ACSIJ Advances in Computer Science: an International Journal, Vol. 2, Issue 5, No.6 , November 2013
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Designing Expert System for Detecting Faults in Cloud Environment

Marzieh Shabdiz[1], Alireza Mohammadrezaei [2] and Hossein Bobarshad[3]

[1] Engineering Department, Tarbiat Modares University
Tehran, Iran
m.shabdiz@modares.ac.ir

[2] Engineering Department, Tarbiat Modares University
Tehran, Iran
a.mohammadrezaei@modares.ac.ir

[3] Faculty of New Sciences and Technologies, University of Tehran
Tehran, Iran
hossein.bobarshad@ut.ac.ir

Abstract

Many fault detection techniques for detecting faults in rule bases system have appeared in the literature. These techniques assume that the rule base is static. This paper presents a new approach by designing Expert system for detecting faults in dynamic environment, such as cloud. Cloud resources are usually not only shared by multiple users but are also dynamically re-allocated per demand. Therefore, rules may be added/deleted in response to certain events happening in the integrated system being controlled by the rules. The approach makes use of spanning trees and Complementary sets to check a dynamic rule base for different kinds of faults underlying directed graph and devises a new method with scripting language on web based tools. This is performed as rules are being added to the dynamic rule base one at a time without the need to rebuild the structures and update rules and paths by expert system.

Keywords**:** *Dynamic Rule bases, Rule base Faults, Spanning Tree, Cloud Environment, Expert System.*

1. Introduction

Developing algorithms to detect rule-based systems against different kinds of faults within the context of large rule–based systems have attracted many of research efforts due to the important role of rule-based systems in various cloud environment, including Expert Systems (ESs), active database systems, and Information Distribution Systems (IDSs) to name a few [1,12].

One of the main concepts of cloud environment is providing almost unlimited resources for a given Service, automatically and dynamically, in a fully-virtual environment. Networks get new devices added to them, but they are seldom re-architected unless a completely new network is purchased. Networks often grow organically like spanning trees. As new nodes are added to a LAN environment the spanning tree evolves over time. Therefore, other nodes and routers in networks should be aware of this growth. One of The challenges of supporting routes in a cloud environment is resources that could be spread over multiple locations and using a transparent transport interconnected mechanism which maintains security and end-to-end segmentation [3].

If a dynamic rule base is fault free at a certain time, then deleting rules may generate unreachability faults, only by making some output vertices unreachable. Adding/deleting rules affect the rule chains in rule bases. Such rule bases are common in active database systems and information distribution systems, many rules may be added at a certain point in time and other rules may be deleted at other points in time.

Where rules are added, as new events occur in the system. In these events, the effects of errors may appear in the performance of these systems. Such faults may cause incorrect or undesired actions. Sometimes, these effects may be harmless, such as redundancy that may cause the systems' performance to be inefficient. On the other hand, contradiction faults may lead to incorrect conclusions [11, 12].

However, in such cases the designer must be knowledgeable of the presence of such faults and their consequences from the practical point of view. Many approaches and algorithms for fault detection have been presented and proposed in the literature.

The Expert System Validation Associate (EVA) program was developed at Lockheed [12]. EVA program was used to check for rule redundancy, inconsistency and contradiction. A decision-table-based processor for checking completeness and consistency in rule-based systems was presented in [11]. The COVER tool was presented in [8]. The tool was designed to build upon the best features of earlier systems. It is used to check rules based on a subset of first-order logic. A Petri-Net based approach for verifying rule bases was presented in [2].

A Transition Directed Graph (TDG), which represents rule sets, was presented in [8, 10]. TDG was used in the development of a set of algorithms to detect inconsistency, contradiction, circularity, Inreach-ability, and redundancy in chained inference rules. To provide those resources, the complete cloud architecture must be built with efficient tools, network, and storage resources [10].

These expert system employed different approaches for detecting some faults. Based on these approaches applications have been developed and used to inspect a rule-based system for known potential faults. This article covers the most frequent errors and how to correct them with design expert system into cloud platform such as Heroku and embedded control system such as Git and other language relative.

2. Rule-Based Systems Faults

A set of well-known faults that may appear in a rule Base is presented in [11]:

1) ***Redundancy/Subsumption***:
   Two rules conclude the same outcome from the same input data. A special case of redundancy is subsumption, where, two rules conclude the same outcome, but one has additional constraints, which may or may not be necessary.
2) ***Contradiction/Conflict***:
   Two rules conclude Different outcomes from the same input data.
3) ***Inconsistency***:
   An antecedent of one rule is mutually exclusive to the consequent of such rule (or a chain of rules).
4) ***Circularity***:
   The rule base contains a cycle inference chain, which may cause a backward-chaining inference engine to enter an endless loop.
5) ***Unreachability***:
   Unreachability occurs if there is no path between any two given vertices.

3. Expert System and Implementing

Many transformation techniques for rule bases have Been suggested in the literature. In this paper, subject essential is implementing expert system on network and integration systems on cloud platforms. Heroku is a polyglot cloud application platform. With Heroku, no need to think about servers at all. Heroku lets us deploy, run and manage applications written in Ruby, Node.js, Java, Python, Clojure and Scala.

Git is a powerful, distributed version control system that many developers use to manage and version source code. The Heroku platform uses Git as the primary means for deploying applications. An application is a collection of source code written in one of these languages, perhaps a framework, and some dependency description that instructs a build system as to which additional dependencies are needed in order to build and run the application. No need to make many changes to an application in order to run it on Heroku. One requirement is informing the platform as to which parts of application are run able. We'll use Git to deploy apps to Heroku in one command. We'll build and run the source application, handling compilation, dependencies, assets and executables so we can focus on code. Code pushed to the heroku remote will be live and running on the platform.

In this approach, a rule base is modeled as a Petri-Net where parameter-value pairs corresponding to places and rules are analogous to transitions. Then the transition/place relationship modeled in a Petri-Net can be summarized in the form of an incident matrix. Decision-table-based processors were presented in [3]. In the figure 1 you can see this situation of nodes in Petri-Net model.

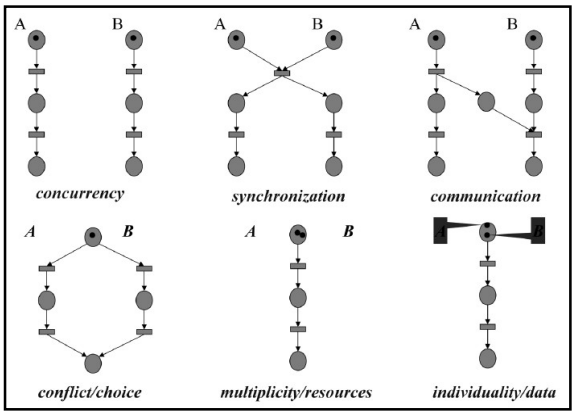

Fig. 1 positions of nodes in Petri-Net model.

In this approach, a decision table is created from the Rules in the rule base systems. A directed-graph-based approach was presented in [6], where the rule base is modeled as a directed graph and the process of anomaly detection is reduced to reachability

among nodes. Each node saves such as separate file in tree structure of Git. A transition-directed-graph-based approach, which is similar to is presented with Simultaneous connections feature in heroku. The *herokuapp.com* routing stack allows many concurrent connections to web dynos [10, 15].

In this paper, we use the transformation technique where the dynamic rule base is modeled as a directed Graph as new rules are being added to the dynamic rule base. In this directed graph, nodes correspond to Propositions and rule identifiers and edges correspond to the rules. Each rule has a rule identifier that in model these nodes appear with MAC address of devices in networks.

A spanning tree/forest will be devised by using Kruskal's like algorithm. Tree structure of GIT Satisfies this problem. During the operation of the algorithm, Complementary sets will be generated. These sets will be used for detecting various kinds of faults while the dynamic rule base is being updated. Spanning Tree's job is to prevent loops from forming. It does this by learning about sub-optimal paths to the root and placing these less desirable links into blocking mode. If there are multiple parallel paths between nodes, then one of them would be selected to be in blocking mode to prevent a loop between the two nodes. This leaves all nodes in the environment using the default root priority. If all nodes have the same root priority, the node with the lowest MAC address will be selected for adding. More complex situations can arise. This would make having multiple links only good for failover for the primary link and not provide increasing bandwidth along that path. The Merge Conflicts feature in Git tool can solves them. That means every edge will pull in the state of the path file on the other tree into the working tree, dynamically. If occurs conflicting in the same file, Git will knowing it and commits again after resolving them. Due to the fact that spanning trees are not unique, such a devised rule base may not be unique. In this case, for every tracked file in tree, Git records information such as its name, number, type, conditions, creation time and last modification time in a file known as the index. To determine whether a file has changed, Git compares current states with those cached in the index. If they match, then Git can skip reading the file again. In addition, for detecting fault pattern with rules, we could routes paths with routing feature in Heroku because inbound requests are received by a load balancer that offers HTTP and SSL termination from here they are passed directly to a set of routers. The routers are responsible for determining the location of nodes and forwarding the HTTP request to one of them. A request's path from the end-node through the Heroku infrastructure to the application allows for full support of HTTP 1.1 features such as chunked responses, long polling, and using an a sync web server to handle multiple responses from a single web process [3, 7, 15]. Heroku executes applications by running a command specified in the Procfile, that is written whit ruby. Also, rules saved in file with extension .rb in git branch. The looping and conditional constructs have the same interpretation as in ruby language. Ruby is an interpreted scripting language for quick and easy object-oriented programming [14].

Features of ruby are:

- Ability to make operating system calls directly
- Powerful string operations and regular expressions
- Immediate feedback during development
- Variable declarations are unnecessary
- Variables are not typed
- Syntax is simple and consistent
- Memory management is automatic
- Everything is an object
- Classes, inheritance, methods, etc.
- Singleton methods
- Mix in by module
- Iterators and closures

Therefore, the focus here is on adding new rules to the dynamic rule base and designing Expert system. Expert systems are part of a general category of computer applications known as artificial intelligence. To design an expert system, one needs a Knowledge engineer, an individual who studies how human experts make decisions and translates the rules into terms that a computer can understand. An expert system has a unique structure, different from traditional computer programming [5]. Components of Expert system and their relationships as shown in Figure 4.

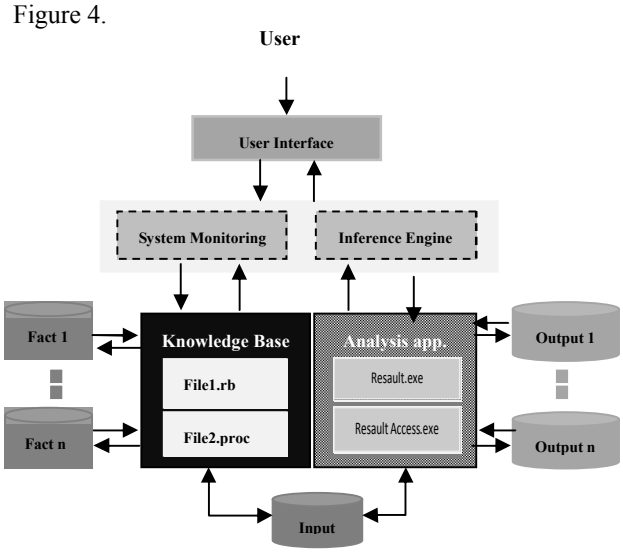

Fig. 2 Component of Expert System

It is divided into two parts, one fixed, independent of the expert system: the inference engine, and one variable: the knowledge base.
An Expert system stores data in its knowledge base as production rules. To query the system involves a consultation being run; whereby the user is asked questions via the user interface until eventually advice is provided. An expert system shell (Git Bash) represents data by storing it in its knowledge base as a series of production rules [1, 19].
In Figure 3 we associate scenarios that have access to the public cloud with all requirements to input expert system. The scenarios are used for experimental environment and experimental operations. They are translated into parameters for routing simulation. The metrics are used to measure performance variability of particular cloud services and setting parameters for cloud environment simulation.

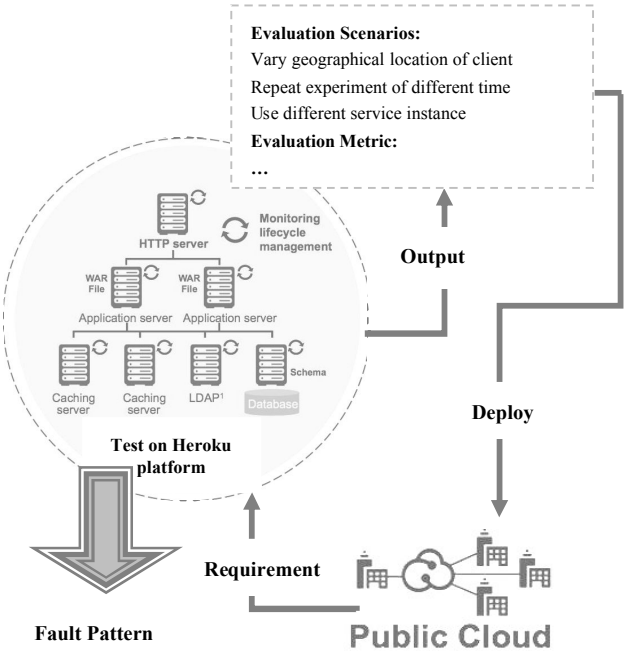

Fig. 3 Association Expert system with cloud

These are many requirements for input expert system such as condition and position nodes. The requirement is translated into inputs to the expert system.
When create an application on heroku platform, it associates a new git remote, typically named Heroku, with the local git repository for application written in ruby. Deployment then is about using git as a transport mechanism, moving application from local system to Heroku. When the Heroku platform receives a git push, it initiates a build of the source application. To build mechanism is typically language specific, such as ruby [15, 16].

4. Fault Detection Algorithm

A spanning tree of an undirected graph G is a tree Formed from graph edges that connects all the vertices of G. formally, let $G = (V, E)$ be an undirected connected graph. A sub graph $T = (V, E')$ of G is a spanning tree of G if T is a tree. An interesting property of a spanning tree is that it represents the minimal subgraph G′ of G such that $V(G') = V(G)$ [10].
By minimal, we mean the one forest is new rules are being added to the dynamic rule base. Initially, there are $|V|$ single-node trees. Adding an edge merges two trees into one. It turns out to be simple to decide whether edge (u, v) should be accepted or rejected. The appropriate data structure or approach is the union/find algorithm. This approach, as presented in DFP_err_Detection algorithm in below.

Algorithm 1. DFP _err_Detection
Require: *r, FP, C, R, S*
1: Chk _Redundancy&Circularity*(r, FP, C, R, S)*
2: **If** *r* contains exclusive vertices **then**
3: Chk _Inconsistency&Contradiction*(r, S)*
4: **end If**
5: Chk _Unreachability*(r, S)*

DFP_err_Detection algorithm checks the current Rule base when a new rule is added as follows:
1.It calls the algorithm Chk_Redundancy& Circularity*(r, FP, C, R, S)* to check if it causes a redundancy or circularity fault pattern. In this call, *r* is the new rule, *FP* is the current fault free dynamic rule base, *C* is the set of circularity fault patterns, *R* is the set of redundancy fault pattern, and *S* is the Complementary sets. This gives algorithm 2.

Algorithm 2. Chk_Redundancy&Circularity
Require: *r, FP, C, R, S*
1: **for** all edges comprising rule *r* **do**
2: Choose the next edge <*u,v*>
3: Delete <*u, v*> from *r*
4: *u.set* = find *(u, S), v.set* = find *(v, S)*
5: **if** *u.set* <> *v.set* **then**
6: Add <*u, v*>, set.union *(S, u, v)* {to FP}
7: **else if** find.path*(u, v, S)* == *'C'* **then**
8: Add *r* to *C* {cycle in the directed graph}
9: **else** Add *r* to *R* {cycle in the undirected graph}
10: **end if**
11: **end for**

2. It checks if the new rule *r* contains exclusive Vertices, then calls the algorithm Chk_Inconsistency & Contradiction*(r, S)* to perform this check as shown in algorithm 3.

Algorithm 3. Chk _Incosistency&Contradiction

```
Require: r, S
for each vertex v in r do
  if   v is an exclusive vertex then
            root.v = find (v, S)
            root.vp = find (vp, S)
    if    (root.v == root.vp)  then
      while (S[root.v]!=0 && S[root.v]!=root.vp)
          root.v = S[root.v]
      end while
     end if
     if  (S[root.v] == root.vp) then
            Display "r causes Inconsistency"
    else    Display "r causes Contradiction"
     end if
    end if
  end for
```

3.The algorithm calls the Chk_Unreachability *(r, S)* to check for potential unreachability faults with algorithm 4.

Algorithm 4. Chk _Unreachability

```
Require: r, S
for each pair of vertices (x, y) in r do
      root.x = find (x, S)
      root.y = find (y, S)
      if    (root.x == root.y)  then
      while (S[root.x]!=0 && S[root.x]!=root.y)
            root.x = S[root.x]
       end while
      if  (S[root.x] == root.y)  then
            Display "r causes Unreachability"
      end if
     end if
  end for
```

The set.union (*S*, *r1*, *r2*) algorithm implemented by (*S*[*r2*] = *r1*) maintains the direction of the edges in the original graph, by using the find algorithm as shown in algorithm 5. Also it specifies the root of the set to which a vertex belongs.

Algorithm 5. Find

```
Require: r, S
          If (S[x] <= 0) then
                Return x
          else
                Return (find(S[x], S))
          end if
```

To determine whether an edge <*x*, *y*> creates a cycle in graph, the algorithm find.path, as shown in algorithm 6, can be used to check. If two nodes *x* and *y* are on the same path in a certain Complementary set *S*. If *x* is reachable from *y*, then they are on the same path and adding an edge <*x*, *y*> does not create a cycle. However, it indicates that there is another path that connects *x* to *y*. Thus there is a redundancy fault pattern. On the other hand, if *x* is not reachable from *y*, then *x* and *y* are not on the same path and adding an edge <*x*, *y*> creates a real cycle. Thus, this is a circularity fault pattern.

Algorithm 6. Find_path

```
Require: x, y, S ,R, C
          while (S[x] != 0 & S[x] !=y)
                x = S[x]
           end while
           If  (S[x] == y) then
                Return R
           else
                Return  C
          end if
```

As mentioned earlier, edges between nodes are paths so that can be used by buffering features in heroku. As a result, each router buffers the header section of all requests, and then delivers them to dyno's web server as fast as internal network. The dyno is protected from slow clients until the request body needs to be read. If need to protection from clients transmitting the body of a request slowly. This will have the request headers available to make a decision as to when to drop the request by closing the connection at the dyno [13, 15]. This will prevent the creation of duplicate path and redundancy.

The process of detecting various types of faults by Formulating faults as reachability problems in the graph-based representation should be followed by a checking rule's identifier vertices that comprise a certain path in the fault patterns. Although the formulation gives set of condition for the existence of various kinds of faults in a rule base, the condition is not sufficient as long as rules with multiple antecedents are considered. To deal with this additional issue, we can estimate the in-degree of the rule identifier vertices in the paths of the fault pattern to specify whether a certain fault satisfies the conditions of representing a real fault. Once these sets of faults have been considered, it would be relatively simple to check for the rest of the well-known faults in a straightforward manner. An inconsistency fault occurs when an antecedent of one Rule is mutually exclusive to the consequent of chain of rules [19, 20]. This means that starting from a vertex (e. g., *A*), we can reach to its exclusive vertex ¬*A*. To check for this kind of anomaly, we first consider the set of exclusive vertices, and then we need only to check whether the exclusive vertices are in the same Complementary set and there is a path between them. A contradiction/conflict fault pattern occurs when two rules conclude different outcomes from the same input data. This means that starting

from one vertex /proposition (e. g. *A*) We can reach to two exclusive vertices (e. g., *C* and ¬*C*). To check for this kind of fault, we first determine the set of exclusive vertices, and then we only need to check whether the exclusive vertices are in the same Complementary set and none of them is the root of the set. If they are in the same set and none of them is a root, then there is a contradiction anomaly, otherwise there is no contradiction anomaly. Unreachability faults occur if there is no path between any two given vertices. To check for that, we first specify whether the two vertices are in the same Complementary set or not. If true, we determine whether there is a path between them, and in this case there is no unreachability anomaly. The benefit of our approach is its ability to detect faults as the dynamic rule base is being updated. If a rule *r* is added to the dynamic rule base, then the new dynamic rule base can be verified against various faults without rebuilding any structures.

5. Algorithm Computational Complexity

DFP_err_Detection algorithm is a variation of Kruskal's spanning tree algorithm without sorting. Therefore, it has a worst-case complexity of *O(nlogn),* where *n* is the number of rules being added to the dynamic rule base.

It calls Chk_Inconsistency& Contradtion algorithm *n* times. The *for* loop for the edge components of each rule is assumed to be constant with a complexity of *O(1)*. The complexity of find is *O(logn)*.Thus, the worst-case complexity of checking for all redundancy and circularity faults is *O(nlogn).* Also this algorithm checked inconsistency and contradiction fault patterns with *O(logn)* complexity. Finally, the worst-case complexity of checking for unreachability faults is *O(n)*. Our approach improves a complexity over Petri-Nets approach, where it complexity for detecting inconsistency and redundancy is *O* (*n2*) [2, 11].

6. Experimental Results

Generally, an empirical study is an integral part of the analysis of algorithms. To study the experimental Complexity of our algorithms, the fault detection algorithms were implemented in ruby and executed on heroku platform. Heroku treats logs as streams of time-ordered events, and collates the stream of logs produced from all of the processes running in all dynos, and the Heroku platform components, into the Logplex a high-performance, real-time system for log delivery. Domains and DNS configuration feature adds experimental WebSocket support to our *herokuapp.com* domain, custom domains and custom SSL endpoints and Maintaining multiple environments. Also, each router maintains an internal per-app request queue. When processing an incoming request, a router sets up an 8KB receive buffer and begin reading the HTTP request line and request headers. It could be sent up to 1MB response in size before the rate at which the client receives the response will affect the dyno even if the dyno closes the connection, the router will keep sending the response buffer to the client. Heroku lets us run application with a customizable configuration and ruby is best choice in this case. Also, in this paper, we used git to keep data in the .git/objects subdirectory. Git heuristically ferrets out renames and copies between successive path files and determine whether a file has changed, Git compares its current status with those cached in the index. If they match, then Git can skip reading the file again [3, 9, 7, and 15]. Some of factors to choose solution in designing expert system are presented in table 1.

Table 1: The main factors in choosing solution

Factors		
Heraku	***Git***	***Ruby***
Logging and monitoring	Merge Conflicts	interpreted scripting language
HTTP routing	Secret Source	quick and easy
Domains and DNS configuration	Ultimate Backups	object oriented programming
Timeouts	Light-Speed Multitask	multiple precision integers
Keep - alive	Branch Wizardry	exception processing model
Routing	Dirty Work	dynamic loading
Request distribution & Request queuing	Quick Fixes	threads
Simultaneous connections	Remote Branches & Trees	Iterators and closures
Request buffering	Integrity	feedback
Memory & swap, CPU load averages	Intelligence	Mix in by module

Also, we added rules, A number of added rules generate a set of faults, and the algorithms detected all these faults. A performance profile, which represents the amount of time the algorithms

consume. This has been compared with the Petri Nets algorithm. The performance measurements have shown in table 2. That our approach outperforms and Faster than the Petri Nets approach.

Table 2: Evaluation Metric

Evaluation Metric	*MIN*	*MAX*
CPU load average	15 Minute	1 Minutes
Resident Memory (RAM)	(25% of total System memory)	7.5 GB
Disk Cache Memory	1/2 Mem	3.5 GB
Swap Memory	1/3 Mem.	2048 MB
Total Memory (GB) (Sum of resident, cache and swap memory)	….	…..
Pages Written to Disk	200	1000
Pages Read from Disk	400	1000
Repository Size	600MB	1000 MB
Rules	10	1000
Run Speed (ms) (Our approach)	0.00456789 ms	108.0673211109 ms
Run Speed (ms) (Petri-Net approach)	9.0009765 ms	376.0005630911 ms

A set of 4 test cases, consisting of 10, 100, 500, and 1000 rules were considered. Each test case uses a randomly-generated set of rules with a number of faults resulting from the random generation of the rule sets.
The result of each case is plotted for our approach and the Petri Nets approach as shown in Figure 4.

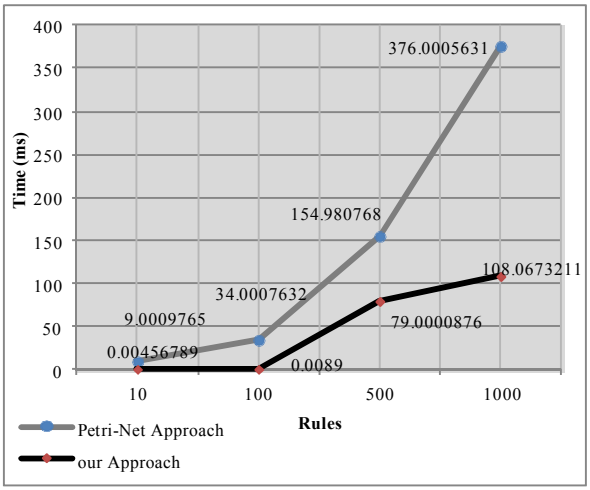

Fig. 4 Result of Comparison Approaches

The performance measurement confirms the earlier theoretical analysis of the various algorithms. Using the timing data, the shapes of the curves are determined.

7. Conclusions

A new approach, based on spanning trees for verifying dynamic environment is presented. The approach uses an algorithm for planning Expert System that checks for various fault patterns in cloud platforms and generates patterns. Addition, an empirical study, which confirms the theoretical analysis, is also presented.
Thanks For the listing name of below for support and encouragement us:

Mr. Amirhossein Mortazavi, Mr. Sam Joseph, Mr. Hossein Bobarshad, Mr. Mehdi Amiri Kordestani, Mr. Hossein Jafari Farhani, Mr. Hossein Shabdiz .

References

[1] A., S. R. a. B. A. P., 1992. Verifying Expert Systems: A Logical Framework and a Practical Tool. *Expert Systems with Applications,* Volume 5, pp. 421-436.

[2] Agarwal, R., 2002. A Petri-Net Based Approach for Verifying the Integrity of Production Systems. *International Journal of Man-Machine Studies,* Volume 36, pp. 447-468.

[3] Amazon, 2013. *Amazon Elastic Compute Cloud*. [Online] Available at: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html#EC2_ConnectToInstance_Linux [Accessed 15 10 2013].

[4] Anon., 16-August-2013. *Engineering Software as a Service: An Agile Approach Using Cloud Computing.* 2 ed. s.l.:Amazon.

[5] Co-Researchers, C. K. a., 2012. Recent Advances in Expert Systems. In: s.l.:s.n., pp. 1-6.

[6] D, T. S. a. V. L. P., 1997. Evaluation of Verification Tools for Knowledge-Based Systems. *International Journal of Human-Computer Studies,* Volume 47, pp. 629-658.

[7] inc., G., 2013. *github.* [Online] Available at: https://github.com/mshabdiz [Accessed 5 10 2013].

[8] Kassem, M. C. S., Mar 2013. A heuristic method for solving reverse logistics vehicle routing problems. *International Journal of Industrial and Systems Engineering.*

[9] lynn, B., 2013. *Git Magic.* 3 ed. s.l.:Amazon.

[10] M., N. D. a. K., 1991. Verification of Rule-Based Knowledge Using Directed Graphs. *Knowledge*

Acquisition, pp. 339-360.

[11] N, R. D. a. R. D. A., 2007. Structural and Syntactic Fault Correction Algorithms in Rule-Based Systems. *International Journal of Computing and Information Sciences (IJCIS),* Volume 2, pp. 1-12.

[12] R., C. J. a. C. C. S., 1987. *Validation of Knowledge-Based Systems.* Arlington, s.n.

[13] Road, N. D. J., 2011. *Heuristic algorithms for container pre-marshalling problems.* Taiwan, Department of Logistics Management, National Kaohsiung First University of Scien.

[14] Ruby, S., 2013. *Agile Web Development with Rails 4.* 4 ed. s.l.:The pragmatic bookshelf.

[15] Services, H. B. C. A., 2013. *heroku.* [Online] Available at: https://www.heroku.com/ [Accessed 8 10 2013].

[16] Shaw, Z. A., 2013. *Learn Ruby The Hard Way.* 2 ed. s.l.:learncodethehardway.

[17] Wardeh, M. B.-C. T. C. F., 2006. *Dynamic Rule Mining for Argumentation Based Systems.* Liverpool, L69 3BX.

[18] W, Y. & de Silva, W., 2008. Multi-robot box-pushing single-agent qlearning vs team q-learning. *IEEE/RSJ International Conference on Intelligent Robots and Systems.*

[19] Y., H., 2004. Detecting Faults In Chained-inference Rules in Information Distribution Systems. *Information Technology and Engineering.*

[20] Zhu, D.-Y. W. K.-J., 2013. Application of hybrid GA–SA heuristics for single-job production–delivery scheduling problem with inventory and due date considerations. *International Journal of Industrial and Systems Engineering.*

Marzieh Shabdiz received her BS in Computer Software Engineering with high honors from Islamic Azad University of Najafabad, Isfahan, IRAN, in 2008, her MS in Information and Communications Technology from Tarbiat Modares University, Tehran, IRAN in 2012. Her research is focused on the cloud environment and intelligent methods to planning solutions. Her research interests include Database and knowledgebase systems, Algorithms, Enterprise System Architecture, Virtualization, Security, Cloud computing (IaaS), Software as a service (SaaS) and GNU/Linux Administration.

Alireza Mohammadrezaei received his BS in Computer Hardware Engineering with high honors from Islamic Azad University South Tehran Branch, Tehran, IRAN, in 2003, his MS in Information and Communications Technology from Tarbiat Modares University, Tehran, IRAN in 2012.

Hossein Bobarshad he is an Assistant Professor at Tarbiat Modares University, Tehran, IRAN. He is also a Member of the Institute of Electrical/Electronics Engineers (IEEE).