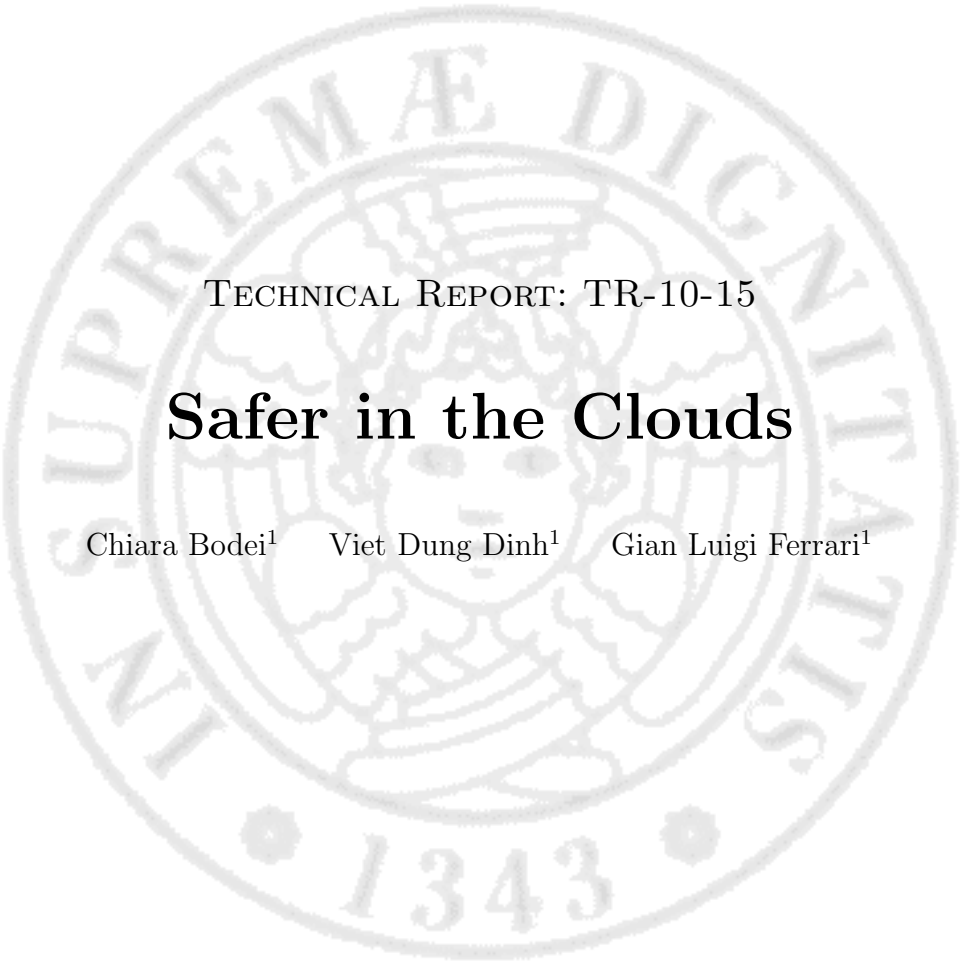
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-10-15

Safer in the Clouds

Chiara Bodei[1] Viet Dung Dinh[1] Gian Luigi Ferrari[1]

September 10, 2010

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Safer in the Clouds *

Chiara Bodei[1], Viet Dung Dinh[1], and Gian Luigi Ferrari[1]

Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
{chiara,dinh,giangi}@di.unipi.it

Abstract. We outline the design of a framework for specifying and reasoning about cloud com- puting systems. The methodology is based on a declarative programming model which takes the form of a λ-calculus enriched with suitable mechanisms to express and enforce application-level security policies governing usages of resources available in the clouds. We will focus on the server side of cloud systems, by adopting a *pro-active* approach, where explicit security policies regolate server's behaviour.

1 Introduction

In the old times, people used to exploit the bakery's oven for their home-made bread. Similarly, people utilised the public mill to obtain flour from their wheat. In both cases, people did not own the physical infrastructure to process their products, neither they invested on it. Instead, they rented usage from a third-party provider. Under this regard, the idea of cloud computing is not completely new, it just updates the above model and adapt it to the Internet world. Cloud Computing customers do not invest on hardware, software or services, but they just pay providers to use them, either on a utility or a subscription basis. They rely on clouds for infrastructures, platforms, and software. Cloud services and the resources they offer over the Internet are therefore used on demand and with a certain degree of flexibility. Usually, these services are based on (a farm of) servers, often virtual ones and are fully managed by their providers. As a consequence, old and new security problems may arise, because if security is related to trust, as Schneier [15] wrote "[cloud computing] moves the trust boundary one step further ... You have to trust your outsourcer completely. You not only have to trust the outsourcer's security, but its reliability, its availability and its business continuity".

In this paper, we describe the design of a framework for specifying and reasoning about cloud computing systems. Our framework takes the form of a declarative model, a dialect of concurrent λ-calculus, for describing and assembling services, and for managing security properties. We also present a methodology for securing the design of services available in the clouds and invoked by possibly untrusted thin clients, such as Web browsers. Next, we informally present the main features of our approach.

* Research supported by the Italian PRIN Project "SOFT".

Services as functions with side effects We adopt the idea of *Software as a service*: each service exposes over the network certain functional behaviour and it is invoked via request/response communication protocols (e.g. SOAP). Services are pluggable entities and composite services are obtained by combining existing elementary or complex services. In our programming model cloud services are viewed as functions. Following [8,9], the service is not a *pure* function as its execution yields a side effect, thus reflecting changes of the service state. For instance, let us consider a simple storage service, offered by a cloud, that gets a string query from the user and accordingly queries the database. The following functional interface can be used to describe the above service.

```
Table fun Q(Query q): Effect e
```

The invocation of the service `Q` with a query value will yield a table value as result. The side effect `e` provides the abstract representation of the modifications to the database such as updates of tables. In other words, the side effects represent the action of accessing service resources. By applying the typing techniques developed in [8,9], we could describe the published interface of the service `Q`, through an annotated functional type, of the form $\texttt{Query} \xrightarrow{e} \texttt{Table}$. When supplied with an argument of type `Query`, the service evaluates to an object of type `Table`. The annotation `e` is the *side effect* of service evaluation that abstractly describes the possible run-time traces of service executions. Since service interfaces play a crucial role in our setting, they have to be *certified* by a trusted party, which guarantees that the side effect abstraction is a sound over-approximation of the actual service behaviour, when acting over service resources.

The main benefit of the service-as-function metaphor, with respect to alternative approaches based on process calculi (see [10] for a survey), is that it provides a *high-level* notion to model services, their composition and interactions, by abstracting from low level networking details.

Security Policies In spite of its undeniable advantages and cost-savings, cloud computing makes data processing inherently risky, as data reside not under the user's control, as effectively said by Diffie in an interview [11]: "... The effect of the growing dependence on cloud computing is similar to that of our dependence on public transportation ... which forces us to trust organizations over which we have no control, ... and subjects us to rules and schedules that we wouldn't apply ... On the other hand, it is so much more economical that we don't realistically have any alternative. ... [Concerning safety] from the view of a broad class of potential users it is very much like trusting the telephone company ... to keep your communications private. "Classical *CIA* (confidentiality, integrity, and availability) concerns are therefore more crucial, and also assuming that the underlying networking infrastructure manages the CIA factors, design flaws can arise and make cloud services *unsafe*. Often security problems do not depend on weird attacks, but simply on the application of careless policies or insufficient policy enforcement. Consequently, it is crucial that safety is addressed when designing a cloud.

Our programming model focuses on application-based security by considering *security policies* as first class programming constructs. We provide explicit constructs to declare and enforce the security policies governing the behaviour of applications, in the style of [5,6]. In our framework, a security policy regulates how resources are granted to and used by services. For instance, let us consider the database service example introduced above. The `Q` service may be unsafe although the code normally runs in most of the cases. An attacker can indeed taint the query string by injecting a command in front; consequently the service would issue dangerous commands such as deleting a file before executing the safe query. This is called an *SQL injection bug*.

Sequences of resource accesses in executions are called *histories*. A *security policy* φ is a regular property of histories. Policies are expressed as languages accepted by an extension of finite state automata, since automata recognize those words that violate the desired property. We refer to the general case of policies as arbitrary safety properties [7,9]. While evaluating a program fragment e protected by a policy φ, written as $\varphi[e]$, the histories must respect φ.

From a methodological perspective, the awareness of security issues from the very beginning of the development process facilitates the design of secure services: security is faced in advance, without sweeping it under the carpet (read it as security patches added later). The database service example above can be moved into a more secure land, by wrapping it inside a suitable security policy φ_{DB}. For instance, the policy can impose that no update operations on the database (i.e. system commands) can be issued during service executions, i.e. the only operations allowed are those in which the database content can only be read. Adding the specified policy to the query interface results in:

$$\texttt{Table fun Q(Query q): Effect e ensuring}\ \varphi_{DB}$$

meaning that each step of service execution must obey the security policy φ_{DB}. Operationally, the run-time structures will enforce the security policy φ_{DB} by monitoring service execution and by catching the occurrences of bad actions, i.e. the actions that violate the policies. Actually, the run-time enforcement mechanism depends on a suitable abstraction of the execution of all the pieces of code (possibly partially) executed so far. This implies that the mechanism enforcing the security policies can make decisions, based on all previous changes of shared resources affected by different user requests. This approach, known under the name of *history-based security*, has been receiving major attention, at both levels of foundations [3,13,16] and of language design/implementation [1,12].

Cloud server Abstractly a cloud server can be seen as a pool of services and computational resources running over a variety of virtual machines. In our programming model, a cloud server is a triple consisting of (i) the history representing the global cloud state (that represents the dependencies among services and resources, as well as virtual machine configurations), (ii) the set of active processes, and (iii) a service store that associates each service name with the script required to load the virtual machine and the resources needed to run the service.

For example, let us consider a cloud server, whose service store provides facilities to convert files to one format to other formats. The initial configuration of the cloud server only includes the service scripts. Notice that these scripts are idle: they are activated by service invocation. We do not model here how clients operate. Clients interactions are asynchronously observed, by means of the server operations required to activate VM as well as the resource needed to operate. For instance, in our running example, the initial configuration includes an empty history ϵ, an empty set of active processes $\mathbf{0}$ and a service store with two possible services $F2F_1$, $F2F_2$ available to convert files having a certain source format:

$$(\epsilon, \mathbf{0}, \{F2F_1 \to p_1, F2F_2 \to p_2\})$$

These two scripts could be characterized by the following types that declare both information about the virtual machines attached to the services and about the costs of service invocations.

$\texttt{Format}_1$ `fun` $\texttt{F2F}_1$`(Format file): Effect` $\texttt{ActivateVM}_1$`;` $\texttt{c}_1$
$\texttt{Format}_2$ `fun` $\texttt{F2F}_2$`(Format file): Effect` $\texttt{ActivateVM}_2$`;` $\texttt{c}_2$

Our cloud server may activate a translation service by the transition:

$$(\epsilon, \mathbf{0}, \{F2F_1 \to p_1, F2F_2 \to p_2\}) \xrightarrow{invoke\ F2F_1} (invoke\, F2F_1, p_1, \{FI2F_1 \to p_1, F2F_2 \to p_2\})$$

This rule spawns the service code in an asynchronous manner. Moreover the server state records the operation that activates the service. Finally, the service scripts are persistent.

Contribution We do not focus here on typing issues. We are concentrated instead on the definition of a semantics-based framework to model cloud servers, and associated reasoning techniques, with a special concern for security properties. We advocate the usage of a declarative model based on functions with side effect to abstractly represent services acting over resources and service assemblies. In particular, the main contributions of the paper are the following.

• The formal semantics of cloud servers as functions, expressed in a suitable dialect of the λ-calculus.
• The modelling of clients interactions is via asynchronous invocation.
• The introduction of a notion of bisimulation equivalence for cloud servers. This bisimulation for cloud servers is proved to be a congruence.
• The presentation of a formal methodology capable of performing *semantics-based* reconfiguration of service clouds.

All the proofs and auxiliary statements presented can be found in the Appendix.

2 The Calculus

We consider a concurrent call-by-value λ-calculus, called $\lambda^{\{\}}$ (*lambda clouds*), enriched with primitives for accessing resources, for declaring and enforcing security policies, and for installing services and managing their invocation.

For simplicity, we assume that resources are objects that are already available in the cloud environment (i.e. resources cannot be dynamically created). We use R (ranged over by r, r_i) to denote the finite set of resources available in the cloud. Resources can be accessed via a given finite set of actions $\alpha_1, \ldots, \alpha_n$. Each action α is characterised by an *arity*, $\mid \alpha \mid$, that corresponds to the number of resources the action operates upon. An event $\alpha(r_1, \ldots, r_k)$ describes the application of the action α on the target resources $r_1, \ldots, r_k$. Traces (ranged over by $\eta, \eta', \ldots$) are finite sequences of events.

Following the approach of [5,6], a security policy is a set of traces that describe the sequences of events satisfying the policy. Security policies are specified via *usage automata* [4]. Usage automata are an extension of Finite State Automata (FSA), where the labels on the edges may contain variables which represents a universally quantified resources, and whose final states denote a violation of the policy.

Usage automata have been proved expressive enough to model security requirements of real-world applications [4]. A usage automaton ϕ gives rise to a finite state automaton when the formal parameters are instantiated with actual resources. Hereafter, we will use the shorthand $\overline{x}$ to indicate that any possible resource different from x and the notation $\eta \models \phi$ to indicate that η drives the automaton ϕ into a state that does not violate the policy, i.e. η respects the policy. Similarly, a sequence of events violates the policy whenever it drives the automaton into a final offending state.

We will simply illustrate some of the features of usage automata with an example and we refer to [4] for more technical details.

Example 1. Let us consider a simple storage service that allows users to store their data. To avoid data being revealed by a third party, it requires that data is encrypted before storing and is logically isolated from other user data. To protect the storage service from external attacks, the storage service runs within the scope of a policy that does not allow to access other user data. To simplify the policy design, we assume to have only two users *(u_1, s_1)* and that each user u_i has the full control of its own data repository s_i. This policy can be easily extended to deal with k users acting over a pool of storage resources. The corresponding automaton, depicted in Fig. 1, describes the policy that regulates the usage of the store resource s_1 by user u_1. Intuitively, the policy constraints the storage service to behave as follows. Firstly, the storage service makes a resource request to *actually* connect to the storage system. This is done by issuing the event *connectStore(u_1,s_1)*. The event *encrypt* is used for encrypting user data, while events *store*(s_i) is used to represent the activity storing the data on the resource s_i. Notice that the policy abstracts from the actual values of data and focusses only on the resources (here users and storages). Automaton transitions are labelled by events. There is a single offending final state (marked by a double circle). To make the description of the automaton more compact we used the notation $\overline{L}$ to indicate any event different from L. We also avoid to report the self loops on the final state (given for every event). Clearly, the sequence of events,

connectStore(u_1,s_1) encrypt; store(s_1) satisfies the policy, while the sequence *connectStore(u_1,s_1) encrypt; store(s_2)* does not.

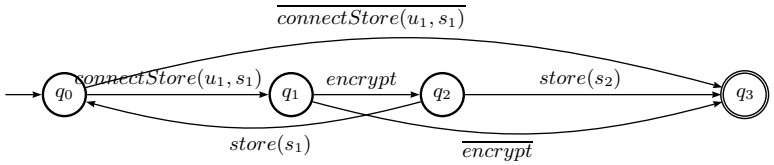


Fig. 1. Usage automaton of the storage service

2.1 Lambda Clouds

We introduce the syntax of $\lambda^{\{\}}$ (see Fig. 2). We assume the existence of a finite repository Π of public service names ranged over by π. A service name can be intuitively thought of as the *URI* of the service. Finally, we use ξ to indicate either a variable or a resource.

$e, e' ::=$		*expressions*
	$\mid \mathbf{0}$	empty value
	$\mid x$	variable
	$\mid r$	resource
	$\mid \alpha(\xi_1, \ldots, \xi_k)$	access event
	$\mid \lambda x.\ e$	abstraction
	$\mid e_1\ e_2$	application
	$\mid e_1 \parallel e_2$	parallel composition
	$\mid$ link e	link component (service constructor)
	$\mid \varphi[e]$	policy framing

Fig. 2. Syntax of $\lambda^{\{\}}$

We assume all standard notions and definitions of lambda calculus. Free and bound variables are defined in the standard way. An expression is closed if it contains no free variable. We denote the set of closed expressions as T_0 and the set of all expressions as T.

The empty value is denoted by $\mathbf{0}$. An event $\alpha \in \Omega$ represents a system- and security-relevant operation. Security policies $\varphi \in \Phi$ are modelled as regular properties of event histories, i.e. properties that are recognizable by a usage automaton, as discussed above. A *policy framing* $\varphi[e]$ enforces regular property of history during execution of e. The parallel composition $\parallel$ allows us to handle concurrency. The constructor *link e* is a new construct, introduced here in order to model the dynamic publication of a service whose code is e.

The values of the calculus are the empty value, variables, resources and lambda abstractions. Hereafter, we used the following abbreviations:

– $\lambda_z e = \lambda_z x.e$ if $x \notin fv(e)$; • $\lambda x e = \lambda_z x.e$ if $z \notin fv(e)$; • $e; e' = (\lambda.e')e$

We model a cloud server as a pool of services and computational resources running over a variety of virtual machines. Formally, a cloud server is a structure of the form

$$(\eta, e, \sigma) \quad \text{where}$$

• η is the history representing the global cloud state, that details the dependencies among services and resources, as well as virtual machine configurations,
• e is the expression that describes the set of active processes, and
• σ is the service store that associates each service name p_i with the expression (script) used to load the virtual machine and the resources required to run the service.

2.2 Operational Semantics

For simplicity, we first define an operational semantics without considering the third component, i.e. the service store. Intuitively, the two-level semantics allows us to reason both on the behavior of individual services and on the behavior of the whole cloud. The behaviour of $\lambda^{\{\}}$-expressions, described in Fig. 3, is defined through a small step operational semantics, called *service semantics*. Configurations are pairs (η, e) consisting of a history η and an expression e. A transition $(\eta, e) \stackrel{(\mu)}{\rightarrow} (\eta', e')$, indicates that, starting from a state described by the history η, the expression e evolves to e', issuing an event μ, and possibly extending the history to η'. Initial configurations have the form $(\emptyset, e)$, where $\emptyset$ denotes the empty history.

We now comment on the operational rules. Rule [EVENT] describes the evaluation of an event $\alpha(r_1, \ldots, r_k)$ that consists in extending the current history with the event, and producing the empty value $\mathbf{0}$. Rules [APP_0],[APP_1] and [APP_2] are standard rules of the call-by-value semantics of lambda calculus. Notice that the whole function body $\lambda_z x.e$ replaces the self variable z after the parameter substitution, so giving an explicit copy-rule semantics for recursive functions. The rule does not change the history component. The rules for the evaluation of parallel expressions are standard. Rules [POL_0], [POL_1] describe the enforcement of policy. The policy framing $\varphi[e]$ enforces the policy φ on the expression e, i.e. the history must respect φ at each step of the evaluation of e and each event issued within e must be checked against φ. When e is just a value, the security policy is simply removed as in [POL_1]. The rule [LINK] asks the repository for having a free service name π to bind to the code e. The result of the evaluation of this transition is the empty value. Moreover, the event $link_\pi$ is issued and appended to the current history η, thus modelling the binding of the service. Notice that the side condition on the service repository Π ensures the uniqueness of the binding: the same service name cannot bind

$$[\text{EVENT}]\ (\eta, \alpha(r_1, \ldots, r_k)) \xrightarrow{\alpha(r_1,\ldots,r_k)} (\eta.\alpha(r_1, \ldots, r_k), \mathbf{0})$$

$$[\text{APP}_0]\quad (\eta, \lambda_z x.e\ v) \xrightarrow{\tau} (\eta, e[\lambda_z x.e/z, v/x])$$

$$[\text{APP}_1]\quad \frac{(\eta, e_1) \xrightarrow{z} (\eta', e_1')}{(\eta, e_1\ e_2) \xrightarrow{z} (\eta', e_1'\ e_2)}$$

$$[\text{APP}_2]\quad \frac{(\eta, e_2) \xrightarrow{z} (\eta', e_2')}{(\eta, v\ e_2) \xrightarrow{z} (\eta', v\ e_2')}$$

$$[\text{PAR}_1]\quad \frac{(\eta, e_1) \xrightarrow{z} (\eta', e_1')}{(\eta, e_1 \parallel e_2) \xrightarrow{z} (\eta', e_1' \parallel e_2)}$$

$$[\text{PAR}_2]\quad \frac{(\eta, e_2) \xrightarrow{z} (\eta', e_2')}{(\eta, e_1 \parallel e_2) \xrightarrow{z} (\eta', e_1 \parallel e_2')}$$

$$[\text{POL}_0]\quad \frac{(\eta, e) \xrightarrow{z} (\eta', e') \ \wedge\ \eta, \eta' \models \varphi}{(\eta, \varphi[e]) \xrightarrow{z} (\eta', \varphi[e'])}$$

$$[\text{POL}_1]\quad \frac{\eta \models \varphi}{(\eta, \varphi[v]) \rightarrow (\eta, v)}$$

$$[\text{LINK}]\quad \frac{\pi \in \Pi, \pi \text{ available}}{(\eta, link\ e) \xrightarrow{link_\pi} (\eta.link_\pi, \mathbf{0})}$$

Fig. 3. Service Semantics

different service codes. Alternatively, one could define a notion of well-formed expressions requiring constraints on semantic of expressions in order to avoid captures of names. Also note that our binding construct does not require the introduction of alpha-conversion.

The whole semantics of a cloud server, called *cloud semantics*, is described in Fig. 4. The inclusion of service store allows us to deal with asynchronous interactions of clients through the rule [INV]. This rule describes the evaluation of service requests. To manage a service invocation through the service name π, the cloud server spawns the code e associated with π. Notice that the event $invoke_\pi$ is appended to the current history η. In our framework, rule [INV] allows us to indirectly model client invocation, through the occurrences of asynchronous events on the server side. Our treatment of service invocation, that is an original feature of our approach, has the consequent benefit to manage a variety of clients, by abstracting from the specific interaction protocols established with the cloud. Our semantic framework handles services as *persistent* entities. Services are not consumed by an invocation: they remain in the service store. Alternatively, one could have introduced a volatile variant, in which the service is removed from service store, after service invocation. However, it is easy to encode volatile services by constraining their unique invocation event. The rule [LINK] adds a new service $\{\pi \rightarrow e\}$ into the service store as an additional side effect. Back to the service for converting formats, where the conversion service is activated by the transition:

$$(\epsilon, \mathbf{0}, \{F2F_1 \rightarrow p_1, F2F_2 \rightarrow p_2\}) \xrightarrow{invoke\ F2F_1} (invoke\ F2F_1, p_1, \{FI2F_1 \rightarrow p_1, F2F_2 \rightarrow p_2\})$$

$$[\text{EVENT}]\quad (\eta, \alpha(r_1,\ldots,r_k), \sigma) \xrightarrow{\alpha(r_1,\ldots,r_k)} (\eta.\alpha(r_1,\ldots,r_k), \mathbf{0}, \sigma)$$

$$[\text{APP}_0]\quad (\eta, \lambda_z x.e\ v, \sigma) \xrightarrow{\tau} (\eta, e[\lambda_z x.e/z, v/x], \sigma)$$

$$[\text{APP}_1]\quad \frac{(\eta, e_1, \sigma) \xrightarrow{z} (\eta', e_1', \sigma')}{(\eta, e_1\ e_2, \sigma) \xrightarrow{z} (\eta', e_1'\ e_2, \sigma')}$$

$$[\text{APP}_2]\quad \frac{(\eta, e_2, \sigma) \xrightarrow{z} (\eta', e_2', \sigma')}{(\eta, v\ e_2, \sigma) \xrightarrow{z} (\eta', v\ e_2', \sigma')}$$

$$[\text{PAR}_1]\quad \frac{(\eta, e_0, \sigma) \xrightarrow{z} (\eta', e_0', \sigma')}{(\eta, e_0\ \parallel\ e_1, \sigma) \xrightarrow{z} (\eta', e_0'\ \parallel\ e_1, \sigma')}$$

$$[\text{PAR}_2]\quad \frac{(\eta, e_1, \sigma) \xrightarrow{z} (\eta', e_1', \sigma')}{(\eta, e_0\ \parallel\ e_1, \sigma) \xrightarrow{z} (\eta', e_0\ \parallel\ e_1', \sigma')}$$

$$[\text{POL}_0]\quad \frac{(\eta, e, \sigma) \xrightarrow{z} (\eta', e', \sigma')\ \wedge\ \eta, \eta' \models \varphi}{(\eta, \varphi[e], \sigma) \xrightarrow{z} (\eta', \varphi[e'], \sigma')}$$

$$[\text{POL}_1]\quad \frac{\eta \models \varphi}{(\eta, \varphi[v], \sigma) \to (\eta, v, \sigma)}$$

$$[\text{LINK}]\quad \frac{\pi \in \Pi, \pi \text{ available}}{(\eta, link\ e, \sigma) \xrightarrow{link_\pi} (\eta.link_\pi, \mathbf{0}, \sigma[\pi \to e])}$$

$$[\text{INV}]\quad (\eta, e, \sigma[\pi \to e']) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel e', \sigma[\pi \to e'])$$

Fig. 4. Cloud Semantics

note that the initial empty process $\mathbf{0}$ indicates that the system waits for the user calling the service. After the call, the system performs an invocation action *invoke* $F2F_1$ and the conversion starts.

3 Abstract Semantics for Clouds

In this section, we introduce the notion of abstract semantics for our framework, by resorting to the notion of bisimilarity.

Applicative bisimulation [2] provides the suitable abstract machinery for semantic reasoning, but not sufficient to deal with the peculiar features of our framework. Basically, the idea behind applicative bisimulation is that in order to reason the equivalence of two functions, we need to know whether their behaviors are the same with all possible closed values. As consequently, applicative bisimulation relies on the output generated by functions, hence it does not capture the events issued by our functions. To clarify this point with an example, let us consider the following cloud servers.

$$(\eta, \alpha; \lambda x.x, \sigma)$$

$$(\eta, \lambda x.x, \sigma)$$

It is easy to see that the two services in the clouds yield the same output. However, the former service during its execution issues an event α and changes its history, while the latter does not.

The management of events is crucial in our framework. By definition, service behaviour is indeed *history-dependent*, i.e. an expression may be executed differently when plugged within different cloud state. For instance, let us consider the cloud servers: $(\eta, \beta; \alpha; \varphi[\gamma], \sigma)$ and $(\eta, \alpha; \beta; \varphi[\gamma], \sigma)$, where η contains neither α or β, and the policy φ states that *never* $\alpha; \beta$ is allowed. After two transitions, the first configuration can make a transition that issues γ, while the second cannot, as illustrated below:

$$(\eta, \beta; \alpha; \varphi[\gamma], \sigma) \xrightarrow{\beta} (\eta.\beta; \alpha; \varphi[\gamma], \sigma) \xrightarrow{\alpha} (\eta.\beta.\alpha; \varphi[\gamma], \sigma) \xrightarrow{\gamma} (\eta.\beta.\alpha.\gamma; \varphi[\mathbf{0}], \sigma)$$
$$(\eta, \alpha; \beta; \varphi[\gamma], \sigma) \xrightarrow{\alpha} (\eta.\alpha; \beta; \varphi[\gamma], \sigma) \xrightarrow{\beta} (\eta.\alpha\beta; \varphi[\gamma], \sigma) \not\rightarrow$$

It leads to the following definition which suitably extends the applicative bisimulation notion. In the following, for the sake of simplicity, we will write $\xrightarrow{\alpha}$, instead of $\xrightarrow{\alpha(r_1,\ldots,r_k)}$. We denote by $\eta\uparrow = \{\eta.\eta' | \eta' \text{ - any history}\}$ the upward-closure of the history η.

Definition 1 (Server Simulation). *A binary relation R_H over T_0 is a* server simulation *w.r.t. a set of histories $H = \eta_o\uparrow$ for some η_o, if whenever eR_Hd then for every history $\eta \in H$:*

- *(1) if $e = \boldsymbol{0}$ then $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \boldsymbol{0})$,*
- *(2) if $e = r$ then $(\eta, d) \xrightarrow{\tau}{}^* (\eta, r)$.*
- *(3) if $e = \lambda_z x.e'$, then $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \lambda_z x.d')$ s.t. $\lambda_z x.e' R_H \lambda_z x.d'$ and for any closed value $v \in T_0$, $e'[\lambda_z x.e/z, v/x]\ R_H\ d'[\lambda_z x.d'/z, v/x]$,*
- *(4) if $(\eta, e) \xrightarrow{\tau} (\eta, e')$ then $e' R_H d$,*
- *(5) if $(\eta, e) \xrightarrow{\alpha} (\eta', e')$ then there exists d' s.t. $(\eta, d) \xrightarrow{\tau}{}^* \xrightarrow{\alpha} (\eta', d')$ and $e'\ R_{H'}\ d'$, where $H' = \eta'\uparrow$.*

where $\xrightarrow{\tau}{}^*$ means zero or more τ transitions. We write $\prec_H$ for the union of all server simulations with respect to a set of initial histories H. If H is the set of all histories, then we call it server similarity and simply write $\prec$ for it.

The first and second clauses ensure that if e can produce a resource r or an empty value, then d can do the same, while the third clause is a variant of applicative simulation. In the forth clause, e evolves to e' by performing an internal transition τ, which does not change the history and after that e' remains equivalent to d. Finally, the fifth clause states that d can generate whatever e can, i.e. if e performs an action α that possibly changes the history into η', then d can perform the same action α after zero or more internal transitions τ^* and generate the same history η'.

It is easy to prove that the relation with respect to a set of history H is included in the one obtained with respect to one of its subsets H'.

Lemma 1. *Let $e, d \in T_0$. If $H' \subseteq H$ and eR_Hd, then $eR_{H'}d$.*

Definition 2. *Let $e, d \in T_0$. We say that d server-simulates e (or e is server-simulated by e) with respect to a set of histories $H = \eta_o\uparrow$ for some η_o if there exists a server simulation R_H s.t. eR_Hd.*

Definition 3 (Server Bisimulation). *A binary relation R_H over T_0, where $H = \eta_o \uparrow$ for some η_o, is a* server bisimulation *if both R_H and its converse R_H^{-1} (i.e. xR_Hy implies $yR_H^{-1}x$) are server simulation with respect to a set of histories H. We write $\sim_H$ for the union of all server bisimulations with respect to a set of histories H. If H is the set of all histories, then we call it server bisimilarity and simply write $\sim$ for it.*

Now we show that bisimulation is a congruence relation using Howe's method [14]. The idea is based on the construction of an auxiliary relation called the *precongruence candidate* $\hat{R}$ in terms of the preorder R which we need to prove a preconguence. It is possible to prove indeed that the preorder is a precongruence if and only if it coincides with the precongruence candidate. The precongruence candidate $\hat{R}$ is a precongruence that contains R, and that is preserved by language constructors. Its definition can be found in the Appendix.

A key property of $\hat{R}$ is that if R is a bisimulation then $\hat{R}$ is a bisimulation, as well. Consequently, if we can show that the precongruence candidate of bisimilarity (union of all bisimulations) is a bisimulation, then bisimilarity and its precongruence candidate coincide, and due to the congruence of the precongruence candidate, bisimilarity is a congruence.
In our setting, we need to show that the precongruence candidate of server bisimulation is also a bisimulation. To prove that $\widehat{\sim}$ is indeed a bisimulation, we need to show that $\widehat{\sim}$ is preserved by computation, i.e. it is preserved under substitution and by tau actions, event actions and abstractions.

Here, we only state the congruence theorem with the main auxiliary lemmata, all the proofs and the auxiliary statements and all the proofs can be found in the Appendix.

Lemma 2 (Substitution). *Let $e_1, e_1', e_2, e_2' \in T$. For every history $\eta \in H$, if $e_1 \widehat{\sim}_H e_1'$ and $e_2 \widehat{\sim}_H e_2'$ then $e_2[e_1/x] \ \widehat{\sim}_H \ e_2'[e_1'/x]$.*

Lemma 3 (Tau Actions). *Let $e, d \in T_0$ and $e \widehat{\sim}_H d$. For every history $\eta \in H$, if $(\eta, e) \xrightarrow{\tau} (\eta, e')$, then $e' \widehat{\sim}_H d$.*

Lemma 4 (Event Actions). *Let $e, d \in T_0$ and $e \widehat{\sim}_H d$. For every history $\eta \in H$ if $(\eta, e) \xrightarrow{\alpha} (\eta', e')$, then there exists d' s.t. $(\eta, d) \xrightarrow{\alpha} (\eta', d')$ and $e' \widehat{\sim}_{H'} d'$, where $H' = \eta' \uparrow$.*

Lemma 5 (Applicative Lemma). *Let $\lambda_z x.e, d \in T_0$. If $\lambda_z x.e \widehat{\sim}_H d$, then there exists d' s.t. for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \lambda_z x.d')$, $\lambda_z x.e \widehat{\sim}_H \lambda_z x.d'$ and for any value v, $e[\lambda_z x.d/z, v/x] \widehat{\sim}_H d'[\lambda_z x.d'/z, v/x]$.*

It is straightforward to prove also the following.

Lemma 6 (Resources and Empty-value). *Let $r, d \in T_0$.*

- *If $r \widehat{\sim}_H d$, then for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, r)$.*
- *If $\boldsymbol{0} \widehat{\sim}_H d$, for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \boldsymbol{0})$.*

We are now ready to state our main result.

Theorem 1 (Congruence). *Server bisimilarity $\sim$ is a congruence.*

Lemma 7. $\forall e, \forall \varphi,\ \varphi[e] \prec e$.

This lemma is particularly useful because it ensures that by instrumenting a program with a policy framing does not add new behavior of the program. As a consequence, behaviour that violates the policy on demand is prevented to occur. This gives rise to a semantics-based methodology of safety refinement process in cycle of software development.

Corollary 1. $\forall e, \forall \varphi,\ \lambda_z x.\varphi[e] \prec \lambda_z x.e$.

Example 2. Back to the storage service `Q` presented in the Introduction, we can specify `Q` as follows:

$$\begin{array}{l} e_{form} = \lambda x.\ e_{process}\ x \\ e_{process} = \lambda y.\ open(db); (query\ db\ y); close(db), \end{array}$$

where e_{form} is the cloud service interface wrapping inside the database. The service gets a query string $\langle strquery \rangle$ from a user, then feeds it to $e_{process}$. In turn, the function $e_{process}$ takes the query y as a parameter, connects to the database db, makes a query to db, by exploiting an auxiliary function *query* with the database and the query string as parameters, then closes the database connection. We abstract from the details of the code of the *query* function here, we just assume that it may perform some internal activities and then issues the database command *dbcmd* and returns a value v. The evolution of the service in the initial state η is as follows:

$$\begin{array}{l} (\eta, e_{form}\ \langle strquery \rangle) \\ \xrightarrow{\tau}\ (\eta, e_{process}\ \langle strquery \rangle) \\ \xrightarrow{\tau}\ (\eta, open(db); ((query\ db\ strquery); close(db)) \\ \xrightarrow{open(db)}\ (\eta.open(db), (query\ db\ strquery); close(db)) \\ \xrightarrow{\tau}{}^{*} \xrightarrow{dbcmd} \xrightarrow{\tau}{}^{*}\ (\eta.open(db).dbcmd, v; close(db)) \\ \xrightarrow{\tau}{}^{*} \xrightarrow{close(db)}\ (\eta.open(db).dbcmd.close(db), \mathbf{0})) \end{array}$$

As previously discussed, the service above is unsafe because it may contain a SQL injection bug: an attacker can try to inject a command in front of query string, e.g. $\langle syscmd; strquery \rangle$, and therefore can execute any dangerous command such as deleting a file, as illustrated by the following trace:

$$\begin{array}{l} (\eta, e_{form}\ \langle syscmd; strquery \rangle) \\ \xrightarrow{\tau}\ (\eta, e_{process}\ \langle syscmd; strquery \rangle) \\ \xrightarrow{\tau}\ (\eta, open(db); (query\ db\ (syscmd; strquery)); close(db)) \\ \xrightarrow{open(db)}\ (\eta.open(db), (query\ db\ (syscmd; strquery)); close(db)) \\ \xrightarrow{\mathbf{syscmd}}\ (\eta.open(db).syscmd, (query\ db\ (strquery)); close(db)) \end{array}$$

To prevent system commands from being executed, we can instrument e_{form} by framing it with a security policy φ_{DB}, which does not allow execution of any *system command*. The corresponding usage automaton is depicted in Fig. 5, where *syscmd* denotes the generic system command. Applying our technical results, we can state that $\lambda x.\varphi_{DB}[e_{process}\ x] \prec \lambda x.e_{process}\ x$. The presence of φ_{DB} in $\lambda x.\varphi_{DB}[e_{process}\ x]$ excludes all generated sequences that contains system commands.

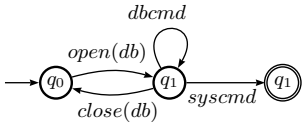


Fig. 5. Usage automaton of the service `Q`

.

3.1 Cloud Bisimulation

Since the introduction of service store adds new transitions and side effects on the server configuration, we need to adapt the notion of equivalence of programs previously used. Let us consider two configurations. A naive approach would be that two configurations contain the same service store. However, with this approach we would not be able to reason about processes of updating and maintenance of services, which is a key property making web-like environment such as cloud environment so popular. Now, observe that if upon a client request one server activates a service code in its service store and makes a transition, the other server should make the same transition, producing the same side effect. Furthermore, the two new processes must be equivalent. This means that two service stores must contain equivalent service codes. Additionally, to make sure that two configurations can make the same transition, the two service store must agree on service names. This leads to the following definitions.

Definition 4. *Two service stores σ, ς are* compatible *($\sigma \equiv \varsigma$) if $dom(\sigma) = dom(\varsigma)$.*

We are now ready for defining cloud bisimulation.

Definition 5 (Cloud Simulation). *A relation on R over T_0 is a cloud simulation w.r.t. a set of histories $H = \eta_o \uparrow$ for some η_o if whenever eR_Hd then for every service store σ, and for every history $\eta \in H$:*

- *(1) if $e = \mathbf{0}$ then there exists service store ς s.t. $\sigma \equiv \varsigma$ and $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} (\eta, \mathbf{0}, \varsigma)$,*
- *(2) if $e = r$ then there exists service store ς s.t. $\sigma \equiv \varsigma$ and $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} (\eta, r, \varsigma)$,*

- *(3) if* $e = \lambda_z x.e'$, *then there exists service store* ς *s.t.* $\sigma \equiv \varsigma$ *and* $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*}$ $(\eta, \lambda_z x.d', \varsigma)$ *s.t.* $\lambda_z x.e' R_H \lambda_z x.d'$ *and for any value* v, $e'[\lambda_z x.e/z, v/x] R_H d'[\lambda_z x.d'/z, v/x]$,
- *(4) if* $e = link\ e'$ $(\eta, e, \sigma) \xrightarrow{link_\pi} (\eta', \mathbf{0}, \sigma[\pi \to e'])$, *then there exist* d', d'' *and service store* ς *s.t.* $\sigma \equiv \varsigma$ *and* $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{link_\pi} (\eta', d'', \varsigma[\pi \to d'])$, $\mathbf{0} R_{H'} d''$ *and* $e' R_{H'} d'$, *where* $H' = \eta' \uparrow$,
- *(5) if* $(\eta, e, \sigma) \xrightarrow{\tau} (\eta, e', \sigma)$ *then* $e' R_H d$,
- *(6) if* $(\eta, e, \sigma) \xrightarrow{\alpha} (\eta', e', \sigma)$, *where* $\alpha \notin \{link_\pi \cup invoke_\pi | \pi \in \Pi\}$, *then there exist* d' *and service store* ς *s.t.* $\sigma \equiv \varsigma$ *and* $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} (\eta', d', \varsigma)$ *and* $e' R_{H'} d'$, *where* $H' = \eta' \uparrow$,
- *(7) if* $(\eta, e, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel \sigma(\pi), \sigma)$ *then there exists service store* ς *s.t.* $\sigma \equiv \varsigma$ *and* $(\eta, d, \varsigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, d \parallel \varsigma(\pi), \varsigma)$ *and* $(e \parallel \varsigma(\pi)) R_{H'} (d \parallel \varsigma(\pi))$, *where* $H' = (\eta.invoke_\pi) \uparrow$,

where $\xrightarrow{\tau}{}^{*}$ *means zero or more* τ *transitions. We write* $\lesssim_H$ *for the union of all cloud simulations with respect to a set of histories* H. *If* H *is the set of all histories, then we call it cloud similarity and simply write* $\lesssim$ *for it.*

All clauses are the same as in server simulation, except the sixth and seventh clauses for creating and invoking a service. Basically, they guarantee equivalence of two service stores during runtime of the system.

Definition 6. *Let* $e, d \in T_0$. *We say that* d *cloud-simulates* e *with respect to a set of histories* $H = \eta_o \uparrow$ *for some* η_o *(or* e *is cloud-simulated by* e*) if there exists a cloud simulation* R_H *s.t.* $e R_H d$.

Definition 7 (Cloud Bisimulation). *A binary relation* R_H *on closed terms in* T_0, *where* $H = \eta_o \uparrow$ *for some* η_o, *is cloud bisimulation if both* R_H *and its converse* R^{-1} *are cloud simulation with respect to a set of histories* H. *We write* $\simeq_H$ *for the union of all cloud bisimulations with respect to a set of histories* H. *If* H *is the set of all histories, then we call it cloud bisimilarity and simply write* $\simeq$ *for it.*

The congruence property of cloud bisimulation is proved similarly to server bisimulation. We use the same candidate precongruence as before, but this time we apply it to cloud bisimulation. The structure of proof is the same. Here, we only state important lemmata, the main results and the relationship between the two kinds of bisimulation.

Lemma 8 (Service Creation). *Let* $link\ e, d \in T_0$. *Assume that for every history* $\eta \in H$ *and service stores* σ, *it holds* $link\ e \widehat{\simeq}_H d$ *and* $(\eta, link\ e, \sigma) \xrightarrow{link_\pi}$ $(\eta.link_\pi, \mathbf{0}, \sigma[\pi \to e])$, *then there exist* d', d'' *and* ς *s.t.* $\sigma \equiv \varsigma$ *and* $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{link_\pi}$ $(\eta', d'', \varsigma[\pi \to d'])$, $\mathbf{0} R_{H'} d''$ *and* $e' R_{H'} d'$, *where* $H' = \eta' \uparrow$.

Lemma 9 (Invocation). *Let* $e, d \in T_0$ *and* $e \widehat{\simeq}_H$. *Assume that for every history* $\eta \in H$ *and service stores* σ *it holds* $(\eta, e, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel \sigma(\pi), \sigma)$

then there exists ς *s.t.* $\sigma \equiv \varsigma$, $(\eta, d, \varsigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, d \parallel \varsigma(\pi), \varsigma)$ *and* $(e \parallel \varsigma(\pi)) R_{H'} (d \parallel \varsigma(\pi))$, *where* $H' = (\eta.invoke_\pi)\uparrow$.

Theorem 2 (Congruence). *Cloud bisimulation* $\simeq$ *is a congruence.*

Lemma 10. $\forall e, \forall \varphi$, $\varphi[e] \lesssim e$.

Corollary 2. $\forall e, \forall \varphi$, $\lambda_z x.\varphi[e] \lesssim \lambda_z x.e$.

Service store in configuration makes cloud bisimulation more difficult to be established than server bisimulation. In the service semantics, expressions of form *link* e seemly behave in the same way regardless expressions e (of course with the same set of available service names π), while in the cloud semantics these expressions must ensure consistency of service store, i.e. expressions e cannot be arbitrary, instead these expressions should be *bisimilar*. In general, server bisimulation is not included by a cloud bisimulation, as shown by the following counter-example.

Example 3. $\forall e, d \in T_0$, *link* $e \sim$ *link* d, but *link* $e \not\simeq$ *link* d.

Lemma 11. $\simeq \subset \sim$.

The methodology outlined in Example 2 can be easily adapted to deal with service store component, thus allowing for securing cloud servers.

4 Concluding Remarks

We have introduced a novel declarative model for cloud computing. The model takes the form of a concurrent λ-calculus enriched with primitive constructs to manage the assembling of services in the cloud, (asynchronous) service invocation, security policies and their enforcement mechanism. Abstract bisimulation semantics provides the formal basis for compositional reasoning of the behaviour of cloud systems. The present work can be seen as first step towards the introduction of formal machineries in the field of cloud computing.

References

1. M. Abadi & C. Fournet (2003): *Access Control Based on Execution History.* In: *NDSS*.
2. S. Abramsky & C.-H. L. Ong (1993): *Full Abstraction in the Lazy Lambda Calculus. Inf. Comput.* 105(2), pp. 159–267.
3. A. Banerjee & D. A. Naumann (2005): *History-Based Access Control and Secure Information Flow.* In: *CASSIS*, LNCS 3362. Springer.
4. M. Bartoletti (2009): *Usage Automata.* In: *ARSPA-WITS*, LNCS 4423. Springer, pp. 32–47.
5. M. Bartoletti, P. Degano & G. Ferrari (2005): *History-Based Access Control with Local Policies.* In: *FoSSaCS*, LNCS 3441. Springer, pp. 316–332.

6. M. Bartoletti, P. Degano, G. Ferrari & R. Zunino (2007): *Secure Service Orchestration*. In: *FOSAD*, LNCS 4667. Springer.
7. M. Bartoletti, P. Degano, G. Ferrari & R. Zunino (2007): *Types and Effects for Resource Usage Analysis*. In: *FoSSaCS*, LNCS 4423. Springer, pp. 32–47.
8. M. Bartoletti, P. Degano, G. Ferrari & R. Zunino (2008): *Semantics-Based Design for Secure Web Services*. *IEEE Trans. Software Eng.* 34(1), pp. 33–49.
9. M. Bartoletti, P. Degano, G. Ferrari & R. Zunino (2009): *Local policies for resource usage analysis*. *ACM Trans. Program. Lang. Syst.* 31(6).
10. R. Bruni (2009): *Calculi for Service Oriented Computing*. In: *SFM 2009*, *LNCS* 5569. Springer.
11. D.Talbot (Nov 2009). *How Secure Is Cloud Computing?* Technology Review. `http://www.technologyreview.com/computing/23951/`.
12. G. Edjlali, An. Acharya & V. Chaudhary (1999): *History-Based Access Control for Mobile Code*. In: *Secure Internet Programming*, LNCS 1603. Springer, pp. 413–431.
13. Philip W. L. Fong (2004): *Access Control By Tracking Shallow Execution History*. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, pp. 43–55.
14. D. J. Howe (1996): *Proving Congruence of Bisimulation in Functional Programming Languages*. *Inf. Comput.* 124(2), pp. 103–112.
15. B. Schneier (2009). *Be Careful When You Come to Put Your Trust in the Clouds*. `http://www.schneier.com/essay-274.html`.
16. C. Skalka & S. F. Smith (2004): *History Effects and Verification*. In: *APLAS*, LNCS 3302. Springer, pp. 107–128.

5 Appendix

A Server Bisimulation

We extend relations on closed terms to open terms, by substituting closed terms for variables.

Definition 8. *Let R be a binary relation over T_0. The binary relation R^o over T is the extension of R to open expressions in T, is defined as follows: eR^oe' if $\sigma(e)R\sigma(e')$ for every closing substitution σ.*

Definition 9 (Precongruence Candidate). *Given a preorder R over T_0, we define the* precongruence candidate *$\widehat{R}$ over T, denoted by $e\widehat{R}e'$, for $e, e' \in T$, by induction on the size of e.*

- *for each variable x, if xR^oe then $x\widehat{R}e$;*
- *for each resource r, if rR^oe then $r\widehat{R}e$;*
- *for each event α, if αR^oe then $\alpha\widehat{R}e$;*
- *for the empty value $\mathbf{0}$, if $\mathbf{0}R^oe$ then $\mathbf{0}\widehat{R}e$;*
- *for $f(\bar{s}), f(\bar{s}') \in T$, if $\bar{s}\widehat{R}\bar{s}'$ and $f(\bar{s}')R^oe$ then $f(\bar{s})\widehat{R}e$, where $\bar{s} = (s_1, ..., s_{arity}(f))$ and $\bar{s}' = (s'_1, ..., s'_{arity(f)})$ for short.*

Now we prove some properties of the candidate precongruence, needed for proving, in turn, that it is a bisimulation.

Lemma 12. *Let R be a preorder over T_0, then the following hold:*

1. *$\widehat{R}$ is reflexive.*
2. *$\widehat{R}$ is constructor respecting, i.e. if $\forall f$, $\bar{e}\widehat{R}\bar{e}'$ then $f(\bar{e})\widehat{R}f(\bar{e}')$.*
3. *$R^o \subseteq \widehat{R}$.*
4. *If we have $e\widehat{R}e'$ and $e'R^oe''$, then $e\widehat{R}e''$.*

Proof. 1. By induction on term size, by definition of $\widehat{R}$ and reflexivity of R.

2. we have $\bar{e}\widehat{R}\bar{e}'$ and, by reflexivity of R,$f(\bar{e}')R^o f(\bar{e}')$. Then by definition of $\widehat{R}$, the property follows immediately.
3. We show that if $eR^o e'$, then $e\widehat{R}e'$, by induction on term e,
   Case: e is a variable: it holds by definition of $\widehat{R}$.
   Case: e is an event, resource or empty process: similarly
   Case: e is a term of form $f(\bar{s})$: by (1) we have $\bar{s}\widehat{R}\bar{s}$. By definition of $\widehat{R}$, $f(\bar{s})\widehat{R}e'$.
   Case: e is a term of form $\varphi[s]$: similarly
4. by induction on term e and transitivity of R:
   Case: e is a variable x: by definition of $\widehat{R}$, $xR^o e'$. By transitivity of R, we have $xR^o e''$. The result follows immediately .
   Case: e is an event, resource or empty process: similarly.
   Case: e is a term $f(\bar{s})$: there exists $\bar{s}'$ s.t $\bar{s}\widehat{R}\bar{s}'$ and $f(\bar{s}')R^o e'$. By transitivity of R, $f(\bar{s}')R^o e''$. The result follows by definition of $\widehat{R}$.

Lemma 13. *Let $e, d \in T_0$. If $H' \subseteq H$ and $eR_H d$, then $eR_{H'} d$.*

Proof. Straightforward.

Lemma 14 (Substitution). *Let $e_1, e_1', e_2, e_2' \in T$. For every history $\eta \in H$, if $e_1 \widehat{\sim}_H e_1'$ and $e_2 \widehat{\sim}_H e_2'$ then $e_2[e_1/x] \widehat{\sim}_H e_2'[e_1'/x]$.*

Proof. By induction on the size of term e_2
Case: e_2 is a variable x: $x\widehat{\sim}_H e_2'$ implies xRe_2', so $e_1' \sim^o e_2'[e_1'/x]$ by definition of $\sim^o$. By property (4) (i.e. $e_1\widehat{\sim}_H e_1'$ and $e_1' \sim^o e_2'[e_1'/x]$), $x[e_1'/x] = e_1'\widehat{\sim}_H e_2'[e_1'/x]$.
Case: e_2 is a variable $y \neq x$: similarly
Case: e_2 is an event or empty process: similarly
Case: e_2 is a term $f(\bar{s})$: since $f(\bar{s})\widehat{\sim}_H e_2'$, there exist s' s.t $\bar{s}\widehat{\sim}_H \bar{s}'$ and $f(\bar{s}')R^o e_2'$. By induction hypothesis, $\bar{s}[e_1/x]\widehat{\sim}_H \bar{s}'[e_1'/x]$ and $f(\bar{s}')[e_1'/x] \sim^o e_2'[e_1'/x]$, so $f(\bar{s})[e_1/x] \sim e_2'[e_1'/x]$.
Case: e is a term of form $\varphi[s]$: similarly.

Lemma 15 (Tau Actions). *Let $e, d \in T_0$ and $e\widehat{\sim}_H d$. For every history $\eta \in H$, if $(\eta, e) \xrightarrow{\tau} (\eta, e')$, then $e'\widehat{\sim}_H d$.*

Proof. By induction on derivation of $(\eta, e) \xrightarrow{\tau} (\eta, e')$:
Case of [APP$_0$] rule: $e = (\lambda_z x.e_1)e_2 : (\eta, (\lambda_z x.e_1)e_2) \xrightarrow{\tau} (\eta, e_1[\lambda_z x.e_1/z, e_2/x])$, where e_2 is a value. We need to show that $e_1[\lambda_z x.e_1/z, e_2/x]\widehat{\sim}_H d$.
Since $e\widehat{\sim}_H d$, w.l.o.g. there exist $d_1, d_2, e_1' \in T_0$ s.t. $\lambda_z x.e_1\widehat{\sim}_H d_1$, $e_2\widehat{\sim}_H d_2$, $e_1\widehat{\sim}_H e_1'$, $\lambda x.e_1' \sim_H d_1$, $d_1\ d_2 \sim_H d$ and d_2 is a value. Otherwise, by choosing a closing substitution for d_1, d_2 and e_1' and definition of $e_2\widehat{\sim}_H d_2$, we can obtain the desired result.
By Lemma 14, we have $e_1[\lambda_z x.e_1/z, e_2/x]\widehat{\sim}_H e_1'[\lambda_z x.e_1'/z, d_2/x]$ (1).
By definition of $\sim$, we have $(\eta, d_1) \xrightarrow{\tau}{}^* (\eta, \lambda_z x.d_1')$ such that $\lambda_z x.e_1' \sim_H \lambda_z x.d_1'$ and $e_1'[\lambda_z x.e_1'/z, d_2/x] \sim_H d_1'[\lambda_z x.d_1'/z, d_2/x]$. Since $d_1\ d_2 \sim_H d$, $d_1'[\lambda_z x.d_1'/z, d_2/x] \sim_H d$ (2).
(1) and (2) implies that $e_1[\lambda_z x.e_1/z, e_2/x]\widehat{\sim}_H d$.
Other cases: we will consider [APP$_1$] rule. The proofs of the other rules are similar.
We have $e = e_1\ e_2$ and $(\eta, e_1) \xrightarrow{\tau} (\eta, e_1')$. Since $e\widehat{\sim}_H d$, w.l.o.g. there exist $d_1, d_2 \in T_0$ s.t. $e_1\widehat{\sim}_H d_1, e_2\widehat{\sim}_H d_2$ and $d_1 d_2 \sim_H d$. By induction hypothesis, $e_1'\widehat{\sim}_H d_1$. Since $\widehat{\sim}_H$ is operator respecting, $e_1' e_2\widehat{\sim}_H d_1 d_2$. This implies that $e_1' e_2\widehat{\sim}_H d$.

Lemma 16 (Event Actions). *Let $e, d \in T_0$ and $e\widehat{\sim}_H d$. For every history $\eta \in H$ if $(\eta, e) \xrightarrow{\alpha} (\eta', e')$, then there exists d' s.t. $(\eta, d) \xrightarrow{\alpha} (\eta', d')$ and $e'\widehat{\sim}_{H'} d'$, where $H' = suff(\eta')$.*

Proof. Obviously, we have $H' \subseteq H$. By induction on derivation of $(\eta, e) \xrightarrow{\alpha} (\eta', e')$:
Case of [EVENT] rule: $e = \alpha$ and $(\eta, \alpha) \xrightarrow{\alpha} (\eta, \mathbf{0})$
Since $\alpha\widehat{\sim}_H d$, we have $\alpha \sim_H d$. So by definition of $\sim_H$ if $(\eta, \alpha) \xrightarrow{\alpha} (\eta.\alpha, \mathbf{0})$, then there exists $d' \in T_0$ s.t $(\eta, d) \xrightarrow{\alpha} (\eta.\alpha, d')$ and $\mathbf{0} \sim_{H'} d'$, where $H' = \eta.\alpha) \uparrow$. This implies that $\mathbf{0}\widehat{\sim}_{H'} d'$.
Other cases: we will consider [APP$_1$] rule. The proofs of the other rules are similar.
We have $e = e_1\ e_2$ and $(\eta, e_1) \xrightarrow{\alpha} (\eta', e_1')$. Since $e\widehat{\sim}_H d$, w.l.o.g. there exist $d_1, d_2 \in T_0$ s.t. $e_1\widehat{\sim}_H d_1, e_2\widehat{\sim}_H d_2$ and $d_1 d_2 \sim_H d$. By induction hypothesis, there exists d_1' s.t $(\eta, d_1) \xrightarrow{\alpha} (\eta', d_1')$, $e_1'\widehat{\sim}_{H'} d_1'$, where $H' = suff(\eta')$. Because $\widehat{\sim}_{H'}$ is operator respecting and $e_2\widehat{\sim}_H d_2$ implying $e_2\widehat{\sim}_{H'} d_2$, so $e_1' e_2\widehat{\sim}_{H'} d_1' d_2$.
Since $(\eta, d_1 d_2) \xrightarrow{\alpha} (\eta', d_1' d_2)$, so there exists d' s.t $(\eta, d) \xrightarrow{\alpha} (\eta', d')$, $d_1' d_2 \sim_{H'} d'$. It implies that $e_1' e_2\widehat{\sim}_{H'} d$.

Lemma 17 (Applicative Lemma). *Let $\lambda_z x.e, d \in T_0$. If $\lambda_z x.e \widehat{\sim}_H d$, then there exists d' s.t. for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \lambda_z x.d')$, $\lambda_z x.e \widehat{\sim}_H \lambda_z x.d'$ and for any value v, $e[\lambda_z x.d/z, v/x] \widehat{\sim}_H d'[\lambda_z x.d'/z, v/x]$.*

Proof. Let $\eta \in H$. By definition of $\widehat{\sim}_H$, w.l.o.g. there exists $c \in T_0$ s.t. $e \widehat{\sim}_H c$ and $\lambda_z x.c \sim_H d$. By Lemma 14, we have $e[\lambda_z x.e/z, v/x] \widehat{\sim}_H c[\lambda_z x.c/z, v/x]$ (1).
By definition of $\sim_H$, there exist d' such that $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \lambda_z x.d')$ and for any value v, $c[\lambda_z x.c/z, v/x] \sim_H d'[\lambda_z x.d'/z, v/x]$ (2).
By property 4, (1),(2) imply that $e[\lambda_z x.d/z, v/x] \widehat{\sim}_H d'[\lambda_z x.d'/z, v/x]$.

Lemma 18 (Resources and Empty-value). *Let $r, d \in T_0$.*

- *If $r \widehat{\sim}_H d$, then for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, r)$.*
- *If $\mathbf{0} \widehat{\sim}_H d$, for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \mathbf{0})$.*

Proof. Straightforward.

Theorem 3 (Congruence). *Server bisimilarity $\sim$ is a congruence.*

Proof. Immediate from the above lemmata.

Lemma 19. *$\forall e, \forall \varphi, \forall H$, $\varphi[e] \prec_H e$.*

Proof. Consider a relation S:

$$S = \{(\varphi[e], e) | \forall e \in T_0 \wedge \forall \varphi\} \cup I_T,$$

where I_T is an identity relation on T_0. We need to show that S is a simulation. Induction on evaluation derivation of e.

Corollary 3. *$\forall e, \forall \varphi$, $\varphi[e] \prec e$.*

Corollary 4. *$\forall e, \forall \varphi$, $\lambda_z x.\varphi[e] \prec \lambda_z x.e$.*

B Cloud Bisimulation

The congruence property proof of cloud bisimulation is similar to that of service bisimulation. Similarly we use the notion of precongruence candidate. Here, we omit the details and only emphasize differences.

Definition 10. *Let $e, d \in T_0$. We say that d are cloud bisimilar if there exists a cloud bisimulation R such that eRd.*

Lemma 20 (Tau actions). *Let $e, d \in T_0$ and $e \widehat{\cong}_H d$. For every history η and service store σ if $(\eta, e, \sigma) \xrightarrow{\tau} (\eta, e', \sigma)$, then $e' \widehat{\cong}_H d$.*

Proof. Similar to the proof of tau lemma for server bisimulation.

Lemma 21 (Event action). *Let $e, d \in T_0$ and $e \widehat{\cong}_H d$. For every history $\eta \in H$ and service stores $\sigma \sim \varsigma$ if $(\eta, e, \sigma) \xrightarrow{\alpha} (\eta', e', \sigma')$, where α is not $link_\pi$ or $invoke_\pi$ for some π, then there exists d' such that $(\eta, d, \varsigma) \xrightarrow{\alpha} (\eta', d', \varsigma')$, $e' \widehat{\cong}_{H'} d'$ and $l' \sim k'$.*

Proof. Similar to the proof of tau lemma for server bisimulation.

Lemma 22 (Applicative lemma). *Let $\lambda_z x.e, d \in T_0$. For every history $\eta \in H$ and service stores $\sigma \sim \varsigma$ If $\lambda_z x.e \widehat{\cong}_H d$, then there exists d' such that $(\eta, d, \varsigma) \xrightarrow{\tau}{}^* (\eta, \lambda_z x.d', \varsigma)$,$\lambda_z x.e \widehat{\cong}_H \lambda_z x.d'$ and for any value v, $e[\lambda_z x.d/z, v/x] \widehat{\cong}_H d'[\lambda_z x.d'/z, v/x]$.*

Proof. Similar to the proof of applicative lemma for server bisimulation.

Lemma 23 (Service Creation). *Let link $e, d \in T_0$. Assume that for every history $\eta \in H$ and service stores σ, it holds link $e \widehat{\cong}_H d$ and $(\eta, link\ e, \sigma) \xrightarrow{link_\pi} (\eta.link_\pi, \mathbf{0}, \sigma[\pi \to e])$, then there exist d', d'' and ς s.t. $\sigma \equiv \varsigma$ and $(\eta, d, \varsigma) \xrightarrow{\tau}{}^* \xrightarrow{link_\pi} (\eta', d'', \varsigma[\pi \to d'])$, $\mathbf{0} R_{H'} d''$ and $e' R_{H'} d'$, where $H' = \eta' \uparrow$.*

Proof. We have $(\eta, link\ e, \sigma) \xrightarrow{link_\pi} (\eta.link_\pi, \mathbf{0}, \sigma[\pi \to e])$.
Since $link\ e \widehat{\simeq}_H d$, so, w.l.o.g. there exists $e' \in T_0$ s.t $e \widehat{\simeq}_H e'$ and $link\ e' \simeq_H d$.

We have that $(\eta, link\ e', \sigma) \xrightarrow{link_\pi} (\eta.link_\pi, \mathbf{0}, \sigma[\pi \to e'])$.
By definition of $\simeq_H$ there exist d', d'' and ς s.t. $\sigma \equiv \varsigma$
$(\eta, d, \varsigma) \xrightarrow{\tau}{}^* \xrightarrow{link_\pi} (\eta.link_\pi, d'', \varsigma[\pi \to d'])$,
$\mathbf{0} \simeq_{H'} d''$, $e' \simeq_{H'} d'$, where $H' = (\eta.link_\pi)\uparrow$, and $H' \subseteq H$. This implies $\mathbf{0} \widehat{\simeq}_{H'} d''$ and $e \widehat{\simeq}_{H'} d'$.

Lemma 24 (Invocation). *Let $e, d \in T_0$ and $e \widehat{\simeq}_H$. Assume that for every history $\eta \in H$ and service stores σ it holds $(\eta, e, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel \sigma(\pi), \sigma)$ then there exists ς s.t. $\sigma \equiv \varsigma$, $(\eta, d, \varsigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, d \parallel \varsigma(\pi), \varsigma)$ and $(e \parallel \varsigma(\pi)) R_{H'} (d \parallel \varsigma(\pi))$, where $H' = (\eta.invoke_\pi)\uparrow$.*

Proof. We have $(\eta, e, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel \sigma(\pi), \sigma)$ and $(\eta, d, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, d \parallel \sigma(\pi), \sigma)$. Obviously, $e \parallel \sigma(\pi) \widehat{\simeq}_{H'} d \parallel \sigma(\pi)$, where $H' = (\eta.invoke_\pi)\uparrow$ (1).
Since $d \simeq_H d$, there exists ς such that $\sigma \equiv \varsigma$ and $(\eta, d, \varsigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, d \parallel \varsigma(\pi), \varsigma)$ s.t. $d \parallel \sigma(\pi) \simeq_{H'} d \parallel \varsigma(\pi)$ (2).
(1),(2) implies $(e \parallel \sigma(\pi)) \widehat{\simeq}_{H'} (d \parallel \varsigma(\pi))$.

Lemma 25 (Resources and Empty-value). *Let $r, d \in T_0$.*

- *If $r \widehat{\simeq}_H d$, then for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, r)$.*
- *If $\boldsymbol{0} \widehat{\simeq}_H d$, for any $\eta \in H$, $(\eta, d) \xrightarrow{\tau}{}^* (\eta, \boldsymbol{0})$.*

Proof. Straightforward.

Theorem 4 (Congruence). *Cloud bisimulation $\simeq$ is a congruence.*

Proof. immediately from the above lemmata.

Lemma 26. $\forall e, \forall \varphi,\ \varphi[e] \lesssim e$.

Proof. Consider a relation S:

$$S = \{(\varphi[e], e) | \forall e \in T_0 \wedge \forall \varphi\} \cup I_T,$$

where I_T is an identity relation on T_0. We need to show that S is a simulation. Induction on evaluation derivation of e.

Corollary 5. $\forall e, \forall \varphi,\ \lambda_z x.\varphi[e] \lesssim \lambda_z x.e$.