Big SaaS: The Next Step Beyond Big Data

Hong Zhu, Ian Bayley, M. Younas, David Lightfoot, Basel Yousef and Dongmei Liu
Applied Formal Methods Research Group
Department of Computing and Communication Technologies
Oxford Brookes University, Oxford OX33 1HX, UK
E-mail: hzhu@brookes.ac.uk

Abstract

Software-as-a-Service (SaaS) is a model of cloud computing in which software functions are delivered to the users as services. The past few years have witnessed its global flourishing. In the foreseeable future, SaaS applications will integrate with the Internet of Things, Mobile Computing, Big Data, Wireless Sensor Networks, and many other computing and communication technologies to deliver customizable intelligent services to a vast population.

This will give rise to an era of what we call Big SaaS systems of unprecedented complexity and scale. They will have huge numbers of tenants/users interrelated in complex ways. The code will be complex too and require Big Data but provide great value to the customer.

With these benefits come great societal risks, however, and there are other drawbacks and challenges. For example, it is difficult to ensure the quality of data and metadata obtained from crowdsourcing and to maintain the integrity of conceptual model. Big SaaS applications will also need to evolve continuously. This paper will discuss how to address these challenges at all stages of the software lifecycle.

1 Introduction

Software-as-a-Service (SaaS) is a cloud computing model in which computer applications are delivered to the users as services [1, 2]. It contrasts with the hitherto more conventional practice of selling applications as products to be owned by the customer, and has led to a revolution in what functions can be offered. Table 1 lists just some of the many successful SaaS applications that have arisen over the past few years.

There is, however, less research on SaaS than on other related areas such as Big Data, Internet of Things (or Cyber-Physical Systems), Wireless Sensor Networks etc. For this reason, it is desired to assess the start of the art for both research and applications. This paper does this and then identifies future directions, recognizes the main challenges, outlines our assumptions and approach, and finally recounts recent progress.

The paper is organized as follows. Section 2 defines the notion of Big SaaS applications. Section 3 identifies the major challenges in their development. Section 4 discusses approaches to solving these problems and reports our preliminary work. Section 5 concludes the paper with a summary.

Table 1 Examples of SaaS Applications

SaaS	**Application Area**
Booking.com	Hotel booking
EasyChair	Conference management
Ebay	Online shopping
Facebook	Web portal and Social networking media
Gmail	Message communication
Just Eat	Online order for Take Away restaurants
Lastminute.com	Travel agency
LinkedIn	Social networking media for professionals
Moodle	Online Learning Platform
ResearchGate	Social networking media for researchers
Rightmove	Estate Agency
SalesForce.com	Customer Relationship Management
WhatsApp	Instant message communication

2 The Growth of SaaS

Those SaaS applications well known to the public today are mostly small, but our vision of the near future is that an era of Big SaaS is emerging. Here, we define Big SaaS applications as those SaaS applications with the following characteristics.

(1) *Big Tenancy*. A Big SaaS application usually serves a large number of tenants and users that may well be interrelated in a complex way.

Examples of this include:

- *Just Eat:* 40,800 takeaway restaurants (in 13 countries) and has 6 million users with active accounts.
- *Booking.com:* 638,960 properties (in 211 countries) with over 800,000 room-nights reserved per day.
- *Rightmove (UK's* largest online estate property advertisement portal): 19,304 agent and new homes advertisers, for more than 1 million properties.

Examples of complex interrelationships include hierarchies (e.g. a tenant may have sub-tenants etc.) and users being associated with many tenants or no particular tenants.

(2) *Big Data*. Large volumes of data will be processed when the number of tenants and users is large.

For example, in January 2014, the Rightmove.com website had a record 100 million visits viewing 1.5 billion

pages.

(3) *Big Code*. For a Big SaaS application, the software will be typically large in size and high in complexity.

Already, SaaS applications are connected to social media or even offer their own domain-specific social networking. SalesForce and Moodle are examples of this. Many already have mobile phone or tablet apps. Inevitably, in the near future, this will extend to Internet of Things, Wireless Sensor Networks, robots etc, making the size and complexity of the code even greater.

(4) *Big Value*. SaaS applications already provide extra services that were hitherto not possible.

For example, *Booking.com* provides two types of cross-tenant services that individual hotel websites cannot: (a) for the hotel customers, access to a network of over 8000 affiliate partners, (b) for property owners, personalized account management to help to optimize revenue. Similarly, *Rightmove.com* claims that property sellers are 5x more likely to find a buyer here than any other website.

Because of this Big Value, SaaS applications generate more revenue and profits with greater productivity than ever before, and it seems likely that this trend will continue. For example, *Rightmove* generated £167m revenue in 2014, up 19% from £140m in 2013, with a similar increase in profits.

So, it seems likely that SaaS applications will advance towards Big SaaS and Big Value in particular.

3 The Challenges

The development of Big SaaS applications poses three types of challenges common to all socio-technical systems.

(1) *Social challenges,* for society as a whole, to accept the changes to various business, finance, legal, ethical and moral aspects;

(2) *Technical challenges,* for industry and researchers, to develop new techniques and novel applications of existing techniques; and finally,

(3) *Engineering challenges,* for engineers and methodologists, to develop new processes, methods and tools to produce applications systematically, efficiently and even automatically.

Recent effort has focused on enabling techniques for SaaS applications. The engineering, on the other hand, is still ad hoc so we will focus only on this. These are what we recognize as the grand challenges to the advance of Big SaaS.

3.1 Societal Risks

For a SaaS application, the risk $Risk_{SaaS}$ of failure is:

$$Risk_{SaaS} = R \times T \times C,$$

where T is the number of tenants reside in the system; R is the failure rate of the system; C is the average consequence of a failure per tenant.

For a software application system that is owned by the customers, the total risk $Risk_{WS}$ of failure globally is:

$$Risk_{WS} = R' \times C' \times S,$$

where S is the number of copies of the system running at the same time globally; R' is the failure rate of the system, and C' is the average consequence of a failure to the customer who runs a copy of the software.

Assume that each tenant runs one copy of the system (i.e. T=S), and that the SaaS is of the same level of reliability as the customer owned software (i.e. $R = R'$). Then, we have that $Risk_{SaaS} = Risk_{WS}$, if C=C'.

From this one can conclude that the two modes of software have equal risks of failure. However, the calculation makes sense only for so-called *individual risks*. There is, however, a concept of *societal risks*, borrowed from safety engineering, where the risks from SaaS are considered greater.

In general, individual risk is the risk for one person of loss of property or life due to system failures. In safety engineering, whether the risk is tolerable can be judged relatively easily for individuals as people knowingly take and accept risks all the time. Travelling in a car brings the risk of an accident but a train crash that kills many people causes an immense public reaction even many more die per year on roads than on trains.

These situations are addressed by estimating *societal risk*, expressed as the relationship between the probability of a catastrophic incident and the number of users affected. It can be represented as an *F-N* curve that plots the expected frequency (*F*) of failure and the number (*N* or more) of users affected by each failure. Figure 1 illustrates the difference between societal risks for SaaS and those for customer-owned software of similar reliability.

These risks are exacerbated if failure recovery is slow, as with the two recent outages of *Salesforce*'s CRM system. They each took more than 10 hours to recover, during which users of more than 100,000 tenants were deprived of the service.

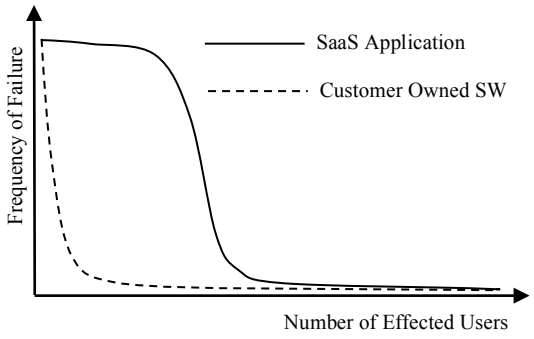

Figure 1 Illustration of the Difference in Societal Risks

Therefore, it is crucial for SaaS application developers to reduce the societal risk significantly to an acceptable level.

3.2 Trustable Crowdsourcing

When there are a large number of tenants, it is highly desirable that a SaaS application supports customization so

that the specific needs of the customers and their users can be accommodated. However, for Big SaaS, such customization cannot be done by the service provider manually. A solution that adopted by almost all existing successful SaaS applications is *crowdsourcing*. This means that the customers perform customization themselves.

For example, *Rightmove* provides a facility for the estate agents to upload themselves information on the properties for sale or to let. Likewise, *Booking.com* enables property owners to set room prices and room availabilities. Similarly, *EBay* enables sellers to enter the information about the goods for sale and the method of payment. Such facilities are fairly simple, however, when compared to *Salesforce*'s facility to let customers build their own applications. An unsolved problem is how to ensure the quality of data and of system configurations obtained by crowdsourcing. This is the second grand challenge to Big SaaS.

3.3 Continuous Evolution

Continuous evolution has been applied to software development practice for web-based systems, as a part of agile methodologies. In this approach, a software system is revised, tested and updated so frequently that the notion of versions and releases no longer makes sense. Moreover, continuous evolution also requires that such updates and releases go live without any interruption to service. This is of paramount importance for Big SaaS but the unprecedented scale and complexity of Big SaaS presents a challenge.

Imagine the situation where hundreds of thousands of tenants each have their own customized version of the system running simultaneously on a number of big clusters distributed around the globe. At the same time numerous new tenants are also performing customization and configuration to join the system. As both of these are happening, developers are committing multiple changes to the system in parallel to fix bugs, to introduce new functions, and to refactor system structure. These changes will inevitably interact with each other while each change may have devastating impact for a large number of users.

After a few days of such frequent modifications, the relations between the components could soon become a spaghetti-like mess. No current software change impact analysis tool could be used here and yet updates will have to go live without interruption to the service. The pressure to complete the testing, verification and validation of each change within a short time with a high adequacy will be several magnitudes higher than ever before.

To enable Big SaaS to be evolved continuously, we must overcome the barriers in software engineering, especially the methods and tools for change impact analysis, for testing, verification and validation, and for on-line refactoring of software structure.

3.4 Conceptual Integrity

Conceptual integrity is one of the key features of a good software design. It means that there is a simple conceptual model of the system in which its structure, functionality and dynamic behavior can be understood.

It appears that the design of a good conceptual model for a Big SaaS application and maintaining its integrity both play a crucial role in development and maintenance. They also play a role in the customization and continuous evolution of the system. Currently, such a conceptual model is rarely formally defined, and often not even documented explicitly, but conveyed instead informally through demonstrations, case studies, online training materials, marketing articles, etc. The advantages of such an approach is that it is user-oriented, but it leaves much scope for ambiguity, incompleteness and misunderstanding.

On the other hand, most online documentation is too developer-oriented, with technical details in place of information about the conceptual model. Ontology and semantic web services can provide user-understandable descriptions of services at the conceptual model level. However, a weakness of ontology based service descriptions is that they are fragmented. Moreover, such documentation and descriptions of services are not verifiable and testable. A link seems missing from the conceptual model to low-level system specification.

4 Research Directions

In this section, we seek for potential solutions to the engineering problems raised in the previous section. We focus on four phases of the software development lifecycle: functional specification, architectural design, implementation and testing. For each of these, we will briefly review the existing work, outline our approach, report the preliminary progresses we have made so far, and point out directions for future research.

4.1 Design: Fault Tolerance Architectures

The societal risk must be addressed by appropriate architectural design of SaaS applications. Chong and Carraro asserted that "*A well-designed SaaS application is scalable, multi-tenant-efficient, and configurable*" [1]. These are the three key differentiators that separate it from a poorly-designed SaaS application. Based on architectural features, they proposed a 4-level maturity model of SaaS applications shown in Figure 2.

Level 1 is ad-hoc, the least mature, and essentially the same as the traditional application service provider (ASP) model of software delivery. Each subsequent level adds one of the three key features (configurability, multi-tenant efficiency, scalable in that order). It is no surprise that almost all successful SaaS applications nowadays employ an architecture model of level 3 and 4, and it seems inevitable that level 4 will be needed for Big SaaS, because, as Chong and Carraro argued, "[such] *a SaaS system is scalable to an arbitrarily large number of customers … without requiring additional re-architecting of the application, and changes*

or fixes can be rolled out to thousands of tenants as easily as a single tenant" [1].

However, this architecture has not addressed the societal risks caused by system level failures. Addressing this problem, in [3] we suggested integrating the architecture with a fault tolerance facility to reduce the consequences of system-scale failures with reduced probability of failure and quicker recovery from failure.

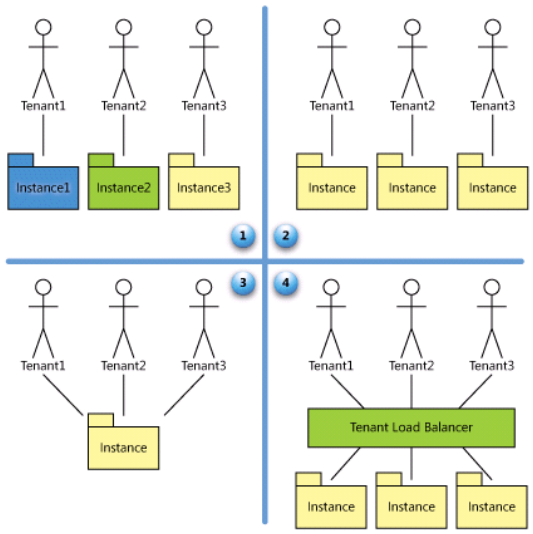

Figure 2 Four-Level SaaS Maturity Model [1]

Fault-tolerance is one of the most challenging issues of distributed and high performance computing [4]. The extensive research in the past few years for cloud computing in particular can be classified according to the fault to be tolerated.

Resource-level fault tolerance aims to achieve high reliability in individual computing resources, such as processor, memory, I/O and network bandwidth, which are lent to users as services, etc. [5,6].

Infrastructure-level fault tolerance techniques include those for virtual machines (VM) or virtual clusters [7], with required availability and reliability via tolerance of underlying hardware failures [8, 9].

At platform level, fault tolerance facilities have been provided in various parallel programming models, such as MapReduce, in which a failed map or reduce task is restarted and/or relocated to a new compute node. The performances of two most commonly used checkpoint / restart techniques for distributed systems, i.e. the Distributed Multi-Threaded Checkpointing and Berkeley Lab Checkpoint/Restart library, have been evaluated in Amazon Elastic Compute Cloud EC2 environment [10].

However, there is no work at application level for SaaS. Moreover, almost all research on fault tolerance in cloud computing assumes that a set of virtual machines are deployed on a number of physical servers and a virtual machine is created for one tenant/user. Thus, they are only applicable to those SaaS applications in the multi-instance architecture of Chong and Carraro's level 2, but not suitable for those in the multi-tenancy architectures of level 3 and 4.

In summary, while some of the above techniques are useful to reduce failure rate of lower level entities, they have not addressed satisfactorily the problem of the high societal risks of Big SaaS. The current practice still relies on traditional periodical backup operations. For example, *Salesforce* backs up all data to a tape storage on a nightly basis. This traditional checkpoint-and-rollback fault tolerance technique is unsatisfactory for Big SaaS applications. In fact, *Salesforce*'s tenants also use third party facilities for backing up their own data.

Addressing this problem, in [3], we proposed a new approach called *tenant-level checkpointing* and implemented a prototype called *Tench*. In this approach, instead of saving the whole system's state, each checkpointing only saves a part of system state related to a specific tenant.

This is important because saving the state of the whole system with one checkpointing operation will cause I/O contention and long delays, as all users of all tenants lose access to the system.

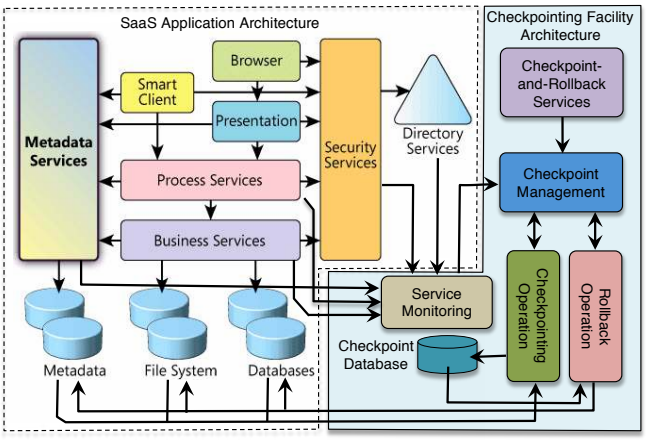

Figure 3 Integration of a fault tolerance facility with SaaS Application Architecture

Figure 3 shows the architecture of such a fault tolerance facility and how it is integrated with the service-oriented SaaS application architecture [1].

In comparison with existing bulk checkpointing techniques, our preliminary theoretical and empirical studies demonstrated that tenant-level checkpointing increase the performance by a factor of O(N), where N is the number of tenants [11]. It has the following advantages.

First, while a SaaS application runs continuously, tenant-level checkpointing can target a specific tenant when the users of the tenant are less active. Thus, a checkpoint can be created without causing too much disruption to normal operations of the system, as requests for services from other tenants are not blocked.

Second, tenants with different quality of service

requirements (e.g., different reliability levels) can be treated differently by having different checkpoint frequencies.

Third, tenant-level checkpointing can be implemented to block only those users of the tenant being checkpointed without affecting any other users. The experiments reported in [3] have shown that the latency of creating a checkpoint for a tenant only depends on the size of the tenant's state. It is independent of the number of tenants.

Moreover, partial checkpointing enables different types of data to be treated differently, with the more important data being checkpointed more frequently. An example of higher priority data would be metadata as it plays an important role in SaaS applications.

Finally, but most importantly, recovery from a system-scale failure can proceed tenant by tenant so that the most important tenants are roll-backed first. This significantly reduces the total outage time and hence the societal risk of system-scale failures.

It is worth noting that VM checkpointing, replication and live migration facilities [12] not only provide fault tolerant solutions to reliability problems, but also balance service work load [13], reduce system energy consumption of data centers [14], and can even the cost of subscription per user [15]. Similar benefits can be obtained from a tenant-level checkpointing facility like Tench for SaaS applications that do not run on virtual machines.

Therefore, tenant level checkpointing could be a viable fault-tolerance solution to Big SaaS' societal risk problem.

4.2 Specification: Algebraic Method

Formal methods have proved their value by their successful applications in safety-critical systems. They can significantly improve software reliability and ensure system safety. Their application in the development of Big SaaS can reduce their societal risk, too.

Although this is considered to be a myth [16, 17], formal methods are widely regarded too expensive to be used. However, the great value of Big SaaS applications makes formal methods viable as its cost would then be justifiable. They can also be easy to learn for ordinary software engineers [18].

Moreover, we believe that formal methods can also provide better solutions to the problems of maintaining conceptual integrity, trustworthy crowdsourcing, and continuous evolution. The following reports our preliminary work on how formal methods address these issues.

4.2.1 Support for Crowdsourcing-Based Customization

As discussed in Section 2, it is highly desirable to include a crowdsourcing-based customization facility in Big SaaS applications. In this approach, services are discovered and composed by the customers with little support from the service provider. One approach to realize such customization is to employ semantic descriptions of the services as illustrated in Figure 4.

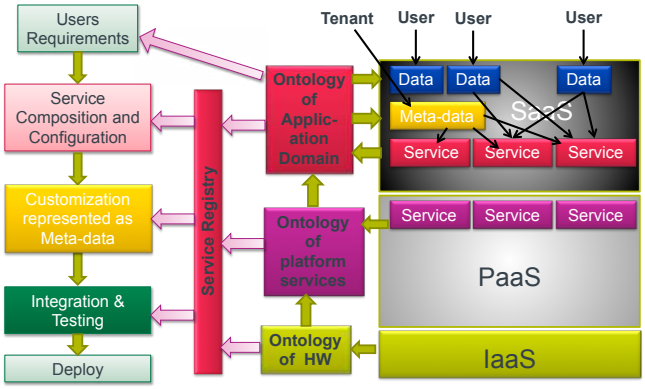

Figure 4. Customization of a SaaS Application in SOA

The results of these customizations and compositions must be of high reliability, due to our requirement to minimize societal risks. To achieve this service semantics need accurate descriptions, which should also be the following:

- *Comprehensible:* easy for users to understand even if they have no IT professional knowledge or skills.
- *Abstract:* the design and implementation details hidden from the users for comprehensibility and also to protect intellectual property.
- *Machine-Searchable* for the discovery, composition and configuration of services.
- *Testable* so that service providers and users can both verify the service's correctness with respect to semantic descriptions.

However, no existing technique satisfies all of these requirements. They tend to fall into two categories. The majorities are based on ontology and use a vocabulary to annotate services. The others are based on the mathematical notations of formal methods.

Semantic Web Services are an example of the former approach [19] and OWL-S was the first major ontology definition language for this purpose [20]. It provides a set of constructs for describing the properties and capabilities of Web Services in a machine-readable format. Formal methods were applied to provide a precise mathematical meaning in a formal ontology. An alternative approach is the Web Service Modelling Ontology (WSMO) [21], which is a conceptual model that uses the Web Services Modelling Language (WSML) [22]. As well as Big Web Services, work has also been carried out on how to specify the semantics of RESTful web services, such as, MicroWSMO/ hRESTS [23], WADL [24] and SA-REST [25].

The above works all take the same approach to specify the semantics of services. That is, a vocabulary is defined by ontology of its application domain to give the meanings of the input and output parameters, as well as the functions of the services. Such descriptions are easy for human developers to understand and efficient for computers to process. However, they cannot provide a verifiable and

testable definition of a service's function, because any ontology is limited to stereotypes formed from the relationship between the concepts and their instances.

Formal methods, as an alternative to the ontological approach, have been developed over the past 40 years to define the semantics of software systems in mathematical notations. One such formal method, algebraic specification was first proposed in the 1970s as an implementation-independent specification technique for defining the semantics of abstract data types. Over these years, it has been advanced to specify concurrent systems, state-based systems and software components, all based on solid foundations of the mathematical theories of behavioural algebras [26] and co-algebras [27]. We argue that it is particularly suitable for the development of Big SaaS.

Algebraic specifications are at a very high level of abstraction. They are independent of any implementation details. One attractive feature they have is that they can be used directly in automated software testing; see Section 4.4. This feature is particularly important for SaaS engineering, because, when services are customized and composed together by the customer, testing must be performed automatically without the developer's support.

In [28], we investigated the application of the algebraic specification method to service-oriented software by extending and combining the behavioural algebra and co-algebra techniques. The algebraic specification language CASOCC, which originally designed for traditional software entities, such as abstract data types, classes and components, was extended to CASSOC-WS for the formal specification of Big Web Services. A tool was developed to automatically generate the signatures of algebraic specifications from WSDL descriptions of Big Web Services. CASOCC-WS was also applied to RESTful web services [29]. A tool was developed to check syntax-level consistency of formal specifications. A case study was conducted applying CASOCC-WS to a real industrial system, GoGrid. Based on these works, a new algebraic formal specification language called SOFIA [43] was proposed to improve the usability of algebraic specification languages when applied to services.

However, algebraic specifications and other formal methods do not directly support efficient searching of services. To bridge the gap between algebraic specification and ontological descriptions, we proposed in [30] to derive the former from the latter, thereby augmenting algebraic specification with the machine-readable and human-understandable attributes of ontology. A software tool called TrS2O (*Tr*anslator from *S*pecification *to O*ntology) has been designed and implemented [30]. It translates formal specifications in SOFIA to ontological descriptions of services in OWL. Figure 6 shows the overall structure of the TrS2O Tool.

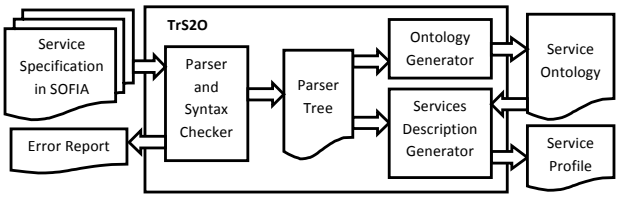

Figure 6. The Overall Structure of The TrS2O Tool

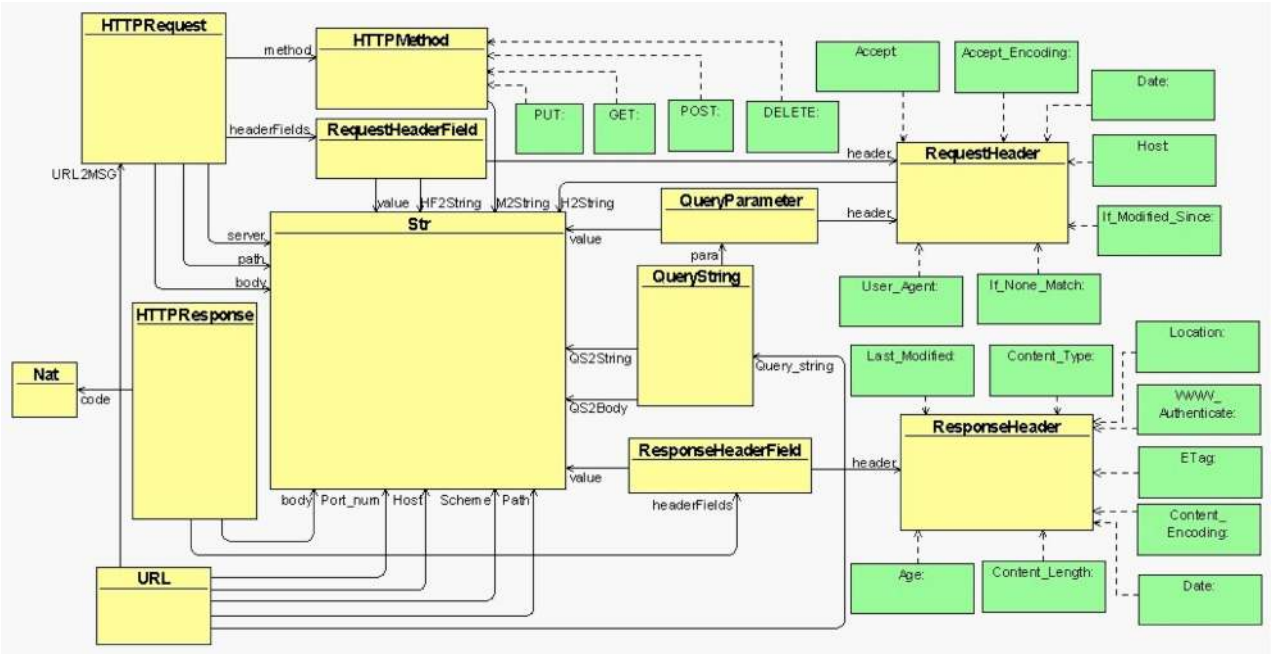

Figure 5 Ontology generated from the SOFIA specification

A case study of the RESTful web service interface of an actual industrial system called GoGrid shows that the approach is practically useful.

4.2.2 Formal Specification of Conceptual Models

One advantage of the algebraic method is that the infrastructure, platform, application domain knowledge, and the services of a SaaS application can all be formally specified in the same language and decomposed into a number of reusable specification packages.

For example, in the case study of GoGrid's RESTful API, we first specified the RESTful web service in a package, then used that to specify the basic constructs of computing infrastructure, and then used both packages to specify the services that GoGrid provides. Figure 5 gives the ontology generated from the SOFIA specification of RESTful web services.

Therefore, the specification of domain concepts can be used to serve as a formal specification of the conceptual model of the system. This specification supports automated testing and its internal consistency can be verified. This enables it to support the maintenance of conceptual integrity, too.

4.3 Implementation: New Paradigm of Programming

Currently, most web-based applications, including those for SaaS, are implemented in many different programming and scripting languages and even several different paradigms. This complicates development and makes it difficult to develop supporting tools. A desirable alternative is to have a new single paradigm that is particularly suitable for SaaS applications.

The agent-oriented paradigm has long been considered suitable for dynamic environments such as the Internet [31], and many research efforts have been reported in the literature [32]. However, the IT industry has been slow to adopt the approach. There are a number of possible reasons for this. First, the notion of agents seems to be too strongly linked to distributed artificial intelligence for software engineers to accept it. Secondly, there are no efficient implementations of agent-oriented programming languages. We now report our work in progress that addresses these problems.

4.3.1 Agent-Oriented Programming Language

To address the first problem, we proposed a simplified model of agent [33, 34]. Agents are service providers that consist of:

- *actions* that the agent can perform, representing the services it provides or requests it can submit,
- *variables*, which represents its internal state of the agent,
- *behaviour rules*, forming the body of the service, that determine how the requests are processed,
- *collaborating agents*, from which the service requests are received. This set can be updated at runtime.

For example, the following is the Hello World example of the language CAOPLE, which we are developing.

```
caste Peer;
  action say(word:  string);
  init say("Hello world!")
end Peer
```

Caste is the classifier of agents so agents are instances of castes. In the above example, the caste Peer is defined. It can take the action of say("Hello world!") and it does this when the agent is created. An agent is therefore an active autonomous computational entity.

Castes can be extended to sub-castes just as classes in object-orientation have subclasses. For example, the following is a sub-caste of Peer.

```
caste GreetingPeer inherits Peer;
  observes all in Peer;
  body
    when exists A in Peer:  say("Hello world!") do
      say("Welcome to the world!")
    end
end GreetingPeer
```

An agent of GreetingPeer observes the actions taken by all agents of Peer, as described in the observes clause, which defines its collaborative agents. When there is an agent in the caste Peer that takes the action say("Hello world!"), it will react with the action say("Welcome to the world!"). In general, an agent communicates with other agents by taking observable actions to send messages and it receives messages by observing the observable actions of its collaborative agents. An action can be targeted to one or a set of specific agents. For example, if the say statement can be changed to one of the following:

```
say("Welcome to the world!") to All in Peer;
say("Welcome to the world!") to A;
```

If the target receiver is omitted, the default is public.

In contrast to the notation of class in object-oriented programming, an agent can be a member of multiple castes at once and its membership can be changed dynamically at runtime by executing one of the caste membership statements:

- **Join** casteID: to become a member of casteID;
- **Quit** casteID: to quit the membership of casteID;
- **Suspend** casteID: to suspend the execution of the body of casteID;
- **Resume** casteID: to resume the execution of the body of casteID;
- **MoveTo** casteID: to quit from the current caste and become a member of the named caste.

Using castes and the inheritance relationships between them, one can encapsulate different behaviours in different contexts together with a set of related state variables, actions, and collaborative agents. The flexible casteship

enables agent to have adaptability and to be easy to compose and configure. For example, the following shows how agent can adapt its behaviour according to the context by change its caste membership.

```
caste CheerfulPeer inherits Peer;
  body
    when exists A in Peer: say("Hello world!")  do
       say("Hi, good morning.");
    end;
end CheerfulPeer
caste SmartPeer inherits Peer;
  observes DateTime: Clock;
  body
    when DateTime: Tick() do
       if DateTime.Day = Monday then Join FriendlyPeer
       else Join CheerfulPeer
       end;
    end;
end SmartPeer
```

The above just a few key features of the agent-oriented programming language CAOPLE. Readers are referred to [34] for more details. In general, we believe that a new programming paradigm such as agent-orientation will enable the implementation of SaaS applications at a high level of abstraction. Thus, it is worth pursuing.

4.3.2 Implementation of CAOPLE Language

Our approach to the implementation of the CAOPLE programming language is to translate CAOPLE source code into machine code for a virtual machine [35].

Our virtual machine, called CAVM, differs from other language specific virtual machines like JVM in that it consists of two parts: a *local execution engine* LEE and a *communication engine CE*. The LEE executes the program's computational code, while the CE realises communication between agents distributed over a computer network.

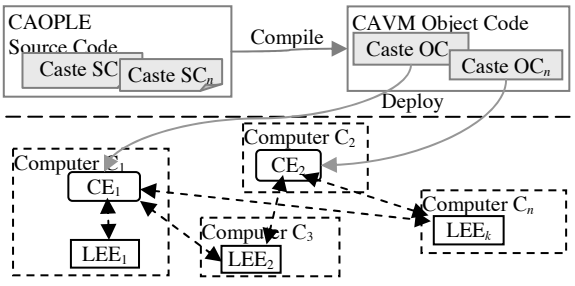

Figure 7 Compiling, deploying and executing CAOPLE code

As illustrated in Figure 7, the castes in a CAOPLE program are compiled so that one Object Code module is generated from each caste Source Code. It is deployed to a Computer node that runs a communication engine. An agent of a caste can be created on any Computer node that runs an execution engine. It will load the object code module of the caste and execute the code on the machine. For cross-machine communications between agents, the messages are send to the communication engine where the caste resides and further distributed to execution engines where the target agents executes. They may be passed through one or more other communication engine. The reader is referred to [35] for more details of the design, implementation and experiment results of CAVM.

4.4 Testing: Specification-Based Test Automation

Automated testing can play at least two roles in the development of Big SaaS: it supports continuous evolution and it ensures the quality of crowdsourcing in service customization.

There are a number of approaches to automated testing for software in general and for service-oriented systems in particular. In [36], we proposed a collaborative approach that realizes automated testing of composite web services through composition of test services, as illustrated in Figure 8. In this approach, each web service is accompanied by a testing service, and the framework of automated testing contains a number of general test services for test case generation, test adequacy measurement, test result correctness checking, etc. A test request for the composition of services is submitted to a test broker, which decomposes the testing task into subtasks if needed and if so, searches for and invokes appropriate test services for each sub-task. The searching and invocation of test services (and the initial registration) employs ontologies both of software testing and of the application domain.

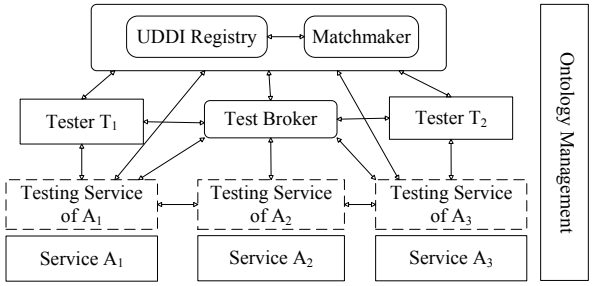

Figure 8. Collaborative Automated Testing of Web Services [36]

This approach was devised for web services and should be applicable to Big SaaS, but we believe a formal specification language like SOFIA would make the test automation efficient without developing various test services.

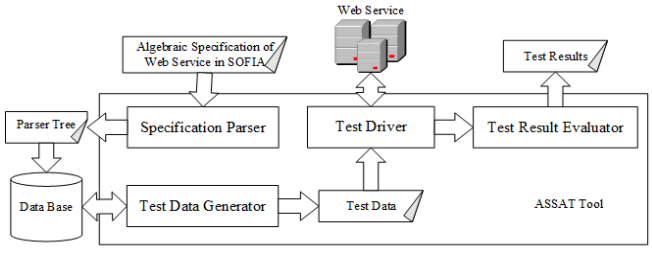

Figure 9. Architecture of ASSAT Testing Tool

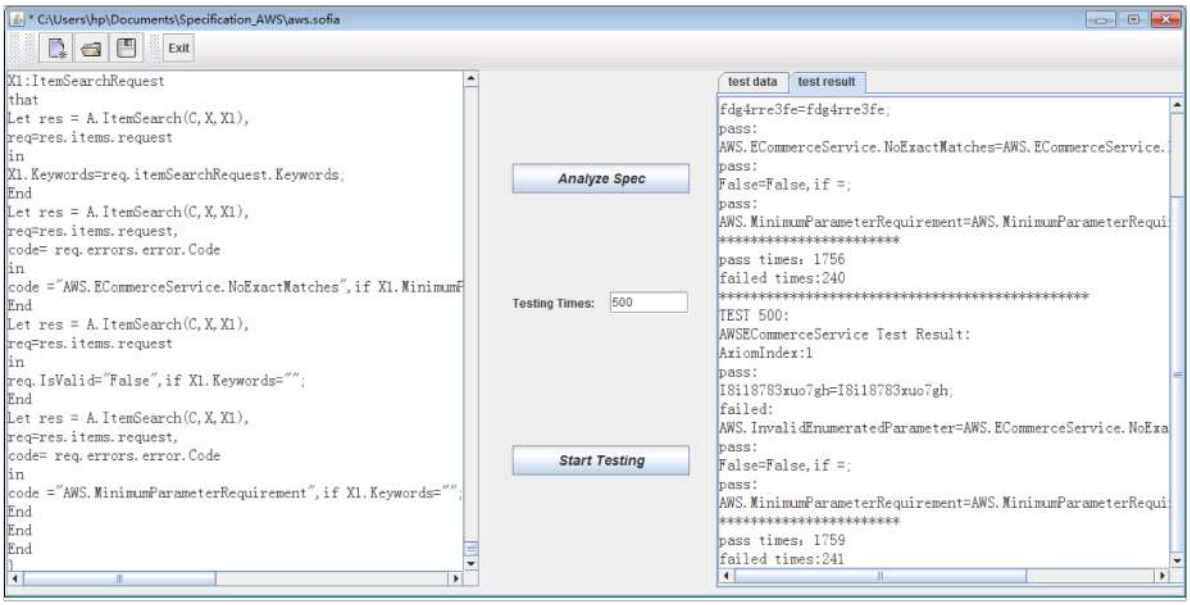

Figure 10. Interface of the ASSAT tool.

Techniques of software test automation based on algebraic specifications have been investigated since 1980s for procedural languages [37, 38], OO software [39, 40], and component-based systems [41], etc. More recently, we have been developing an automated testing tool called *ASSAT* [42] for testing web services based on formal specification written in SOFIA [43]. Figure 9 shows the architecture of the tool and Figure 10 shows its GUI. Such testing tools can achieve complete automation of the whole testing process including test case generations, test invocation and test result correctness checking.

Although SOFIA and ASSAT were originally developed for web services, the principles underlying the language and the implementation of the tool are applicable to Big SaaS. It is worth further research to adapt them to Big SaaS and evaluate their effectiveness.

It is worth noting that there are two approaches to the quality assurance of customization. The first is brutal force approach. In this approach, all possible compositions of services and all possible configurations of the SaaS application are tested up to a certain level of combination adequacy, say the coverage of all 2-way or 3-way combinations, before the system is released to the users. This approach is viable only when the number of possible service compositions and configurations is small. Unfortunately, even for a SaaS application of modest scale, there could be a huge number of test cases even to cover 2-way or 3-way combinations of services and configurations.

The second is the automated online testing approach. During the development process, testing focus at the individual services to ensure each service is correct with respect to its specification. The most popular and important combinations and configurations of the services are also tested. When a user builds his or her own customized version of the system, the customization, which is a composition and configuration of the services, it is then tested automatically against the specification. In this approach, automated testing plays a crucial role to support customization of services. It requires testing to be performed with little human involvement because crowdsourcing-based customization is conducted by the users.

5 Conclusion

In this paper we argue that an era of Big SaaS is emerging. It differs from existing SaaS applications in the number of tenants/users and the complexity of their relationships, as well as in the size and complexity of the program code. They will possess and utilize Big Data to provide great added value to their services. Developing Big SaaS applications will impose grave challenges to software and service engineering to reduce the societal risks to an acceptable level, to enable trustable crowdsourcing-based customization, to maintain conceptual integrity of the system and to support continuous evolution. We argued that these challenges must be met in all stages of the software development lifecycle.

In particular, in the specification phase, an algebraic specification language can support formal development of service-oriented systems to improve reliability. It also helps to maintain conceptual integrity by providing a formal definition of the conceptual model. It supports crowdsourcing-based customization by linking formal specification to the ontological description of services. Moreover, testing can be automated based on algebraic specifications. This also helps with continuous evolution.

Also, for the architectural design phase, a tenant-level

checkpointing facility could play a significant role in reducing societal risks. In the implementation phase, a new paradigm of programming is desirable and we are exploring the potential of an agent-oriented programming language. In the testing phase, automation is essential and formal specification will make this possible.

References

[1] F. Chong and G. Carraro, Architecture strategies for catching the long tail, Microsoft Corporation, April 2006. URL: https://msdn.microsoft.com/en-us/library/aa479069.aspx. Last access on 3 May 2015.

[2] W-T. Tsai, X. Bai, Y. Huang: Software-as-a-service (SaaS): perspectives and challenges. SCIENCE CHINA Information Sciences 57(5), pp1-15 (2014)

[3] B. Yousef, H. Zhu and M. Younas, Tenant level checkpointing of meta-data for multi-tenancy SaaS, In Proc. of IEEE SOSE 2014, Oxford, UK.

[4] T. Kraska and B. Trushkowsky, The new database architectures, IEEE Internet Comp. 17(3), pp72-75, 2013.

[5] I. Jangjaimon and N.-F. Tzeng, Design and implementation of effective checkpointing for multithreaded applications on future clouds," in Proc. of IEEE CLOUD 2013, pp. 438-445.

[6] I. Jangjaimon and N.-F. Tzeng, "Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing, IEEE Transactions on Computers 64(2), pp396–409, 2015.

[7] P. Lu, B. Ravindran, and C. Kim, VPC: Scalable, low downtime checkpointing for virtual clusters, in Proc. of SBAC- PAD 2012, pp203–210.

[8] A. Agbaria and R. Friedman, Virtual-machine-based heterogeneous checkpointing, Software: Pract. & Exp. 32(12), pp1175-1192, 2002.

[9] T. C. Bressoud and F. B. Schneider, Hypervisor-based fault tolerance, ACM Trans. Comp. Syst. 14(1), pp80-107, 1996.

[10] B. Azeem and M. Helal, Performance evaluation of checkpoint/ restart techniques: For MPI applications on Amazon cloud, in Proc. of INFOS 2014, ppPDC49-PDC57.

[11] H. Zhu, B. Yousef, and M. Younas, Evaluation of a tenant level checkpointing technique for SaaS applications, Proc. of IEEE CLOUD 2015. (*In press*)

[12] H. Liu, H. Jin, X. Liao, C. Yu, and C.-Z. Xu, Live virtual machine migration via asynchronous replication and state synchronization," IEEE Transactions on Parallel and Distributed Systems 22(12), pp1986–1999, 2011.

[13] D. Singh, J. Singh, and A. Chhabra, High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms, in Proc. of CSNT2012, pp698– 703.

[14] S. Mondal and J. Muppala, Energy modeling of virtual machine replication schemes with checkpointing in data centers, in Proc. of BDCloud 2014, pp633–640.

[15] S. Yi, A. Andrzejak, D. Kondo, Monetary cost-aware checkpointing and migration on Amazon cloud spot instances, IEEE Trans.on Serv. Comp.5(4), pp512-524, 2012.

[16] A. Hall, Seven myths of formal methods, IEEE Software 7(5), pp11-19, 1990.

[17] J. P. Bowen, M. G. Hinchey, Seven more myths of formal methods, IEEE Software 12(4), pp34-41, 1995.

[18] H. Zhu and B. Yu, An experiment with algebraic specifications of software components, in Proc. of QSIC 2010, pp190-199.

[19] S. A. Mallraith, T. C. Son, & H. Zeng, Semantic web services, IEEE Int. Systems, 2001(March/April), pp46-53.

[20] D. Martin, et al., Semantic Markup for Web Services (W3C member submission): W3C. 2004.

[21] J. Bruijn, et al., Web service modeling ontology (WSMO), (W3C member submission): W3C. 2005.

[22] J. Bruijn, et al., The web service modelling language WSML: An overview, in Proc. of the 3rd European Semantic Web Conference, pp590-604, 2006.

[23] J. Kopecky, K. Gomadam, & T. Vitvar, hRESTS: An HTML microformat for describing RESTful web services, in Proc. of WI-IAT 2008, pp619-625.

[24] M. J. Hadley, Web application description language (WADL) SMLI TR-2006-153, Sun Microsystems, 2006.

[25] J. Lathem, K. Gomadam, & A. P. Sheth, SA-REST and mashups: Adding semantics to RESTful services, in Proc. of ICSC, pp469-476, 2007.

[26] J. A. Goguen, & G. Malcolm, A hidden agenda. Theoretical Computer Science 245(1), pp55-101, 2000.

[27] F. Bonchi, & U. Montanari, A coalgebraic theory of reactive systems. Electronic Notes in Theoretical Computer Science 209, pp201-215, 2008.

[28] H. Zhu and B. Yu, Algebraic Specification of Web Services, Proc. of QSIC 2010, pp457-464.

[29] D. Liu, H. Zhu & I. Bayley, Applying algebraic specification to cloud computing -- A case study of Infrastructure-as-a-Service GoGrid, in Proc. of ICSEA 2012, pp407-414.

[30] D. Liu, H. Zhu, and I. Bayley, Transformation of algebraic specifications into ontological semantic descriptions of Web Services, International Journal of Services Computing 2(1), pp58-71, 2014.

[31] N. R. Jennings, On agent-based software engineering. Artificial Intelligence 117, pp277–296, 2000.

[32] B. Henderson-Sellers and P. Giorgini, (Eds.), Agent-oriented Methodologies, Idea Group Publishing, 2005.

[33] H. Zhu, SLABS: A formal specification language for agent based systems. SEKE 11(5), pp529–558, 2001.

[34] H. Zhu, Towards an agent-oriented paradigm of information systems. Handbook of Research on Nature Inspired Computing for Economy and Management, Jean-Philippe Rennard (Ed), Chapter XLIV, pp679–691, 2006.

[35] B. Zhou and H. Zhu, A virtual machine for distributed agent-oriented programming, in Proc. of SEKE 2008, pp729-734.

[36] H. Zhu and Y. Zhang, Collaborative testing of Web Services, IEEE Trans. on Services Comp. 5(1), pp116-130, 2012.

[37] J. Gannon, P. McMullin, R. Hamlet, Data abstraction, implementation, specification, and testing, ACM Trans. on Program. Lang. and Syst. 3(3), pp211–223, 1981.

[38] G. Bernot, M.-C. Gaudel, and B. Marre, Software testing based on formal specifications: a theory and a tool, Software Engineering Journal 6(6), pp387–405, 1991.

[39] R.K. Doong & P.G. Frankl, The ASTOOT approach to testing object-oriented programs, ACM TSEM 3(2), pp101–130, 1994.

[40] M. Hughes, D. Stotts, Daistish: systematic algebraic testing for OO programs, in Proc. of ISSTA 1996, pp53–61.

[41] B. Yu, L. Kong, Y.Zhang, and H. Zhu, Testing Java components based on algebraic specifications, in Proc. of ICST 2008, pp190–199.

[42] D. Liu, Y. Liu, X. Zhan, H. Zhu and I. Bayley, Automated testing of Web Services based on algebraic specification, in Proc. of IEEE SOSE 2015.

[43] D. Liu, H. Zhu, and I. Bayley, SOFIA: An Algebraic Specification Language for Developing Services, in Proc. of IEEE SOSE 2014, pp70–75.