PARALLEL VERIFICATION EXECUTION WITH VERIFY ALGEBRA IN A CLOUD ENVIRONMENT

Kan Luo[1] Siyuan Wang[1] An Wei[2] Wei Yu[1] Kai Hu[1]

[1]School of Computer Science and Engineering, Beihang University, Beijing, China
[2]China Mobile (Hangzhou) Information Technology Co.,Ltd

ABSTRACT

Soft-as-a-Service (SaaS) is a software delivery model that contains composition, development and execution on cloud platforms. And massive SaaS applications need verifying before deployed. To get the verify results of a large quantity of applications in a tolerate time, verify algebra (VA) is used to cut down the number of combinations to be verified. VA is an effective way to acquire the verify statue by using previous results. In VA, the verify result is calculated without knowing the process of verification. In this way, the verification task can be distributed to servers and executed in any order. This paper proposes method called component disassembly tree to decompose a complex SaaS application. And designs a parallel verification framework in cloud environment. The Optimization of execution is discussed. The proposed parallel schema is simulated in MapReduce.

KEYWORDS

Verification, SaaS, Components Combinations

1. INTRODUCTION

Software as a Service (SaaS) is a new way for software development and delivery based on the cloud platform. In SaaS, MTA (Multi-Tenancy Architecture) [2] is a key feature. It allows all tenants software to share the same code on the basis of configuration data stored in databases or data stores. Each element represents a unique component in the SaaS system, and a set of components represents a tenant application. Tenants customize their applications using components stored in the SaaS database according to their individual requirements. However, a tenant application can be insecure or vulnerable. For those applications, some interesting properties, such as dead-lock and safety need to verifying before deployed on the cloud. With the number of components in database increasing, the workload of verification grows up greatly. For instance, supposing there are 4 layers (GUI layer, workflow layer, service layer and data layer) has 10^5 components respectively and there will be 10^{20} ($10^5*10^5*10^5*10^5$) possible combinations in total. It is impossible to get 10^{20} different application verified one by one. Combinatorial verifying is new verifying technique to verifying component-based applications. It verifies combinations among components which has been verified individually.

Formal method can be applied in a cloud environment leveraging the computing power offered. We have proposed the concept, VaaS (Verification-as-a-Service) on MTA, a scalable cloud-based on-demand service that uses formal models for verification. The VaaS on MTA can verify SaaS software and address behaviour, performance, and attribute aspects of software models, while it has features of SaaS software such as automated provisioning, scalability, fault-tolerant computing, and concurrent processing[2]. In a cloud environment, different combinations can be allocated to different processors for execution in parallel. One simple way to perform combinatorial verifying in a cloud environment is: split the verification tasks, then allocate the tasks to different servers in cloud, finally summary the results. However, this is not efficient. While computing and storage resources have increased significantly, the number of combinations to be considered is still too high. Verifying all of the combinations in a SaaS system with millions of components can consume all the resources of a cloud platform.

In our previous work, we study the rule of merging the verifying status of combinations and propose a Verify Algebra System [17] to cut down component combinations to be verified. Verify algebra is an algebraic system, which defines the five statuses of verify results and four operations of merging previous verify results. According to the combinatorial structure, our proposed VA can reduce the number of combinations to be verified by using existing verification statuses and then getting the results of unknown statuses by algebraic computation. In VA, the verify status record the verify result of a certain combination. we calculate the verify result without knowing the process of verification. And the verify results are merged according to the rules of VA and the results will not be affected by the processing order and merging order. In this way, we propose a parallel and asynchronous computing mechanisms such as MapReduce, automated redundancy and recovery management, automated resource provisioning, and automated migration for scalability.

Our contribution can be summarized as: we put forward a schema that leveraging Formal Method with the computing power offered by cloud environment to check software correctness. Further, we propose a new verification framework with VA and shared databases. All verify results are saved in shared databases. the process of verification is designed where previous verifying results are used to get unknown combinations status.

This paper is structured as follows: Section II discusses the related work; Section III introduces the component disassembly tree. Section IV and Section V discusses VA parallel execution and analysis; Section VI illustrates TA experiments using the proposed solutions; and Section VII concludes this paper

2. Related work

2.1. Verification-as-a-Service

VaaS on MTA has been designed in[1]. VaaS is an architecture that can be used to verify models, similar to SaaS, beneficial from the computing power offered in a cloud. A VaaS hosts verification software in a cloud environment, and these services can be called on demand, and can be composed to verify a software model. In VaaS, Bigragh is selected as the modelling language for illustration as it can model mobile applications. A Bigraph models can be verified by first converting it to a state model, and the state model can be verified by model-checking tools. The VaaS services combination model and execution model are also presented in[8].

In a SaaS, tenants can have their customized applications stored in the SaaS databases, and often the applications are not stored as a unit in the databases. Instead, each tenant application is decomposed into its GUIs, workflows, services, and data components, and each component is stored in the database together with components of the same kinds[9]. For example, a SaaS GUI database contains all the GUI components used by all the tenants. The MTA VaaS design follows the SaaS design, for example, it has databases to store verification software, models to be verified, and verification results; it provides customization support; etc.

Essentially, an MTA VaaS is like a SaaS except the principal task is for software verification rather than general computing. A VaaS also has following unique features:

- Only formal verification software is stored in a VaaS;
- As formal verification often involves model transformation, thus a VaaS may contains transformation software;
- A VaaS also contains software for parsing formal models; and
- A VaaS may support incremental verification where subsystems are verified before whole systems are verified. Support for incremental verification includes storing model architecture and intermediate verification results, and algorithms to select only those compositions or combinations that need to be verified.

A tenant can develop a new verification application, i.e., a tenant application, by identifying and reusing GUIs (G), Workflows (W), Services (S), and Data components (D) in the VaaS database. All selected components can be linked, and then compiled to produce executable code.

In VaaS, however, there are too many compositions to be verified. Because the number of verification tasks will grow exponentially as complexity of tenant application grows. To address this issue, verification algebra rules are introduced into VaaS Architecture.

2.2. Verify Algebra

In VA, each combination can be in one of the following five states:

- ***Infeasible** (X):* some components are not permitted to be combined. For example, there would be a conflict when two components do same things but cause different results such as two GUI components, one of which paints the background BLUE but the other paint RED.

- ***Failed (F)***: Verification to combination on a certain property is not passed.

- ***Passed (P)**:* Verification to combination on a certain property is passed.

- ***Irrelevant (N)***: For some combinations which are impossible to be created, but is still feasible to be verified, so there is no need to verify these combinations.

- ***Unknown (U)***: the status of a combination is certain but not currently known.

Verify algebra method is capable of reducing the number of combinations to be verified by using existing verification status and then getting the results of unknown status by algebraic computation. To be specific, the verification state of one combination on a certain property is $V(Com_1,p)$ and the state of another combination is $V(Com_2,p)$, we can determine the $V(Com_1 \cup Com_2,p)$ from $V(Com_1,p)$ and $V(Com_2,p)$. To do this, four kinds of binary operations are defined as follows:

(1) Rules for operator $\otimes$

One combination contains many different sub-combinations. The status of one combination is composed by merging the result of its sub-combinations. The operation $\otimes$ merges the ***passed*** sub-combinations.

(2) Rules for operator ⊕

The operation ⊕ merges the ***failed*** sub-combinations

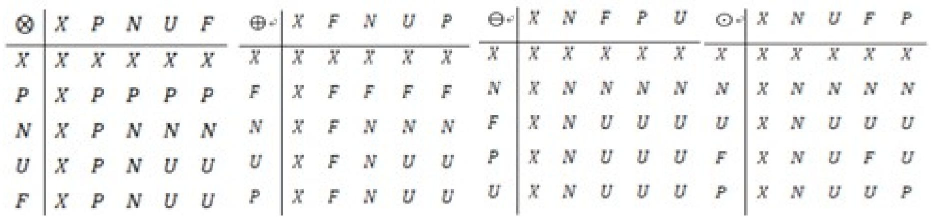
.

(3) Rules for operator ⊖

the operator ⊖ combines part of the proper subsets of one combination.

(4) Rules for operator ⊙

The operator ⊙ combines all the proper subsets of one combination.



⊗	X	P	N	U	F
X	X	X	X	X	X
P	X	P	P	P	P
N	X	P	N	N	N
U	X	P	N	U	U
F	X	P	N	U	U

⊕	X	F	N	U	P
X	X	X	X	X	X
F	X	F	F	F	F
N	X	F	N	N	N
U	X	F	N	U	U
P	X	F	N	U	U

⊖	X	N	F	P	U
X	X	X	X	X	X
N	X	N	N	N	N
F	X	N	U	U	U
P	X	N	U	U	U
U	X	N	U	U	U

⊙	X	N	U	F	P
X	X	X	X	X	X
N	X	N	N	N	N
U	X	N	U	U	U
F	X	N	U	F	U
P	X	N	U	U	P

Fig. 1. Combination results

2.3. Workflow Patterns

In SaaS, components are combined by single or multi patterns. The combination patterns of components are mainly expressed by the workflow. The pattern of workflow has been discussed by Van Der Alast et al. in[13]. They classify workflow patterns into six categories, namely Basic Control Flow Patterns, Advanced Branching and Synchronization Patterns, Structural Patterns, Patterns involving Multiple Instances, State-based Patterns, and Cancellation Patterns. In the six classes of workflow patterns, five patterns can be expressed by Basic Control Patterns. Furthermore, tenant applications of multi-patterns in SaaS platform can be disassembled into multi-tiered combinations that contains only the Basic Control Patterns. For this reason, emphasis of analysing combination patterns will be put on Basic Control Patterns. There are five kinds of workflow patterns in Basic Control Patterns.

- **Sequence:** The sequence pattern is the most common pattern used to model consecutive steps in a workflow process. A component in TA is enabled after the completion of another component in the same workflow pattern process, as **Error! Reference source not found.** (a).

- **Parallel Split:** Multiple components are simultaneously enabled after the completion of another component, thus allowed to be executed in parallel or in any order, as **Error! Reference source not found.** (b).

- **Synchronization:** Synchronization is a point in workflow process where multiple parallel components converge into one single component, thus synchronizing execution of multiple components, as **Error! Reference source not found.** (c).

- **Exclusive Choice:** One of several components is chosen based on a decision or workflow control data in pattern of Exclusive Choice, as **Error! Reference source not found.** (d).

- **Simple Merge:** The completions of two or more alternative components come together without synchronization in pattern of Simple Merge. In other words, the merge will be triggered once any of incoming transitions are triggered, as **Error! Reference source not found.** (e).

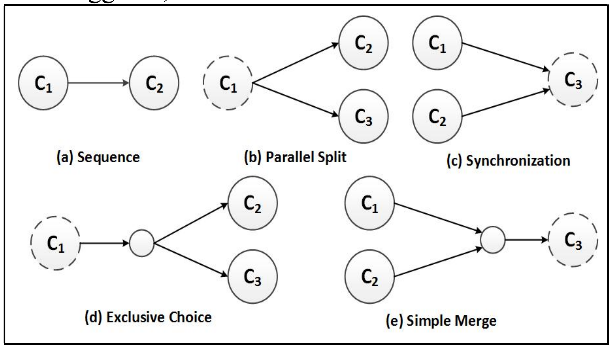


Fig. 2. Basic workflow patterns

3. Component Disassembly Tree

3.1. Abbreviation and Definition

TA: Tenant application
C: Component, a single component in SaaS databases, the basic unit in components combination. Supposing that all single components are tested or verified to be correct.
Com: Combination, constructed by more than one component.

Definition Ⅰ: (*Sub-Combination*) If the components set Com comprises Com', it can be claimed that combination Com' is a sub-combination of combination Com. Sub-combination is defined as $Com' \subset Com$.

Definition Ⅱ: (*Similar Sub-Combination*) If $\boldsymbol{Com' \subset Com}$ and the work flow pattern of $\boldsymbol{Com'}$ is the same as $\boldsymbol{Com}$, It can be said that $\boldsymbol{Com'}$ is a similar ***Sub-Combination*** of $\boldsymbol{Com}$. ***Sub-Combination*** is defined as $\boldsymbol{Com' \sqsubset Com}$.

In practical applications, most ***TA***s are under hybrid patterns. For ***Com*** under hybrid patterns, it is complex to get its sub-combinations with same basic pattern and therefore, combination verification algebra can be directly used in this case. To solve this problem, a method is proposed that ***TA*** is disassembled and substitute by combinations with more simple patterns. In this method, disassembly means splitting ***TA*** into several sub-combinations and substitution means replacing atomic sub-combination with component.

The combination containing only one kind of workflow pattern is single-patterns combination. Multi-patterns combination is the components combination that contains more than one kind of Basic Control patterns. The rules of Single-pattern combination can be generalized to multi-patterns. To do this, the component disassembly tree is proposed. Component disassembly tree is a tree whose root node is ***TA*** itself and whose leaf nodes are components and other nodes of which are sub-tenant applications or combinations of ***TA***.

The component disassembly tree of ***TA*** shown above is shown in **Error! Reference source not found.**

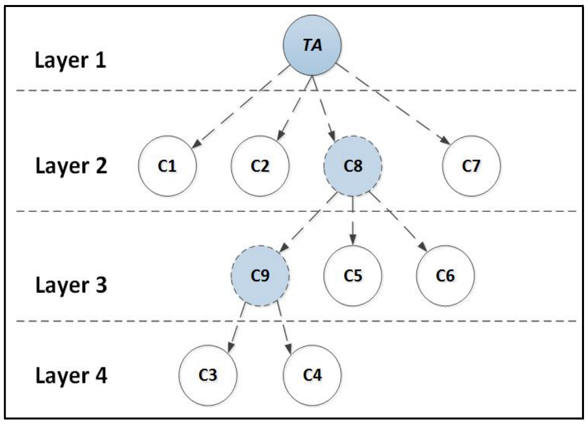


Fig. 3. Component disassembly tree of TA

A method is designed to automatically disassemble workflow graph of ***TA*** to a component disassembly tree. Supposed that the workflow graph has no circles and only has a source vertex and a destination vertex. The process of the method is illustrated as follows.

StepI: Initialize the component disassembly tree to contain a root with value ***TA***.

StepII: Find all paths of the workflow graph using DFS.

StepIII: Search paths from head and tail to find the same nodes. Make the nodes found to the child nodes of root. And remove the nodes found from all paths. If the paths are empty, end process. otherwise go to ***StepIV***.

StepIV: Add a child of root with value ***Com***

StepV: Sort and then group the paths by first elements. For these groups, if a group only has one path and the path only has one node, make the node to be a child of ***Com***. Else, add ***Com*** a child node with value ***Com***, and regard this ***Com*** node as root and the group as paths, go to ***StepIII***.

After component disassembly tree of tenants' application is derived, the process of combination verification algebra under hybrid patterns can be performed as follows:

Step1**:** Get component disassembly tree of ***TA***.
Step2**:** Check if the depth of component disassembly tree is more than 2. If the depth is no more than 2, it means that ***TA*** is under single pattern and go to ***Step5***. Otherwise, go to ***Step3***.
Step3**:** Choose leaf nodes which have same parent node from bottom of the tree. It is obvious that their parent is sub-tenant application consisting of these leaf nods and the parent node is under single pattern. Therefore, the verification status can be derived via the process represented in last section. After that, remove these leaf nodes from the tree until the parent node has no child nodes and its status is known.
Step4**:** Check if there is node at the bottom layer. If yes, continue ***Step3***. Otherwise, decrease depth of the tree by 1 and go to ***Step2***.
Step5**:** the verification status of ***TA*** can be derived via the process represented in last section. End the process.

4. Parallel Verification Framework

When tenant application is submitted to SaaS platform, component combination verification service will use component combination algebra to verify if this application can satisfy some specific property. When this system is verifying some combination, it may need to verify its sub-combination, this may need tons of computing power, therefore, we design a distributed verification framework base on component combination algebra to accelerate this process.

To use a distributed system to distribute the verification of application, we need component combination verification transaction satisfy these properties to avoid impact between different verification task or let failed verification result contaminate database of verification result.:

- Atomicity of verification task. The operations of a verification task either execute all or execute none, it can't execute only part of it. If some error occurs when execute these operations, system should rollback to the initial state.
- Consistency of verification result. These component combination verification transactions should keep system in consistence. A verification task execute at different node or time should always produce same verification result.
- Isolation of verification task. No verification task could disturb other verification task's execution. Whether these tasks are executed parallel in different node or execute serial in same node, it will always produce same result.
- Durability of verification task. After execution of a verification task, the result of this verification task will store in database, it will not be rollback by system.

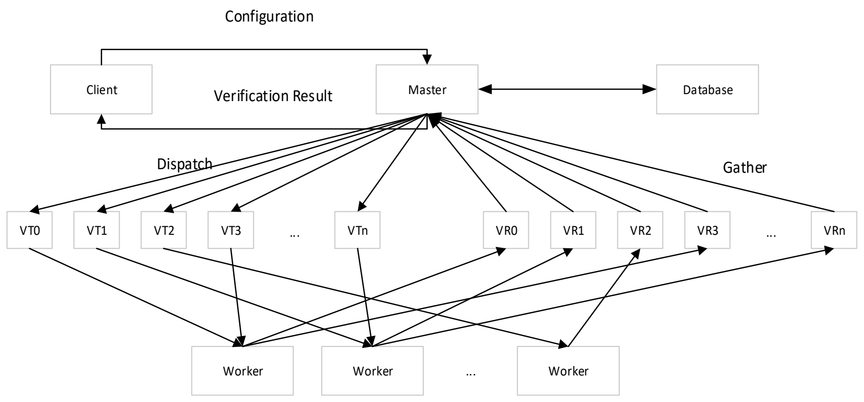


Fig. 4. Parallel verification process

When define component combination verification transaction, we need decompose component combination first. We define the smallest execution unit as component combination verification task CT, multiple component combination task together constituted a component combination verification transaction T. System will execute these component combination verification task to get component combination verification result CR. All component combination verification results together constituted component combination verification result TR. The framework of this system is show in the picture above. The definitions of these concepts are listed below.

Definition 1: A component combination verification task CT is a pair of combination model M and property r, it can represent as $CT =< M,r >$.

Definition 2: A component combination verification transaction T is constitute of multiple component combination verification tasks, it can represent as $T = \{CT_1, CT_2, \ldots, CT_n\}$.

Definition 3: A component combination verification result CR is a tuple of component combination model M, verification property r and verification status s. It can represent as: $CR = < M, r, s >, s \in \{F, P\}$

Definition 4: A combination transaction's verifications result TR is a set of CR, it can represent as $TR = \{CR_1, CR_2, \ldots, CR_n\}$.

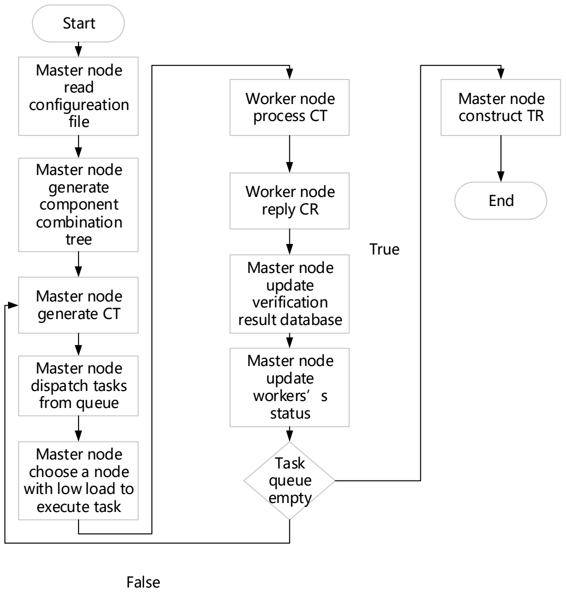

Start
Master node read configureation file
Master node generate component combination tree
Master node generate CT
Master node dispatch tasks from queue
Master node choose a node with low load to execute task
Worker node process CT
Worker node reply CR
Master node update verification result database
Master node update workers' s status
Task queue empty
True
False
Master node construct TR
End

Fig. 5.VA execution flow chart

In order to decompose tenant applications, we use component combination tree to define the relation between components. After system convert user tenant application to a component combination tree, it will use this tree to generate component combination verification task and assign these tasks to different node of worker swarm to verify this application. Figure 10 shows the procedure of this algorithm:

1. Master node read tenant application configuration file and parse component combination tree from it, each node in component combination tree represent a component combination, the component combination represent by leaf can't be divide further, the component combination represent by root node is the combination of whole tenant application.
2. Master node using property need to be verified and the component represented by leaf node to construct a CT. Then master node will put these tasks into a priority queue, the lower the depth of the leaf nodes, the higher the priority.
3. Master node extract component combination verification task from priority queue and assign it to a worker node with low load. Worker node will send verification result to master node after it finish verification.
4. Master node will use verification result CR to mark the verification status of corresponding node, then delete this leaf node. Master node will also put the result into database. If the verification status of leaf node will influence the verification status of parent node, it will recursively mark the verification result of parent node, remove the subtree of parent node

and abort all verification task associated with this subtree. Then go to step 5 if the verification result of root node is confirmed, otherwise go to step 2.

Master node will use component combination result to construct a component combination verification transaction result and send it back to user.

5. Analysis

Whether using distributed system or a single computer system to verify a tenant application, it's possible to encounter a scenario which it only needs to verify one submodule then produce the result of component combination verification transaction. But because of the low probability of this situation, in most cases, it's required to verify majority of component combination to complete the verification of tenant application. Let the time of produce verification task from tenant application configuration file as t_g, average execution time of verification task as t_e. Assume we use component combination tree to break a tenant application into n sub-component combination, then generate n component verification task from it. Using serial execution of verification algorithm, it will cost

$$T_s = n * (t_a + t_e) \tag{1}$$

If we use distributed component combination verification service to verify this application, assume there is k worker node. Because the master node needs to spend extra time to manage the status of worker nodes, assume the overhead of this is t_g. Then, the time use to verify this application is

$$T_a = n * t_g + \frac{n}{k} * t_e + k * t_g \tag{2}$$

From the above equation, it can be seen that the execution time of verification process depends on the number of worker node in addition to the verification transaction. When the time of node management is negligible, if the number of nodes is smaller than number of tasks, then the verification time will decrease as the number of nodes increase, if the number of nodes is larger than number of tasks, add more worker node won't bring any improvement. Due to the large number of verification task, master node needs to spend much time to distribute verification task and manage worker node status, this cannot be ignored. If we use a distributed system to verify a tenant application, adding more nodes will decrease verification time in the begging, but after certain threshold, adding more nodes will increase the time of execution. We can calculate T_d will get minimum

$$T_{d\ min} = 2 * \sqrt{n * t_g * t_e} + n * t_g \tag{3}$$

$$k = \sqrt{\frac{n * t_e}{t_g}} \tag{4}$$

Compared serial execution method with the parallel execution method, when the number of verification task is large, using parallel method can significantly reduce the verification time.

6. Experiments

The experiment is organized as follows: Firstly, verification of component combination in SaaS is simulated; Then use the method proposed in this paper to decrease the quantity of combination to be formally verified; Finally, compare the quantities of combinations using this method with that not using this method to prove the validity and efficiency of the method in this paper.

6.1. Environment of the Simulations

(1) Hardware Environment

The simulation performs is a Hadoop distributed computing environment constructed by several virtual machines. A virtual machine is used as master node and the other seven virtual machines are used as computing nodes. All VMs have same configuration which is one CPU, 2G memory and 10G hard disk space.

(2) Software Environment

All nodes are deployed with Ubuntu Server 12.04 LTS as operating system and Hadoop 2.5.2 as run-time environment of MapReduce and Apache HBase 1.0.1.1 as distributed database. The combination verification algebra method proposed in this paper is implemented using Java programming language.

(3) Related parameters

This experiment is designed by reference to EasySaaS Architecture proposed in[17]. Components in the experiment are divided four classes: GUI components, workflow components, service components and data components. The total number of components is 100, 30% of which is GUI components, 30% of which is workflow components, 20% of which is service components, 20% of which is data components. Each tenant application consists of ten components, four GUI components, three workflow components, two service components, one data component , So there are 4.22e11 possible combinations. The error rate is set to 0.1% and failing combinations are randomly generated at the beginning of the experiment.

Before verification starting, all tenant applications are disassembled into combinations under basic pattern, using Disassembly Tree proposed in section V. Table 1 indicates the number of combinations decomposed from applications. **Total workload** is the number of combinations to be verified for checking all applications on a certain property, and those combinations are decomposed into basic workflow pattern using component disassembled tree. After combinations status are merged with our verify algebra, the minimum combinations need to be verified is shown in **Workload after merging**.

Table 1. Heading and text fonts.

Applications number	Workload after merging	Total workload
200	159595	167400
400	311520	334800
600	459831	502200
800	603482	669600
1000	745079	837000

6.2. Simulation Experiment

This experiment is about combining different combinations on same property. In this experiment, different operations are used to be verified in parallel. The result of this experiment is shown in Fig.6. It shows the ratio of reduced time in parallel execution compared with serial execution.

From the result, it can be concluded that the ration of reducing the quantity of combinations to be verified has relation with the scale and operations described above.

When ***TA*** is on a small scale, the efficiency of combination verification method is low. But as the scale grows, the efficiency is promoted gradually. The main reason is that when verification scale grows, the probability that tenants' applications have same combinations goes higher, therefore more verify results in shared databases are reused. Meantime, as more and more sub-combinations have been verified, combination verification algebra can use the known status more times to calculate the unknow status.

For another, the efficiency of parallel verification is related to the operation type. operation with $\otimes$ and $\oplus$ have much less strict requirements for status of sub-combinations than that with $\ominus$ and $\odot$. For operator $\otimes$ and $\oplus$, status of combination can be derived as long as one of its sub-combination has a certain status. Operator $\otimes$ and $\oplus$ have very high efficiency. For operator $\ominus$, when all sub-combinations of a combination have same status, the status of the combination can be worked out. Therefore, operator $\ominus$ has lowest efficiency. Operator $\odot$ is a little less strict than operator $\ominus$ but much lower than operator $\otimes$ and $\oplus$.

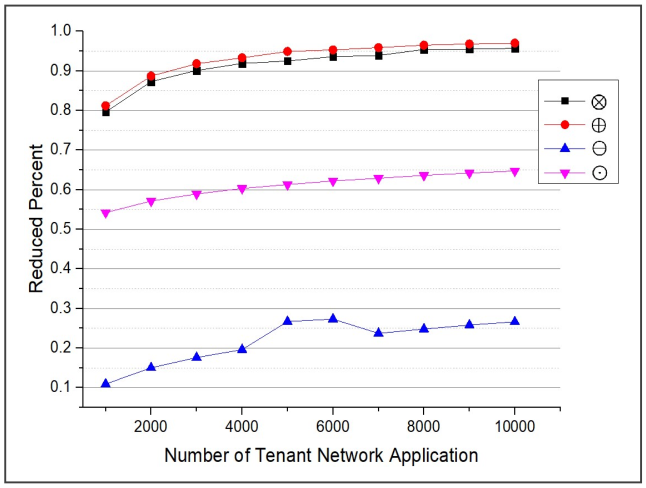


Fig. 6. Reduced percent in parallel verification

7. CONCLUSION

The VA defines the five states of verify result and operations of combining operation, provides a foundation for parallel combinatorial verifying.

In cloud platform, the computing power is utilized to check the application correctness. We have proposed VaaS on MTA, a scalable cloud-based on-demand service that uses formal models for verification.

Further, we propose a new verification framework with VA and shared databases. All verify results are saved in shared databases. the process of verification is designed where previous verifying results are used to get unknown combinations status.

By simulation experiments, it can be proved that parallel verification execution proposed in this paper is reasonable and correct and can decrease the number of the verification transactions effectively.

ACKNOWLEDGEMENTS

This work was supported by National Natural Science Foundations of China (No. 61672074,91538202). Funding of Ministry of Education and China Mo-bile MCM20160203,

REFERENCES

[1] Yuan D M, Ren R W. Research on the SDN-Based Architecture of Space-Sky Information Network[C]//Applied Mechanics and Materials. Trans Tech Publications, 2014, 644: 2854-2856.

[2] De Cusatis C, Cannista R, Hazard L. Managing multi-tenant services for software defined cloud data center networks[C]//Adaptive Science & Technology (ICAST), 2014 IEEE 6th International Conference on. IEEE, 2014: 1-5.

[3] Nunes B A A, Mendonca M, Nguyen X N, et al. A survey of software-defined networking: Past, present, and future of programmable networks[J]//Communications Surveys & Tutorials, IEEE, 2014, 16(3): 1617-1634.

[4] Open Networking Foundation. SDN architecture, version 1.1, 2016.02.

[5] Monsanto C, Reich J, Foster N, et al. Composing software defined networks[C]//Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). 2013: 1-13

[6] Shin Y Y, Kang S H, Kwak J Y, et al. The study on configuration of multi-tenant networks in SDN controller[C]//Advanced Communication Technology (ICACT), 2014 16th International Conference on. IEEE, 2014: 1223-1226.

[7] Reich J, Monsanto C, Foster N, et al. Modular SDN programming with pyretic[J]. Technical Report of USENIX, 2013.

[8] AuYoung A, Banerjee S, Lee J, et al. Corybantic: Towards the modular composition of SDN control programs[C]//Twelfth ACM Workshop on Hot Topics in Networks (HotNets-XII), College Park, MD. 2013.

[9] Pelle I, Lévai T, Németh F, et al. One tool to rule them all: A modular troubleshooting framework for SDN (and other) networks[C]//Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. ACM, 2015: 24.

[10] Dixit A, Hao F, Mukherjee S, et al. Towards an elastic distributed SDN controller[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 7-12.

[11] White S A. Business process modeling notation[J]. Specification, BPMI. org, 2004.

[12] Alves A, Arkin A, Askary S, et al. Web Services Business Process Execution Language Version 2.0[J]. Working Draft. WS-BPEL TC OASIS, May 2005.

[13] van Der Aalst W M P, Ter Hofstede A H M, Kiepuszewski B, et al. Workflow patterns[J]. Distributed and parallel databases, 2003, 14(1): 5-51.

[14] Al-Shaer E, Marrero W, El-Atawy A, et al. Network configuration in a box: Towards end-to-end verification of network reachability and security[C]//Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on. IEEE, 2009: 123-132.

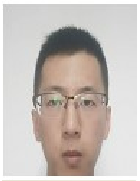
[15] Zakharov V A, Smelyanskii R L, Chemeritsky E V. A Formal Model and Verification Problems for Software Defined Networks[J]. Modelirovanie i Analiz Informatsionnykh Sistem [Modeling and Analysis of Information Systems], 2013, 20(6): 36-51.

[16] ter Hofstede A H M, Orlowska M E. On the complexity of some verification problems in process control specifications[J]. The Computer Journal, 1999, 42(5): 349-359.

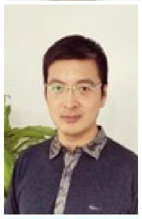
[17] Tsai W T, Colbourn C J, Luo J, et al. Test algebra for combinatorial testing[C]//Automation of Software Test (AST), 2013 8th International Workshop on. IEEE, 2013: 19-25.

AUTHOR

Kan Luo, born in Hunan province, China in 1994. He received the B.E. degree in Internet of thing from the Sichuan University, China in 2016. He is currently a postgraduate at Beihang University. His researches focus on blockchain and formal verification in Institute of System in Beihang University. His direction is verification algebra in cloud environment.



Siyuan Wang, he was born in Anhui province, China. He received bachelors in computer science from Xidian University in 2017. Currently, he is a graduate student of Beihang University, major in blockchain and distributed system. He focus on distributed verification and parallel computation on software engineering.



An wei, born in 1974, graduated from Tianjin University in 1996 with a bachelor's degree in engineering, then worked in China National Software, Unisplendour Corporation Limited, Beijing Global Safety Technology Co.,Ltd and other well-known enterprises, engaged in software development, project management, consulting and other work, he is in currently engaged in China Mobile(Hangzhou) Information Technology Co.,Ltd ，in digital currency product management, technology research and development, etc., for the virtual currency, block chain technology in-depth study.

Wei Yu, he was born in Neijiang, Sichuan province, China in 1993. He received the B.E. degree in computer science and technology from the Beijing Jiaotong University, China in 2016. Yu Wei is currently a postgraduate at Beihang University. His research directions are related to high performance blockchain and smart contracts.

Kai Hu is a professor at Beihang University, China. He received his PhD degree from Beihang University in 2001. From 2001 to 2004, he did the post-doctoral research at Nanyang Technological University, Singapore. Since 2004, he is the leader of the team of LDMC in the Institute of Computer Architecture (ICA), Beihang University. His research interests concern embedded real time systems and high performance computing. He has good cooperation with IRIT and INRIA Institute of France on study of AADL and synchronous languages