An Ambient ASM Model for Client-to-Client Interaction via Cloud Computing

Károly Bósa
Christian Doppler Laboratory for Client-Centric Cloud Computing, JKU, Linz, Austria

Keywords: Cloud Computing, Ambient Abstract State Machines, Ambient Calculus.

Abstract: In our former work we have given a high-level formal model of a cloud service architecture in terms of a novel formal method approach which combines the advantages of the mathematically well-founded software engineering method called *abstract state machines* and of the calculus of mobile agents called *ambient calculus*. This paper presents an extension for this cloud model which enables client-to-client interaction in an almost direct way, so that the involvement of cloud services is transparent to the users. The discussed solution for transparent use of services is a kind of switching service, where registered cloud users communicate with each other, and the only role the cloud plays is to switch resources from one client to another.

1 INTRODUCTION

In (Bósa, 2012b) we proposed a new formal method approach which is able to incorporate the major advantages of the *abstract state machines (ASMs)* (Börger and Stark, 2003) and of *ambient calculus* (Cardelli and Gordon, 2000). Namely, one can describe formal models of distributed systems including mobile components in two abstraction layers such that while the algorithms of executable components (*agents*) are specified in terms of ASMs; their communication topology, locality and mobility described with the terms of ambient calculus in our method.

In (Bósa, 2013) we presented a high-level formal model of a cloud service architecture in terms of this new method. In this paper, we extended this formal model with a *Client-to-Client Interaction (CTCI)* mechanism via a cloud architecture. Our envisioned cloud feature can be regarded as a special kind of services we call *channels*, via which registered cloud users can interact with each other in almost direct way and, what is more, they are able to share available cloud resources among each other as well.

Some use cases, which may claim the need of such CTCI functions, can be for instance: dissemination of large or frequently updated data whose direct transmitting meets some limitations; or connecting devices of the same user (in the later case an additional challenge can be during a particular interpretation of the modeled CTCI functions, how to wrap and transport local area protocols, like *upnp* via the cloud).

The rest of the paper is organized as follows. Section 2 informally summarizes our formerly presented high-level cloud model. Section 3 gives a short overview on the related work as well as ambient calculus and ambient ASM. Section 4 introduces the definitions of some non-basic ambient capability actions which are applied in the latter sections. Section 5 describes the original model extended with the CTCI functions. Section 6 demonstrates how a request for a shared service is processed by the discussed model. Finally, Section 7 concludes this paper.

2 OVERVIEW ON OUR MODEL

Roughly our formal cloud model can be regarded as a pool of resources equipped with some infrastructure services, see Figure 1a. Depending whether these abstract resources represent only physical hardware and virtual resources or entire computing platforms the model can be an abstraction of *Infrastructure as a Service (IaaS)* or *Platform as a Service (PaaS)*, respectively. The basic hardware (and software) infrastructure is owned by the cloud provider, whereas the softwares running on the resources are owned by some users. We assume that these softwares may be offered as a *service* and thus used by other users. Accordingly, we apply a relaxed definition of the term service cloud here, where a user who owns some applications running on some cloud resources may become a soft-

Bosa K..
An Ambient ASM Model for Client-to-Client Interaction via Cloud Computing.
DOI: 10.5220/0004490904590470
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 459-470
ISBN: 978-989-8565-68-6

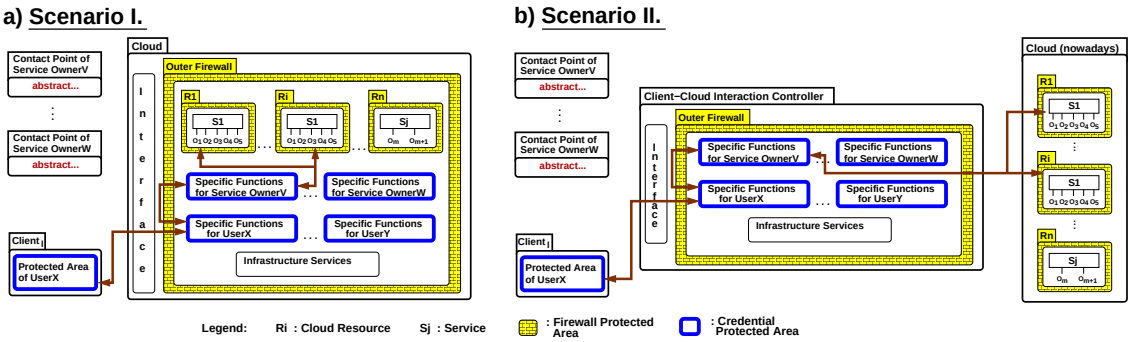

Figure 1: Application of our Model According to Different Scenarios.

ware service provider at the same time. Thus, from this aspect the model can be regarded as an abstraction of a mixture of *Software as a Service (SaaS)* and of IaaS (or a mixture of SaaS and PaaS).

We make a distinction between two kinds of cloud users. The normal *users* are registered in the cloud and they subscribe to and use some (software) services available in the cloud. The *service owners* are users as well, but they also rent some cloud *resources* to deploy some *service instances* on them.

For representing service instances, we adopt the formal model of *Abstract State Services (AS^{2}s)* (Ma et al., 2008; Ma et al., 2009). In an AS^2 we have views on some hidden database layer that are equipped with *service operations* denoted by unique identifers $o_1,\ldots,o_n$. These service operations are actually what are exported from a service to be used by other systems or directly by users. The definition of AS^{2}s also includes the *pure data services* (service operations are just database queries) and the *pure functional services* (operation without underlying database layer) as extreme cases.

In our approach the model assumes that each service owner has a dedicated contact point which resides out of the cloud. It is a special kind of client that can also act as a server for the cloud itself in some cases. Namely, if a registered cloud user intends to subscribe to a particular service, she sends a subscription request to the cloud, which may forward it to such a special kind of client belonging to the corresponding service owner. This client responses with a special kind of action scheme called *service plot*, which algebraically defines and may constrain how the service can be used by the user[1]. (e.g.: it determines the permitted combination of service operations). This special kind of client is abstract in the current model.

The received service plots, which may be composed individually for each subscribing user by service owners, are collected with other cloud functions available for this particular user in a kind of personal user area by the cloud. Later, when the subscribed user sends a service request, it is checked whether the requested service operations are allowed by any service plot. If a requested operation is permitted then it is triggered to perform, otherwise it is blocked as long as a plot may allow to trigger it in the future. Each triggered operation request is authorized to enter into the user area of the corresponding service owner to whom the requested service operation belongs. Here a scheduler mechanism assigns to the request a one-off access to a cloud resource on which an instance of the corresponding service runs. Then the service operation request is forwarded to this resource, where the request is processed. Finally, the outcome of the performed operation returns to the area of the initiator user, where the outcome is either stored or send further to a given client device. In this way, the service owners have direct influence to the service usage of particular users via the provided service plots.

Regarding our proposed cloud model one of the major questions can be whether it is adaptable to the leading cloud solutions (e.g.: Amazon S3, Microsoft Azure, IBM SmartCloud, etc.). Since due to the applied ambient concept the relocation of the system components is trivial, we can apply our model according to different scenarios. For instance, all our novel functions including the client-to-client interaction can be shifted to the client side and wrapped into a middleware software which takes place between the end users and cloud in order to control the interactions of them, see Figure 1b. The specified communication topology among the distributed system components remains the same in this later case.

3 RELATED WORK

It is beyond the scope of this paper to discuss the vast literature of formal modeling mobile systems and SOAs, but we refer to some surveys on these

[1]For an algebraic formalization of plots *Kleene algebras with tests (KATs)* (Kozen, 1997) has been applied.

fields (Boudol et al., 1994; Cardelli, 1999; Schewe and Thalheim, 2007).

One of the first examples for representing various kinds of published services as a pool of resources, like in our model, was in (Tanaka, 2003).

In (Ma et al., 2012) a formal high-level specifications of service cloud is given. This work is similar to ours in some aspects. Namely, it applies the language-independent AS^{2}s with algebraic plots for representing services. But it principally focuses on service specification, service discovery, service composition and orchestration of service-based processes; and it does not apply any formal approach to describe either static or dynamically changing structures of distributed system components.

In the rest of this section, we give a short summary on ambient calculus and ambient ASM, respectively, in order to facilitate the understanding of the latter sections.

3.1 Ambient Calculus

The ambient calculus was inspired by the π-calculus (Milner et al., 1992), but it focuses primarily on the concept of locality and process mobility across well defined boundaries instead of channel mobility as π-calculus. The concept of *ambient*, which has the following main features, is central to the calculus:

- An ambient is defined as a bounded place where computation happens.
- Each ambient has a name, which can be used to control access (entry, exit, communication, etc.). Ambient names may not be unique.
- An ambient can be nested inside other ambients.
- An ambient can be moved. When an ambient moves, everything inside it moves with it (the boundary around an ambient determines what should move together with it).

The ambient calculus includes only the mobility and communication primitives depicted in Table 1. The main syntactic categories are *processes* (including both ambients and agents) and *actions* (including both *capabilities* and *communication primitives*). A reduction relation $P \longrightarrow Q$ describes the evolution of a term P into a new term Q (and $P \longrightarrow^* Q$ denotes a reflexive and transitive reduction relation from P to Q). A summarized explanation of the primitives and the relevant reduction rules is given below:

Parallel Composition. Parallel execution is denoted by a commutative and associative binary operator[2], which complies the rule:

[2]The parallelism in ambient calculus is always interpreted via *interleaving*.

Table 1: The Mobility and Communication Primitives of Ambient Calculus.

P,Q,R::=	processes
$P \mid Q$	parallel composition
$n[\,P\,]$	an ambient named n with P in its body
$(\nu\, n)P$	restriction of name n within P
0	inactivity (**skip** process)
$!P$	replication of P
$M.P$	(capability) action M then P
$(x).P$	input action (the input value is bound to x in P)
$\langle a \rangle$	async output action
$M_1.M_2 \ \ldots\ .M_k.P$	a path formation on actions then P
M::=	capabilities
IN n	entry capability (to enter n)
OUT n	exit capability (to exit n)
OPEN n	open capability (to dissolve n's boundary)

$$P \longrightarrow Q \Longrightarrow P \mid R \longrightarrow Q \mid R$$

Ambients. An ambient is written as $n[P]$, where n is its name and a process P is running inside its *body* (P may be running even if n is moving):

$$P \longrightarrow Q \Longrightarrow n[\,P\,] \longrightarrow n[\,Q\,]$$

Ambients can be embedded into each other such that they can form a hierarchical tree structure. An ambient body is interpreted as the parallel composition of its elements (its local ambients and its local agents) and can be written as follows:

$$n[\,P_1 \mid \ldots \mid P_k \mid m_1[\ldots] \mid \ldots \mid m_l[\ldots]\,] \text{ where } P_i \neq m_i[\ldots]$$

Replication. $!P$ denotes the unlimited replication of the process P. It is equivalent to $P \mid !P$. There is no reduction rule for $!P$ (the term P under $!$ cannot start until it is expanded out as $P \mid !P$).

(Name) Restriction. $(\nu\, n)P$ creates a new (unique) name n within a scope P. n can be used to name ambients and to operate on ambients by name. The name restriction is transparent to reduction:

$$P \longrightarrow Q \Longrightarrow (\nu\, n)P \longrightarrow (\nu\, n)Q$$

Furthermore, one must be careful with the term $!(\nu\, n)P$, because it provides a fresh value for each replica, so

$$(\nu\, n)!P \neq !(\nu\, n)P$$

Inactivity. 0 is the process that does nothing.

Actions and Capabilities. An action defined in the calculus can precede a process P. P cannot start to execute until the preceding actions are performed. Those actions that are able to control the movements of ambients in the hierarchy or to dissolve ambient boundaries are restricted by capabilities. By using capabilities an ambient can allow some processes to

perform certain operations without publishing its true name to them (see the entry, exit and open below).

Communication Primitives. The *input actions* and the *asynchronous output actions* can realize local anonymous communication within ambients, e.g.:

$$(x).P \mid \langle a \rangle \longrightarrow P(x/a)$$

where an input action captures the information a available in its local environment and binds it to the variable x within a scope P. In case of the modeling of a real life system, communication of (ambient) names should be rather rare, since knowing the name of an ambient gives a lot of control over it. Instead, it should be common to exchange restricted capabilities to control interactions between ambients (from a capability the ambient name cannot be retrieved).

Entry Capability. The capability action IN m instructs the surrounding ambient to enter a sibling ambient named m. If a sibling ambient m does not exist, the operation blocks until such a sibling appears. If more than one sibling ambient called m can be found, any of them can be chosen. The reduction rule for this action is:

$$n[\ \text{IN}\ m.P \mid Q\ ] \mid m[\ R\ ] \longrightarrow m[\ n[\ P \mid Q\ ] \mid R\ ]$$

Exit Capability. The capability action OUT m instructs the surrounding ambient to exit its parent ambient called m. If the parent is not named m, the operation blocks until such a parent appears. The reduction rule is:

$$m[\ n[\ \text{OUT}\ m.p \mid Q\ ] \mid R\ ] \longrightarrow n[\ P \mid Q\ ] \mid m[\ R\ ]$$

Open Capability. The capability action OPEN n dissolves the boundary of an ambient named n located in the same ambient as OPEN n. If such an ambient cannot be found in the local environment of OPEN n, the operation blocks until an ambient called n appears. The relevant rule is:

$$\text{OPEN}\ n.P \mid n[\ Q\ ] \longrightarrow P \mid Q$$

Path Formation on Actions. It is possible to combine multiple actions (e.g.: capabilities and input actions). For this, a path formation operation is introduced on actions $(M_1.M_2\ \ldots\ .M_k)$. For example. (IN n.IN m).P is interpreted as IN n.IN m.P (P does not start to execute until the preceding capabilities are performed).

3.2 Ambient ASM

In (Börger et al., 2012) the ambient concept (notion of "nestable" environments where computation can happen) is introduced into the ASM method. In that article an ASM machine called MOBILEAGENTSMANAGER is described as well, which gives a natural formulation for the reduction of three basic capabilities (ENTRY, EXIT and OPEN) of ambient calculus in terms of the *ambient ASM* rules. For this machine an ambient tree hierarchy is always specified initially in a dynamic derived function called *curAmbProc*. The machine MOBILEAGENTSMANAGER transforms the current value of *curAmbProc* according to the capability actions given in *curAmbProc*. Since one of the main goals of (Börger et al., 2012) is to reveal the inherent opportunities of the new ambient concept introduced into ASMs, the presented definitions for moving ambients are unfortunately incomplete.

In (Bósa, 2012b) we extended this ASM machine given in (Börger et al., 2012), such that it fully captures the calculus of mobile agents and it can interpret the agents' algorithms (given in terms of ASM syntax in *curAmbProc* as well) in the corresponding contexts. By this one is able to describe formal models of distributed systems including mobile components in the mentioned two abstraction layers.

Since the definition of ambient ASM is based upon the semantics of ASM without any changes, each specification given this way can be translated into a traditional ASM specification.

4 DEFINITIONS

As Cardelli and Gordon showed in (Cardelli and Gordon, 2000) the ambient calculus with the three basic capabilities (ENTRY, EXIT and OPEN) is powerful enough to be Turing-complete. But for facilitating the specification of such a compound formal model as a model of a cloud infrastructure, we defined some new *non-basic capability* actions encoded in terms of the three basic capabilities. Table 2 summarizes the definitions of these non-basic capabilities.

4.1 Applied Notations

In the rest of this paper, the term $P \longrightarrow^* Q$ denotes multiple reduction. In addition, $P \xrightarrow{asm}{}^* Q$ denotes one or more steps of some ASM agents.

In the reductions presented in the latter sections, the names of some ASM agents are followed by subscripts which contain some enumerated expressions between parenthesis. Such a subscript refers to (a relevant part of) the current state of an agent, e.g.: $\text{AGENT}_{(ctr\_state:=RunningState,\ x:=a)}$.

We also apply the following abbreviations:

$M_1.\ \ldots\ M_n \equiv M_1.\ \ldots\ .M_n.0$ where 0 = inactivity
$n[\,] \equiv n[0]$ where 0 = inactivity
$(\nu\ n_1, \ldots, n_m)P \equiv (\nu\ n_1) \ldots (\nu\ n_m)P$

Table 2: A Summary of the Definitions of some Non-Basic Capabilities.

Names	New Reduction Relations (Based on the Definitions)	Definitions of the New Capabilities
1) Renaming	$n[\ n\ \text{BE}\ m.P \mid Q\ ] \longrightarrow^* m[\ P \mid Q\ ]$	$n\ \text{BE}\ m.P \equiv (\nu\ s)(s[\ \text{OUT}\ n \mid m[\ \text{OPEN}\ n.\text{OUT}\ s.P\ ]\ ] \mid \text{IN}\ s.\text{IN}\ m)$
2) Seeing	$n[\ ] \mid \text{SEE}\ n.P \longrightarrow^* n[\ ] \mid P$	$\text{SEE}\ n.P \equiv (\nu\ r, s)(\ r[\ \text{IN}\ n.\text{OUT}\ n.r\ \text{BE}\ s.P\ ] \mid \text{OPEN}\ s\ )$
3) Wrapping	$n[\ m\ \text{WRAP}\ n.P\ ] \longrightarrow^* m[\ n[\ P\ ]\ ]$	$m\ \text{WRAP}\ n.P \equiv$ $(\nu\ s, r)(\ s[\ \text{OUT}\ n.\text{SEE}\ n.s\ \text{BE}\ m.r[\ \text{IN}\ n\ ]\ ] \mid \text{IN}\ s.\text{OPEN}\ r.P\ )$
4) Allowing Code	$\text{ALLOW}\ key.P \mid key[\ Q\ ] \longrightarrow^* P \mid Q$	$\text{ALLOW}\ key.P \equiv \text{OPEN}\ key.P$
5) Drawing in (an Ambient)	$m[\ Q \mid \text{ALLOW}\ key\ ] \mid n[\ n\ \text{DRAWIN}_{key}\ m.P\ ]$ $\longrightarrow^* n[\ Q \mid P\ ]$	$n\ \text{DRAWIN}_{key}\ m.P \equiv$ $key[\ \text{OUT}\ n.\text{IN}\ m.\text{IN}\ n\ ] \mid \text{ALLOW}\ m.P$
6) Drawing in Then Release a Lock	$m[\ Q \mid \text{ALLOW}\ key\ ] \mid$ $n[\ \text{DRAWIN}_{key}\ m\ \text{THENRELEASE}\ lock.P\ ]$ $\longrightarrow^* lock[\ n[\ Q \mid P\ ]\ ]$	$n\ \text{DRAWIN}_{key}\ m\ \text{THENRELEASE}\ lock.P \equiv$ $key[\ \text{OUT}\ n.\text{IN}\ m.\text{IN}\ n\ ] \mid \text{SEE}\ m.lock\ \text{WRAP}\ n.\text{ALLOW}\ m.P$
7) Concurrent Server Process	$m[\ Q \mid \text{ALLOW}\ key\ ] \mid \text{SERVER}^n_{key}\ m.P$ $\longrightarrow^* \text{SERVER}^n_{key}\ m.P \mid n_k^{uniq}[\ Q \mid P\ ]$	$\text{SERVER}^n_{key}\ m.P \equiv$ $(\nu\ next)(next[\ ] \mid$ $!(\nu\ n)(\text{OPEN}\ next.n[$ $n\ \text{DRAWIN}_{key}\ m\ \text{THENRELEASE}\ next.P\ ]\ )\ )$

4.2 Non-Basic Capabilities

Below we give an informal description of each non-basic capability in Table 2. It is beyond the scope of the paper to present detailed explanations and reductions of their ambient calculus-based definitions, but we refer to our former works (Bósa, 2013) and (Bósa, 2012a) for more details.

1. Renaming. This capability is applied to rename an ambient comprising this capability. Such a capability was already given in (Cardelli and Gordon, 2000), but our definition differs from Cardelli's definition. In the original definition, the ambient *m* was not enclosed into another, name restricted ambient (it is called *s* in our definition), so after it has left ambient *n*, *n* may enter into another ambient called *m* (if more than one *m* exists as sibling of *n*).

2. Seeing. This operation was defined in (Cardelli and Gordon, 2000) and it is used to detect the presence of a given ambient.

3. Wrapping. Its aim is to pack an ambient comprising this capability into another ambient.

4. Allowing Code. This capability is just a basic OPEN capability action. It is applied if an ambient allows/accepts an ambient construct (which may be a bunch of foreign codes) contained by the body of one of its sub-ambients (which may was sent from a foreign location). The name of the sub-ambient can be applied for identifying its content, since its name may be known only by some trusted parties.

5. Draw in (an Ambient) The aim of this capability is to draw in a particular ambient (identified by its name) into another ambient (which contains this capability) and then to dissolve this captured ambient in order to access to its content. For achieving this, a mechanism (contained by the ambient *key*) is applied which can be regarded as an abstraction of a kind of protocol identified by *key*. The ambient *key* enters into one of the available target ambients which should accept its content in order to be led into the initiator ambient.

6. Draw in Then Release a Lock. This capability is very similar to the previous one, but after *m* has been captured by *n* (and before *m* is dissolved), *n* is wrapped by another ambient. The new outer ambient is usually employed as release for a lock[3].

7. Concurrent Server Process. This ambient construct can be regarded as an abstraction of a multi-threaded server process. It is able to capture and process several ambients having the same name in parallel. In the definition *n* is a replicated ambient whose each replica is going to capture another ambient called *m*. Since there is a name restriction quantifier in the scope of the replication sign, which bounds the name *n*, a new, fresh and unique name (denoted by n_k^{uniq}) is generated for each replica of *n*. One of the consequences of this is that nobody knows from outside the true name of a replica of the ambient *n*, so each replica of *n* is inaccessible from outside for anybody (even for another replica of *n*, too).

5 THE EXTENDED FORMAL MODEL

In the formal model discussed in this section, we assume that there are some standardized public ambient names, which are known by all contributors. We distinguish the following kinds of public names: addresses (e.g.: *cloud*, $client_1$, ..., $client_n$), message types (e.g.: *reg*(*istration*), *request*, *subs*(*cription*),

[3]In ambient calculus the capability OPEN *n.P* is usually used to encode locks (Cardelli and Gordon, 2000). Such a lock can be released with an ambient like *n*[*Q*] whose name corresponds with the target ambient of the OPEN capability.

returnValue, etc.) and parts of some common protocols (e.g.: *lock*, *msg*, $intf$, *access*, *out*, o_1, ..., o_s, *op*). All other ambient names are non-public in the model which follows:

$curAmbProc := root[\ Cloud \mid Client_1 \mid \ldots \mid Client_n\ ]$[4]

In this paper, we focus on the cloud service side and we leave the client side abstract.

5.1 User Actions

In the model user actions are encoded as messages. A user can send the following kinds of messages to the cloud:

$MsgFrame \equiv msg[\ \text{IN}\ cloud.\text{ALLOW}\ intf.content\ ]$
where *content* can be:

$RegMsg \equiv reg[\ \text{ALLOW}\ CID.\langle UID_x\rangle\ ]$

$SubsMsg \equiv subs[\ \text{ALLOW}\ CID.\langle UID_x, SID_i, pymt\rangle\ ]$

$RequestMsg \equiv request[\ \text{IN}\ UID_x \mid$
$o_i[\ \text{ALLOW}\ op.\langle client_k, args_i\rangle\ ] \mid$
$\vdots$
$o_j[\ \text{ALLOW}\ op.\langle client_k, args_j\rangle\ ]\ ]$

$AddClMsg \equiv addCl[\ \text{IN}\ UID_x \mid$
$\text{ALLOW}\ CID.\langle client_k, path_l, UID_{(\text{on}\ client_l)}\rangle\ ]$

$AddChMsg \equiv addCh[\ \text{ALLOW}\ CID.\langle UID_x, cname\rangle\ ]$

$SubsToChMsg \equiv subsToCh[$
$\text{ALLOW}\ CID.\langle UID_x, cname, uname, client_k, pymt\rangle\ ]$

$ShareInfoMsg \equiv share[\ \text{IN}\ CHID_i \mid$
$\text{ALLOW}\ CID.\langle sndr, rcvr, info\rangle\ ]$

$ShareSvcMsg \equiv share[\ \text{IN}\ CHID_i \mid \text{ALLOW}\ CID.$
$\langle sndr, rcvr, info, o_i, argsP, argsF\rangle\ ]$

In the definitions above: the ambient *msg* is the frame of a message; the term IN *cloud* denotes the address to where the message is sent; the term ALLOW $intf$ allows a (server) mechanism on the target side which uses the public protocol $intf$ to capture the message; and the *content* can be various kind of message types. The term ALLOW *CID* denotes that the messages are sent to a service of a particular cloud which identifies itself with the non-public protocol/credential *CID* (stands for *cloud identifier*).

The first three kinds of messages were introduced in the original model. In a *RegistrationMsg* the user x provides her identifier UID_x that she is going to use in the cloud. By a *SubscriptionMsg* a user subscribes to a cloud service identified by SID_i; the information represented by *pymt* proves that the given user has paid for the service properly.

Again, cloud services provide their functionalities for their environment (users or other services) via actions called service operations in our model. In a *RequestMsg* a user who has subscribed to some services before can request the cloud to perform some service operations belonging to some of these services. o_i and o_j are the unique names of these service operations and denote service operation requests; $client_k$ is the identifer of a target location (usually a client device) to where the output of a given operation should be sent by the cloud; and $args_i$ and $args_j$ are the arguments of the corresponding requested service operations. Furthermore, the term IN UID_x represents the address of the target user area within the *Cloud* and ALLOW *op* denotes that the request will be processed by a service plot, which expects service operation requests (and which interacts with the request via the public protocol *op*).

The rest of the message types is new in the model. With *AddClMsg* a user can register a new possible target (client) device or location for the outcomes of the requests initiated by her. Such a message should contain the chosen identifier $client_k$ of the new device, the address $path_l$ of the device and the user identifier $UID_{(\text{on}\ client_l)}$ used on the given target device.

By *AddChMsg* users can open new channels, by *SubsToChMsg* users can subscribe to channels and by *ShareInfoMsg* and *ShareSvcMsg* users can share information as well as service operations with some other users registered in the same channel. For the detailed description of the arguments lists of these last four messages, see Section 5.3.1, Section 5.3.2 and Section 5.3.3.

5.2 The Cloud Service Architecture

The basic structure of the defined cloud model, which is based on the simplified *Infrastructure as a Service (IaaS)* specification given in (Bósa, 2012b), is the following:

$Cloud \equiv (\nu\ fw, q, rescr_1, \ldots rescr_m)cloud[$
$interface \mid$
$fw\ [\ rescr_1[\ service_1\ ] \mid \ldots \mid rescr_l[\ service_1\ ] \mid$
$rescr_{l+1}[\ service_2\ ] \mid \ldots \mid rescr_m[\ service_n\ ] \mid$
$q[\ !\text{OPEN}\ msg \mid$
$BasicCloud functions \mid CTCI functions \mid$
$UID_x[\ userIntf\ ] \mid \ldots \mid UID_y[\ userIntf\ ] \mid$
$UID_v^{owner}[\ ownerIntf\ ] \mid \ldots \mid UID_w^{owner}[\ ownerIntf\ ]$
$]\ ]\ ]$
where
$interface \equiv \text{SERVER}_{intf}^{n}\ msg.\text{IN}\ fw.\text{IN}\ q.n\ \text{BE}\ msg$

In the cloud definition above, the names of the ambients fw, q and $rescr_1, \ldots rescr_m$ are bound by

[4]The ambient called *root* is a special ambient which is required for the ASM definition of ambient calculus, see (Börger et al., 2012) and (Bósa, 2012b).

name restriction. The consequence of this is that the names of these ambients are known only within the cloud service system, and therefore the contents of their body are completely hidden and not accessible at all from outside of the cloud. So each of them can be regarded as an abstraction of a firewall protection.

The ambient expression represented by $interface$ "pulls in" into the area protected by the ambients fw and q any ambient construct which is encompassed by the message frame *msg*. The purpose of the restricted ambients fw and q is to prevent any malicious content which may cut loose in the body of q after a message frame (*msg*) has been broken (by OPEN *msg*) to leave the cloud together with some sensitive information. For more details we refer to (Bósa, 2013).

The restricted ambients $resrc_1,\ldots,resrc_m$, represent computational resources of the cloud. Within each cloud resource some service instances can be deployed. A service may have several deployed instances in a cloud (see instances of $service_1$ in $resrc_1,\ldots,resrc_l$ above).

Every user area is represented by an ambient whose name corresponds to the corresponding user identifier UID_i. Furthermore, the user areas extended with service owner role are denoted by UID_i^{owner}. The terms denoted by *BasicCloud functions* are responsible for cloud user registration and service subscription. Finally the terms denoted by *CTCI functions* encode the client-to-client interaction.

It is beyond the scope of this paper to describe all parts of this model in details (e.g.: the structure of service instances $service_i$, functions of a service owner area *ownerIntf*, the service plots and the ASM agents in *BasicCloud functions*). For the specification of these components, we refer to (Bósa, 2013).

5.2.1 User Access Layers

A user access layer (or user area) may contain the following mechanisms: accepting user requests (!ALLOW *request*), accepting new plots (!ALLOW *newPlot*), accepting outputs of service operations (!ALLOW *returnValue*) and some service plots.

$userIntf \equiv$
 !ALLOW *request* | !ALLOW *newPlot* | *clientRegServer* |
 !ALLOW *returnValue* | *sortingOutput* |
 $\text{PLOT}_{SID_i} | \ldots | \text{PLOT}_{SID_j}$ |
 $client_1[\ posting_{client_1}\ ] | \ldots | client_k[\ posting_{client_k}\ ]$
where
 $sortingOutput \equiv !(o, client, a).output[$
  IN *client*.ALLOW *CID* | $\langle o, client, a\rangle$]]
 $clientRegServer \equiv \text{SERVER}^n_{CID}\ addCl.$
  $(client, path, UID).(n$ BE *client* | $posting_{client}$)
 $posting_{client_i} \equiv \text{SERVER}^n_{CID}\ output.(o, client, a).$
  OUT $client_i.forwardTo_{client_i}.returnValue[$
   IN $UID_{(\text{on } client_i)}$ | $\langle o, client, a\rangle$]]
 $forwardTo_{client_i} \equiv n$ BE *outgoingMsg*.
  OUT $UID_x.leavingCloud.path_i$
 $leavingCloud \equiv$ OUT q.OUT fw.
  OUT *cloud.outgoingMsg* BE *msg*

This paper extends the user areas with some new functionalities. *clientRegServer* is applied to process every *AddClMsg* sent by the corresponding user. It creates new communication endpoint for target (client) devices. Each such an endpoint is encoded by an ambient whose name $client_i$ corresponds to the given identifier provided in a message *AddClMsg*. By these endpoints outputs of service operations can immediately be directed to registered (client) devices after they are available. Of course, if no target device or a non-registered one is given in a *RequestMsg*, the outcome will be stored in the area of the user.

Every service operation output, which is always delivered within the body of an ambient called *returnValue*, consists of three parts: the name of the performed service operation, the identifier of a target location to where the output should be sent back and the outcome of the performed service operation itself.

sortingOutput distributes every service operation output among the communication endpoints in an ambient called *output*, see Section 6. The mechanism $posting_{client_i}$, which resides in each such a communication endpoint, is responsible to wrap each output of service operations which reaches the corresponding endpoint again into an ambient *returnValue* and to forward it to the specified user $UID_{(\text{on } client_i)}$ on the corresponding device $client_i$.

5.3 Client-to-Client Interaction

Again, the client-to-client interaction in our model is based on the constructs called *channels*. These are represented by ambients with unique names denoted by $CHID_i$ which contain some mechanisms whose purpose is to share some information and service operations among some subscribed users, see below:

$CTCI functions \equiv$
 $CHID_1[\ channelInt f\ ] | \ldots | CHID_l[\ channelInt f\ ] |$
 $\text{SERVER}^n_{CID}\ addCh.(UID, cname).$
  CHMGR($n, UID_x, cname$) |
 $\text{SERVER}^n_{CID}\ subsToCh.(UID, cname, uname, client, pymt).$
  CHSUBSMGR($n, UID, cname, uname, client, pymt$)
where
 $channelInt f \equiv$
  $\text{SERVER}^n_{CID}\ share.($
  $(sndr, rcvr, info).\langle$sndr,rcvr,$info$,**undef**,**undef**,**undef**$\rangle$ |
  $(sndr, rcvr, info, o, argsP, argsF).$
   SHARINGMGR($n, sndr, rcvr, info, o, argsP, argsF$))

Every cloud user can create and own some channels by sending the message *AddChMsg* to the cloud, where an instance of the ASM agent CHMGR, which is equipped with a server mechanism, processes such a request and creates a new ambient with unique names for the requested channel, see Section 5.3.1.

If a user would like to subscribe to a channel she should send the message *SubsToChMsg* to the cloud. The server construct belongs to the ASM agent CHSUBSMGR is responsible for processing these messages, see Section 5.3.2. In the subscription process the owner of the channel can decide about the rights which can be assigned to a subscribed user. According to the presented high-level model, the employed access rights are encoded by the following static nullary functions: *listening* is a default basic right, because everybody who joins to a channel can receive shared contents; *sending* authorizes a user to send something to only one user at a time; and *broadcasting* permits a user to distribute contents to all member of the channel at once.

Both *ShareInfoMsg* and *ShareSvcMsg* are processed by the same server which belongs to the ASM agent SHARINGMGR and which is located in the body of each ambient $CHID_i$, see Section 5.3.3. In the case of *ShareInfoMsg* the server first supplements the arguments list of the message with three additional **undef** values, such that it will have the same number of arguments as *ShareSvcMsg* has. Then an instance of the ASM agent SHARINGMGR can process the *ShareInfoMsg* similarly to *ShareSvcMsg* (the first three arguments are the same for both messages).

5.3.1 Establishing a New Channel

CHMGR is a parameterized ASM agent, which expects *UID* of the cloud user who is going to create a new channel and *cname* which is the name of this channel as arguments. The additional argument *n* is the unique name of an ambient which was provided by the surrounding server construct and in which the current *AddChMsg* is processed by an instance of this agent (such an argument is also applied in the case of the other ASM agents below).

First the agent checks whether the given *UID* has already been registered on the cloud and whether the given name *cname* has not been used as a name of an existing channel yet (the unary function *ownerOfCh* returns the value **undef** if there is no assigned owner to this name). If it is the case, the agent generates a new and unique identifier denoted by *CHID* for the new channel with the usage of the function **new** which provides a unique and completely fresh element for the given set each time when it is applied. The abstract ASM macro STORECHANNEL inserts into an abstract database a new entry with all the details of the new channel which are the channel identifier, the channel name and the identifier of the owner.

Then it calls the abstract derived function *createChannel*, which creates an ambient called *CHID* with the terms denoted by *channelIntf* in its body which encode the functions of the new channel. By the abstract tree manipulation operation called NEWAMBIENTCONSTRUCT[5] introduced in (Bósa, 2012b), this generated ambient construct is placed into the ambient tree hierarchy as sibling of the agent.

```
CHMGR(n, UID, cname) ≡
 ctr_state : {InitialState, EndState}
 initially ctr_state := InitialState
 if ctr_state = InitialState then
  ctr_state := EndState
  if UID ∈ userIds then
   if ownerOfCh(cname) = undef then
    let CHID = new(channelIds) in
     STORECHANNEL(CHID, cname, UID)
     let CHConstruct = createChannel( CHID ) in
      NEWAMBIENTCONSTRUCT(CHConstruct)
where
 CHConstruct ≡ CHID[ OUT n | channelIntf ]
```

Although a channel is always created as a sibling of the current instance of CHMGR, but as a first step it leaves the ambient *n* which was provided by the surrounding server construct and in which the message was processed (see the underlined moving action in *CHConstruct* above). After that it is prepared to serve as a channel for client-to-client interaction (it is supposed that the name *cname* of every channel is somehow announced among the potential users).

5.3.2 Subscribing to a Channel

CHSUBSMGR is a parameterized ASM agent, which expects the following as arguments: *UID* of the user who is going to subscribe to the channel, *cname* which is the name of the channel, *uname* is the name that the user is going to use within the channel, *client* which is the identifier of a registered client device to where the shared content will be forwarded and *pymt* which is some payment details if it is required. A user can register to a channel with different names and various client devices in order to connect these devices via the cloud.

First the agent checks whether the given *UID* and *cname* have already been registered on the cloud and whether the given *uname* has not been used as a name of a member of the channel yet. If it is the case,

[5]This is the only way how an ASM agent can make changes in the ambient tree hierarchy contained by dynamic derived function *curAmbProc* (Bósa, 2012b).

the agent informs the owner of the channel about the new subscription by applying the abstract ASM macro CONFIRMRIGHTS, who responses with a set of access rights to the channel that she composed based on the information given in the subscription.

CHSUBSMGR(*n,UID,cname,uname,client,pymt*) ≡
ctr_state : {*InitialState, EndState*}
initially *ctr_state* := *InitialState*

if *ctr_state* = *InitialState* **then**
ctr_state := *EndState*
if *UID* ∈ *userIds* **then**
if *ownerOfCh*(*cname*) ≠ **undef then**
if *uname* ∉ *members*(*cname*) **then**
let *owner* = *ownerOfCh*(*cname*) **in**
let *rights* =
CONFIRMRIGHTS(*owner,uname,cname,pymt*) **in**
if *rights* ≠ ∅ **then**
STOREMEMBER(*UID,uname,client,cname,rights*)
if *rights* ≠ {*listening*} THEN
let *CHID* = *idOfCh*(cname) **in**
NEWAMBIENTCONSTRUCT(*returnValue*[
OUT *n*.IN *UID*.⟨*cname, client*, IN *CHID*⟩])

If the subscription has been accepted by the owner and besides *listening* some other rights are granted to the new user, an ambient construct is created and sent as a message *returnValue* to the user by NEWAMBIENTCONSTRUCT. This message contains the capability IN *CHID* by which the new user can send messages called *ShareInfoMsg* and *ShareSvcMsg* into the ambient *CHID* which represents the corresponding channel (the owner of a channel also has to subscribe in order to receive this information and to be able to distribute content via the channel).

5.3.3 Sharing Information via a Channel

Every server construct in which the agent SHARINGMGR is embedded is always located in an ambient which represents a particular channel and whose name corresponds to the identifier of the channel. In order to be able to perform its task, it is required that each instance of SHARINGMGR knows by some static nullary function called *myChId* the name of the ambient in which it is executed.

SHARINGMGR is a parameterized ASM agent, which expects the following arguments: *sndr* is the registered name of the sender, *rcvr* is either the registered name of a receiver or an asterisk "*", *info* is either the content of *ShareInfoMsg* or the description of a shared service operation in *ShareSvcMsg*. The last three arguments are not used in the case of the message *ShareInfoMsg* and the value **undef** is assigned to each of them by the surrounding server construct. In the message *ShareSvcMsg o* denotes the unique identifier of the service operation that *sndr* is going to share, *argsP* denotes the arguments of *o* that *rcvr* can freely modify if she calls the operation and *argsF* denotes those part of the argument list of *o*, whose value is fixed by *sndr*.

The agent first generates a new and unique operation identifier for the service operation *o* in the control state *InitialState*. This new identifier which is stored in the nullary location function *shOp* will be announced to the channel member(s) specified in *rcvr*. In the control state *SharingState* the agent checks whether the *sndr* is a registered member of the channel by calling the function *members*(*cname*). Then if the given value of *rcvr* is equal to "*" the agent broadcasts the content of the current message to all members of the channel, see code branch bordered by the first rectangular frame below. Otherwise if the value of *rcvr* corresponds to the name of a particular member of the channel, the agent sends the content of the current message only to her, see the code branch bordered by the second rectangular frame below.

SHARINGMGR(*n, sndr, rcvr, info, o, argsP, argsF*) ≡
ctr_state : {*InitialState, SharingState, EndState*}
initially *ctr_state* := *InitialState*

if *ctr_state* = *InitialState* **then**
ctr_state := *SharingState*
if *o* ≠ **undef then** //svc. sharing
let *newOpId* = **new**(*sharedOpIds*) **in**
shOp = *newOpId*
else *shOp* = **undef** //msg. sharing

if *ctr_state* = *SharingState* **then**
ctr_state := *EndState*
let *cname* = *getChannelName*(*myChId*) **in**
if *sndr* ∈ *members*(*cname*) **then**
let *rights* = *getRights*(*cname, sndr*) **in**

if *rcvr* = "*" **then** //broadcasting a msg.
if *boradcasting* ∈ *rights* **then**
forall *M* ∈ *members*(*cname*) **do**
let *UID* = getId(*M*), *client* = getAddress(*M*) **in**
if *shOp* = **undef then**
NEWAMBIENTCONSTRUCT($sharedM_{content_1}$)
else
NEWAMBIENTCONSTRUCT($sharedM_{content_2}$)
let UID_{sndr} = getId(*sndr*) **in**
NEWAMBIENTCONSTRUCT(*sharedPlot*)

else //sending a msg.
if *sending* ∈ *rights* **and** *rcvr* ∈ *members*(*cname*)
then
let *UID*= getId(*rcvr*), *client*= getAddress(*rcvr*) **in**
if *shOp* = **undef then**
NEWAMBIENTCONSTRUCT($sharedM_{content_1}$)
else
NEWAMBIENTCONSTRUCT($sharedM_{content_2}$)
let UID_{sndr} = getId(*sndr*) **in**
NEWAMBIENTCONSTRUCT(*sharedPlot*)

where
$sharedM_{content_i} \equiv$ *returnValue*[
OUT *n*.OUT *myChId*.IN *UID*.⟨*cname, client*, $content_i$⟩]

$content_1 \equiv$ {"sender:" *sndr*, "content:" *info*}
$content_2 \equiv$ {"sender:" *sndr*, "operation:" *shOp*,
"arguments:" *argsP*, "description:" *info*}
$sharedPlot \equiv$
newPlot[OUT *n*.OUT *myChId*.IN *UID* | PLOT_{shOp}]
$\text{PLOT}_{shOp} \equiv \text{SERVER}^{s}_{op}\ shOp.trigger_o$
$trigger_o \equiv (\nu\ tmp)$
(*client*, *argsP*).(OUT *UID*.IN UID_{sndr}.*s* BE *request* |
o[ALLOW $op.\langle tmp, (argsP \setminus argsF) + argsF\rangle$] |
tmp[ALLOW *output* | *CID*[(*o*, *c*, *a*).OUT UID_{sndr}.
IN *UID.tmp* BE *returnValue*.$\langle shOp, client, a\rangle$]])

Apart from the number of users to whom the information is sent the both code branches mentioned above define the same actions. Accordingly at the end of the processing of *ShareInfoMsg* the agent sends to the member(s) specified in *rcvr* the message $sharedM_{content_1}$, which contains the sender *sndr* and the shared information *info*.

At the end of the processing of *ShareSvcMsg* two ambient constructs are created by NEWAMBIENTCONSTRUCT. The first one is the message $sharedM_{content_2}$ and it is sent to the member(s) specified in *rcvr*. It contains the sender *sndr*, the new operation identifier *shOp*, the list of public arguments *argsP* and the informal description of the shared operation denoted by *info*.

The second ambient construct is the plot PLOT_{shOp} enclosed by the ambient *newPlot* and equipped with some additional ambient actions (see the underlined capabilities in the definition of *sharedPlot*) which move the entire construct into the user area of the channel member(s) specified in *rcvr*, where the plot will be accepted by the term !ALLOW *newPlot*.

The execution of the shared service operation *shOp* can be requested in a usual *RequestMsg* as normal service operations. The PLOT_{shOp} is a plot, which can accept service operation requests for *shOp* several times. It is special plot, because instead of triggering the execution of *shOp* as in the case of a normal operation a normal plot does, see (Bósa, 2013), it converts the original request to another request for operation *o* by applying the term $trigger_o$. This means that it substitutes the operation identifier *o* for *shOp*, it completes its arguments list with *argsF* and it forwards the request for *o* to the user area of the user *sndr* who actually has access to trigger the execution of the operation *o*, see Section 6.

To the new request the name restricted ambient *tmp* is attached, whose purpose is similar to the communication endpoints of registered clients. Namely, it is placed into the user area of *sndr* temporary and it is responsible to forward the outcome of this particular request from the user area of *sndr* to the user area of the user who initiated the request, see Section 6.

6 PROCESSING OF SHARED SERVICES

In the following we present a draft of a reduction how a particular request for a shared operation is processed in our model. We assume that the user UID_{sndr} has already shared the service operation o_i^{shared} with another user called UID_x. *shOp* is used as the shared identifer of o_i^{shared}. According to our example the user UID_x sends an operation request for *shOp* and expects to receive the outcome at the client device $clnt_1$.

6.1 Triggering of Shared Operations

After a message *RequestMsg* has arrived at the restricted cloud area protected by ambient *fw* and *q* and the message frame *msg* has been dissolved, the ambient *request* enters into the user area which is identified by UID_x (see the numbered ambient actions in the depicted reduction outlines below):

$$
\begin{aligned}
&Cloud \mid RequestMsg \\
&= (\nu\ fw, q, rescr_1, \ldots rescr_m)cloud[\\
&\quad \overbrace{\text{SERVER}^{n}_{intf}\ msg}^{(2nd)}.\overbrace{\text{IN}\ fw.\text{IN}\ q}^{(3rd)}.\overbrace{n\ \text{BE}\ msg}^{(4th)} \mid \\
&\quad fw\,[\ rescr_1[\ service_1\ ] \mid \ldots \mid rescr_i[\ service_1\ ] \mid \\
&\quad\ rescr_{i+1}[\ service_2\ ] \mid \ldots \mid rescr_m[\ service_n\ ] \mid \\
&\quad\ q[\ \overbrace{!\ \text{OPEN}\ msg}^{(5th)} \mid \\
&\quad\quad BasicCloud\,functions \mid CTC\,functions \mid \\
&\quad\quad UID_x[\ userIntf \mid \text{PLOT}_{shOp}\ ] \mid \\
&\quad\quad UID_{sndr}[\ userIntf \mid \text{PLOT}_{(o_i^{shared}\ldots)}\ ] \mid \\
&\quad\quad UID_v^{owner}[\ ownerIntf\ ] \\
&\quad]\,]\,] \mid \\
&\quad msg[\ \overbrace{\text{IN}\ cloud}^{(1st)}.\overbrace{\text{ALLOW}\ intf}^{(2nd)}.request[\ \overbrace{\text{IN}\ UID_x}^{(6th)} \mid \\
&\quad\quad shOp[\ \text{ALLOW}\ op.\langle clnt_1, argsP\rangle\ ]\,]\,]
\end{aligned}
$$

After the ambient *request* has arrived at the ambient UID_x, it is dissolved and PLOT_{shOp} captures and converts the request to another request for o_i^{shared}:

$$
\begin{aligned}
\longrightarrow^{*}\quad &(\nu\ fw, q, rescr_1, \ldots rescr_m)cloud[\\
&\quad \vdots \\
&\quad UID_x[\ \overbrace{!\text{ALLOW}\ request}^{(7th)} \mid \\
&\quad\quad !\text{ALLOW}\ newPlot \mid clientRegServer \mid \\
&\quad\quad !\text{ALLOW}\ returnValue \mid sortingOutput \mid \\
&\quad\quad clnt_1[\ posting_{clnt_1}\ ] \mid \\
&\quad\quad \overbrace{\text{SERVER}^{s}_{op}\ shOp}^{(8th)}.\overbrace{trigger_{o_i^{shared}}}^{(9th)} \mid \\
&\quad\quad request[\\
&\quad\quad\quad shOp[\ \overbrace{\text{ALLOW}\ op}^{(8th)}.\langle clnt_1, argsP\rangle\ ]\,] \\
&\quad]\,]
\end{aligned}
$$

In the 8th step the bound names s and tmp, which located under a replication sign in the server construct, get new unique names, respectively (denoted as s^{uniq} and tmp^{uniq} in the rest). In the 9th step, the path formation on $(client, argsP).\text{OUT}\ UID.\text{IN}\ UID_{sndr}$ is applied in $trigger_{o_i^{shared}}$, which captures the given arguments of the original request and forwards the ambient s^{uniq} with the request for o_i^{shared} in its body to the user area of UID_{sndr}.

The captured arguments will be changed in the new request. Namely, the original target location $clnt_1$, to where the output of the request should be sent, is replaced with tmp^{uniq}. Furthermore, the *argsP* which is given as set of identifiers to which some values are assigned is complemented with *argsF* which is specified in the same way. In the term $(argsP \setminus argsF) + argsF$ below, the employment of the set difference of *argsP* and *argsF* ensures that the user UID_x cannot overwrite any fixed argument of o_i^{shared} (even if she somehow knows the syntax of o_i^{shared}).

After the ambient s^{uniq} has arrived at the ambient UID_{sndr} it is renamed to *request* in the 10th step and then it is processed as a normal request for o_i^{shared}. This means that the abstract plot $\text{PLOT}_{(o_i^{shared}\ldots)}$ into which the current service operation request fits, triggers the execution of o_i^{shared}, for more details see (Bósa, 2013). The only difference compared to the processing of a normal request is that the ambient tmp^{uniq} is left behind in UID_{sndr}:

$\longrightarrow^*$ $(\nu\ fw, q, rescr_1, \ldots rescr_m)cloud[$

$\vdots$

$UID_{sndr}[\ \overbrace{!\text{ALLOW}\ request}^{(11th)} \mid$
$!\text{ALLOW}\ newPlot \mid clientRegServer \mid$
$!\text{ALLOW}\ returnValue \mid sortingOutput \mid$
$clnt_j[\ posting_{clnt_j}\ ] \mid \overbrace{\text{PLOT}_{(o_i^{shared}\ldots)}}^{(12th)} \mid$
$s^{uniq}[\ \overbrace{s^{uniq}\ \text{BE}\ request}^{(10th)} \mid o_i^{shared}[\ \overbrace{\text{ALLOW}\ op.}^{(12th)}$
$\langle tmp^{uniq}, (argsP \setminus argsF) + argsF \rangle\ ] \mid$
$tmp^{uniq}[\ \text{ALLOW}\ output \mid CID[$
$(o, c, a).\text{OUT}\ UID_{sndr}.\text{IN}\ UID_x.$
$tmp^{uniq}\ \text{BE}\ returnValue.\langle shOp, clnt_1, a \rangle\ ]\ ]\ ]$
$]\ ]$

6.2 Redirecting the Outcome

The output for a service operation request is always a triple which is always delivered within the body of an ambient called *returnValue* and which consists of three parts: the name of the performed service operation, the identifier of a target location to where the output should be sent back and the outcome of the performed service operation itself.

After the request has been processed by an instance of the corresponding service located on one of the cloud resources and the output of o_i^{shared} has arrived at UID_{sndr}, the mechanism denoted as *sortingOutput* forwards the output in an ambient called *output* to the given communication endpoint of the specified target location, which is tmp^{uniq} in the current case, see the 14th, 15th and 16th steps:

$\longrightarrow^*$ $(\nu\ fw, q, rescr_1, \ldots rescr_m)cloud[$

$\vdots$

$UID_{sndr}[\ !\text{ALLOW}\ request \mid$
$!\text{ALLOW}\ newPlot \mid clientRegServer \mid$
$\overbrace{\underline{!\text{ALLOW}\ returnValue}}^{(13th)} \mid \overbrace{\underline{!(o, client, a).output}}^{(14th)}[$
$\overbrace{\underline{\text{IN}\ client}}^{(15th)}.\overbrace{\underline{\text{ALLOW}\ CID}}^{(17th)} \mid \langle o, client, a \rangle\ ]\ ] \mid$
$clnt_j[\ posting_{clnt_j}\ ] \mid \text{PLOT}_{(o_i^{shared}\ldots)} \mid$
$tmp^{uniq}[\ \overbrace{\text{ALLOW}\ output}^{(16th)} \mid CID[$
$\overbrace{(o, c, a)}^{(18th)}.\overbrace{\underline{\text{OUT}\ UID_{sndr}.\text{IN}\ UID_x.}}^{(19th)}$
$tmp^{uniq}\ \text{BE}\ returnValue.\langle shOp, clnt_1, a \rangle\ ]\ ] \mid$
$returnValue[\ \langle o_i^{shared}, tmp^{uniq}, outcome \rangle\ ]$
$]\ ]$

The ambient terms in tmp^{uniq} capture the output triple and set back the target location of the output to $clnt_1$ and the performed operation to *shOp*; then tmp^{uniq} is moved with this new output triple in its body to the user area of UID_x, see the 18th and 19th steps.

After the ambient tmp^{uniq} has arrived at the ambient UID_x, it will be renamed to *returnValue*, see the 20th step. Then the body of the ambient *returnValue* will be processed similarly in the 21st, 22nd and 23rd steps as before, but this time the output triple will be redirected to the communication endpoint of $clnt_1$ by the *sortingOutput* mechanism, see below:

$\longrightarrow^*$ $(\nu\ fw, q, rescr_1, \ldots rescr_m)cloud[$

$\vdots$

$UID_x[\ !\text{ALLOW}\ request \mid$
$!\text{ALLOW}\ newPlot \mid clientRegServer \mid$
$\overbrace{\underline{!\text{ALLOW}\ returnValue}}^{(21st)} \mid \overbrace{\underline{!(o, client, a).output}}^{(22nd)}[$
$\overbrace{\underline{\text{IN}\ client}}^{(23rd)}.\overbrace{\underline{\text{ALLOW}\ CID}}^{(24th)} \mid \langle o, client, a \rangle\ ]\ ] \mid$
$clnt_1[\ \overbrace{posting_{clnt_1}}^{(24th)}\ ] \mid$
$\text{SERVER}_{op}^{s}\ shOp.trigger_{o_i^{shared}} \mid$

$$tmp^{uniq}[\overbrace{tmp^{uniq}\ \text{BE}\ returnValue.\ \langle shOp, clnt_1, outcome\rangle}^{(20th)}\ ]\ ]$$

Then in the 24th step the functionality denoted by $posting_{clnt_1}$ send the output triple to the client device $clnt_1$, where only the user who initiated the request can access to it (via the user identifier $UID_{(\text{on } clnt_1)}$).

7 CONCLUSIONS

In this paper we extended our formerly given cloud model with the high-level formal definitions of some client-to-client interaction functions, by which not only information, but cloud service functions can be also shared among the cloud users. Our approach is general enough to manage situation in which a user who has access to a shared service operation to share it again with some other users via a channel (who in turn may share it again, etc.).

Furthermore, if we apply the scenario proposed in Section 2 and depicted on Figure 1b, according to which we shift (among others) the client-to-client functionality to client side and wrap into a middleware, then no traces of the user activities belonging to the shared services will be left on the cloud, since all the service operations which are shared via a channel are used on behalf of its initial distributor. This consideration can lead one step into the direction of anonym usage of cloud services. The consequence of this that if a cloud user who has contracts with some service providers completely or partially shares some services via a channel, then she should be aware of the fact that all generated costs caused by the usage of these shared services will be allocated to her.

ACKNOWLEDGEMENTS

This research has been supported by the Christian Doppler Society.

REFERENCES

Börger, E., Cisternino, A., and Gervasi, V. (2012). Ambient Abstract State Machines with Applications. *J.CSS (Special Issue in honor of Amir Pnueli)*, 78(3):939–959.

Börger, E. and Stark, R. F. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Bósa, K. (2012a). A Formal Model of a Cloud Service Architecture in Terms of Ambient ASM. Technical report, Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC), Johannes Kepler University Linz, Austria.

Bósa, K. (2012b). Formal Modeling of Mobile Computing Systems Based on Ambient Abstract State Machines. *Semantics in Data and Knowledge Bases*, 7693 of LNCS:18–49.

Bósa, K. (2013). An Ambient ASM Model for Cloud Architectures. *Formal Aspects of Computing*. Submitted.

Boudol, G., Castellani, I., Hennessy, M., and Kiehn, A. (1994). A Theory of Processes with Localities. *Formal Aspects of Computing*, 6:165–200. 10.1007/BF01221098.

Cardelli, L. (1999). Mobility and Security. In Bauer, F. L. and Steinbrüggen, R., editors, *Foundations of Secure Computation Proc. NATO Advanced Study Institute*, pages 3–37. IOS Press. Lecture Notes for Marktoberdorf Summer School 1999 (A summary of several Ambient Calculus papers).

Cardelli, L. and Gordon, A. D. (2000). Mobile Ambients. *Theor. Comput. Sci.*, 240(1):177–213.

Kozen, D. (1997). Kleene Algebra with Tests. *Transactions on Programming Languages and Systems*, 19(3):427–443.

Ma, H., Schewe, K.-D., Thalheim, B., and Wang, Q. (2008). Abstract State Services. In *Object-Oriented and Entity-Relationship Modelling/International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 406–415.

Ma, H., Schewe, K.-D., Thalheim, B., and Wang, Q. (2009). A Theory of Data-Intensive Software Services. *Service Oriented Computing and Applications*, 3(4):263–283.

Ma, H., Schewe, K.-D., Thalheim, B., and Wang, Q. (2012). A Formal Model for the Interoperability of Service Clouds. *Service-Oriented Computing and Applications*. To appear.

Milner, R., Parrow, J., and Walker, D. (1992). A Calculus of Mobile Processes, Parts I. and II. *Information and Computation*, 100(1):1–77.

Schewe, K.-D. and Thalheim, B. (2007). Personalisation of Web Information Systems - A Term Rewriting Approach. *Data & Knowledge Engineering*, 62(1):101 – 117.

Tanaka, Y. (2003). *Meme Media and Meme Market Architectures: Knowledge Media for Editing, Distributing, and Managing Intellectual Resources*. Wiley.