Trustworthy Cloud Certification: A Model-Based Approach

Marco Anisetti[1], Claudio A. Ardagna[1], Ernesto Damiani[1], and Nabil El Ioini[2(✉)]

[1] Università degli Studi di Milano, DI, Crema, Italy
{marco.anisetti,claudio.ardagna,ernesto.damiani}@unimi.it
[2] Free University of Bozen, Bolzano, Italy
nabil.elioini@unibz.it

Abstract. Cloud computing is introducing an architectural paradigm shift that involves a large part of the IT industry. The flexibility in allocating and releasing resources at runtime creates new business opportunities for service providers and their customers. However, despite its advantages, cloud computing is still not showing its full potential. Lack of mechanisms to formally assess the behavior of the cloud and its services/processes, in fact, negatively affects the trust relation between providers and potential customers, limiting customer movement to the cloud. Recently, cloud certification has been proposed as a means to support trustworthy services by providing formal evidence of service behavior to customers. One of the main limitations of existing approaches is the uncertainty introduced by the cloud on the validity and correctness of existing certificates. In this paper, we present a trustworthy cloud certification approach based on model verification. Our approach checks certificate validity at runtime, by continuously verifying the correctness of the service model at the basis of certification activities against real and synthetic service execution traces.

Keywords: Certification · Cloud · FSM · Model verification

1 Introduction

Cloud computing paradigm is radically changing the IT infrastructure, as well as traditional software provisioning and procurement. Cloud-based services are becoming the primary choice for many industries due to the advantages they offer in terms of efficiency, functionality, and ease of use. Several factors characterize the choice between functionally-equivalent services at infrastructure, platform, and application layers, among which Quality of Service (QoS) stands out [17]. However, the dynamic and opaque nature of the cloud makes it hard to preserve steady and transparent QoS, which often affects the trust relationship between providers and their customers, and limits customer movement to the cloud.

In response to the need of a trustworthy and transparent cloud environment, several assurance techniques have been defined [6,12,16,22,23]. Among them,

P. Ceravolo et al. (Eds.): SIMPDA 2014, LNBIP 237, pp. 107–122, 2015.
DOI: 10.1007/978-3-319-27243-6_5

certification approaches are on the rise [2,10,26]. Cloud certification in fact is beneficial for both customers having trusted evidence on the correct behavior of the cloud (and corresponding cloud/service providers) in the treatment and management of their data and applications, and providers having trusted evidence on the truthfulness of their claims.

Software certification has a long history and has been used in several domains to increase trust between software system providers and customers. A certificate allows providers to gain recognition for their efforts to increase the quality of their systems, and supports trust relationships grounded on empirical evidence [26]. However, cloud computing introduces the need of re-thinking existing certification techniques in light of new challenges, such as services and processes owned by unknown parties, data not fully under control of their owners, and an environment subject to rapid and sudden changes. Certification processes need then to be tailored to accommodate these new challenges.

An increasing trend in software system certification is to test, monitor, and/or check a system behavior according to its model [4,7,24]. If this paradigm fits well the certification of a static system, it opens the door to potential inconsistencies in cloud environments, where cloud services and corresponding processes are subject to changes when deployed in the production environment, and could therefore differ from their counterparts verified in a lab environment. In this scenario, online certification is a fundamental requirement and evidence collection becomes a continuous and runtime process. In general we need to answer the following question: "*does my service behave as expected at runtime when deployed in the production environment?*". The correct answer to this question passes from the continuous verification of the model used to provide a solid evidence on service behavior. In this paper, we present an approach to continuous model verification at the basis of a sound cloud service certification. Our approach builds on testing and monitoring of execution traces to discover differences between the model originally used to verify and certify a service in a lab environment, and the one inferred in the production environment.

The remainder of this paper is structured as follows. Section 2 presents our certification process and reference scenario. Section 3 presents two different approaches to model generation. Section 4 describes our approach to model verification. Section 5 presents a certification process adaptation based on model verification in Sect. 4. Section 6 illustrates an experimental evaluation of our approach. Section 7 discusses related work and Sect. 8 draws our concluding remarks.

2 Certification Process and Reference Scenario

We describe the certification process at the basis of the approach in this paper and our reference scenario.

2.1 Certification Process

A certification process for the cloud involves three main parties [2], namely a *cloud/service provider* (service provider in the following), a *certification authority*,

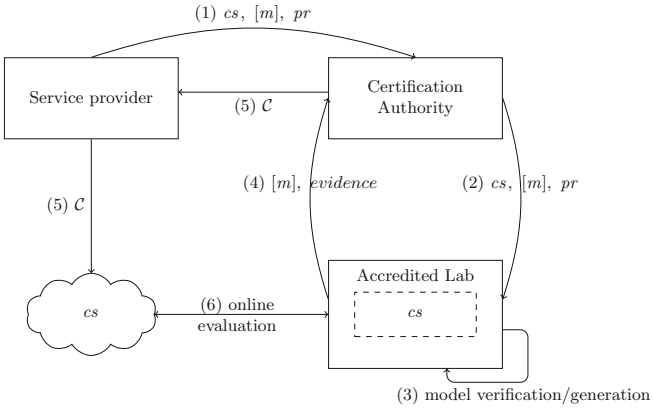

Fig. 1. Certification process

and an *accredited Lab*. It is aimed at collecting the evidence proving a property *pr* for a cloud service *cs* (Target of Certification – ToC). The process, being defined for the cloud, must combine both *offline* and *online evaluation*. Offline evaluation usually considers a copy of the ToC deployed in a lab environment, which can be tested according to the selected property. While offline evaluation verifies services in a controlled setting and provides many advantages such as improved performance and reduced costs, it does not represent alone a suitable approach for the cloud. For this reason, online evaluation has attracted increasing attention, since it provides rich information coming from real services and processes running with production configurations. It also permits to study the behavior of services at runtime, and verify that service certificates are valid over time and across system/environment changes. Online evaluation involves the real ToC, which can be evaluated by monitoring real traces of execution or by testing specific aspects of its behavior.

Figure 1 shows our certification process, taken as reference in this paper. A service provider initiates the certification process by requesting a certificate for one of its cloud services *cs* (ToC) for a given property *pr* (step 1). We note that the request *may* contain the model m of the service to be certified ($[m]$ in Fig. 1). The certification authority receiving the request starts the process by involving its accredited lab (step 2). The latter is delegated by the certification authority to carry out the service evaluation. The accredited lab checks the correctness of model m (step 3), if available, or generates it according to information (e.g., interface description, implementation documentation, source code) available on the service. Model m is then used to collect the evidence, which is delivered to the certification authority (step 4). The certification authority evaluates whether the evidence is sufficient or not to prove the requested property for the service and issue a certificate $\mathcal{C}$ to it (step 5). Upon certificate issuing (step 5), the accredited lab starts a continuous and online evaluation process (step 6), involving model verification and refinement (step 3). We note that,

although some solutions for online evaluation exist [5,27], online evaluation is often preceded by offline evaluation to target those scenarios which are difficult to evaluate on real systems (e.g., Denial of Service, security attacks).

In summary, we consider a certification process whose evidence collection is driven by a model of the ToC [2]. The model is an automaton representing the ToC behavior. It is used to exercise the ToC and collect the evidence needed to verify the property of interest. The model is verified both offline, when provided by the service provider, and online, where its validity is continuously evaluated together with the validity of the corresponding certificate. Our certification process can support different types of evidence (e.g., test-based, monitoring-based) to assess the quality of service. On one hand, test-based certification is grounded on results retrieved by test case executions on the ToC, while monitoring-based certification builds on evidence retrieved by observing real executions of the ToC. The choice of defining a cloud certification scheme based on system modeling is driven by the fact that model-based approaches are common in the context of software evaluation [2,7,19,20,24]. In particular, model-based approaches have been used to analyze software behavior, to prove non-functional properties of software, to infer system executions, and to generate test cases at the basis of software evaluation.

2.2 Reference Scenario

Our reference scenario is a travel planner *TPlan* delivered at cloud application layer (SaaS),[1] implementing functionality for flight and hotel reservation. It is implemented as a composite service and includes three main parties: *(i)* the client requesting a travel plan; *(ii)* the travel agency (i.e., *TPlan*) implementing service *TPlan* and acting as the service orchestrator; and *(iii)* the component services which are invoked by the travel agency within *TPlan* process. The travel agency implements *TPlan* as a business process using a service *BookFlight* for flight reservation, a service *BookHotel* for hotel reservation, and a service *BankPayment* for payment management. Table 1 summarizes the details about the operations of partner services, including *TPlan*.

Upon logging into the system by means of a public interface provided by *TPlan* (operation `login`), customers submit their preferences, which are distributed to the partner operations `findOffers` of services *BookFlight* and *BookHotel*. Once the customer has selected the preferred flight and hotel (calling operations `bookOffer` of services *BookFlight* and *BookHotel*, respectively), service *BankPayment* is invoked to conclude the reservation (operation `makePayment` of service *BankPayment*). We note that *BankPayment* is invoked only in case both *BookFlight* and *BookHotel* are correctly executed. The customer can also cancel a previous transaction and logout from the system (operations `cancelTPlan` and `logout` of *TPlan*).

[1] We note that, though for simplicity a SaaS scenario is considered in the paper, the proposed approach applies to services insisting also on platform (PaaS) and infrastructure (IaaS) layers.

Table 1. Operations of service travel planner

Service	Operation	Description
TPlan	< *tokenID* > `login`(*username*, *password*)	Provides password-based authentication, and returns an authentication token
TPlan	< *TPlanID* > `saveTPlan`(*hotelBookings*, *flightBookings*)	Saves hotel and flight reservations
TPlan	< *confirmation* > `cancelTPlan`(*TPlanID*)	Cancels a plan
TPlan	< *confirmation* > `logout` (*tokenID*)	Disconnects a user and destroys the authentication token
BookHotel	<*confirmation*> `login` (*tokenID*)	Provides a token-based authentication
BookHotel	< *hotelsList* > `findOffers`(*check-in*, *check-out*)	Searches for hotel offers and returns a list of offers
BookHotel	< *confirmation* > `bookOffer` (*offerID*)	Books a specific offer from the offer list
BookHotel	< *confirmation* > `cancel`(*bookingID*)	Cancels an existing reservation
BookHotel	< *confirmation* > `logout`(*tokenID*)	Disconnects a user from service BookHotel
BookFlight	< *confirmation* > `login`(*tokenID*)	Provides a token-based authentication
BookFlight	< *flightsList* > `findOffers`(*departure-day*, *return-day*)	Searches for flight offers and returns a list of offers
BookFlight	< *confirmation* >`bookOffer`(*offerID*)	Books a specific offer from the offer list
BookFlight	< *confirmation* > `cancel`(*bookingID*)	Cancels an existing reservation
BookFlight	< *confirmation* >`logout`(*tokenID*)	Disconnects a user from service BookFlight
BankPayment	< *transactionID* >`makePayment`(*tokenID*, *TPlansID*)	Executes a payment
BankPayment	< *confirmation* > `cancelPayment`(*transactionID*)	Cancels a transaction

3 System Modeling

The trustworthiness of a model-based cloud certification process is strictly intertwined with the trustworthiness of the considered model, or in other words depends on how much the model correctly represents the ToC. The latter depends on *(i)* the amount of information available on the system to be modeled and *(ii)* how the model is generated. We consider two types of models depending on the amount of available information: *(i) workflow-based models* that consider information on service/operation conversations, *(ii) implementation-based models* that extend workflow-level models with details on operation and service implementation. In the following, we give a quick overview on workflow-based and implementation-based models.

3.1 Workflow-Based Model

A workflow-based model represents the ToC in terms of operations/services. At this level we differentiate between two types of workflow: *(i) single-service workflow*, which models the sequence of operation invocations within a single service and *(ii) composite-service workflow*, which models a sequence of operation invocations within a composite service. We note that, in the latter case, operations

belong to different composed services. We also note that single-service workflow can be considered as a degeneration of the general case of composite-service workflow, modeling the internal flow of a single service. Some approaches already used workflow-based models for service verification (e.g., [15,20]). Merten et al. [20] presented a black-box testing approach for service model generation, which entirely relies on service interface and addresses requirements of single-service workflow. Their approach focuses on the definition of a data-sensitive behavioral model in three steps as follows. First, it analyzes the service interface and generates a dependency automaton based on input/output dependencies between operations. Second, a saturation rule is applied, adding possible service invocations from the client to the model (e.g., directly calling an operation without following the subsequent calls). Third, an additional step verifies whether the generated dependencies are semantically meaningful or not. Fu et al. [15] addressed the issue of correctness verification of composite services using a model based approach. To this aim, they developed a tool to check that web services satisfy specific Linear temporal logic properties. Their approach relies on BPEL specifications, which are translated in guarded automata. Automata are then translated into Promela language and checked using the SPIN model checker.

3.2 Implementation-Based Model

An implementation-based model extends the workflow-based model with implementation details coming from the service providers. These details can be used in different ways depending on the type of information they carry on. If they include implementation documentation, they can be used to manually build an automaton representing the behavior of the single operations. Otherwise, if the providers provide traces of the internal operation execution, methods such as the one in [19] can be used to automatically generate the internal behavioral model of the service. Each of the states in the workflow-based model can be further extended in the implementation-based model. At this level we can combine techniques for extracting the service model based on data value constraints [14] and techniques that generate a finite state machine based on service component interactions [8].

Example 1. Figure 2 presents a *workflow-based model* (*composite-service*), a *workflow-based model* (*single service*), and an *implementation-based model*. Workflow-based model (composite-service) is driven by the flow of calls between services in the composition. For instance, *BookFlight* and *BookHotel* are two partner services that are invoked by *TPlan*. *Workflow-based model* (*single service*) is driven by the flow of operation calls within a service, which are annotated over the arcs. For instance, service *BookHotel* exposes different operations to book a hotel room (see Table 1), which are used to generate its FSM model [20]. The model includes the input/output dependencies between operations. Also, for simplicity, a transition is triggered iff the execution of the annotated operation call is successful. Node *Env* in Fig. 2 represents requests sent by the client to different operations. *Implementation-based model* is driven by the flow of code

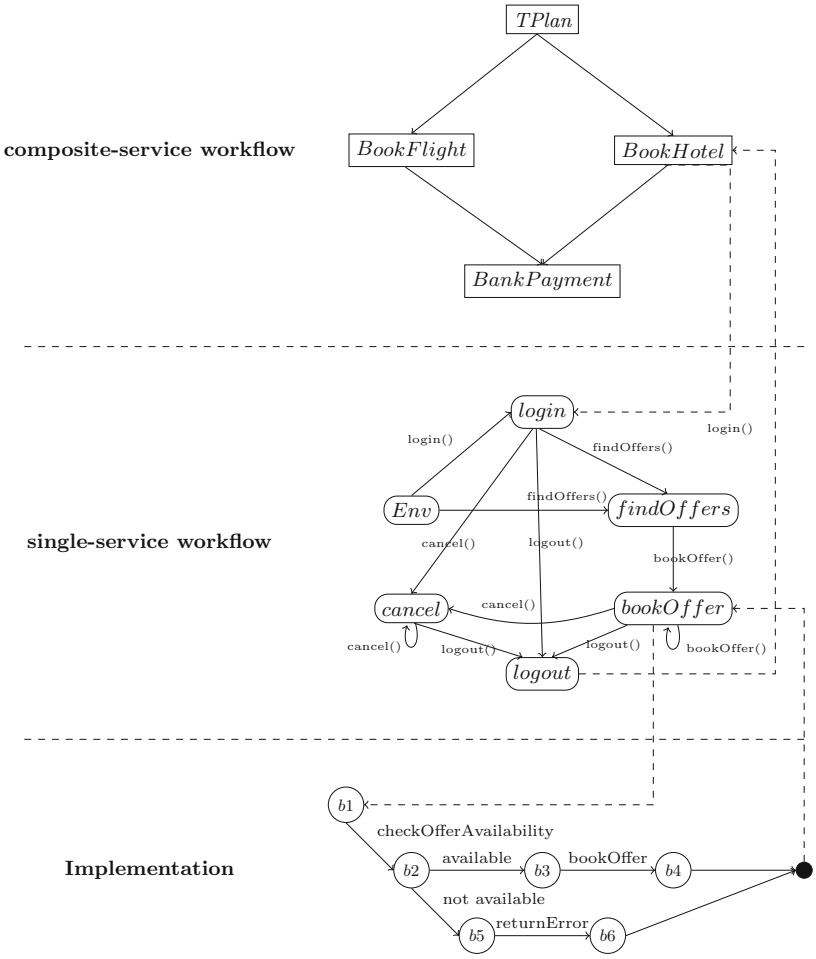

Fig. 2. Three-level modeling for *TPlan* service

instructions within a service operation and provides a fine-grained representation of the considered service.

All generated models can be represented as a Finite State Machine (FSM) defined as follows.

Definition 1 (Model m). *Let us consider a service model m generated by the accredited lab. Model m can be defined as $m{=}(S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ is a finite set of input, $\delta : S \times \Sigma \mapsto S$ is the transition function, $s_0{\in}S$ is the initial state, and $F \subseteq S$ is the set of final states.*

We note that each transition annotation in Σ corresponds to a valid operation/code instruction call at any levels of the cloud stack, including infrastructure

and platform levels. The service model represents the different execution paths for the service. A path can be formally defined as follows.

Definition 2 (Path pt_i). *Given a service model m, a path pt_i is a sequence of states $pt_i=\langle s_0,\ldots,s_n\rangle$, with $s_0 \in S$ and $s_n \in S$ denoting the initial state and a final state, respectively, s.t. $\forall_{i=0}^{n-1} s_i$, $\exists$ a transition $(s_i \times \sigma \mapsto s_{i+1}) \in \delta$.*

When a certification process is considered, the minimum amount of information required for system modeling is mandated by the certification authority and usually involves a combination of workflow and implementation information. The approach in this paper supports both workflow-based and implementation-based modeling, as well as manual modeling.

There is however a subtlety to consider. Sometimes the correctness of a cloud service model, and in turn of the corresponding certification process, passes through the modeling of the configurations of the cloud infrastructure where the service is deployed. For instance, the activities to be done to certify a property *confidentiality against other tenants* depend on the real service deployment, such as how resources are shared among tenants. If a tenant shares a physical machine with other tenants, confidentiality can be guaranteed by encrypting the physical storage; if a tenant is the only one deployed on the physical machine, no encryption is required to preserve the property. This difference, if not correctly modeled, can result in scenarios where laboratory configurations guarantee the property, which is no more valid with production configurations. In this paper, we assume that configurations are not modeled in the FSM leaving such modeling issue for our future work.

4 Model Verification

According to the process in Fig. 1, model verification is under the responsibility of the accredited lab that starts the verification of the ToC model in a lab environment. If the verification is successful the certification process continues and eventually leads to certificate issuing based on the collected evidence. We note that, in those cases where the model is generated by the accredited lab itself, model verification is successful by definition. Upon certificate issuing, the certificate is linked to the ToC deployed in the production environment. However, in addition to common errors that could inadvertently be added during model definition, the ToC can be affected by events changing its behavior once deployed in the cloud production environment, and therefore impairing the correctness of the original modeling effort usually done in a lab environment. It is therefore fundamental to provide an approach to online model verification, which allows to continuously evaluate the correctness and trustworthiness of certification processes and the validity of the corresponding issued certificates. The accredited lab is responsible for online model verification, and checks the consistency between the observed ToC behavior and the original model.

In the following of this section, we present our model verification approach. The approach checks model correctness by collecting execution traces from real

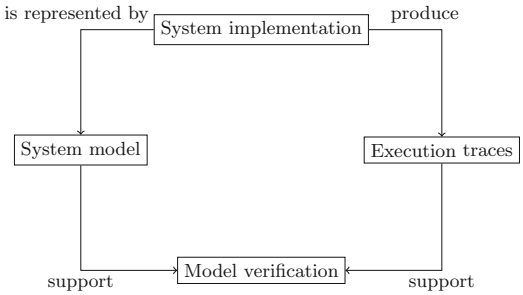

Fig. 3. Model verification process

ToC executions and projecting them on the model itself. Figure 3 shows the conceptual design of our approach.

4.1 Execution Trace Collection

Execution trace collection represents the first step of our verification process. Traces are collected either by monitoring real executions of the ToC involving real customers or by observing the results of ad hoc test-based synthetic executions (synthetic traces). Execution traces can be formally defined as follows.

Definition 3 (Trace T_i). *An execution trace T_i is a sequence $\langle t_1,\ldots,t_n\rangle$ of actions, where t_j can be either an operation execution op_j or a code instruction execution ci_j.*

We note that a trace T_i composed of a sequence of operation executions op_j refers to a workflow-based model, while a trace T_i also including code instruction executions ci_j refers to an implementation-based model. We also note that execution traces can be collected at multiple provider sites, depending on the considered workflow.

Example 2. According to our reference scenario in Sect. 2.2, two types of traces can be collected depending on the considered level (see Fig. 2). At workflow level, a trace represents a sequence of operation invocations, which may belong to a single (e.g., T_i=⟨*BookHotel*.`login`(), *BookHotel*.`findOffers`(), *BookHotel*.`bookOffers`()⟩) or multiple (e.g., T_i=⟨*BookHotel*.`login`(), *BookHotel*.`findOffers`(), *BookHotel*.`bookOffers`(), *BankPayment*.`makePayment`()⟩) services. At implementation level, a trace is represented by a sequence of code instruction executions (e.g. T_i=⟨`checkOfferAvailability`, `notavailable`, `returnError`⟩).

4.2 Service Model Verification

Model verification is a function that takes as input the service model m=(S, Σ, δ, s_0, F) and collected execution traces T_i, and produces as output either

success (1), if the traces conform to the service model, or *failure* (0), otherwise, with a description of the type of inconsistency found. Formally, we can define model verification as follows.

Definition 4 (*MV*). *Model Verification is a function* $MV{:}\mathcal{M}\times\mathcal{T}\rightarrow\{0,1\}\times\mathcal{R}$ *that takes as input the initial service model* $m=(S,\Sigma,\delta,s_0,F)\in\mathcal{M}$ *and execution traces* $T_i\in\mathcal{T}$*, and produces as output either:*

- [1,∅] *iff i)* $\forall T_i\in\mathcal{T}$, $\exists$ *a finite path* $pt_j=\langle s_0,\ldots,s_n\rangle$ *in* $m\in\mathcal{M}$ *s.t.* T_i *is consistent with* pt_j *(denoted* $T_i\equiv pt_j$*) and ii)* $\forall pt_j=\langle s_0,\ldots,s_n\rangle$ *in* $m\in\mathcal{M}$, $\exists T_i\in\mathcal{T}$ *s.t.* $T_i\equiv pt_j$*, or*
- [0,r]*, otherwise, where* $r\in\mathcal{R}$ *describes the reason why a failure is returned.*

We note that model verification is based on a *consistency function* $\equiv$ between collected traces and service model paths, as defined in the following.

Definition 5 (Consistency Function $\equiv$). *Given a trace* $T_i=\langle t_1,\ldots,t_n\rangle\in\mathcal{T}$ *and* $pt_j=\langle s_0,\ldots,s_n\rangle$ *in* $m\in\mathcal{M}$, $T_i\equiv pt_j$ *iff* $\forall t_k\in T_i$, $\exists$ $(s_{k-1}\times\sigma\mapsto s_k)\in\delta$ *s.t.* t_k *and* σ *refer to the same operation/code instruction.*

Definitions 4 and 5 establish the basis for verifying the consistency between observed traces and paths. A failure in the model verification means that there is an inconsistency between the service model and the execution traces, which can affect an existing certification process and invalidate an issued certificate (see Sect. 5). Several types of inconsistencies can take place, which can be reduced to three main classes as follows.

- *Partial path discovery:* it considers a scenario in which a trace is consistent with a subset of a path in the model. In other words, given a trace $T_i=\langle t_1,\ldots,t_n\rangle\in\mathcal{T}$ and a path $pt_j=\langle s_0,\ldots,s_l\rangle$ in $m\in\mathcal{M}$, $\exists$ a subset $\overline{pt}_j$ of pt_j s.t. $T_i\equiv\overline{pt}_j$. This means that while mapping a trace to paths in the model, only an incomplete path is found. We note that, for traces that stop before a final state in m, an inconsistency is raised after a pre-defined timeout expires. The timeout models the expected elapsed time between two operations/instructions execution.
- *New path discovery:* it considers a scenario in which a trace is not consistent with any path in the model. In other words, given a trace T_i and a model m, $\forall pt_j\in m$, $T_i\not\equiv pt_j$. This means that a new path is found, that is, at least a new transition and/or a new state is found in the traces.
- *Broken existing path:* it considers a scenario in which real traces do not cover a path in the model, and the synthetic traces return an error for the same path. In other words, given a path pt_j, $\nexists T_i$ s.t. $T_i\equiv pt_j$. This means that the model includes a path that is not available/implemented in the real ToC.

We note that the above classes of inconsistency are due to either a bad modeling of the service or a change in the production service/environment. Broken existing path inconsistencies can also be due to unexpected failures within the deployed ToC. We also note that additional inconsistencies can be modeled by

Table 2. Execution traces of operation *BookHotel*.

# Trace	Trace values
1	`login(),findOffers(),bookOffer(),logout()`
2	`bookOffer(),cancel()`
3	`findOffers(),login(),bookOffer(),logout()`
4	`Env,cancel()`
5	`Env,login(),logout()`
6	`Env,cancel(),login(),cancel(),logout()`
7	`Env,bookOffer(),login(),bookOffer(),logout()`
8	`bookOffer(),login(),bookOffer(),cancel()`
9	`findOffers(),bookOffer(),login(),bookOffer(),logout()`
10	`findOffers(),login(),logout()`

a combination of the above three. As an example, let us consider a single operation annotating a transition in model m (e.g., `login`() of service *BookHotel*), which is updated to a new version with a new slightly different interface. In this case, two inconsistencies are raised. First, a new path is discovered such that it contains the new interface for operation `login`(); then, a broken existing path is discovered having the original operation `login`().

Example 3 Let us consider the execution traces in Table 2. By projecting the list of traces over the model in Fig. 2 (*single service workflow*), we find some inconsistencies. For instance, trace 2 shows a partial path inconsistency. The sequence of calls (*BookHotel*.`bookOffer`(), *BookHotel*.`cancel`()) maps to a sub-path of the model (*login*, *bookOffer*, *cancel*). Trace 10 shows a new path discovery inconsistency. The sequence of calls (*BookHotel*.`findOffers`(), *BookHotel*. `login`(), *BookHotel*.`logout`()) is supported by the service, while the model does not have this path (i.e., the model is missing a transition from node *findOffers* to node *login*). Finally, let us consider a scenario in which a failure in the authentication mechanism makes function `login`() unreachable. In this case, a broken existing path inconsistency is raised for each path involving function `login`().

5 Certification Process Adaptation

Inconsistencies raised by our model verification in Sect. 4 trigger a certification process adaptation, which could result in a certificate refinement. Certification process adaptation is executed during online evaluation by the accredited lab. It is aimed at incrementally adapting the certification process according to the severity of the model inconsistency, reducing as much as possible the need of costly re-certification processes [3].

The accredited lab, during online evaluation, collects the results of our model verification including possible inconsistencies. Then, it adapts the entire evaluation process following four different strategies.

- *No adaptation:* model verification raises negligible inconsistencies (e.g., a *partial path discovery* that does not affect the certified property). Accredited lab confirms the validity of the certificate.
- *Re-execution:* model verification raises one or more *broken existing path* inconsistencies. Accredited lab triggers additional testing activities by re-executing test cases on broken paths, to confirm the validity of the certificate.
- *Partial re-certification:* model verification raises critical inconsistencies (e.g., a *new path discovery* or a *partial path discovery* that affects the certified property). Accredited lab executes new evaluation activities, such as exercising the new portion of the system for certificate renewing.
- *Re-certification: re-execution* or *partial re-certification* fail and the accredited lab invalidates the certificate. A new certification process re-starts from step 2 in Fig. 1 by adapting original model m according to the model verification outputs (i.e., r in Definition 4).

The role of model verification is therefore twofold. On one side, it triggers *re-execution* of testing activities; on the other side, it adapts the model for partial or complete re-certification. In any case, model verification supports a trustworthy cloud certification process, where accidental and/or malicious modifications to certified services are identified well in advance, reducing the window of time in which an invalid certificate is perceived as valid.

6 Experimental Evaluation

We implemented a Java-based prototype of our model verification approach. The prototype is composed of two main modules, namely, *consistency checker* and *model adapter*. *Consistency checker* receives as input a service model and a set of real and synthetic traces,[2] and returns as output inconsistent traces with the corresponding type of inconsistency. *Model adapter* receives as input the results of the consistency checker and, according to them, generates as output a refined model.

To assess the effectiveness of our prototype, we generated an experimental dataset as follows. We first manually defined the correct implementation-based model m_{cs} of a generic service cs composed of 20 different paths; we then randomly generated 1000 inconsistent models by adding random inconsistencies (Sect. 4.2) to m_{cs}. Inconsistent models are such that 10 %, 20 %, 30 %, 40 %, or 50 % of the paths are different (e.g., missing, new) from the paths in m_{cs}. To simulate realistic customer behaviors, we extended m_{cs} by adding a probability P_i to each path $pt_i \in m_{cs}$, such that $\sum_i P_i = 1$. Probabilities are taken randomly from a normal probability distribution such that there exist few paths whose probability of being invoked tends to 0. Real and synthetic traces are then produced using m_{cs} extended with probabilities.

[2] We remark that synthetic traces are generated by ad hoc testing.

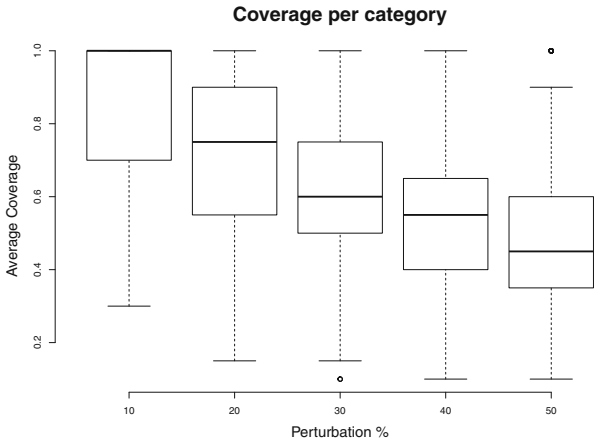

Fig. 4. Refined model coverage of m_{cs}.

The experimental evaluation proceeded as follows. First, using the *consistency checker*, we verified inconsistent models in the dataset and retrieved inconsistent traces together with the type of inconsistency. Then, using the *model adapter*, we built a refined model of each inconsistent model, and evaluated how much these models approximate the correct model m_{cs}. Our results show that the refined models covered 64 % of paths in m_{cs} on average, with an increase of 28 % on the average coverage of the inconsistent models. Also, 17 % of the inconsistent models were able to cover the entire model m_{cs} (i.e., 100 % coverage), while 0 % of the inconsistent models in the dataset covered the entire model by construction. Figure 4 shows a Box and Whisker chart presenting more detailed results on the basis of the rates of differences (i.e., 10 %, 20 %, 30 %, 40 %, 50 %) introduced in the inconsistent models. The Box and Whisker chart in Fig. 4 splits the dataset into quartiles. Within the box, containing two quartiles, the horizontal line represents the median of the dataset. Two vertical dashed lines, called whiskers, extend from the bottom and top of the box. The bottom whisker goes from box to the smallest non-outlier in the data set, and the top whisker goes from box to the largest non-outlier. The outliers are plotted separately as points on the chart. Figure 4 shows that while we increased the perturbation level, we decreased the ability to achieve a full coverage of the initial model. In fact, the median value decreases in such a way that the complete coverage is achieved in the last category (50 %) as an outlier. Nevertheless, in all the perturbation levels, 50 % of the models recover at least 40 % of the inconsistencies. Additionally, in the first case (10 % perturbation), more than 50 % of the models were completely recovered; therefore, both the median and the maximum coverage are equal to 1.

In summary, model adapter does not achieve full coverage of m_{cs} for all refined models. This is mainly due to the fact that paths at low probability are invoked with low probability. If these paths are already specified in an inconsistent model, then a synthetic trace can be generated to evaluate them (if no

real traces are observed) and the refined model covers 100 % of m_{cs}. Otherwise, they remain hidden impairing the ability of model adapter to produce a refined model that covers 100 % of m_{cs}. In our future work, we plan to apply fuzzing and mutation techniques to inconsistent models as a way to reveal hidden paths.

7 Related Work

Model-based verification of software components is increasingly becoming the first choice technique to assess the quality of software systems, since it provides a systematic approach with solid theoretical background. In the context of cloud certification, model-based approaches are used to provide the evidence to certify service quality. The work proposed in [7] starts from a model of the service under certification as a Symbolic Transition System (STS), and generates a certification model as a discrete-time Markov chain. The Markov chain is then used to prove dependability properties of the service. In [4], the authors uses STSs to model services at different levels of granularity. The model is then used to automatically generate the test cases at the basis of the service certification process. Spanoudakis et al. [25] present the EU FP7 Project CUMULUS [9], which proposes a security certification scheme for the cloud based on the integration of different techniques for evidence collection and validation. Additionally, they support also an incremental approach to certify continuously evolving services. A different strategy proposed by Munoz and Mãna in [21] focuses on certifying cloud-based systems using trusted computing platforms. In [13], a security certification framework is proposed. The framework relies on the definition of security properties to be certified. It then uses a monitoring tool to check periodically the validity of the defined properties. In [1], the OPTET project aims to understand the trust relation between the different stakeholders. OPTET offers methodologies tools and models, which provide evidence-based trustworthiness. Additionally, it puts high emphasis on evidence collection during system development, enriched by monitoring and system adaptation to maintain its trustworthiness. In [11], an extension to the Digital Security Certificate [18] is proposed. This certificate contains machine-readable evidence for each claim about the system quality. The certificates are verified by continuously monitoring the certified properties. Differently from the above works, our approach provides a certification process whose trustworthiness is verified by continuously checking the equivalence between a service model and its implementation. Our approach also supports model adaptation to reflect changes that might happen while services are running.

8 Conclusions

In the last few years, the definition of assurance techniques, increasing the confidence of cloud users that their data and applications are treated and behave as expected, has attracted the research community. Many assurance techniques in the context of audit, certification, and compliance domains have been provided,

and often build their activities on service modeling. The correctness of these techniques however suffers by hidden changes in the service models, which may invalidate their results if not properly managed. In this paper, we presented a model-based approach to increase the trustworthiness of cloud service certification. Our approach considers service models at different granularity and verifies them against runtime execution traces, to the aim of evaluating their correctness and, in turn, the validity of the corresponding certificates. We also developed and experimentally evaluated a first prototype of our model verification approach.

Acknowledgments. This work was partly supported by the Italian MIUR project SecurityHorizons (c.n. 2010XSEMLC) and by the EU-funded project CUMULUS (contract n. FP7-318580).

References

1. OPTET Consortium: D3.1 initial concepts and abstractions to model trustworthiness (2013). http://www.optet.eu/project/
2. Anisetti, M., Ardagna, C., Damiani, E.: A certification-based trust model for autonomic cloud computing systems. In: Proceedings of the IEEE Conference on Cloud Autonomic Computing (CAC 2014), London, UK, September 2014
3. Anisetti, M., Ardagna, C., Damiani, E.: A test-based incremental security certification scheme for cloud-based systems. In: Proceedings of IEEE SCC 2015, New York, NY, USA, June–July 2015
4. Anisetti, M., Ardagna, C., Damiani, E., Saonara, F.: A test-based security certification scheme for web services. ACM Trans. Web **7**(2), 1–41 (2013)
5. Antunes, N., Vieira, M.: Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. In: 2011 IEEE International Conference on Services Computing (SCC), pp. 104–111, July 2011
6. Ardagna, C., Asal, R., Damiani, E., Vu, Q.H.: From security to assurance in the cloud: a survey. ACM Comput. Surv. **48**(1), 1–50 (2015)
7. Ardagna, C., Jhawar, R., Piuri, V.: Dependability certification of services: a model-based approach. Computing **97**(1), 51–78 (2015)
8. Biermann, A., Feldman, J.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **C–21**(6), 592–597 (1972)
9. Certification infrastrUcture for MUlti-layer cloUd Services. http://www.cumulus-project.eu/
10. Cimato, S., Damiani, E., Zavatarelli, F., Menicocci, R.: Towards the certification of cloud services. In: Proceedings of the IEEE SERVICES 2013, Santa Clara, CA, USA, June–July 2013
11. Di Cerbo, F., Bisson, P., Hartman, A., Keller, S., Meland, P.H., Moffie, M., Mohammadi, N.G., Paulus, S., Short, S.: towards trustworthiness assurance in the cloud. In: Felici, M. (ed.) CSP EU FORUM 2013. CCIS, vol. 182, pp. 3–15. Springer, Heidelberg (2013)
12. Doelitzscher, F., Reich, C., Knahl, M., Passfall, A., Clarke, N.: An agent based business aware incident detection system for cloud environments. J. Cloud Comput. **1**(1), 1–19 (2012)
13. Egea, M., Mahbub, K., Spanoudakis, G., Vieira, M.R.: A certification framework for cloud security properties: the monitoring path. In: Felici, M., Fernández-Gago, C. (eds.) A4Cloud 2014. LNCS, vol. 8937, pp. 63–77. Springer, Heidelberg (2015)

14. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 213–224. ACM, New York, NY, USA (1999)
15. Fu, X., Bultan, T., Su, J.: Analysis of interacting bpel web services. In: Proceedings of the 13th International Conference on World Wide Web, WWW 2004, pp. 621–630. ACM, New York, NY, USA (2004)
16. Hudic, A., Tauber, M., Lorunser, T., Krotsiani, M., Spanoudakis, G., Mauthe, A., Weippl, E.: A multi-layer and multitenant cloud assurance evaluation methodology. In: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 386–393, December 2014
17. Jianqiang, H., Changguo, G., Huaimin, W., Peng, Z.: Quality driven web services selection. In: IEEE International Conference on e-Business Engineering, 2005, ICEBE 2005, pp. 681–688, October 2005
18. Kaluvuri, S., Koshutanski, H., Di Cerbo, F., Mana, A.: Security assurance of services through digital security certificates. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 539–546, June 2013
19. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 501–510. ACM, New York, NY, USA (2008)
20. Merten, M., Howar, F., Steffen, B., Pellicione, P., Tivoli, M.: Automated inference of models for black box systems based on interface descriptions. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 79–96. Springer, Heidelberg (2012)
21. Munoz, A., Māna, A.: Bridging the gap between software certification and trusted computing for securing cloud computing. In: Proceedings of IEEE SERVICES 2013, Santa Clara, CA, USA, June 2013
22. Pearson, S.: Toward accountability in the cloud. IEEE Internet Comput. **15**(4), 64–69 (2011)
23. Rasheed, H.: Data and infrastructure security auditing in cloud computing environments. Int. J. Inf. Manage. **34**(3), 364–368 (2014). ISSN: 0268-4012. http://dx.doi.org/10.1016/j.ijinfomgt.2013.11.002, http://www.sciencedirect.com/science/article/pii/S026840121300145X
24. Ravindran, K.: Model-based engineering methods for certification of cloud-based network systems. In: 2013 Fifth International Conference on Communication Systems and Networks (COMSNETS), pp. 1–2, January 2013
25. Spanoudakis, G., Damiani, E., Mana, A.: Certifying services in cloud: the case for a hybrid, incremental and multi-layer approach. In: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE), pp. 175–176, October 2012
26. Sunyaev, A., Schneider, S.: Cloud services certification. Commun. ACM **56**(2), 33–36 (2013)
27. Wu, C., Lee, Y.: Automatic saas test cases generation based on soa in the cloud service. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, pp. 349–354 (2012)