SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Verification Algebra for Multi-Tenant Applications in VaaS Architecture

Kai Hu[1,4], Ji Wan[1], Kan Luo[1,*], Yuzhuang Xu[1], Zijing Cheng[2], and Wei-Tek Tsai[1,3]

[1]*School of Computer Science and Engineering, Beihang University, China*
[2]*Beijing Institute of Satellite Information Engineering, China*
[3]*Arizona State University, USA*
[4]*Blockchain Key Laboratory of Yunnan Province, Yunnan Innovation Institute of Beihang University, China*

SUMMARY

This paper proposes an algebraic system, Verification Algebra (VA), for reducing the number of component combinations to be verified in Multi-Tenant Architecture (MTA). MTA is a design architecture used in SaaS (Software-as-a-Service) where a tenant can customize its applications by integrating services already stored in the SaaS databases or newly supplied services. Similar to SaaS, VaaS (Verification-as-a-Service) is a verification service in a cloud which leverages the computing power offered by a cloud environment with automated provisioning, scalability, and service composition. In VaaS Architecture, however, there is a challenging problem called 'Combinatorial Explosion' that it is difficult to verify a large number of compositions constructed by both quantities of components and various combination structures even with computing resources in cloud. This paper proposes rules to emerge combinations status for future verification, on the basis of the existing results. Both composition patterns and properties are considered and analyzed in VA rules.

Received 30 May 2019; Revised 23 February 2020; Accepted 3 December 2020

KEY WORDS: MTA; SaaS; Combinatorial Explosion; Verification; Workflow Patterns

1. INTRODUCTION

Software-as-a-Service(SaaS) is a scalable computation and new delivery model on top of other cloud platforms, which are Platform-as-a-Service(PaaS) and Infrastructure-as-a-Service (IaaS). Compared to PaaS and IaaS, software can be automatically customized and adjusted depending on user requirements and business development to obtain a better cost control [1]. SaaS often supports three important features: customization, scalability, and multi-tenancy archi-tecture (MTA). MTA is a key feature of SaaS. It allows all tenant softwares to share the same code on the basis of configuration data saved in databases or data stores. In this way, multiple tenant applications can be developed swiftly. Each tenant application feels like a dedicated application, while the compositions are different [2].

Although each component provided by a software vendor in the SaaS database can be ensured to be correct, it is still a challenging task to guarantee the quality of tenant application composed by hundreds of thousands of compo-nents in SaaS. Any software should be identified to be correct before deployment. Thus, verifying and validating those tenant applications becomes an indispensable work. Many cloud-based quality-assurance (QA) techniques are available including testing, verification, and simulation, and all can take the advantage of the enormous computation

*Correspondence to: Kan Luo,Beihang University, China.
Email:looken@foxmail.com

power offered, such as Testing-as-a-Service (TaaS) [3] and Simulation-as-a-Service (SIMaaS) [4]. TaaS provides testing services on a cloud and people can submit their softwares for testing, and SIMaaS provides simulation services.

However, checking applications based on testing have limitations: (1) A tenant application cannot run the test service until the code is available. (2) It cannot verify software correctness but detect bugs.

Distinguished with testing using test case, the correctness of unimplemented software is checked with formal methods. Formal verification addresses the correctness of software with respect to a specification or property with formal methods. It is widely used in cryptographic protocols, combinational circuits, digital circuits, and software [5]. These systems are always converted into formal forms first and then verified by mathematical methods. The methods used to verify these systems are finite state machines, labeled transition systems, Petri nets, timed automata, hybrid automata, process algebra. In some cases, verifying can be an alternative solution to testing for ensuring the quality of a tenant application. However, verifying application in SaaS with the new feature MTA creates new issues in a cloud environment:

(1) There is a challenging problem, 'Combinatorial Explosion', in the process of verifying MTA applications, as combinations grow exponentially with the number of components. For instance, supposing that the 4 layers (GUI layer, workflow layer, service layer and data layer) has 10^5 components respectively there will be $10^{20}(10^5*10^5*10^5*10^5)$ possible combinations in total. Even if we use cloud computing resources, it is unworkable to verify every tenant application within the scope of tolerable time.
(2) After deployment, applications need to update when necessary, e.g. components addition, substitution or dele-tion. With software being frequently modified, large number of tenant applications are recreated and in need of verifying. However, checking ever-increasing applications cannot be handled by a traditional verification scheme, which is incapable of verifying similar applications because once an application is modified, it will be treated as a new application and the new verification must start from the scratch. Massive components or combi-nations are repetitively verified, increasing the costs of tenant application management.

Faced with above challenges, we have proposed the concept, MTA VaaS (Verification-as-a-Service) [5], which is capable of verifying SaaS software and exhibits the features of SaaS software such as automated provisioning, scala-bility, fault-tolerant computing, and concurrent processing. In VaaS, although the concurrent computing power of cloud environment is utilized to increase the speed of verifying tenant applications, it is impossible to ensure all ap-plications are correct in SaaS.

Furthermore, this paper proposes VA (Verification Algebra) to improve the MTA VaaS Architecture. Similar to Test Algebra [7–9], VA is an algebra system that can compute the results of combinations according to existing veri-fying results of components and combinations,and the number of component combinations to be verified can be effec-tively reduced. When combining components or sub-combinations, both workflow patterns and properties should be considered. Therefore, two kinds of algebraic rules are designed, one kind of rules is for workflow patterns, another kind of rules is for properties to be verified.

Particularly, rules for workflow patterns are based on the principle that the status of combinations depends on its components or sub-combinations. Different workflow patterns have different rules when getting the verifying status of combinations. Rules for properties are based on the relationship of properties. By defining all relationship of ab-stract properties to be verified, the rules for properties can be separately designed.

In our previous work [16], we proposed rules for verifying combinations under basic workflow patterns, and some operating rules have also been proposed. However, multi-workflow pattern was not taken into consideration. In most cases, tenant applications are combined under multiple patterns. To verify hybrid patterns in tenant applications, a graph-based method is proposed to disassemble application of complex structure into a tree with single workflow patterns as leaf nodes. By recursively verifying the branches of the workflow tree using VA rules, the application can be verified. Our contribution can be summarized as:

(1) We put forward a schema that leverages formal methods with the computing power offered by a cloud environ-ment to check software correctness, making up for the shortages of testing.
(2) We study the rules of merging the verifying status of combinations and propose a Verification Algebra System to cut down component combinations to be verified.
(3) We design the process of verification where previous verifying results are used to get unknown combinations status, saving time and cost in verification process.

This paper is organized as follows: Section 2 reviews related works; Section 3 describes the MTA VaaS architec-ture we have proposed; Section 4 presents VA rules for properties to be verified; Section 5 shows the graph-based method to construct verification space and describes processes of constructing, pruning and verifying disassembled trees of multi-workflow patterns; Section 6 shows the result of the simulation experiment to validate the feasibility and efficiency of VA method proposed; Section 7 concludes this paper.

2. RELATED WORK

2.1. Test Algebra

In SaaS, once tenant applications are composed, they need to be tested, but a SaaS system can have millions of components, and hundreds of thousands of tenant applications. Testing tenant application becomes a challenge as new tenant applications and components are added into the SaaS system continuously. Combinatorial testing is a popular testing technique to test an application with different configurations. It often assumes that each component within a configuration has been tested already, but interactions among components in the configuration may cause failures. Traditional combination testing methods to detect the presence of faults only work when the problem size is small. When the problem gets larger, combinations among components and properties may cause 'Combinatorial Explosion' issue. As components increase, the number of combinations quickly rises exponentially [19]. It will be hard to identify faults even if we use cloud computing resources.

As IT industry moves to Big Data and cloud computing where hundreds of thousand processors are available, potentially, a large number of processors with distributed databases can be used to perform large-scale combinatorial testing. One simple way of performing combinatorial testing in a cloud environment is [21]:

(1) Partition the testing tasks.
(2) Allocate these testing tasks to different processors in the cloud platform for test execution.
(3) Collect results done by these processors.

Test Algebra can be used to identify faults in combinatorial testing for SaaS applications [7, 22]. Test Algebra eliminates some configurations or interactions from future testing based on the existing testing results, and it can merge testing results from different processors so that testing results can be obtained quickly. The Test Algebra de-fines rules to identify faulty configurations and interactions [11]. By using the rules defined in the Test Algebra, a col-lection of configurations can be tested concurrently in different servers and in any order, the results obtained will be still the same due to the algebraic constraints.

Tsai et al. [8] proposed a Test Algebra to identify faults in combinatorial testing for SaaS applications. It allocates workloads into different clusters of computers and performs the Test Algebra analyses from 2-way to 6-way configu-rations. Wu et al. [9] put forward a MapReduce design of Test Algebra concurrent execution in cloud environment. Qi et al. [12] proposed the Test-Algebra-Based fault location analysis for concurrent combinatorial testing.

Test Algebra identifies candidate configurations to be eliminated, reducing the workload of future testing. Meanwhile, large-scale combinatorial testing can be carried out in a cloud platform with a large number of processors to perform test execution in parallel to identify faulty interactions. Large-scale combinatorial verification in a cloud platform bring new challenges. Inspired by Test Algebra, we can also merge the verifying status of combinations and cut down component combinations to be verified in verification tasks. To our knowledge, we are the first to propose verification algebra rules

DOI: 10.1002/stvr

which are used to eliminate unnecessary combinatorial verification tasks. Furthermore, the structure of combination is also considered in verification algebra rules. The verification algebra method can improve combination verification efficiency in MTA VaaS. Compared with original verification results, verification status for a component combination is actually the same after applying the verification algebra method.

2.2. *Formal Method*

Many formal methods and verification techniques have been developed with three main approaches: model checking [23], deductive verification [24], and equivalence checking [25]. According to these approaches, many model-ing languages and verification tools are available such as BIP [26, 27], Algebra of communicating shared resources (ACSR) [28], AADL [29,30], FIACRE [31,32], Timed Abstract State Machine (TASM) [33], Bigraph [34,35], SPIN [36].

Schaefer et al. [**?**] proposed that formal verification process can be cast as workflows in business process model-ing. Single steps in the verification process constitute verification tasks that can be flexibly combined to verify work-flows. Verification can be carried out using services in a cloud. Zahoor et al. [38]proposed a bounded model-checking approach for verification of declarative service composition using Satisfiability solving. A holistic approach to verify the correctness of Hadoop systems using model checking techniques is proposed. They model Hadoop's parallel archi-tecture to validate startup ordering and address data locality, deadlock freeness and non-termination. Moscato et al. [39] proposed a methodology based on Multi-Agent Model that allows for description, composition, and verification of cloud-based services. The methodology uses a modeling profile to describe services as agents in a multi-agent en-vironment and it is based on Model-Driven Engineering (MDE). The methodology includes a verification process that exploits formal methods during the service life cycle. They developed a system that can be used for describing and analyzing composite services.

In addition, Service-Level Agreements (SLA) is an important issue for cloud services, but public clouds provide few guarantees in terms of performance and dependability [40]. Manciniet al. [41] proposed a VaaS concept for sys-tem-level formal verification-as-a-service SyLVaaS, a Web-based tool VaaS. But its goal is to show system correctness, such as faults, variation in system parameters, and external inputs.

Furthermore, "verification as a service" is used with different meanings, e.g., system verification via model checking, and these services are available in many different applications, like identity verification services. However, these "Verification-as-a-Service" are different from MTA VaaS. MTA VaaS integrates formal methods as services run-ning in a cloud taking advantages of distributed computation in the cloud, and these services are organized in a SaaS manner.

2.3. *Workflow Patterns*

In SaaS, components are combined under single or multi patterns. The combination patterns of components are ex-pressed by the workflow. The pattern of workflow has been discussed by Van Der Alast et al. [43]. 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 platforms can be divided into multi-tiered combinations that contain only the Basic Control Patterns. For this reason, emphasis of analyzing combination patterns will be put on Basic Control Pat-terns. There are five kinds of workflow patterns in Basic Control Patterns as shown in *Figure 1*.

- **Sequence:** The sequence pattern is the most common pattern used to model consecutive steps in a workflow pro-cess.
- **Parallel Split:** Multiple components are simultaneously enabled after the completion of another component, thus allowed to be executed in parallel or in any order.

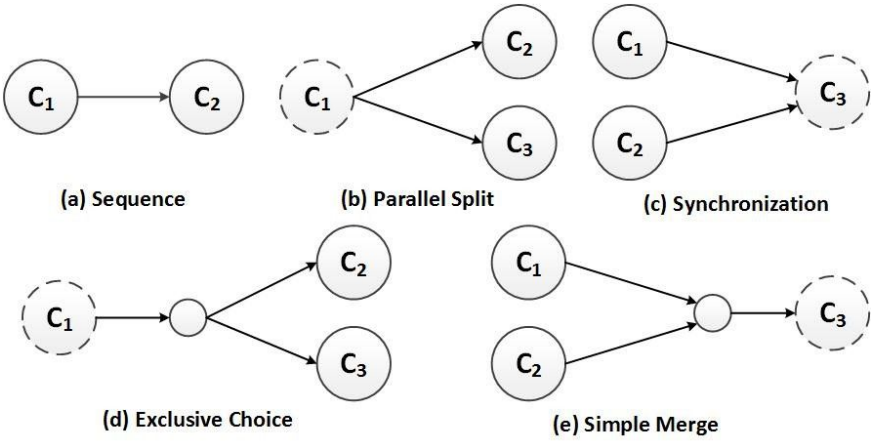

Figure 1. Basic Workflow Patterns

- **Synchronization:** Synchronization is a point in the workflow process where multiple parallel components con-verge into one single component.
- **Exclusive Choice:** One of several components is chosen based on a decision or workflow control data in the pat-tern of Exclusive Choice.
- **Simple Merge:** The completions of two or more alternative components come together without synchronization in the pattern of Simple Merge. In other words, the merging will be triggered once any of incoming transitions is triggered.

This paper names the combination containing only one kind of workflow pattern as single-pattern combination. Similarly, multi-pattern combination is the component combination that contains more than one kind of Basic Control pattern. This paper will study the rules of Single-pattern combination first, and then generalize them to multi-patterns.

3. MTA VAAS ARCHITECTURE

The MTA VaaS architecture has been designed by Zhao et al. [5]. MTA VaaS is an architecture that can be used to verify models. Similar to SaaS, MTA VaaS is beneficial from the computing power offered in a cloud. A VaaS host ver-ification software is in cloud environment, and these services can be called on demand, and those services can be composed to verify software models. In VaaS, Bigragh is selected as the modeling language for illustration as it can model mobile applications. A Bigraph model can be verified by converting it to a state model, and the state model can be verified by model-checking tools. The VaaS service combination model and execution model are also presented by Zhao et al. [5].

In SaaS, tenants can customize their applications stored in the SaaS databases, and 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 [42]. For ex-ample, a SaaS GUI database contains all the GUI components used by the tenants.

The design of MTA VaaS is similar to SaaS. For example, it has databases to store verification softwares, models to be verified, and verification results. The MTA VaaS also provides customization service support.

Essentially, MTA VaaS is alike to SaaS except the principal task. Principal task of MTA VaaS is for software verifi-cation rather than general computing. It is similar to a TaaS, but a TaaS runs test scripts. However, VaaS also has unique features not in SaaS or TaaS:

- Only formal verification software is stored in VaaS.
- As formal verification often involves model transformation, thus VaaS may contain transformation software.

- VaaS also contains softwares for parsing formal models.
- 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 needed to be verified.

VaaS may be an open system where users can publish verification services (VS) to create a community. In the future, VaaS can be integrated with TaaS and SIMaaS.

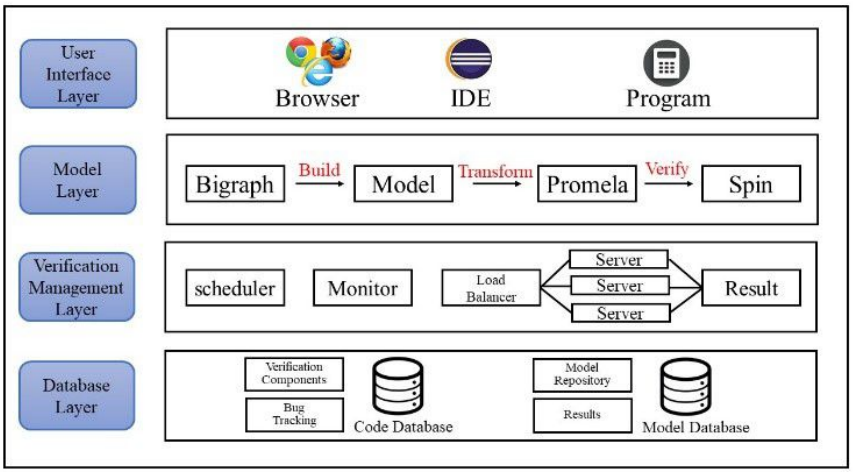

Figure 2. MTA VaaS Architecture

Figure 2 shows a sample VaaS architecture including User Interaction Layer, Model Layer, Verification Management Layer, and Database Layer [5].

A tenant can develop a new verification application, i.e., a tenant application which identifies and reuses UIs (G), Workflows (W), Services (S), and Data components (D) in the VaaS database. All components selected can be linked, then components are compiled to produce an executable code.

Models to be verified are stored in the VaaS database. A model may consist of sub-models. As verification is often applied to a specific property, a verification task (VT) consists of two components $< M, VR >$, where M represents a model, and VR represents a verification rule. As the type of model can be a composite model, its sub-models may be verified first before combinations of sub-models are verified.

However, there are too many compositions to be verified in VaaS. Since the number of verification tasks may grow exponentially as the complexity of tenant application grows. To address this issue, verification algebra rules are proposed in the VaaS Architecture.

4. THE THREE GOLDEN RULES

After applications are customized by a tenant, those applications can be verified on properties, such as $deadlock$, $soundness$, $well-structured$. Those properties are not independent of each other, each property is related to the other properties. For example, a web application is proved to be $soundness$ and $well-structured$ after being verified. It can be inferred that this web application is $coverability$, and the property $coverability$ can be derived from the former two properties the system already possesses. In this circumstance, the knowledge between $soundness$, $well-structured$, $coverability$ can be described for short:

$$soundness + well-structured \Rightarrow coverability$$

With the knowledge of the relation between properties, the workload of verification decreases, as many tenant applications only have to be verified on two properties rather than three properties. The third property $coverability$ in the system is inferred through previous results and knowledge

without verification. In this way, considering the relationship between properties can help to reduce the number of combinations to be verified.

In this section, firstly we introduce abbreviations and definitions in combinatorial verification. Then we intro-duce the algebraic rules for merging verification status.

4.1. Abbreviations and Definitions

In SaaS, tenant application is a combination of a set of components customized by tenants according to their own demands. This paper uses symbols to simplify presentation:

TA: Tenant Application.

C: Component, the basic unit in component combination. Components in SaaS are provided by component vendors. We suppose that all components of SaaS should pass the verification tasks.

Com: Combination constructed by more than one components, and it also has a certain function.

W: the workflow pattern of a Com.

p: A certain property to be verified. In SaaS, a set of properties should be specified before verification task and then verification of these properties will be conducted.

p set: Properties set to be verified on and *p set* $= \{p_1, p_2, p_3, ..., p_n\}$.

In SaaS platform, a combination is composed by two or more components. Even if we use the same set of components, we can get many different combinations because of different workflow patterns. Thus it is meaningless to study the relationships among combinations without considering workflow patterns. The definition of sub-combination is given as follows:

Definition I: (Sub-Combination) If the component set $Com^{'}$ is contained in the component set Com, we say that combination $Com^{'}$ is a sub-combination of combination Com. Sub-combination is defined as $Com^{'} \subset Com$.

Definition II: (Similar Sub-Combination) If $Com^{'} \subset Com$ and the workflow pattern of $Com^{'}$ is the same as Com, we claim that $Com^{'}$ is a similar sub-combination of Com. Sub-combination is defined as $Com^{'} \sqsubset Com$.

SC: Sub-Combination.

SSC: Similar Sub-Combination.

SSC Set: All SSCs of a Com.

Definition III: (Verification Transaction) We name verification transaction as the verification process of TA or Com. Verification transaction is relevant to the workflow pattern W, and the properties $p\ set$.

T: Verification transaction. T can be expressed by a 2-tuple, as $T = \langle W, p\ set \rangle$

Since TA has a large number of components and properties to be verified, it is difficult to specify and verify TA directly. So we start the verification transaction from subsets of components and properties of TA transaction, then we merge the results. If a certain verification task can be executed on Com, its execution result is $passed(operational)$ or $failed(fault)$. To present the verification results of Com, a valuation function V is proposed. For a combination Com and a certain property p, $V(Com, p)$ indicates the current knowledge about the verification status consistent with the components in Com. Three more status are defined besides $passed$ and $faulty$. So every Com has one of these five status:

- **Infeasible (X):** If some components are not permitted to be combined, then $V(Com, p) = X$. For example, there would be a conflict when two components do the same thing but cause different results. Such as two GUI components, one of which paints the software background in blue, but another one paints the software background in red.
- **Faulty (F):** Verification of combination Com on property p is not passed.
- **Passed (P):** Verification of combination Com on property p is passed.
- **Irrelevant (N):** Some components are permitted to be combined while it is no sense to be created, so there is no need to verify these combinations, and $V(Com, p) = N$ represents such case.
- **Unknown (U):** The status of Com is certain but not currently known.

DOI: 10.1002/stvr

At any stage of verification, a Com must be in one of five possible status. We use symbol $\succ$ and $=$ to show the priority sequences of these five status.

The verification algebra method aims to reduce the number of combinations to be verified. It merges existing verification status and then gets the results of unknown status by algebraic computation. In verification algebra system, the most important operation is to merge the status of two combinations or two properties. There are four kinds of ways for merging operations:

(I) Merging status of the same ***Com*** on the same ***p***
(II) Merging status of the same ***Com*** on different ***p***
(III) Merging status of different ***Com*** on the same ***p***
(IV) Merging status of different ***Com*** on different ***p***

Operation (I) cannot reduce the workload of verification, because merging the two same status is meaningless to new Com or p. Operation (IV) can be transferred into operation (II) and operation (III). Therefore, we focus on operation (II) and (III) and formalize these two operations using two binary operators. Here we use $\odot$ for operation (II) and $\otimes$ for operation (III). These two operators are defined as follows:

Definition IV:(Operator $\otimes$) Given a property p, Com_1 and Com_2 are $SSCs$ of Com. If the verification status of Com_1 is $V(Com_1, p)$, the status of Com_2 is $V(Com_2, p)$, and $Com_1 \cup Com_2$ are known previously. Then:

$$V(Com, p) = V(Com_1 \cup Com_2, p) = V(Com_1, p) \otimes V(Com_2, p)$$

Definition V:(Operator $\odot$) Given two different properties p_1 and p_2. $V(Com, p_1)$ denotes the verification status of Com on property p_1, and $V(Com, p_2)$ denotes the verification status of Com on property p_2. then:

$$V(Com, p_1) = V(Com, p_1) \odot V(Com, p_2)$$
$$V(Com, p_2) = V(Com, p_2) \odot V(Com, p_1)$$

In the formulas above, $V(Com, p_1) \odot V(Com, p_2)$ stands for combining status on p_2 to status on p_1 in the same combination. In other words, it is used to merge unknown p_1 status with known p_2 status.

Thus, $V(Com, p_1) = V(Com, p_1) \odot V(Com, p_2)$ represents calculating $V(Com, p_1)$ based on $V(Com, p_2)$. On the contrary, $V(Com, p_2) = V(Com, p_2) \odot V(Com, p_1)$ means calculating $V(Com, p_2)$ based on $V(Com, p_1)$.

4.2. Algebraic Rules for Operator $\otimes$

In SaaS, a TA can be decomposed into multiple sub-combinations. When verifying TA on a specific property, the verification status of their sub-combinations will probably determine the result of the whole TA. For instance, a web application is made up of three combinations with property $non - deadlock$. And the web application is also $non - deadlock$. This situation just presents one kind of relationship among combinations and their sub-combinations. All relationships are formally interpreted by attributes below. In this paper, attributes of properties only focus on the combinations with the same workflow pattern.

The validation status processing rule is an extension of the test status processing rule. There have been numerous research works about testing state processing rules [7–15]. Inspired by the state merging rules in these research works we use **Attr** to present attribute. In other words, **Attr** is used to indicate the verification status relationship among different combinations on the same property. The attributes of property can be defined as follows:

Attr 1: $Com^{'}$ is a SSC of Com. If $V(Com^{'}, p) = P$, then $V(Com, p) = P$.

Attr 2: $Com^{'}$ is a SSC of Com. If $V(Com^{'}, p) = F$, then $V(Com, p) = F$.

Attr 3: $Com_1, Com_2, ..., Com_n$ is $SSC\ Set$ of Com. If $V(Com_1, p) = V(Com_2, p) = ... = V(Com_n, p) = F$ or P, then $V(Com, p) = F$ or P.

Attr 4: $Com_1, Com_2, ..., Com_n$ are $SSCs$ of Com and $Com = Com_1 \cup Com_2 \cup ... \cup Com_n$. If $V(Com_1, p) = V(Com_2, p) = V(Com_3, p) = ... = V(Com_n, p) = F$ or P, then $V(Com, p) = F$ or P.

DOI: 10.1002/stvr

Supposing each p has only one of the four attributes above, and **Attr** is also affected by the workflow of applications, the mapping relation is formalized as:

$$f : (W, p) = Attr$$

Where W is the workflow pattern of the Com, p is the property to be verified. For example, when verifying a Com constructed by $Com_1, Com_2, \ldots, Com_n$ with Sequence pattern, if p is $non - deadlock$, then if any one of $Com_1, Com_2, \ldots, Com_n$ fails on p, then Com will fail on p. That is:

$$f(Sequence, non - deadlock) = Attr2$$

Another example is that, when p is $Bandwidth\ Guarantee$, and a Com with $Exclusive\ Choice$ pattern is made up of $Com_1, Com_2, \ldots, Com_n$. Only if all $SSCs$ of Com do not go beyond bandwidth, then Com will meet $bandwidthguarantee$ property. That is:

$$f(Exclusive\ Choice, Bandwidth\ Guarantee) = Attr3$$

Assume that Com_1 and Com_2 are two $SSCs$ and their verification status on p are $V(Com_1, p)$ and $V(Com_2, p)$. Rules of combining Com_1 and Com_2 by operator $\otimes$ are determined by $Attr$, so operating rules for $Attr$ are discussed as follows:

(1) Operating Rules for Attr 1

According to the definition of Attr 1, the rule of operation follows the sequence below:

- If $V(Com_1, p) = X$ or $V(Com_2, p) = X$, then $V(Com_1, p) \otimes V(Com_2, p) = X$
- Else, if $V(Com_1, p) = P$ or $V(Com_2, p) = P$, then $V(Com_1, p) \otimes V(Com_2, p) = P$
- Else, if $V(Com_1, p) = N$ or $V(Com_2, p) = N$, then $V(Com_1, p) \otimes V(Com_2, p) = N$
- Else, if $V(Com_1, p) = U$ or $V(Com_2, p) = U$, then $V(Com_1, p) \otimes V(Com_2, p) = U$
- Else, $V(Com_1, p) \otimes V(Com_2, p) = U$

So the priority sequence of the five verification status is X $\succ$ P $\succ$ N $\succ$ U $\succ$ F, and operation table used by operator $\otimes$ to combine status of two $Coms$ is as follows:

$\otimes$	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

(2) Operating Rules for Attr 2

Similar to rules for $Attr1$, the priority sequence of five verification status is X $\succ$ F $\succ$ N $\succ$ U $\succ$ P. Operation table used by operator $\otimes$ to combine status of two $Coms$ is as follows:

$\otimes$	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

(3) Operating Rules for Attr 3

According to the definition of $Attr3$, the rules of operation follow the sequence below:

- If $V(Com_1, p) = X$ or $V(Com_2, p) = X$, then $V(Com_1, p) \otimes V(Com_2, p) = X$
- Else, if $V(Com_1, p) = N$ or $V(Com_2, p) = N$, then $V(Com_1, p) \otimes V(Com_2, p) = N$
- Else, $V(Com_1, p) \otimes V(Com_2, p) = U$

So the priority sequence of the five verification status is X $\succ$ N $\succ$ F = P = U . Operation table used by operator $\otimes$ to combine status of two $Coms$ is as follows:

$\otimes$	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

(4) Operating Rules for Attr 4

According to the definition of Attr 3, the rules of operation follows the sequence below:

- If $V(Com_1, p) = X$ or $V(Com_2, p) = X$, then $V(Com_1, p) \otimes V(Com_2, p) = X$
- Else, if $(Com_1, p) = N$ or $V(Com_2, p) = N$, then $V(Com_1, p) \otimes V(Com_2, p) = N$
- Else, if $V(Com_1, p) = U$ or $V(Com_2, p) = U$, then $V(Com_1, p) \otimes V(Com_2, p) = U$
- If $V(Com_1, p) = F$ and $V(Com_2, p) = F$, then $V(Com_1, p) \otimes V(Com_2, p) = F$
- If $V(Com_1, p) = P$ and $V(Com_2, p) = P$, then $V(Com_1, p) \otimes V(Com_2, p) = P$
- If $(Com_1, p) \neq V(Com_2, p)$, then $V(Com_1 \cup Com_2, p) = U$

For $Attr4$, the priority sequence of the five verification status is X $\succ$ N $\succ$ U $\succ$ F = P. Operation table used by operator $\otimes$ to combine status of two Coms is as follows:

$\otimes$	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

4.3. Algebraic Rules for Operator $\odot$

When verifying a TA, every property p in $p\ set$ must be verified. A certain kind of dependence relationship may exist between two properties p_m and p_n. Therefore, the verification status of Com on p_m has relations with the verification status of Com on p_n. We use ***PR*** to represent the verification status relationship between two properties of the same combination Com. The relationships can be divided into following four kinds:

PR1: If $V(Com, p_m) = P$, then $V(Com, p_n) = P$
PR2: If $V(Com, p_m) = P$, then $V(Com, p_n) = F$
PR3: If $V(Com, p_m) = F$, then $V(Com, p_n) = F$
PR4: If $V(Com, p_m) = F$, then $V(Com, p_n) = P$

The dependence relationship between p_m and p_n is one of the four relationships demonstrated above. And $\rightarrow$ is used to represent dependence relationship. $p_m \overset{PR}{\rightarrow} p_n$ represents that the verification status of p_n depends on p_m. Furthermore, it is easy to know that the dependence relationship is the inherent attribute of property, and it has nothing to do with combination patterns.

We use ***dp*** to describe the depth of this dependency relationship. ***dp*** is defined as follows:

Definition VI:(*dp*) It is used to describe the depth of dependency relationship between properties in $p\ set$. $dp(p)$ represents dependency depth of p. A higher value of $dp(p)$ indicates that p depends on other properties more, or few properties rely on p. On the contrary, a lower value of $dp(p)$ indicates that p depends on other properties less, or more properties rely on p.

Assume that verification status of Com on p_m and p_n are $V(Com, p_m)$ and $V(Com, p_n)$. Rules of combining this two status with operator $\odot$ are related with the dependence relationship between p_m and p_n. Rules for the four relationships are discussed as follows:

(1) Operating Rules for PR1

- If $p_m \overset{PR1}{\rightarrow} p_n$ and $V(Com, p_m) = P$, then $V(Com, p_n) = V(Com, p_n) \odot V(Com, p_m) = P$
- If $p_m \overset{PR1}{\rightarrow} p_n$ and $V(Com, p_m) = F$, then $V(Com, p_n) = V(Com, p_n) \odot V(Com, p_m) = U$

For $PR1$, operation table used by operator $\odot$ to combine status on p_m to p_n is as follows:

$\odot$	X	P	F	N	U
$V(Com, p_n)$	X	P	U	U	U

(2) Operating Rules for PR2
Similar to rules for $PR1$, operation table used by operator $\odot$ to combine status on p_m to p_n is as follows:

$\odot$	X	P	F	N	U
$V(Com, p_n)$	X	F	U	U	U

(3) Operating Rules for PR3
Similarly, operation table used by operator $\odot$ to combine status on p_m to p_n is as follows:

$\odot$	X	P	F	N	U
$V(Com, p_n)$	X	F	U	U	U

(4) Operating Rules for PR4
For PR4, operation table used by operator $\odot$ to combine status on p_m to p_n is as follows:

$\odot$	X	P	F	N	U
$V(Com, p_n)$	X	P	U	U	U

In order to describe the relationship among properties more vividly, we give some properties and their definitions in *Table I*. These properties are commonly used in formal verification.

There are a lot of researches in process verification area. The rationality of process has a close relationship with properties such as soundness and reachability. Those relationships of process

Table I. Definitions of Partial Properties

Property	**Definition**
soundness	The software will not fail or crash under unusual circumstances
well-structured	The software has the characteristics of high cohesion and low coupling
coverability	Software products can be maintained and upgraded with less time and cost
boundness	The elements in the module of software are closely connected to each other
liveness	In software, liveness refers to a set of properties of concurrent systems, that require a system to take turns in critical sections
reachability	Reachability refers to the software from design to implementation, enabling people with disabilities, the elderly, non-native groups to obtain the same information and services
safety	The system controlled by software is always in a safe state that does not endanger people's life, property and ecological environment
strong connection	Modules have strong connectivity among softwares
deadlock	A deadlock is a blockage that occurs during the execution of two or more processes that compete for resources or communicate with each other

DOI: 10.1002/stvr

Table II. Relationship of Partial Properties

No	Relationship
1	$soundness \Leftrightarrow liveness + boundness$
2	$soundness + well-structured \Rightarrow safety$
3	$soundness + well-structured \Rightarrow coverability$
4	$coverability \Rightarrow safety + boundness$
5	$liveness \Rightarrow boundness$
6	$liveness \Rightarrow strongconnection$
7	$deadlock \overset{exclusive}{\longleftrightarrow} liveness$
8	$deadlock \overset{exclusive}{\longleftrightarrow} reachability$

model are illustrated with some properties examples [16] as shown in *Table II*. Each property may have multiple relationships with other properties. There is not a consistent one-to-one match between each property and relationship in *Table I* and *Table II*.

Then we will give examples to illustrate how components, properties and tenant applications work in real-world. There are many Cloud Computing Providers providing customized services for components, such as Amazon Web Service [17], Google Cloud [18] and Alibaba Cloud Computing [20]. Users can customize the composition of components according to their functional requirements

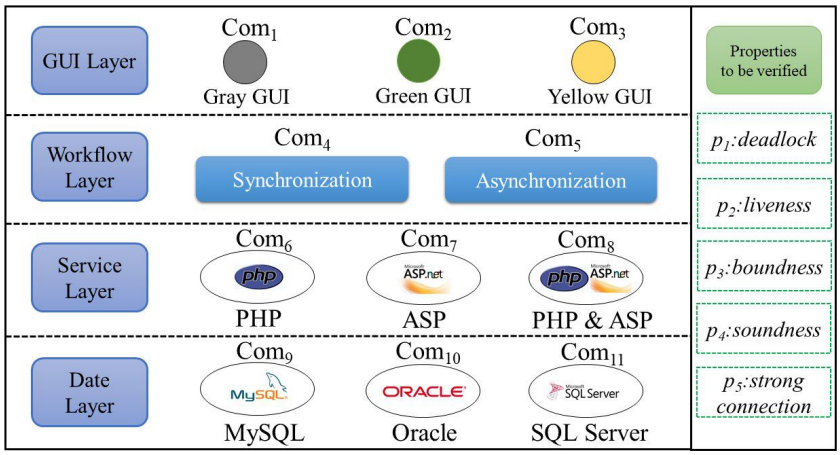

Figure 3. Example of Tenant Application

Example 1. As shown in *Figure 3* , assume that a tenant application TA_1 named "Virtual Private Server (VPS)" is decomposed into GUIs, Workflows, Services, and Data components. All the combinations in *Figure 3* have the same workflow pattern. $TA_1 = \{Com_2, Com_6, Com_8\}$ denotes a green GUI tenant application VPS that supports PHP and ASP. Assume that the attribute of different combinations on property p_1 is Attr 2, while Com_6 is a SSC of TA_1 obviously. According to the operating rules for Attr 2 in Section 4.2, if $V(Com_6, p_1) = F$, we can infer that $V(TA_1, p_1) = F$. In addition, Com_6 is a SSC of Com_8, according to the operating rules for Attr 2, we know that $V(Com_8, p_1) = F$.

Then we illustrate the relationship PR of two properties. We can find the relationship between $p_1(deadlock)$ and $p_2(liveness)$ in *Table II*:

$$deadlock \overset{exclusive}{\longleftrightarrow} liveness$$

It can be converted to following inequality:

$$p_1 \overset{PR2}{\rightarrow} p_2$$

Assume that $V(Com_9, p_1) = P$, then we can infer that $V(Com_9, p_2) = F$ according to the operating rules for PR2.

DOI: 10.1002/stvr

Example 2. $TA_2 = \{Com_1, Com_4, Com_6, Com_9$ denotes a grey GUI synchronization tenant application VPS that supports PHP and MySQL. Assume that the attribute of different combinations on property p_2 is Attr 3, obviously $\{Com_1, Com_4, Com_6, Com_9\}$ is $SSC\ Set$ of TA_2, if $V(Com_1, p_2) = V(Com_4, p_2) = V(Com_6, p_2) = V(Com_9, p_2) = P$, according to operating rules for Attr 3, we can infer that $V(TA_2, p_2) = P$. In addition, the relationship between p_2 and p_3 in *Table II* shows that:

$$liveness \Rightarrow boundness$$

It can be converted to the following inequality:

$$p_2 \overset{PR1}{\rightarrow} p_3$$

As the verification status of the TA_2 on p_2 $V(TA_2, p_2) = P$ was previously known, according to the operating rules for PR1, we can infer that $V(TA_2, p_3) = P$.

5. VERIFICATION PROCESS

To cut down combinations to be verified, previous verifying results are used to calculate unknown combination verification status through verification algebra which are formally described in the previous section. We start the verification transaction from subsets of components and properties of TA transaction, then we merge the verification status results. Operator $\otimes$ can merge the status of different combinations on the same property. Operator $\odot$ can merge the status of the same combination on different properties.

In this section, the detailed process of cutting down workload by using verification algebra is presented. The entire verification transaction contains the process of merging different combinations on the same property and the process of merging the same combination on different properties. During the process of merging different combinations, workflow patterns of combinations must be taken into consideration. Owing to only basic pattern combinations can be handled with the verification algebra method, a method named ***Component Disassembly Tree*** is proposed to decompose a hybrid workflow pattern TA into a group of combinations with the basic pattern.

5.1. Overview of algebraic verification process

For a tenant application, we can use the algebraic rules represented above to decrease the scale of verification status space by calculating some sub-combinations. Those rules can sequentially shorten the time of verifying TA. According to the analyses in the previous section, it can be known that verification status can be combined by two operators, $\otimes$ and $\odot$. Operator $\otimes$ corresponds to merging the status of different Com on the same property p, and operator $\odot$ corresponds to merging the status of the same Com on different properties p. In summary, a verification transaction T would have two kinds of basic operations:

- When verifying TA on a property p_1, operator $\otimes$ is used to calculate the status of $Com_1 \cup Com_2$ by merging the status of sub-combinations Com_1 and Com_2. In this way, the verification status of $TA(Com_1 \cup Com_2 \cup ... \cup Com_n)$ on p_1 can be known eventually.
- When verifying TA on a property p_2, if p_2 depends on another property p_3 , and the verification status on p_3 has already been known previously. Then the verification status on p_2 can be calculated by operator $\odot$.

In brief, the basic idea of VA is vertically reaching the verification status of TA on a property by operator $\otimes$, and then horizontally extending the status to that on other properties by the operator $\odot$. The whole process of combination verification algebra is represented as follows:

Step 1: Get all dependency depth of properties by algorithm in *Algorithm 1*.

Step 2: Choose the property p which has the lowest dp from $p\ set$, then start sub-process of merging different Com on the same p.

Step 3: After getting the verification status of TA on p, start sub-process of merging TA on different p.

Step 4: Remove properties which have already been verified from property set.

Step 5: Check out if there is any element left to be verified in the property set. If the property set is not empty, jump into ***Step 1***. Otherwise, generate the verification report and end the process.

Algorithm 1 Property Dependency Depth Algorithm

Require:
Input propSet // Input property set
Ensure:
Output propDepthSet //Output property dependency depth set
1: **foreach** prop ∈ PropSet
2: prop.dependence ← 0 // Initialize property dependency number
3: **foreach** temprop ∈ PropSet
4: **if** isDepend(temprop,prop) //prop depends on temprop
5: prop.dependPropList.add(temprop)
6: prop.dependence←prop.dependence+1
7: **endif**
8: **endfor**
9: **endfor**
10: **foreach** prop ∈ PropSet
11: temprop ← prop. dependPropList.Top()
12: prop.depth ← prop.dependence //Initialize property dependency depth
13: **while** isNotNull(temprop.next())
14: prop. depth ← prop. depth+temprop. dependence
15: temprop ← temprop. dependProp.next()
16: **endwhile**
17: PropDepthSet.add(prop, prop.depth)
18: **endfor**
19: **return** PropDepthSet

Discussion. As illustrated, the Property Dependency Depth Algorithm firstly initializes input property set $PropSet$ and output property set $PropDepthSet$, then traverses property set $PropSet$ and calculates how many times each property $prop$ depends on other property $temprop$. Function $isDepend()$ is to judge whether two properties have dependencies (Lines 1-9). Then it traverses property set $PropSet$ and initializes dependency depth $prop.depth$ of each property $prop$. Last, the algorithm adds up all the property dependency number $temprop.dependence$ of property $prop$. The result of property dependency depth is saved to $prop.depth$ (Lines 13-16).

Example 3. Consider three properties (p_1, p_2, p_3), where p_2 depends on p_1, and p_3 depends on both p_1 and p_2. So $p_3.dependPropList = \{p_1, p_2\}$ and $p_3.dependence = 2$, $p_2.dependPropList = \{p_1\}$ and $p_2.dependence = 1$; meanwhile, $p_1.dependence = 0$. Then we can add up all property dependency number of a certain property and get the result of property dependency depth. So $p_3.depth = p_3.dependence + p_2.dependence + p_1.dependence = 3$, and $p_2.depth = p_2.dependence + p_1.dependence = 1$, while $p_1.depth = 0$.

In order to avoid repeated verification and increase the efficiency of verification, verification status of all Com need to be stored in a status database *STATUS-DB* In VA. There are five data tables in *STATUS-DB*, namely *X-TABLE*, *F-TABLE*, *P-TABLE*, *N-TABLE* and *U-TABLE*. Those tables are applied to save the five status *X*, *F*, *P*, *N* and *U*.

In SaaS, not only does TA customized by a tenant need to be verified, but also all sub-combinations of TA should be verified. Sub-tenant Customizations of TA are applications, which are combinations of a set of sub-components customized by subtenants according to their own demands,so VA can support the demands of Sub-tenant Customizations. In the verification process, all $V(Com, p)$ will be stored in the data table of database. Any new Com that is not verified will be stored in *U-TABLE*. The purpose of the combination verification algebra process is to turn the status of verification records of *U-TABLE* into *F* or *P*, then the records should be moved into *F-TABLE* or *P-TABLE*.

5.2. *Sub-process of Merging Different Com on p*

If TA is under a single pattern, the verification status of TA can be directly merged by operator $\otimes$. However, if TA is under hybrid patterns, we cannot use operator $\otimes$ directly. Then we discuss how to expand the merging process to the cases under hybrid patterns.

(1) Process under single pattern

For combination Com and property p, if $f(W,p) = Attr1$, $V(Com,p)$ cannot be status *X* or *N*, while Com is created and meaningful to be verified. Then we can use the operating table extensively, and the following rules can be concluded:

- ***Rule1-1*** $\exists\ Com^{'}$ in ***SSC Set of Com***, $V(Com^{'},p) = P$ then $V(Com,p) = P$.
- ***Rule1-2*** $\exists\ Com^{'}$ in ***SSC Set of Com***, $(Com^{'},p) = F$, $V(Com,p) = U$. Because $V(Com,p)$ can be *F* or *P*, according to the operating table.

In order to verify Com on p, we should find at least one similar sub-combination which can pass the verification according to the rule ***Rule1-1***. If no similar sub-combination is found, we should decompose those similar sub-combinations iteratively or utilize verification service in VaaS. The detail of sub-process is described as follows:

Step1: For all $Com^{'}$ ($Com^{'}$ in ***SSC Set of Com***) on p, firstly search the verification record of all $V(Com^{'},p)$. If there is one or more records of $V(Com^{'},p)$ in *P-TABLE*, it can be concluded that $V(Com^{'},p) = P$. According to the rule Rule1-1 ,the label status of Com on p should be PASS, and end the process. If all $V(Com^{'},p)$ are not in *P-TABLE*, add $V(Com,p)$ into U-TABLE.

Step2: For all $Com^{'}$ ($Com^{'}$ in ***SSC Set of Com***) on p, all $Com^{''}$ ($Com^{''}$ in ***SSC Set of*** $Com^{'}$). Check if all of the $V(Com^{''},p)$ is in *F-TABLE* or not. If there is a $V(Com^{''},p)$ not in *F-TABLE*, add $V(Com^{'},p)$ into *U-TABLE*.

Step3: Sort the records $V(Com,p)$ of *U-TABLE* by the component number of $Com^{'}$ in ascending order, and put them in a candidate-set. Call the combination verification service provided by VaaS to verify each $Com^{'}$ in candidate-set on p, then move records $V(Com^{'},p)$ into *P-TABLE* or *F-TABLE* according to verification results.

Step4: If any $Com^{'}$ ($Com^{'}$ in ***SSC Set of Com***) passes verification on p, then verification of Com passes. Otherwise, Com fails the verification. End the process.

When $f(W,p) = Attr2$, $f(W,p) = Attr3$, $f(W,p) = Attr4$, the process steps is similar to $f(W,p) = Attr1$.

(2) Process under hybrid workflow patterns

In practical applications, most TAs are under hybrid patterns. For Com under hybrid patterns, it is complex to get its sub-combinations with the same basic pattern. Therefore, combination verification algebra cannot be used in this case directly.

We propose a method to solve this problem. In this method, TA is disassembled and substitute by combinations with more simple patterns. Disassembly means splitting TA into several sub-combinations and substitution means replacing atomic sub-combination with component. An example of disassembling and substituting TA under hybrid patterns is represented as follows to illustrate the method.

Example 4. The structure of TA which consists of seven components under hybrid patterns is shown in *Figure 4*. When dealing with TA, partial components which are inside the dotted line box can be regarded as a whole, which is substituted with a new component C_8. So TA is divided into two parts: $TA^{'}$ and C_8 as shown in *Figure 5*. We know that $TA^{'}$ is under sequence pattern, but C_8 is still under a hybrid pattern. So C_8 can be disassembled and substituted as *Figure 6*.

After dealing with C_8, TA is eventually divided into three parts: $TA^{'}$, $C_8^{'}$ and C_9, and each part is under the basic pattern which can be handled by combination verification algebra process.

Component Disassembly Tree is designed to represent the process above. It is defined as follows.

Definition VII: (Component Disassembly Tree) Component disassembly tree is a tree, root node of the tree is TA itself , and leaf nodes of the tree are components and other nodes which are sub-tenant applications or combinations of TA.

The Component Disassembly Tree of TA is shown in *Figure 7*.

DOI: 10.1002/stvr

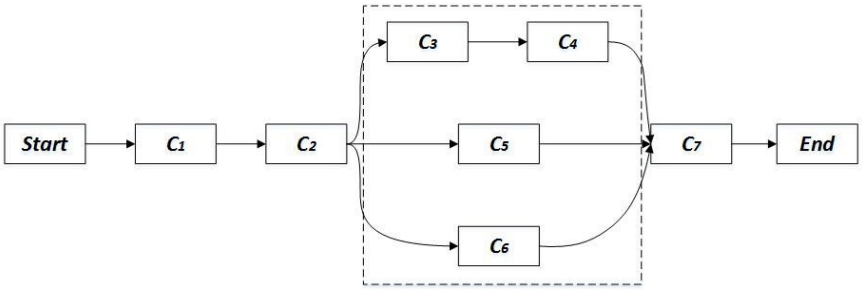

Figure 4. Structure of a TA under Hybrid Patterns

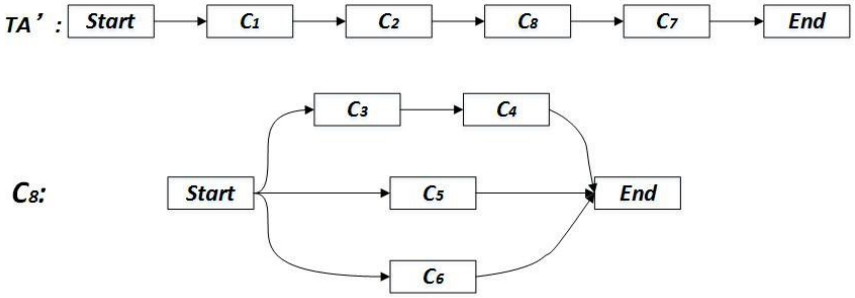

Figure 5. Disassembly Graph of TA

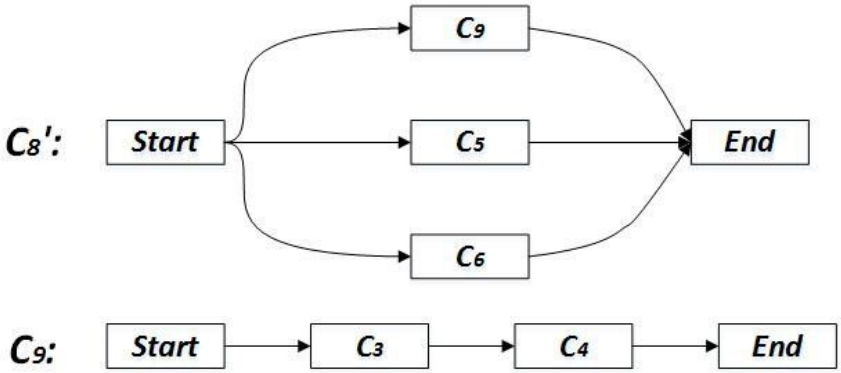

Figure 6. Disassembly Graph of C_8

A method is designed to automatically disassemble workflow graph of TA to a Component Disassembly Tree. Suppose that the workflow graph has no circle, and there is only a pair of source vertex and destination vertex in the workflow graph, the detail of the method is illustrated as follows.

Step1: Initialize the Component Disassembly Tree and make it contain a root with value 'TA'.

Step2: Find all paths of the workflow graph by DFS.

Step3: Search paths from head and tail to find the same nodes. Make the nodes found to be the child nodes of root. And remove the nodes found from the whole paths. If the paths are empty, end process. Otherwise, go to ***Step4***.

Step4: Add a child of root with value 'Com'.

Step5: Sort and group the paths by the first element. For these groups, if a group only has one path ,and there is only one node in the path, then make the node to be a child of 'Com'. Else, add a child node with value 'Com', regard this 'Com' node as root and the group as paths, and go to ***Step3***.

5.3. Sub-process of Merging the Same Com on different p

After TA has been verified on p, the process of merging the same Com on different p by operator $\odot$ is represent as follows:

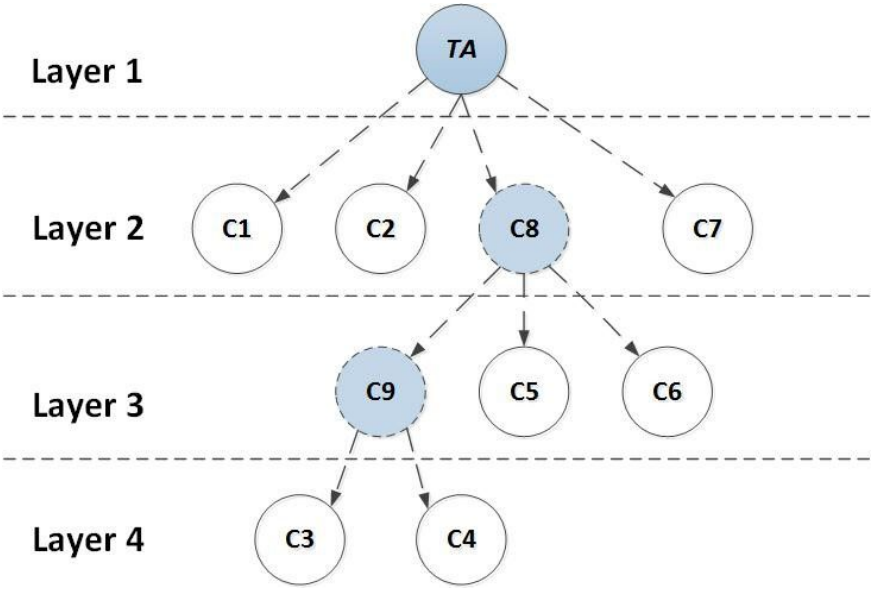

Figure 7. Component Disassembly Tree of TA

Step1: Move p from set to be verified $p\ set_1$, then add p to verified set $p\ set_2$.

Step2: If there is a property $p^{'}$ in $p\ set_1$ has a dependency relationship with property p in $p\ set_2$, move $p^{'}$ into the candidate set and go to ***Step3***. Otherwise, operator $\odot$ cannot be used to merge verification status and end the process.

Step3: Use operator $\odot$ to merge the status on p to status on $p^{'}$. If $(Com, p^{'}) = P$ or $(Com, p^{'}) = F$, move verification record into *P-TABLE* or *F-TABLE*, and move $p^{'}$ into $pset_2$. Otherwise, move $p^{'}$ into $pset_1$.

Step4: If candidate set is not empty, go to ***Step2***. Otherwise, check whether there is a newly-added property in $pset_2$. If yes, go to ***Step2***. Otherwise, end the process.

6. SIMULATION

A simulation experiment is designed to prove the validity and performance of the verification algebra method.

6.1. Simulation Experiment setup

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

At the beginning of the simulation experiment, simulation system should be initialized. The initialization process will complete the following works:

- Simulate a certain number of components according to default parameters for later combinations.
- Simulate a certain number of properties according to default parameters and randomly generate the dependency relationships among properties.
- Randomly generate the failing combinations and corresponding properties according to the existing components and properties.
- Generate tenant applications by combining the components which are randomly chosen.

Table III. Merging Results with Atrr 1,2,3,4

Applications Quantity	Total workload	Attr1	Attr2	Attr3	Attr4
200	167400	97092	105462	155682	80352
400	334800	140616	161373	308016	160704
600	502200	175770	200880	451980	236034
800	669600	180792	214272	595944	301320
1000	837000	209250	242730	736560	351540

After initialization, start the verification process proposed in Section 4. When the process is done, perform statistical analysis of the actual formal verification times, and compare the result with that do not use the verification algebra method.

Three groups of simulation experiments are designed to analyze the combination verification algebra method, we design the simulation experiments from three perspectives:

- Simulation experiment of combining different combinations on the same property.
- Simulation experiment of combining the same combination on different properties.
- Simulation experiment of combining different combinations on different properties.

6.2. Environment of the Simulations

(1) Hardware Environment

The simulation is tested on a machine with a Hadoop distributed computing environment made up of several virtual machines. A virtual machine is used as the master node, and the other seven virtual machines are used as computing nodes. All VMs have the 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, 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 is implemented in Java programming language.

(3) Related parameters

This experiment is designed by referring to Easy SaaS Architecture proposed in [44]. Components in the experiment are divided into four classes: GUI components, Workflow components, Service components and Data components. The total number of components is 100, 30% of which are GUI components, 30% of which are Workflow components, 20% of which are Service components, and 20% of which are Data components. Each tenant application consists of ten components: four GUI components, three Workflow components, two Service components, a data component. According to rules of the Permutation and Combination, there are $C_{30}^{4} \times C_{30}^{3} \times C_{20}^{2} \times C_{20}^{1} = 4.23 \times 10^{11}$ possible combinations. The error rate is set to 0.1%, and failing combinations are randomly generated at the beginning of the experiment.

Before verification starts, all tenant applications are disassembled into combinations under basic pattern by the ***Component Disassembly Tree*** proposed in Section 5.

6.3. Presentation and Analysis of Experiment Results

(1) Simulation experiment I

Simulation experiment I is about combining different combinations on the same property. In this experiment, properties with different attributes are used to be verified among combinations. In our simulation, 100 tenant applications have 83700 sub-combinations which are under basic workflow patterns.

The results of merging combinations on four kinds of properties by Attr operating rules are presented in *Table III*.The total number of applications to be verified is presented in **Applications Quantity**. In addition, **Total workload** is the number of combinations to be verified when Attr rules are not used to merge verification status. Attr is used to indicate the verification status relationships

among different combinations on the same property. The four columns **Attr1**, **Attr2**, **Attr3** and **Attr4** respectively represent the number of combinations to be verified for four different properties, which have four different Attr rules. After combinations status are merged by our verification algebra method, the minimum combinations to be verified are shown in column **Attr1**, **Attr2**, **Attr3** and **Attr4** respectively. In addition, we show the reduction rate in *Figure 8*. We also generate a large number of tenant applications to check whether the verification algebra method can handle the 'Combinatorial Explosion' in the process of verifying MTA applications. The experimental result is shown in *Figure 9*.

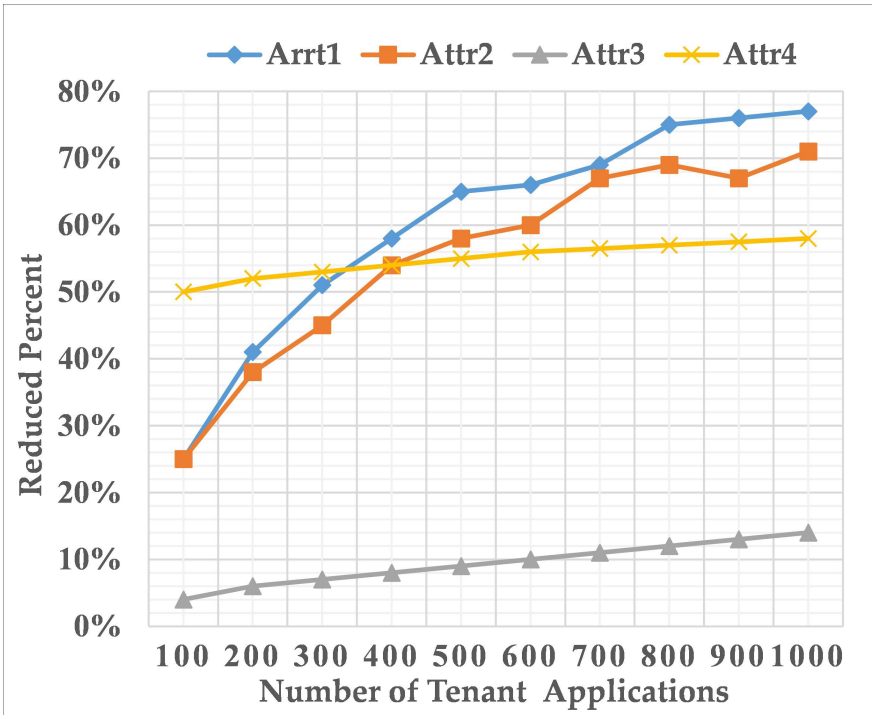

Figure 8. Reduced Percent Lines of Different Combinations on the Same Property

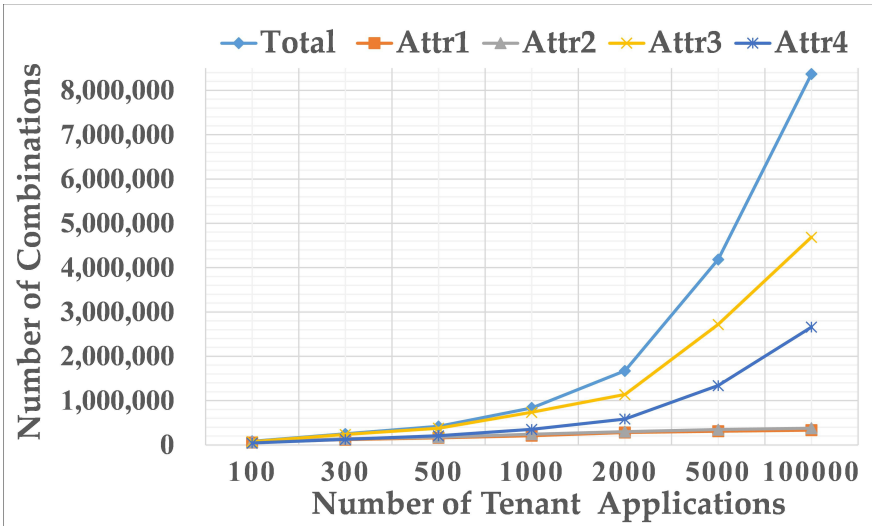

Figure 9. Verification Workload with Attr 1,2,3,4

From the result, it can be concluded that:

- The ratio of reducing the number of combinations to be formally verified has relation with the scale of TA and the attributes presented in Section 3.
- When the magnitude of tenant application is small, the efficiency of verification algebra method is low. But as the scale grows, efficiency of the method is gradually improved. The main reason is that there are more combinations in the system that need to be verified, so the probability that tenant applications have the same combinations goes higher. Meantime, as more and more sub-combinations have been verified, the combination verification algebra can use the combinations with known status to calculate the unknown verification status of the combinations.
- When the magnitude of tenant application is large, combinations with **Attr1** and **Attr2** are relatively stable, while combinations with **Attr3** and **Attr4** increase rapidly. When TAs change from 1000 to 100000 , **Total workload** has rapidly increased by 100 times, but the number of combinations with **Attr1** and **Attr2** is twice as much as before. There are still a large number of combinations with **Attr3** and **Attr4** left to be verified. When the verification status among the combinations satisfy some specific relationships such as **Attr1** and **Attr2**, the verification algebra method is helpful to solve the 'Combinatorial Explosion' in the process of verifying MTA applications.
- Combinations with **Attr1** and **Attr2** have much less strict requirements for status of sub-combinations than those with **Attr3** and **Attr4**. For **Attr1** and **Attr2**, status of combination can be derived as long as one of its sub-combination has a certain status, so **Attr1** and **Attr2** have high efficiency. For **Attr3**, only when all sub-combinations of a combination have the same status, the status of the combination can be deduced. Therefore, **Attr3** has the lowest efficiency. **Attr4** is similar to **Attr3** but a little less strict than **Attr3**, so **Attr4** has a slightly higher efficiency than **Attr3** but much lower than **Attr1** and **Attr2**.

(2) Simulation experiment II

Simulation experiment II is about combining different properties on the same combination. Although there are four kinds of relationship between properties, the essence of them is the dependency relationship. The degree of dependency relationship is the proportion of properties with **PR** accounts in all 50 properties. There are 50 properties with **PR1** in this experiment, and combinations are verified with different degree of dp.

We generate 1000 applications and properties with different degree of dp. The number of combinations which are reduced is shown in *Figure 10*. The reduced percent line in different number of applications simulation is shown in *Figure 11*.

According to the *Figure 10*, the number of cut-down combinations increases with the growth of the dp degree. In *Figure 11*, it can be concluded that the reduced ratio of combinations to be verified is determined by the degree of dp, independent of the number of applications.

(3) Simulation experiment III

Simulation experiment III is about combining different combinations on different properties. In addition, we check whether the verification algebra method changes the verification results in this experiment. Four kinds of attributes and 50 properties are considered, and 20% of those properties have the **PR1** relationship. In addition, we will compare the verification status with original after applying verification algebra, in order to check whether the verification algebra method changes the validation status of a component.

During the process of verification, we save the verification status of each component combination in $file_1$. After applying the verification algebra method, we save the verification statuses in $file_2$. The verification status of each combination is saved in a line. The sequences of two files' component verification are the same. Then we compare the hash values of $file_1$ and $file_2$, from the comparison results we find that their $hash\ values$ are the same.

The result of this experiment is shown in *Figure 12*.

From the result, it can be concluded that:

- For properties with the same attribution, as the number of tenant applications grows, the ratio of reduction is increasing.

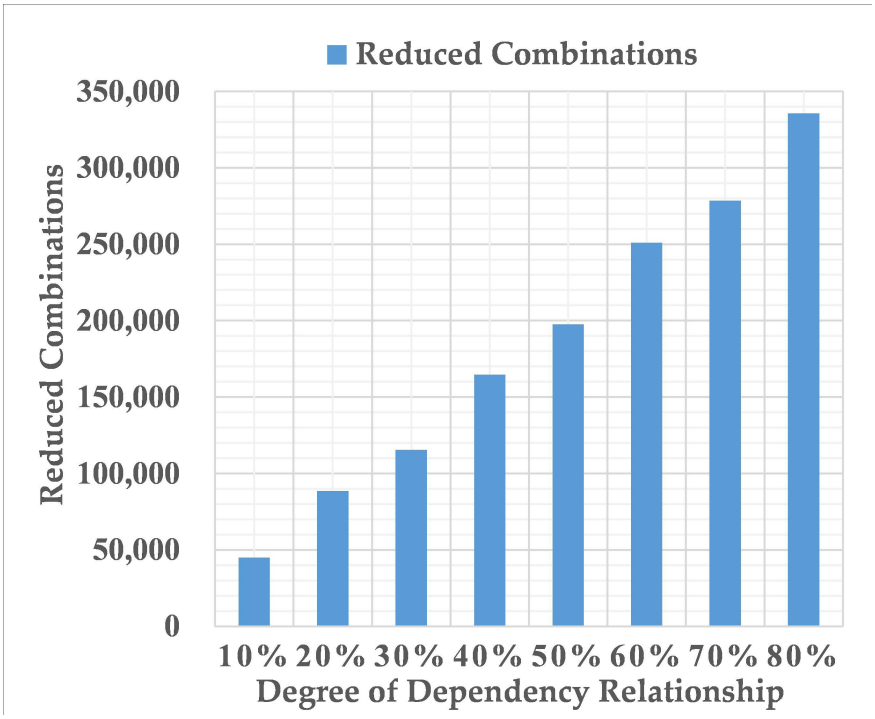

Figure 10. Reduced Combinations with Different dp

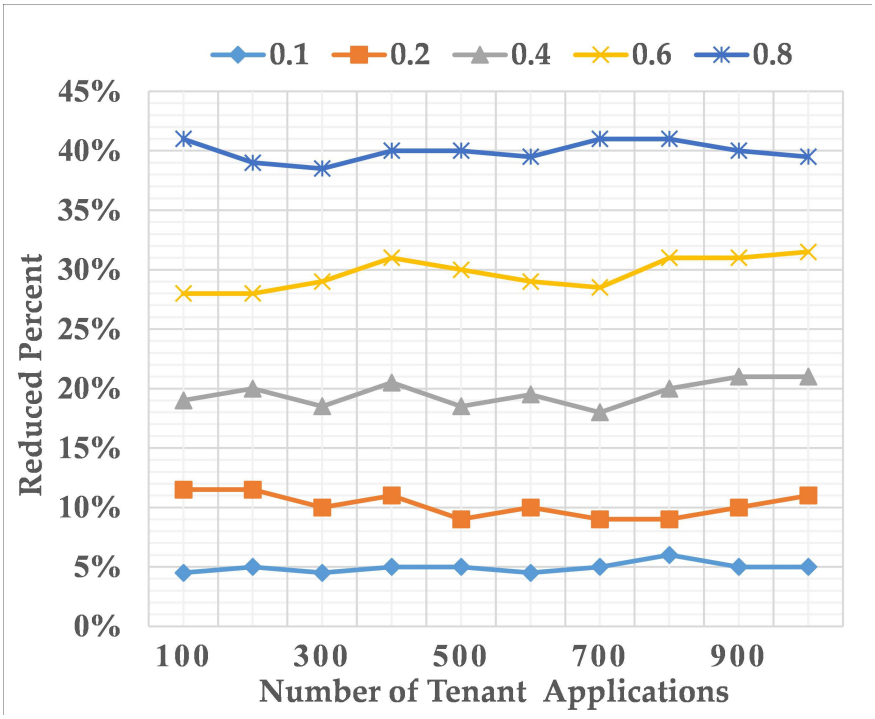

Figure 11. Reduced Percent Lines of Different properties on the Same Combination

- By comparing this experiment with the two former experiments, it can be concluded that the influence of property relationships is fixed. Although both attribute and property relationship are related to the reduction ratio. The attribute has a bigger impact on the efficiency of verification transaction reduction. We find that the attributes of patterns and properties are the keys for the verification algebra.

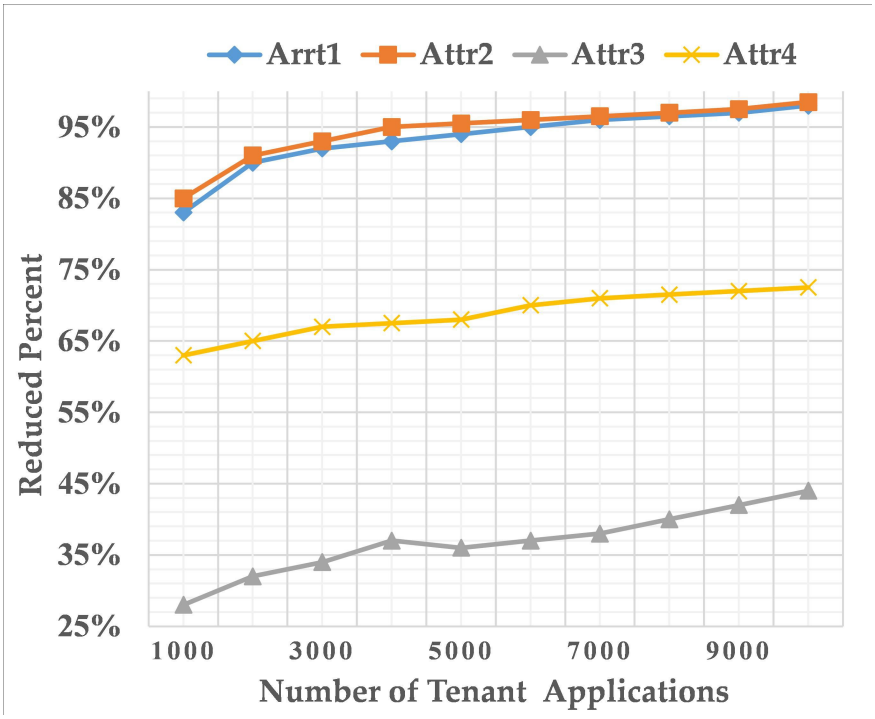

Figure 12. Reduced Percent Lines of Different Combinations on Different Properties

- The method of verification algebra proposed in this paper does not change the verification status of combinations. Unknown combination verifying status can be inferred from previous verification results after applying verification algebra. The result of a full verification cycle for a component composition equals the result of applying the verification algebra method to a partial verification status of a component composition.

7. CONCLUSIONS

This paper proposes an algebraic method VA to address the 'Combinatorial Explosion' problem in MTA VaaS. Four kinds of combination attributes between combination and sub-combination and four kinds of property relationships among properties are addressed to analyze relationships of elements. Then algebraic rules of these attributes and relationships are discussed in detail. According to the algebraic rules, a complete process is proposed to apply the algebraic rules to decrease the number of verification transactions in VaaS. By using previous verification results, VA can calculate unknown combination verification status and cut down component combinations verified in verification tasks. Since only basic pattern combinations can be handled by VA, a decomposing hybrid workflow method named ***Component Disassembly Tree*** is proposed. It can decompose a hybrid workflow pattern TA into a group of combinations with the basic pattern. By simulation experiments, it can be proved that the verification algebraic method proposed is efficient, and greatly decrease the number of the verification transactions in VaaS systems. And the result of a full verification cycle for a component composition equals the result of applying the verification algebra to partial component compositions.

However, the algebraic method VA proposed in this paper still has some shortcomings. The number of combinations that need to be verified after applying VA is affected by the number of initial combinations. There may still be a large number of verification tasks left after applying VA if large quantities of combinations need to be verified initially. Furthermore, only partial properties and relationships can be merged by the VA. In the future, we will propose more rules similar to Attr and PR operating rules to expand the application scenarios of VA.

8. ACKNOWLEDGMENTS

This work was supported by the Project of National key research and development of China under Grant 2018YFB1402702, National Natural Science Foundation of China under Grant 61672074 and 61672075, Funding of Ministry of Education and China Mobile MCM20180104, State Key Laboratory of Software Development Environment (No. SKLSDE-2020ZX-21), Science and Technology Major Project of Yunnan Province(202002AA100007).

REFERENCES

1. Cusumano M. Cloud computing and SaaS as new computing platforms[J]. *Communications of the ACM*, 53(4): 27-29, 2010.
2. Tsai W T, Bai X Y, Huang Y. Software-as-a-service (SaaS): perspectives and challenges[J]. *Science China (Information Sciences)* ,57(5): 1-15, 2014.
3. Yu L, Tsai W T, Chen X, et al. Testing as a Service over Cloud[C]. *In Proc. IEEE Int'l Symposium on Service Oriented System Engineering*,181-188, 2010.
4. Tsai W T, Huang Y, Bai X.Y, Gao J, Scalable architecture for SaaS[C], *In Proc. IEEE Int'l Symposium on Object Component Service oriented Real-time Distributed Computing*, 2012.
5. Zhao Y, Sanán D, Zhang F, et al. Formal specification and analysis of partitioning operating systems by integrating ontology and refinement[J]. *IEEE Transactions on Industrial Informatics, 12(4): 1321-1331*, 2016.
6. Hu K, Lei L, Tsai W T. Multi-tenant Verification-as-a-Service (VaaS) in a cloud[J]. *Simulation Modelling Practice and Theory*, 122-143, 2016.
7. Tsai W T, Colbourn C J, Luo J, et al. Test algebra for combinatorial testing[C]. *In Proc. Int'l Workshop on Automation of Software Test*, 19-25, 2013.
8. Tsai W T, Luo J, Qi G, et al. Concurrent test algebra execution with combinatorial testing[C]. *In Proc. IEEE Int'l Symposium on Service Oriented System Engineering*, 35-46, 2014.
9. Wu W, Tsai W T, Jin C, et al. Test-algebra execution in a cloud environment[C]. *In Proc. IEEE Int'l Symposium on Service Oriented System Engineering*, 59-69, 2014.
10. Tsai W T, Qi G. Integrated fault detection and test algebra for combinatorial testing in TaaS (Testing-as-a-Service)[J]. *Simulation Modelling Practice and Theory*, 108-124, 2016.
11. Arcuri A, Briand L. Formal analysis of the probability of interaction fault detection using random testing[J]. *IEEE Transactions on Software Engineering*, 38(5): 1088-1099 , 2011
12. Qi G, Tsai W T, Colbourn C J, et al. Test-algebra-based fault location analysis for the concurrent combinatorial testing[J]. *IEEE Transactions on Reliability*, 67(3): 802-831, 2018.
13. Tsai W T, Qi G. Integrated TaaS with fault detection and test algebra[J]. *Combinatorial Testing in Cloud Computing*, 115-128, 2017.
14. Tsai W T, Qi G. Adaptive reasoning algorithm with automated test cases generation and test algebra in SaaS system[J]. *Combinatorial Testing in Cloud Computing*, 83-99, 2017.
15. Tsai W T, Qi G. Test Algebra Execution in a Cloud Environment[J], *Combinatorial Testing in Cloud Computing*, 69-82, 2017.
16. Yuzhuang X , Kai H , Jiehua H , et al. An algebraic approach for verifying compositions of SDN components[C]. *In Proc. Int'l Conf. on Communications in China (ICCC Workshops)*, 2016.
17. Wittig M, Wittig A, Whaley B. Amazon web services in action[M]. *Manning*, 2016.
18. Moreno I S, Garraghan P, Townend P, et al. An approach for characterizing workloads in google cloud to derive realistic resource utilization models[C]. *In Proc. IEEE Int'l Symposium on Service Oriented System Engineering*, 49-60, 2013.
19. Grindal M, Offutt J, Andler S F. Combination testing strategies: a survey[J]. *Software Testing, Verification and Reliability*, 2005, 15(3): 167-199.
20. Zhang G, Ravishankar M N. Exploring vendor capabilities in the cloud environment: A case study of Alibaba Cloud Computing[J]. *Information & Management*, 56(3): 343-355, 2019.
21. Huang R, Xie X, Chen T Y, et al. Adaptive random test case generation for combinatorial testing[C].*In Proc. IEEE Annual Computer Software and Applications Conference*, 2012.
22. Tsai W T, Qi G. Integrated fault detection and test algebra for combinatorial testing in TaaS (Testing-as-a-Service)[J]. *Simulation Modelling Practice and Theory*, 108-124, 2016.
23. Baier C, Katoen J P. Principles of model checking[M]. *MIT press*, 2008.
24. Alur R. Formal verification of hybrid systems[C]. *In Proc. ACM Int'l Conf. on Embedded software*, 273-278, 2011.
25. Huang S Y, Cheng K T T. Formal equivalence checking and design debugging[M]. *Springer Science & Business Media*, 2012.
26. Lei Pi. Architecture description language semantic and behavior analysis. PH.d thesis, 2011.
27. Basu A, Bozga M, Sifakis J. Modeling heterogeneous real-time components in BIP[C]. *In Proc. IEEE Int'l Conf. on Software Engineering and Formal Methods*, 3-12, 2006.
28. Lee I, Choi J Y, Kwak H H, et al. A family of resource-bound real-time process algebras[C]. *In Proc. Int'l Conf. on Formal Techniques for Networked and Distributed Systems*, 443-458, 2001.
29. Feiler P H, Gluch D P, Hudak J J. The architecture analysis & design language (AADL): An introduction[R]. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

30. Feiler P H, Lewis B A, Vestal S. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems[C]. *In Proc. IEEE Conf. on Computer Aided Control System Design*, 1206-1211.
31. Berthomieu B, Bodeveix J P, Farail P, et al. Fiacre: an intermediate language for model Verification in the Topcased Environment. In Proc. European Congress on Embedded Real-Time Software, 2008.
32. Pi L, Yang Z, Bodeveix J P, et al. A comparative study of FIACRE and TASM to define AADL real time concepts[C]. *In Proc. IEEE Int'l Conf. on Engineering of Complex Computer Systems*, 347-352, 2009.
33. Ouimet M, Lundqvist K. The TASM Language Reference Manual, Version 1.1[J]. *Cambridge*, 2006.
34. Bigraph.http://www.itu.dk/research/pls/wiki/index.php/A-Brief-Introduction-To-Bigraphs.
35. Yu L, Tsai W T, Wei X, et al. Modeling and analysis of mobile cloud computing based on bigraph theory[C]. *In Proc. IEEE Int'l Conf. on Mobile Cloud Computing, Services, and Engineering*, 67-76, 2014.
36. Mikk E, Lakhnech Y, Siegel M, et al. Implementing statecharts in PROMELA/SPIN[C]. *In Proc. IEEE Workshop on Industrial Strength Formal Specification Techniques*, 90-101, 1998.
37. Schaefer I, Sauer T. Towards verification as a service[C]. *In Proc. Int'l Workshop on Eternal Systems*, 2011.
38. Zahoor E, Munir K, Perrin O, et al. A bounded model checking approach for the verification of web services composition[J]. *International Journal of Web Services Research*, 10(4): 62-81, 2013.
39. Moscato F. Model driven engineering and verification of composite cloud services in metamorp (h) osy[C]. *In Proc. Int'l Conf. on Intelligent Networking and Collaborative Systems*, 635-640, 2014.
40. Baset S A. Cloud SLAs: present and future[J]. *ACM SIGOPS Operating Systems Review*, 46(2): 57-66, 2012.
41. Mancini T, Mari F, Massini A, et al. SyLVaaS: System level formal verification as a service[J]. *Fundamenta Informaticae*, 149(1-2): 101-132, 2016.
42. Tsai W T, Shao Q, Li W. OIC: Ontology-based intelligent customization framework for SaaS[C]. *In Proc. IEEE Int'l Conf. on Service-Oriented Computing and Applications*, 1-8, 2010.
43. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros. Workflow Patterns[J]. *Distributed and Parallel Databases*, 14(1):5-51, 2003.
44. Tsai W T, Huang Y, Shao Q. EasySaaS: A SaaS development framework[C]. *In Proc. IEEE Int'l Conf. on Service-Oriented Computing and Applications*, 1-4, 2011.

Kai Hu is a professor at Beihang University, China. He received his Ph.D. 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. Email: hukai@buaa.edu.cn

Ji Wan received the MEng degree in computer science and technology from Zhengzhou University, China, in 2019. He is currently working toward the PhD degree at the School of Computer Science and Engineering, Beihang University. His research interests include distributed systems, block chain and urban computing. Email:wanji@buaa.edu.cn

Kan Luo is graduated Sichuan University in 2016 with a bachelor's degree. And he received the MEng degree at the School of Computer Science and Engineering, Beihang University, in 2019. His reseach direction is software engineering and formal methods. He is the corresponding author. Email:looken@foxmail.com

Yuzhuang Xu, born in 1991, received the MEng degree at the School of Computer Science and Engineering, Beihang University, in 2017. He is now technical researcher of China UnionPay Electronic Payment Research Institute. His research intersts include Blockchain and distributed systems. Email: xuyuzhuang@unionpay.com

Zijing Cheng, born in 1972, received his Ph.D. degree at the School of Computer Science and Engineering, Beihang University. He is a researcher at State Key Laboratory of Space-Ground Integrated Information Technology, Beijing Institute of Satellite Information Engineering. His research interests include aerospace internet of things, satellite network communications, and space information networks. Email: linuxdemo@126.com

Wei-Tek Tsai received his B.S. from Computer Science and Engineering from Massachusetts Institute of Technology at Cambridge in 1979, M.S. and Ph.D. in Computer Science from University of California at Berkeley in 1982 and 1986. He is now Professor in the School of Computer Science and Engineering at Beihang University, Beijing, China, and Emeritus Professor at Arizona State University. He has authored more than 400 papers in software engineering, service-oriented computing, and cloud computing. He travels widely and has held various professorships in US, Asia and Europe. Email: wtsai@asu.edu