Protecting Software as a Service in the Clouds by Validation

Tien-Dung Cao and Kevin Chiew

School of Engineering - Tan Tao University,
Long An province, Vietnam
{dung.cao,kevin.chiew}@ttu.edu.vn

Abstract. The cloud computing has provided customers with various services at its SaaS layer though, few work has been done on the security checking of messages exchanged between a customer and a service provider at SaaS so as to protect SaaS. In this paper we propose a validation model to investigate the SaaS security issue. Rather than installing a set of probes as we have done for the testing web services, in this model we introduce a validation service that plays the role of a firewall and protects our SaaS by verifying the correctness of messages with respect to a set of predefined security rules and forwarding them to their real destinations if they pass the verification or rejecting them otherwise. We develop a prototype model based on the tool known as RV4WS which was developed in our early study on web service runtime verification, as well as a checking engine RVEngine to verify our checking algorithm for the model. A survey on how to use this model for the services deployed on Google App Engine, Window Azure and Oracle Java Cloud Service is also presented.

Keywords: SaaS, Cloud Computing, Security Checking, Rule Specification.

1 Introduction

The cloud computing [1] has been witnessed to grow tremendously in recent years as driven by the ubiquitous availability of high capacity networks, low cost computers and storage devices, as well as the widespread adoption of service-oriented architecture and utility computing. Cloud computing is the delivery of computing as a service rather than a product. The current cloud computing architecture provides clients with three layers of services [2] for them to interact with the clouds:

– IaaS (Infrastructure as a Service) is the fundamental layer providing services for deploying, running and managing virtual machines, networks and storage.
– PaaS (Platform as a Service) is the layer above IaaS by delivering the services for programming and execution, like deploying, monitoring, testing, security, analyzing, etc.

– SaaS (Software as a Service) is the top layer that is the URI software applications providing customers with the shared services.

Given the above layered model, our SaaS (a.k.a, service) is built using the existing PaaS and IaaS such as Google App Engine [3], Window Azure [4], Oracle Java Cloud Service [5], and Amazon S3 [6], due to security and reliability of PaaS and IaaS. In a cloud, however, the services may play the roles of consumers and providers. Therefore, without an adequate protection mechanism our service may breakdown due to customer's and/or provider's unpredictable behaviors. For example, during the communication between a customer and its provider, the messages from the customer/provider may contain untrusted data like virus links which may harm the service, or the customer/provider sends/responds the same message several times within a short duration or accesses a cloud service from different devices (e.g., mobile phones and PCs) at the same time which may bring with unexpected results to the service. Moreover, an untrusted service existing in a composite of services may harm all composition.

To protect the services, an important step is to go through a solid security testing and verification of the software at runtime. The security testing of a software implementation is usually executed via runtime verification. Presently the approaches to runtime verification of software are usually carried out by collecting the messages exchanged and verifying them against a set of constraints [7],[8], in which message collecting is generally practiced by installing a set of points of observation (a.k.a., a set of probes). However, these approaches are not applicable to a cloud environment due to two reasons: (1) SaaS in a cloud uses a dynamic and virtual infrastructure, and it does not function well to install a set of points of observation because of some limitations, for example, only Servlet is supported in Google App Engine, and (2) the points of observation do not allow us to make several decisions like reject, modify or ignore the unexpected messages (from both directions) which are necessary to protect the services.

Given the above situation, in this paper we firstly survey on several clouds and propose the corresponding validation model for security checking of services in these clouds. A validation module which plays the role of a firewall, is actually performing as a kind of intermediary between the customer side and the provider side, serving as the probes to collect messages, and verifying them with respect to a set of predefined security constrains. Secondly, our prototype model is developed with a tool known as RV4WS (Runtime Verification for Web Services) which was developed in our early study on automated runtime verification for web services [9]. Besides, we survey on how to use this prototype for the service composition that are deployed on two popular clouds, namely Google App Engine and Microsoft Window Azure.

The remaining sections of the paper are organized as follows. In Section 2, we review related work on testing of SaaS, and present our security model in Section 3 which includes a validation architecture, rule model and an algorithm to check the correctness of a sequence of messages with respect to a set of security rules, followed by showing our implementation details of prototype development with open discussions in Section 4 before concluding the paper in Section 5.

2 Related Work on Testing of SaaS in the Clouds

There is few published work focusing on either active or passive testing and verification of a cloud application though, some approaches have been proposed to deal with the testing of web services and can be considered as a cloud application if they are applied to a cloud environment.

To protect a web service, Gruschka and Luttenberger [10] proposed a mechanism by validating the SOAP (Simple Object Access Protocol) messages, aiming to filter out the SOAP messages by detecting the malicious ones so as to improve the availability web services. They set up a web service firewall that can validate all incoming and outgoing SOAP messages against the schema, and forward valid messages or reject invalid messages.

Salva *et al.* [11] proposed a security testing method for stated web services. In this work, the security rules defined by the Nomad language are used to construct the test purposes. This security rule set expresses the different properties such as the web service availability, authorization and authentication by means of malicious requests. Using these test purposes, the test cases are then generated from the symbolic specification of web services to test against the service implementation.

The approaches proposed in [8] and [12] focus on invariant satisfiability. These invariants are constructed from the specification and are later on checked based on the collected traces. These approaches use a sniffer-based module which may not be easy to set up on a cloud environment to collect the traces.

Chan *et al.* [13] presented a graph-theoretic model of computing clouds together with a family of model-based testing criteria for testing cloud applications. Their approach is proposed particularly for clouds-in-the-small to predicate the behaviors of applications though, it may not be viable to our study scenario which focuses on protecting SaaS via security testing.

A recent model-based testing process proposed by Endo and Simao [14] suggested using finite state machines to model and support the test case generation for the verification of service-oriented applications. This process focuses on the functional verification of SaaS rather than the security checking.

In [15], Salva defined a proxy-tester as a product between the specification and its canonical tester, which is an intermediary between the client and its implementation. Whenever the proxy-tester receives a message either from the client or from the implementation, it will analyze this message by means of ***ioco*** for passive testing to detect faults.

Our motivation of this study is based on the idea presented in [10] and the features of clouds. Since a cloud environment is dynamic, virtual and limitation of supported technologies, making it difficult to install a sniffer-based [8] [12] module, we build a firewall, which is either installed totally outside the Clouds or a part depending on the concrete cloud environment, to verify all communicating messages before forwarding them to their destinations so as to protect our services. However, unlike the work in [10] that checks the correctness of messages by comparing the structure of those messages against the schema defined in WSDLs file, we focus on the security issue to protect our service from the

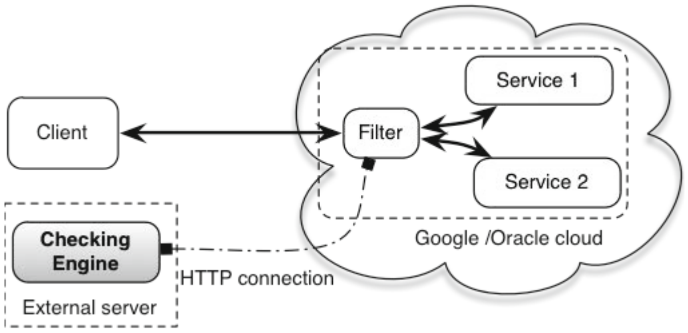

Fig. 1. Testbed architecture using Servlet Filter

untrusted customers or the mistake of partners. For example, the partner responses a message two time, the second one will be rejected. We firstly predefine a set of security rules by giving its syntax, and then verify all messages based on the set of rules defined by this syntax. Amongst these rules, some aspects are considered as time constraint (i.e., future and past time), data correlation, message filtering out by its content, and service behavior.

3 Security Model of SaaS in a Cloud

3.1 A Survey on the Existing Clouds

In this section, we conduct a survey on how the validation model can deploy on the popular PaaS such as Google App Engine (GAE), Window Azure, Oracle Java Cloud Service. Google App Engine [3] supports Java technologies with restrictions at the moment for developing and deploying the services, only Servlet and Rest Web Services are supported. Window Azure [4] supports more standards and it also allows us to configure an application on this environment to call the other one though an HTTP Proxy which is installed outside of the cloud. It allows a service to call to the other one outside the cloud by using the Service Bus or Window Azure Connect [16]. Oracle Java Cloud Service [5] supports full standards of Java EE which allows to call other applications though an HTTP Proxy. However, a little modification of source code is required.

With the services (also service composition) that are developed using Servlet of Java EE, using a Filter[1] as a transparent proxy module, we can capture all communicating message among consumers and providers. This module communicate with Checking Engine that is installed outside the cloud via HTTP Protocol to validate these messages before forwarding them to corresponding Servlet. The Filter is a specialized Servlet which can intercept and transform any requests and responses, therefore it can deploy on the same environment

[1] `http://www.oracle.com/technetwork/java/filters-137243.html`

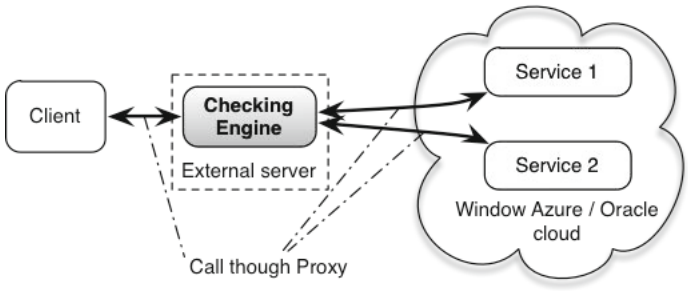

Fig. 2. Testbed architecture though Proxy

with the services. With this architecture, any kind of messages exchanged between a consumer and a provider can be observed without modifying any source code, whereas we do not need to install the checking engine inside the cloud. We can also configure a filter to capture the messages between partners of a service composition. Figure 1 shows the testbed architecture using Servlet Filter. However, an environment, such as Window Azure or Oracle Java Cloud Service, allows the applications to configure to use an HTTP Proxy to call other applications. The checking engine can be integrated into a proxy as shown in Figure 2. However, the consumer is required to call the service though this proxy.

3.2 Security Checking

There have been approaches [8,12] proposed for web services and security checking, which only focus on the behavior of the applications and ignore the data part (i.e., the contents of messages exchanged between a customer and a service provider). However, the security of a service oriented application needs to consider the following aspects, namely (1) *Data Constraints*, i.e. limitations on the structures and contents of the messages sent to the service, (2) *Control-Flow Constraints*, and (3) *Data-aware Control Flow Constraints*, i.e. the authorized sequence of messages depends on relationships between values of multiple messages. To support our model for security checking, in this section we present a checking algorithm that can verify the correctness of a timed trace with respect to a set of constraints. For this purpose, in what follows, we give some formal definitions as preliminaries to the checking algorithm. Our rule definitions and checking algorithm inherit from our previous study [9] on runtime verification of web services.

Rule Definition. The rule definitions include two parts, namely syntax and semantics.

For the syntax part, we consider each message as an atomic action, and use one or several messages to define a formula as a boolean expression. We also use the operation NOT to indicate that a message is prohibited to appear in the trace within a duration. During the formula definition, the constraint on the values of message parameters may be considered. Finally, from these formulas, the rule is defined in two parts, namely supposition (or condition) and context. A set of data correlations are included as an option.

Definition 1. *(Atomic action).* An atomic action is either an input message or an output message, formally denoted as

$$A \ := \ Event(Const) | \neg A$$

where

- *Event* represents an input/output message name;
- $Const := P \approx V | Const \wedge Const | Const \vee Const$ where
  - P are the parameters. These parameters represent the relevant fields in the message;
  - V are the possible parameters values;
  - $\approx \in \{=, \neq, <, >, \leq, \geq\}$;
- $\neg A$ means $not(A)$.

Definition 2. *(Formula).* A formula is recursively defined as

$$F \ := \ start(A) \ | \ done(A) \ | \ F \wedge F \ | \ F \vee F \ | \ O^{d \in [m,n]} F$$

where

- A is the atomic action;
- $start(A)$: A is being started;
- $done(A)$: A has been finished;
- $O^{d \in [m,n]} F$: F was *true* in d units of time ago if $m > n$, and F will be *true* in the next d units of time if $m < n$ where m and n are natural numbers.

Definition 3. *(Data correlation).* A data correlation is a set of parameters that have the same data type where each different parameter represents a relevant field in a different message, for which the operator = (equal) is used to compare the equality amongst parameters. A data correlation is considered as a property on data.

By putting the time constraints into an interval, we support two types of rules, namely *obligation* and *prohibition*. Obligation means that all traces must satisfy the constraints; whereas prohibition is the negation of an obligation constraint.

Definition 4. *(Rule with data correlation).* Let α and β be formulas, and CS be a set of data correlations based on α and β (CS is defined based on the messages of α and β). A rule with data correlation is defined as $\mathcal{R}(\alpha|\beta)/CS$[2] where $\mathcal{R} \in \{\mathcal{O}$: Obligation; $\mathcal{F}$: Prohibition;$\}$. The constraint $\mathcal{O}(\alpha|\beta)$ or $\mathcal{F}(\alpha|\beta)$ (where $\mathcal{F}(\alpha|\beta)$ = $\mathcal{O}(NOT \ \alpha|\beta)$) respectively means that it is obligated or prohibited to have α *true* when context β holds within the conditions of CS.

Example 1. If we have such a constraint that we do not allow to submit the same login request twice within a period of time, say 3 seconds, then we can use *userId* to distinguish among requests with the following formula:

[2] CS is an optional part.

$$\mathcal{F}(start(loginRequest)|O^{d\in[0,3]S}\ done(loginRequest))\ /$$
$$\{\{loginRequest.userId,\ loginRequest.userId\}\}$$

Example 2. We also define a rule to control the client behavior. For example, if a client wants to send a confirmation request to a service provider, then the client must firstly receive a response from the previous operation within maximum 5 minutes where the content of this response is accepted.

$$\mathcal{O}(start(confirmRequest)|O^{d\in[5,0]M}\ done(xxxResponse(resp = "accept")))$$

For the semantics part, we have the following definition for a rule model.

Definition 5. *(Rule model).* A model of rules corresponds to a pair $r = (P_r, C_r)$ where

- P_r is a total function that associates every integer x with a propositional formula.
- C_r is a total function that associates every integer x with a pair (α, d) where α is a formula and d a positive integer.

Intuitively, $\forall x$, $p \in P_r(x)$ means that proposition p is *true* at time x; while (α, d) $\in C_r(x)$ means that context of formula α holds (is evaluated *true*) at time t where

- $t \in [x, x + d]$ if we focus on future time.
- $t \in [x - d, x]$ if we focus on past time.

Checking Algorithm. Given the above rule definitions, in what follows we present our algorithm to check a message's security property with respect to a set of constraints. Our algorithm will deal with two cases of rules, namely rules with future time and rules with past time, in which we use two global variables, namely *currlist* and *rulelist*, in which *currlist* is a list of enabled rules that have been activated and *rulelist* is the list of defined rules that are used to verify the system. The full verification algorithm is presented in [9] and it is summarized as follows.

a) Rules with Future Time

Given that each rule has two parts (i.e., the supposition and context parts), a rule will be evaluated as either *true* or $false$ or $undefined$ if its supposition has been enabled and the current message belongs to its context. At any occurrence time t of message *msg*, our algorithm checks the correctness of a rule by two steps.

- Step 1. Examine the list of enabled rules *currlist* to evaluate their context if the time constraints are valid. If the context of a rule is evaluated to be $true/false$, then it will be removed from the enabled list *currlist* and the corresponding verdict is returned. Otherwise (i.e., the context is $undefined$, meaning incomplete context), we wait for the arrival of the next message and return *true* to the verdict.

Algorithm 1. Checking algorithm for future time rules

```
  Input  : timed event: $(msg, t)$
  Output: $true/false$
1 $verdict \longleftarrow true$
    1. For each $r \in currlist$
        – IF the time constraints of $r$ at $t$ are validation
            • IF $msg$ belongs to the context of $r$
                ∗ Update context of $r$ by $msg$
                ∗ IF the evaluation of the context of $r$ is $true/false$
                    · Remove $r$ from $currlist$
                    · $verdict \longleftarrow verdict \wedge true/false$
        – ELSE: $verdict \longleftarrow false$
    2. For each $r \in rulelist$
        – IF $msg$ belongs to the supposition of $r$
            • Update the activated time for $r$ by $t$
            • Add $r$ into $currlist$ (activated)
```

– Step 2. Examine the list of rules *rulelist* to activate[3] them if their supposition contains the current message *msg*.

Algorithm 1 shows how to check the correctness of a message with a set of future time rules, in which we assume that the rules are *Obligation* (the *Prohibition* rules are the negation of the verdict of the *Obligation* rules), and do not consider data correlation.

b) Rules with Past Time

For a rule with past time, the context part will happen before its supposition, meaning that the context part must be evaluated to be $true/false$ whenever its supposition handles the current message. Upon the arrival of any timed event (msg, t), our algorithm checks correctness of a rule with past time by two steps.

– Step 1. Examine the list of enabled rules *currlist* to check the correctness of current message *msg*. If t satisfies their time constraints and *msg* belongs to their supposition, then remove them from list *currlist*. At the same time, if their context is evaluated to be $false/undefined$, then a $false$ verdict will be assigned; otherwise, a *true* verdict is admitted. On the other hand, if *msg* does not belong to their supposition and *msg* is found in their context, then we update their context by *msg* and wait the next message to evaluate these rules.
– Step 2. Examine the list of rules *rulelist* and activate them if their context contains the current message *msg*.

Algorithm 2 shows how to check the correctness of a message with a set of past time rules under the assumption that the rules are *Obligation*.

[3] If a rule exists in the current enable rule list, it will still be activated.

Algorithm 2. Checking algorithm for past time rules

```
Input  : timed event: $(msg, t)$
Output: $true/false$
1 $verdict \longleftarrow true$
  1. For each $r \in currlist$
     – IF the time constraints of $r$ at $t$ are validation
        • IF $msg$ belongs to the supposition of $r$
           ∗ Remove $r$ from $currlist$
           ∗ IF the evaluation of the context of $r$ is $false/undefined$
              · $verdict \longleftarrow verdict \wedge false$
        • ELSE IF the context of $r$ contains $msg$
           ∗ Update the context of $r$ by $msg$
     – ELSE: $verdict \longleftarrow false$
  2. For each $r \in rulelist$
     – IF $msg$ belongs to the context of $r$
        • Update the activated time for $r$ by $t$
        • Add $r$ into $currlist$ (activated)
```

4 Implementation and Discussion

With the above rule definitions and checking algorithm, in what follows we present the implementation of our prototype, as well as applying to a simple example of service composition which can well demonstrate the effectiveness of our security model and method.

To support our security checking method, in the context of WebMov[4] project, we developed a tool known as RV4WS [17][18] (Runtime Verification for Web Services) that checks a timed trace with respect to a set of security rules predefined by the syntax as introduced in Section 3.2. In this tool, a checking engine [19] (i.e., RVEngine) is developed independently in Java language and used as a library of the tool. To support several types of systems, this engine defines an interface known as IParseData which allows us to parse the different structure of messages by implementing it. This interface provides two operations, namely (1) *getMessageName()* which returns the message name by analyzing the structure or content of a message, and (2) *queryData()* which allows us to query a data value from a specific field of a message. The query path of the latter operation depends on the structure of messages. For example, it is an XPath in the case of SOAP message of web services.

To demonstrate the effectiveness of our prototype, we developed a simple service composition where 4 services (i.e., Shopping, Login, Stock and Cart) are developed using Servlet and deployed in Google App Engine [3]. In this composition, Shopping service proposes an interface that allows a client to search a book from Stock service by sending an ISBN (International Standard Book Number). If a book is found, the client can add it into a temporary cart

[4] `http://webmov.lri.fr`

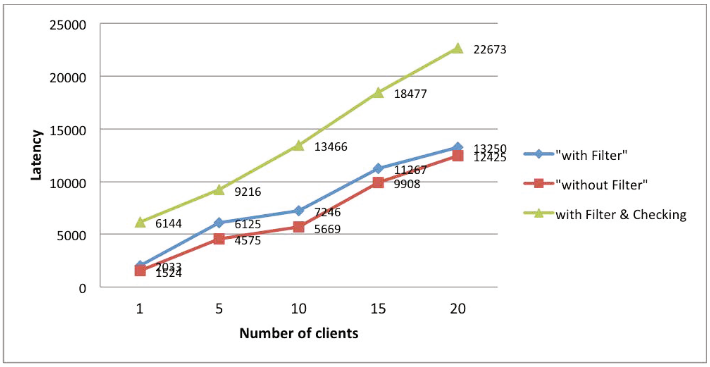

Fig. 3. Latency measurements of composition on GAE Cloud

(i.e., Cart service) and search the other ones. Finally, when one or several books were added into the cart, the client can checkout. However, to use these services, the client is required firstly to access to the Login service to get a ticket. And then, every time when the client is accessing Shopping service, the ticket is required and Shopping service will contact to Login service to validate this ticket. A Filter is also configured to collect all communicating messages from 4 services and the client. These messages will be sent to Checking Engine which is installed on a local machine to validate. We implemented our prototype with 20 instances of mocked client written in Java, which simulate real client applications. They run in a loop and perform the following actions, namely call to Login Service to get a ticket and use this ticket to search a book, add the found book into a temporary cart and check out the cart. In this paper we ignore the correctness of RVEngine because it is proved in [9], but we are interested in the latency of messages while applying our prototype to protect the services. The latency measurement is important since it may lead to an issue if it is equal to or longer than the time set for a connection. Figure 3 shows the obtained latency measurement with GAE. This is the average of total time to complete the sequence of actions with 1, 5, 10, 15 and 20 clients running in parallel. We observe that the difference of time execution between non validation and validation of one instance is 4620 ms (6144-1524) while 18[5] messages were passed to the checking engine. It means that each message is delayed 256 ms for capturing of filter, sending/receiving to checking engine and checking time. However, if many instances are executed in parallel, then the latency can be reduced. For example, with 10 instances, the difference of time is $13466 - 5669 = 7797$ ms for all 180 messages. However, all these latencies depend on the network between our local machine and the Google cloud at different moment.

[5] *loginReq*$\rightarrow$ *loginResp*$\rightarrow$ *bookReq*$\rightarrow$ *verifyReq*$\rightarrow$ *verifyResp*$\rightarrow$ *stockReq*$\rightarrow$ *stockResp*$\rightarrow$ *bookResp*$\rightarrow$ *addCartReq* $\rightarrow$ *verifyReq*$\rightarrow$ *verifyResp*$\rightarrow$ *cartAddReq* $\rightarrow$ *cartAddResp*$\rightarrow$ *addCartResp* $\rightarrow$ *checkReq*$\rightarrow$ *checkCartReq* $\rightarrow$ *checkCartResp* $\rightarrow$ *checkResp*.

This validation model provides us with some other choices than those described in this paper. We discuss several choices of our validation model in the following.

- *Integrate with a Passive Testing.* All communicating messages of service are passed to it before forwarding to the final destination. Beside the security checking, it also collects the traces. Therefore, a passive testing method can also apply to this step to verify the behavior of the service with respect to a formal specification using these input/output messages. Both on-line [15] or off-line [20] [12] methods can be integrated in this service however, the data correlation must be considered in the case when traces are mixed by many sessions.
- *Integrate with an Analytical Method.* With the model given in Figures 1 and 2 where all communicating messages are collected, an analytical method can be used at the checking engine part for the purpose of understanding and optimizing service usage. Either *off-site* analytical methods such as the measurement of a service's potential audience (opportunity), share of voice (visibility), or buzz (comments) that is happening on the service or *on-site* analytical methods such as service analytics measuring a client's journey can use our model.
- Using this model with a String Solver such as [21] [22], we can apply it to the problem of finding client-side code injection vulnerabilities of the web applications.

5 Conclusion

In this paper, we have proposed a proxy-tester model to protect the security of SaaS. Our contributions are multi-fold. First, we have proposed a firewall model as a validation service for the security checking of SaaS. Second, we have presented an algorithm to check the correctness of messages for both past time and future time cases. Third, we have developed a tool and a checking engine for our model. Fourth, we have conducted a survey on how to use our tool to validate the services that are deployed on several available PaaS. We have also conducted experiments on a simple of service composition which is deployed in GAE and verified the effectiveness of our model. In the future, we plan to investigate our prototype for the real applications and also on Window Azure clouds.

Acknowledgement. Part of the work contributed by K. Chiew was conducted during his visit in Zhejiang University from April to August 2012, which was partly supported by the Chinese Ministry of Science and Technology under Planned-supported Project No. 2011BAD21B0202 and the Natural Science Foundation of China (NSFC) under grant No. 60970081.

References

1. Introduction to cloud computing architecture. White Paper, Sun Microsystems, 1st edn. (June 2009)

2. Lenk, A., Klems, M., Nimis, J., Tai, S., Sandholm, T.: What's inside the cloud? an architectural map of the cloud landscape. In: ICSE Workshop on Software Engineering Challenges of Cloud Computing, pp. 23–31 (2009)
3. Google app engine, `http://code.google.com/appengine/`
4. Window azure, `http://www.windowsazure.com/en-us/`
5. Oracle java cloud service, `https://cloud.oracle.com/mycloud/f?p=service:java:0`
6. Amazon s3, `http://aws.amazon.com/s3/`
7. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 193–303 (2009)
8. Cavalli, A., Benameur, A., Mallouli, W., Li, K.: A passive testing approach for security checking and its pratical usage for web services monitoring. In: NOTERE 2009, Montreal, Canada (2009)
9. Cao, T.D., Castanet, R., Felix, P., Chiew, K.: An approach to automated runtime verification for timed systems: Applications to web services. Journal of Software 7(6), 1338–1350 (2012)
10. Gruschka, N., Luttenberger, N.: Protecting Web Services from DoS Attacks by SOAP Message Validation. In: Fischer-Hubner, S., Rannenberg, K., Yngstrom, L., Lindskog, S. (eds.) Security and Privacy in Dynamic Environments. IFIP, vol. 201, pp. 171–182. Springer, Boston (2006)
11. Salva, S., Laurencot, P., Rabhi, I.: An approach dedicated for web service security testing. In: 5th International Conference on Software Engineering Advances, Nice, France, August 22-27, pp. 494–500 (2010)
12. Morales, G., Maag, S., Cavalli, A., Mallouli, W., de Oca, E., Wehbi, B.: Timed extended invariants for the passive testing of web services. In: IEEE International Conference on Web Service, Miami, Florida, USA, pp. 592–599 (2010)
13. Chan, W., Mei, L., Zhang, Z.: Modeling and testing of cloud applications. In: IEEE Asia-Pacific Services Computing Conference, Singapore, December 7-11, pp. 111–118 (2009)
14. Endo, A.T., Simao, A.: Model-based testing of service-oriented applications via state models. In: IEEE International Conference on Services Computing, pp. 432–439 (2011)
15. Salva, S.: Passive testing with proxy tester. International Journal of Software Engineering and Its Applications 5(4), 1–16 (2011)
16. Using windows azure connect to integrate on-premises web services, `http://msdn.microsoft.com/en-us/library/windowsazure/hh697512.aspx`
17. Cao, T.D., Castanet, R., Felix, P., Morales, G.: Testing of web services: Tools and experiments. In: IEEE Asia-Pacific Services Computing Conference, Jeju, Korea, pp. 78–85 (December 2011)
18. Cao, T.D., Phan-Quang, T.T., Felix, P., Castanet, R.: Automated runtime verification for web services. In: IEEE International Conference on Web Services, Miami, Florida, USA, July 5-10, pp. 76–82 (2010)
19. Nguyen, K.D.: The development of a testing framework for web services. Master's thesis, Poles Universitaire Française in Ho Chi Minh City (December 2010)
20. Cavalli, A., Gervy, C., Prokopenko, S.: New approaches for passive testing using an extended finite state machine specification. Information and Software Technology 45, 837–852 (2003)
21. Hampi: A solver for string constraints, `http://people.csail.mit.edu/akiezun/hampi/index.html`
22. Kaluza string solver, `http://webblaze.cs.berkeley.edu/2010/kaluza/`