionable invariants in Fig. 3, that will be used for a specific application, e.g., anomaly detection.

3.3 Dynamic Detecting Likely Invariant

This mainly focuses on the approach of detecting functional likely invariant which not only solves the problems of blind detection to improve the efficiency but also reduces the possibility of missing important functional invariants compared with the traditional hypothesis verification approach such as Daikon.

3.3.1 Dynamic invariant detection

The whole running process of a program is close and invisible unless it needs interaction. So inserting some probes in the detection points without destroying the logic integrity of the program can obtain the information of the running program. When the probes are executed, the value of variables at those detection points will be thrown out. Analysing these feature data could help revealing the information of data flow and control flow of the program for discovering program invariant. The process of inserting the tracking code is called instrument and the location of the inserted probes is called instrumented program point.

There are four steps in the process of dynamic likely invariant detection, as shown in figure1: (1) Insert track code into the source program. (2) Select test cases. (3) Run program over the test suites. (4) Analyze the data trace and report likely invariants.

3.3.2 Trace of Program

Step 1 to step 3 is the process of generating and collecting the trace of program. The trace which implies the values of the instrument variables at program execution period is the base of dynamic detection. For example, suppose that X={x1,x2,…,xn} represents the instrument variables set and (d1,d2,…dn) represents the record of program running once (di means the value of xi after program execution).n items of running record will be obtained that constitute data trace file of the program on the detecting point when the instrument program is run over n items of test cases.

Therefore, to accomplish invariant detection based on the program data trace becomes to discover the parsing expression of the relation pattern by analyzing the instances.

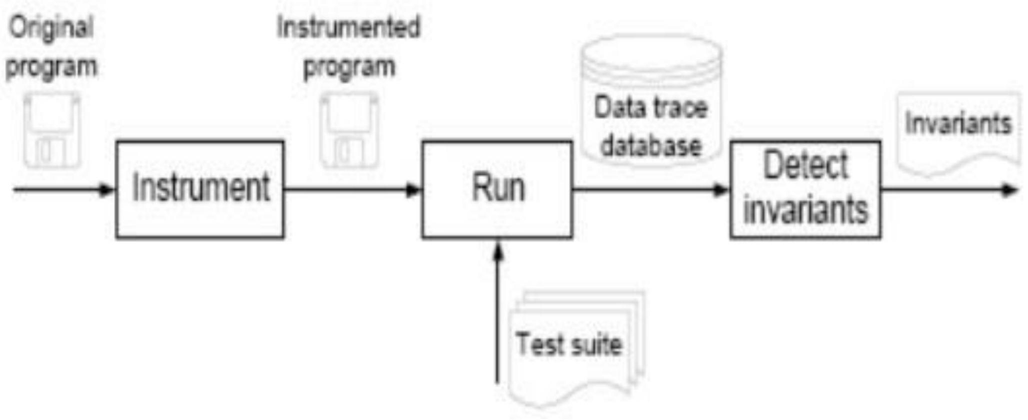

Fig.3.3.1 An overview of dynamic invariant detection

Table- 3.3.2 Values of instrument variables

Order	X1	X2	X3
1	2	4	3453
2	23	25	55
…	…	…	…
1000	4	6	3125

3.3.3 Classies of invariants

An invariant is the description of the property of a program, whose form is determined by the relationship between the constants, variables and expression on the instrumented program point. Function relation that abounds in a program is the most important data relation and has wide applications. The invariants can be generally classified into functional invariants and non-functional invariants .Functional invariants can be described in mathematic relation such as invariant in linear relation $y=a*x+b$, whereas those can not be described in mathematic relation are called nonfunctional invariants such as invariants in comparison relation $x<y$ and in range relation $a \leq x \leq b$ etc. Functional invariant should be considered with high priority because of its wide application and volume of existence.

3.4 Detection Approach

Once the invariants have been defined, we are able to estimate the expected output ˆ y(t) of the system, given the input x(t). Let y(t) be the actual output of the system for the input x(t). At this point, to define an anomaly detector, we have to select a function δ computing the distance of ˆ y(t) from y(t). For this purpose, we use the residual function:

$$Rxy(t) = |y(t) - \hat{y}(t|\hat{\theta})| \quad (2)$$

An alert is raised at time t if $Rxy(t) > \tau$ where τ represents the tolerance of the detection system. The number of estimated violations and the number of raised alerts heavily depend on the threshold value τ, as discussed in [6]. When τ =0, invariants are broken for almost each entry of the logs. Clearly, it is very difficult that the predicted values is exactly the same of the system monitored value. If invariants with coefficient of determination larger or equal to 0.70 are accepted, up to 30% of the variation is not explained by the model. A threshold depending on the prediction interval (π) of the output with respect to the provided input is then considered. In [6], it is shown that when adopting this threshold, the number of alerts is largely reduced, but anomalies likely causing SLA violations are detected anyway. In this we show that all the anomalies happening in the system and reported in the application logs can be detected.

Starting from the invariant mining framework presented in [6], we implemented an anomaly detection system. Its schematic diagram is depicted in Figure 1. The Invariant specification Workbench GUI allows the user to interact with the system

As an instance, of the incoming data to be used as training logs, a subset can be selected for instructing the detection system. The Log Analyser component, invoked by the workbench, takes such logs as inputs and generates time series as a flow matrix, which is used by the Flow invariant miner to infer the flow invariants. Invariants are then stored in XML format. The latter two components act as a single module for producing the invariants used for the detection. It is worth noting that the miner is able to automatically identify invariants and evaluate the goodness of fit exploiting the common format of time series data, while the analysis and the creation of the flow matrix is to be tailored on specific log format of the application.

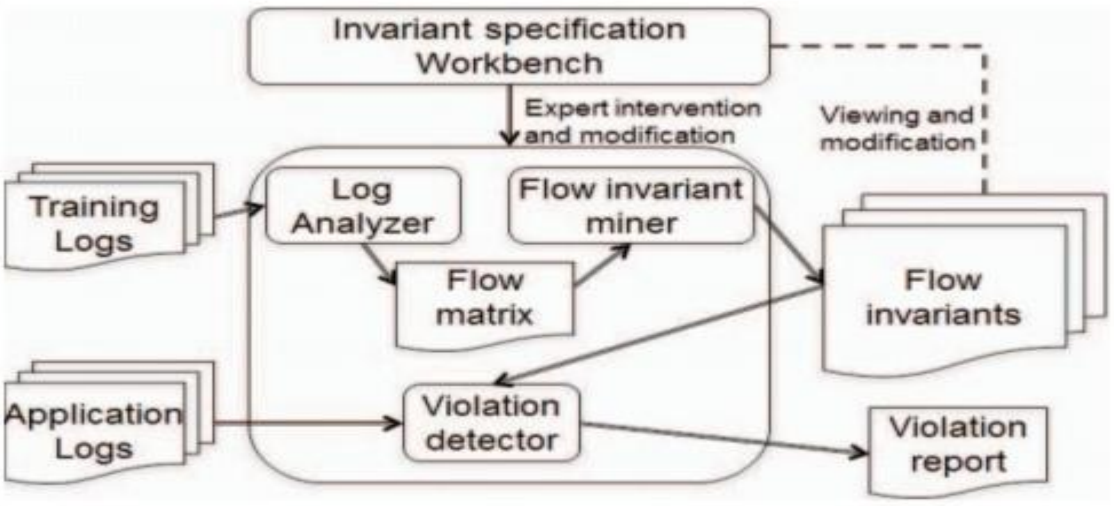

Fig. 3.4 Input, output, and main components of the implemented tool for mining invariants and online detecting anomalies.

The tool supports repeated execution of time-series generation and invariant generation for different sampling times with the help of the workbench module. It may also be required to define a new invariant that the operations team would like to monitor, in addition to the automatically identified ones, or to discard some invariants based on their prediction capability. The workbench module provides such functionalities, too. The Violation Detector component uses mined invariants and runtime application logs to check the ones that, possibly, are broken because of some anomalies and, in that case, rise alerts and generates violation reports. Clearly, also runtime application logs needs to be opportunely parsed to make them understandable by the detector.

3.5 Failure Signature Representation

The failure representation is based on our previous work [5] on system invariants discovery. The concept of invariants was motivated by the observation that most of the system attributes in the measurement data are strongly correlated. For example, the resource utilizations of the system such as CPU and memory usages always increase or decrease in accordance with the change of system workloads. Furthermore, the system structure and design also introduce a lot of correlated attributes. Based on the above observations, we build an ensemble of models to correlate the large amount of monitoring data collected from various points of the system. If the discovered correlations can continually hold under different user scenarios and workloads, they are regarded as invariants of the information system.

After we learn all the models, we also validate them using the operational data from different system workloads. Only those correlations that always keep high fitness value during the validation are regarded as the invariants of the system. Since the learned invariants reflect the system internal properties and are robust under normal system dynamics such as the workload variations, they can benefit many system management tasks. In the following, we use the status of invariants to represent system failures.

The discovered invariants can be illustrated by a network graph as shown in Figure 1(a), in which each node represents one system attribute, and each link represents the invariant relationship (1) between the two end attributes. Based on the invariants graph, we can inspect the system runtime status by examining the consistencies of learned invariants during the operation. We set a threshold for the residual R to determine whether the invariant model is broken or not. The threshold value is based on the residual values computed from historical data. In real situations, a system failure usually leaves evidences on a variety of invariant residuals instead of just a few broken invariants. The condition of each invariant, i.e., being normal or broken, provides a view of failure characteristics, because different types of failures usually introduce different subsets of broken invariants. Therefore we can use the status of system invariants network under the failure to represent the characteristics of that failure.

Figure 3.5(b) presents an example to illustrate the status of invariants network under the failure. In a typical situation, the failure starts with a relatively small number of broken invariants, followed by a gradual increase of broken ones until they get saturated after some time. In order to cover all those evidences, we record the status of invariants at every sampling interval, and include the union of all broken invariants during the failure period into the signature representation. The length of the failure period varies with different failures. If it is a transit failure or performance problem, the system may go back to the normal state after a short time.

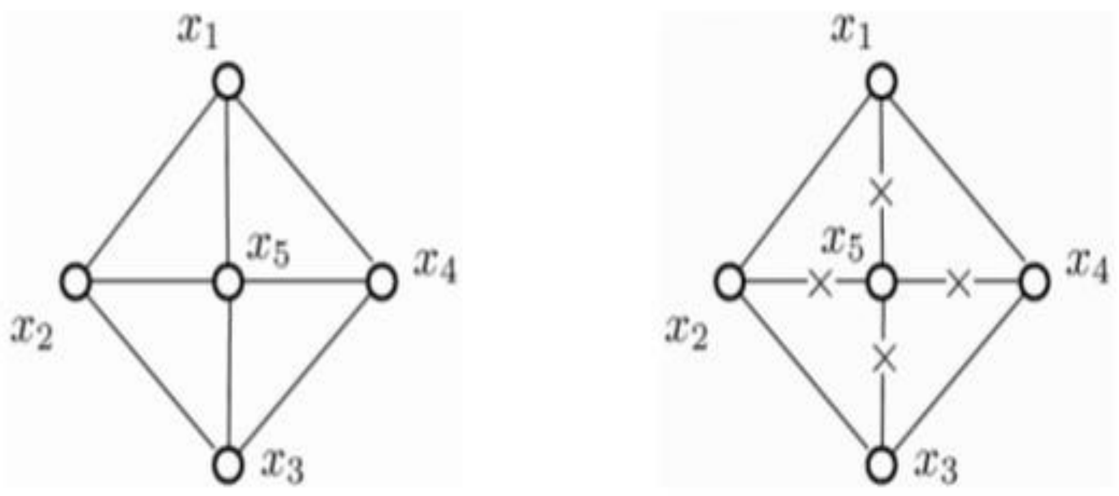

fig. 3.5 The status of invariants network when the system is (a)in the normal state and (b) under a failure

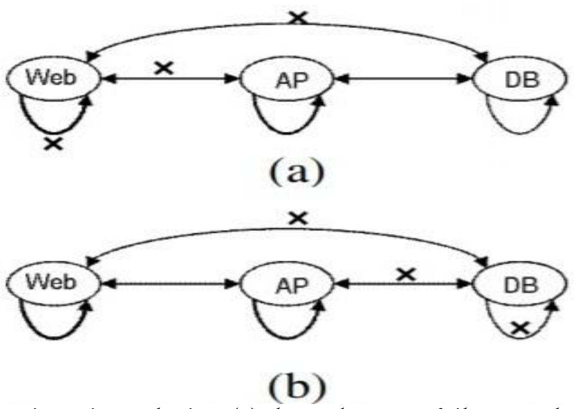

fig3.5 The status of system invariants during (a) the web server failure, and (b) the database server failure.

There have been several papers [2] [8] recently dealing with the instance based failure diagnosis. However, those methods used the raw system measurements as the failure signature. Compared with them, our graph based representation provides more evidences about the failure source because it includes the correlation changes between system attributes during the failure. Such information is especially important when the failure symptoms are noisy. Figure 2 presents an example to illustrate such fact, in which three units, Web, AP, and DB, represent typical components in a multi-tiered web system: the web server, the application server, and the database server. Note each node in Figure 6 represents one component that includes a number of attributes, and each line denotes a set of invariants formed by the attributes originating from two end components of the line. The line that connects the same component corresponds to the internal invariants of that component. Figure 6(a) presents the situation of web server failure, whereas Figure 6(b) presents the case of database failure. In those two situations, if the numbers of abnormal attributes, i.e., those violating their thresholds, are the same in the web server and database server, the measurements based failure representation cannot tell which server has the problem. However, in our representation, we can compare the sizes of the following two sets of broken invariants to get more clues: those between the web server and the application server, and those between the application server and the database server. If more invariants are broken between the web server and the application server, the web server is more likely to encounter a failure [9].

4. Methodology

4.1 Mining Techniques

The invariant mining step shown in Fig. 3 aims to infer recurring patterns among the attributes of the workload units. Likely patterns represent invariants, i.e., properties holding across different executions of batch work. . In the Google dataset we noted that 54,976 jobs assume the values R0, low and D0 for attributes R, P, and D, respectively, meaning that a significant number of jobs experiencing no task resubmissions have low priority and small duration. Similarly, in the SaaS dataset, 10,701 processing stages assume the value IT3, L1_REJ, Invalid_File (for S, E and R, respectively), indicating that the stage IT3 exiting with code L1_REJ fail because of an invalid file. There are a number of considerations underlying the choice of the clustering, association rules and decision list mining techniques. First, production systems might generate unlabeled workload data, which prevents the use of many machine learning techniques. More important, as pointed out in [10], invariants should be comprehensible and useful to practitioners. Alternative invariant based classifiers can been applied, e.g. neural or Bayesian networks; however, their output, e.g., probabilities and/or weights, have small explicative power for practical purposes.

4.1.1 Clustering

Clustering is an unsupervised technique and the invariants obtained specify the values of all the attributes .Clustering methods are mainly suitable for finding interrelationships between data to make a assessment of sample structure. It is required because for humans it is very difficult to understand data in a high dimensional space. It can be noted that the 30,025 stage concentrate around a few tens data points [11]. A similar consideration can be done in the Google dataset. This technique identifies clusters of data points and it has been applied by k-medoids algorithm. The medoid of a cluster is assumed to be invariant that characterizes the data points of the cluster. The k-medoid is used to find out clusters from the given data and has high computation cost and not sensitive to noisy data. The number of clusters K the workload units will be assigned to is an input parameter of K-medoids. The medoid of a cluster is assumed to be invariant that characterizes the data points of the cluster. Points belonging to the same cluster are characterized by the same invariant.

We assume that the points belonging to the same cluster are characterized by the same invariant. Clusters are sorted by decreasing size, beforehand: likely invariants are deemed to be the ones representing larger clusters. Clustering is an unsupervised technique (i.e., it does not require labelled training data). The invariants obtained specify the values of all the attributes.

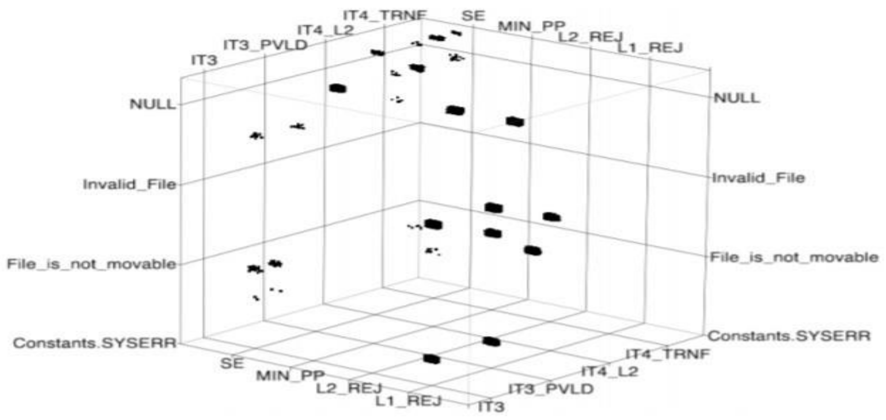

Fig.4.1.1 3D scatterplot of the workload units in the SaaS dataset

4.1.2 Association Rules

The second technique is frequent item set mining, which extracts frequently observed patterns in a database in the form of item sets or association rules. This technique is well known in the field of market basket analysis, where it is used to find out sets of products that are frequently bought together. We apply the association concept to values of attributes. Let B ={i1,...,im}be a set of items, any S ⊆ B an item set, and T the bag of transactions under consideration(a transaction is a set of items). The absolute support (the relative support) of S is the number of transactions in T (the percentage of transactions in T) that contain S. More formally, let U ={X ∈ T | S ⊆ T}be the set of transactions in T that have S as a subset (i.e., contain all the items in S and possibly some others). Then suppabs(S) = |U| = |{X ∈ T|S ⊆ T}| is the absolute support of S, and supprel(S) = |U| |T| ×100% is the relative support of S. Here |U| and |T| are the number of elements in U and T, respectively.

The support threshold (s) is an input of the algorithm: the smaller it is, the larger the number of association rules that will be returned by the algorithm. Association rules returned by either Apriori or GSP are assumed to represent an invariant. Rules are sorted by decreasing values of the support, i.e. by decreasing likelihood.

4.1.3 Decision Tree

A decision tree is a supervised technique and an ordered set of classification rules. Given a workload unit abstracted by the value of the attributes, the list is scanned until a rule is found that matches the attributes. We use Naïve Bayes algorithm which is based on bayes theorem with an assumption of independence among predictors. A naïve bayes classifier assumes that the presence of a particular feature in a class is unrelated to presence of any other feature. So it is very easy to build and particularly used for very large data sets. In this study, the rules in the list that aim to catch the correct workload units are deemed to be invariants, they are sorted by decreasing number of correct units they detect.

Lists some of the 91 classification rules obtained for the Google dataset with PART. For instance, a job where T=T2 and R=R0 is classified as KILLED regardless the value of the remaining attributes because it matches the rule at line2;similarly, by looking at line 4 and 6 it can be noted that a job where T=T0 and R=R0 and P=High and D=D2 is classified as FINISHED if (i) it has been run on the server type B (regardless the CPU usage) or (ii) its CPU usage has been C0 in the case the server type is C.

1. if (T=T2 and R=R1) then KILLED 2
2. else if (T=T2 and R=R0) then KILLED 3
3. else if (T=T0 and R=R0 and P=High and D=D2 and S=B) 5 then FINISHED 6
4. else if (T=T0 and R=R0 and P=High and D=D2 and C=C0 7 and S=C) then FINISHED 8
5. else if (P=MEDIUM) then FAILED 10
6. default FINISHED

Differently from clustering and association rules, decision list is a supervised technique because the model is learned f222s+rom a labeled dataset (i.e., beside the attributes matrix, the construction of the tree requires the knowledge of the label of each workload unit). In this study, the rules in the list that aim to catch the correct workload units are deemed to be invariants; they are sorted by decreasing number of correct units they detect.

5. Conclusion

Invariants can be mined for a variety of service computing systems, including cloud systems, web service infrastructures, data centres, enterprise systems, IT services , network services. The identification and analyses of their violations support a range of operational activities such as anomaly detection, capacity planning. The results provide suggestions to practitioners to establish the configuration of the mining algorithms and to select the number of invariants. A small number of invariants allows to reach a high coverage, i.e. they can characterize the most of the executions. Finally, we presented a general heuristic for selecting the likely invariants from a dataset. Finally, we presented a general heuristic for selecting a set of likely invariants from a dataset. All these results aim to fill the gap between past scientific studies and the concrete usage of likely system invariants by operations engineers.

6. References

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, Dynamically Discovering Likely Program Invariants to Support Program Evolution, IEEE Trans. Soft. Eng., 27(2001) 99–123.

[2] H. Chen, H. Cheng, G. Jiang, K. Yoshihira, Invariants Based Failure Diagnosis in Distributed Computing Systems, in Proc. 29th IEEE Int. Symp. Reliable Dis. Sys., (2010) 160–166.

[3] G. Jiang, H. Chen, and K. Yoshihira, Modeling and Tracking of Transaction Flow Dynamics for Fault Detection in Complex Systems, IEEE Trans. on Depen. Sec. Com., 3(2006) 312–326.

[4] G. Jiang, H. Chen, and K. Yoshihira, Efficient and scalable algorithms for inferring likely invariants in distributed systems, IEEE Trans. on Data and Knowledge Eng., 19(2007) 1508–1523.

[5] H. Chen, H. Cheng, G. Jiang, and K. Yoshihira, Exploiting Local and Global Invariants for the Management of Large Scale Information Systems, in Proc. 8th IEEE Int. Conference on Data Mining, (2008) 113–122.

[6] F. Frattini, S. Sarkar, J. Khasnabish, and S. Russo, Using Invariants for Anomaly Detection: The Case Study of a SaaS Application, in Proc. 25th Int. Symp. Software Reli. Eng. Work., (2014) 383–388.

[7] X. Chen, C.-D. Lu, and K. Pattabiraman, Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study, in Proc. 25th Int. Symp. Soft. Reli. Eng., (2014) 167–177.

[8] C. Pacheco and M. D. Ernst, Eclat: Automatic Generation and Classification of Test Inputs, in Proc. 19th Eur. Conf. Object Oriented Pro., Springer (2005), 504–527.

[9] C. Csallner and Y. Smaragdakis, Dynamically discovering likely interface specifications, in Proc. 28th Int. Conf. Software Eng. (2006) 861–864.

[10] L. Mariani, S. Papagiannakis, and M. Pezzè, Compatibility and regression testing of COTS-component-based software, in Proc. 29th Int. Conf. Soft. Eng. (2007) ACM.

[11] J. Cobb, J. A. Jones, G. M. Kapfhammer, and M. J. Harrold, Dynamic Invariant Detection for Relational Databases, in Proc. 9th Int. Workshop on Dynamic Analysis, *ACM* (2011) 12–17.

About The License