A Formal Model of a Cloud Service Architecture in Terms of Ambient ASM

Károly Bósa

August 21, 2012

Abstract

This document presents a formal model of a *cloud service system* in terms of Ambient Abstract State Machine in two abstraction layer:

- while the spatial locations, mobility and some security considerations (accessibility) are described by a hierarchy of some ambient constructs;
- the algorithmic functionalities are defined by abstract state machine agents (which reside on various locations in the ambient hierarchy).

The proposed model applies a novel approach for client-cloud interaction by which service owners, who may be different from the cloud provider, are able to fully control the usages of their services in the case of each subscription, respectively.

1 Introduction

One of our goals of our research project is to compose formal models of some new kinds of distributed service oriented and cloud architectures. For this, we searched for a software engineering method by which both the algorithms of the concurrent system components as well as their spatial locality and their mobility can be formally described.

Nowadays one of the most outstanding methods for formal modeling interaction of concurrent components of (mobile) network applications in heterogeneous and dynamically changing environments is a calculus of mobile agents called *ambient calculus* [5, 7]. This concept is simple, succinct and sufficient enough to describe efficiently locality as well as phenomena related to physical mobility (mobile hardware) and virtual mobility (mobile software). Additionally, besides mobility it supports the reasoning of some degree of security as well (e.g.: some combinations of operations can be interpreted as crossing firewalls or decoding of ciphertexts, etc.). But one of its main drawbacks is that it is not capable to describe the algorithmic specification of the executable agents which appear in its ambient constructs.

In [3] the definition of the ambient concept (and of the three basic actions of ambient calculus for moving ambients as well) is given in terms of the mathematically well-funded software engineering method called *abstract state machine (ASM)* [8, 1, 2]. The definition of *ambient ASM* is based upon the semantics of traditional ASM method without any changes. Since one of the main goals of [3]

is to reveal the inherent opportunities of the new ambient concept introduced into ASM, the presented ASM definitions for moving ambients are unfortunately incomplete.

In [4], we extended and completed these ASM rules given in [3] such that a new method is created in terms of ASM rules which fully capture the calculus of mobile agents and by which one is able to describe formal models of concurrent systems including mobile components in two different abstraction layers: the long range interaction and movements of system components are specified in terms of moving ambients; while the algorithm of agents are defined in terms of ASM syntax.

The main purpose of this paper is to give a formal model of a cloud service system in the two abstraction layers mentioned above. The described model can be regarded as a basic model, which is going to be extended significantly with the next project phases (e.g.: with a sophisticated identity management and a role based access control mechanism, etc.). Besides the usual basic functionalities like (like user registration, subscription to various services and providing access to cloud resources) our current cloud specification includes the following two features:

- the cloud is able to adapt its services to heterogeneous client devices in order to contribute to fundamental infrastructure for clients in cloud computing supporting provisioning, management, and update functionality; and
- the cloud architecture provides interaction between users and service owners (different from the cloud provider), such that each service owner is able to control the service subscriptions and service usages.

The rest of the paper is organized as follows. First Section 2 shows a general high-level overview on the new formal model of a cloud service system presented in this paper. Section 3 gives a short summary on ambient calculus and ambient ASM, which our model is based on. Section 4 introduces some notations and definitions applied in the latter sections. Section 5 discusses the entire cloud system specification including a simplified client specification, the main user actions and the cloud service infrastructure.

The paper comprises two appendices as well. Appendix A gives the formal model of some typical service operation plots by which a service provide can determine and specify how a particular user can use some of her service(s). Finally Appendix B demonstrates the interactions among various system components of the discussed cloud service system model by an example in which the processing of a particular user request is presented.

2 An Overview on our Model

In this section, we give a general and informal overview on the cloud service architecture described by our model in terms of ambient ASM in the latter sections. The basic structure of the presented model is based on the simplified *Infrastructure as a Service (IaaS)* specification given in [4].

Since our model is only a basic model which is going to be extended with some sophisticated identity and role based access control managements in the

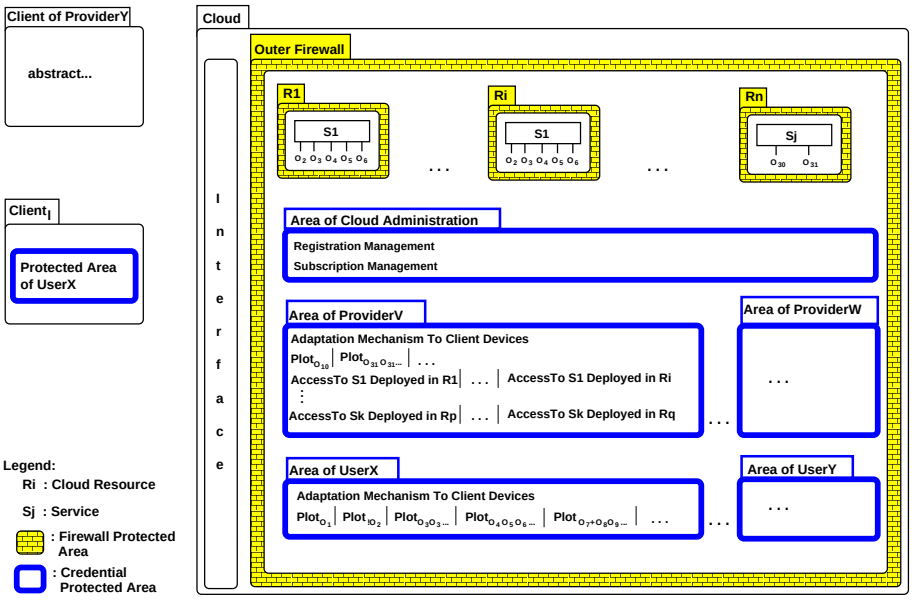

Figure 1: The High-Level Structure of our Cloud Model

near future, we currently apply only some relatively primitive abstract mechanisms for handling identities and access control.

In our model, we make a distinction between two kinds of cloud users. The *normal users* are registered to the cloud and they subscribe to and use some services provided by the cloud. The *service owners* (or providers) are normal users as well, but they rent some cloud *resources* to deploy some *instances* of their own *services*.

In Figure 1, the structure of the model is depicted. The entire cloud is encompassed with a firewall. This firewall is equipped with an interface mechanism which lets through all incoming messages, which are compatible with the interface. This is because the major purpose of the firewall is to prevent any information to leave the cloud without authorization (to keep sensitive information within the cloud). Therefore, each component within the outer firewall is protected either with a firewall (e.g.: cloud resources) or with credential (e.g.: cloud user areas) to prevent unauthorized accesses from outside.

In each resource some particular service instances can be deployed (one service may have several deployed instances in a cloud). For representing the services we adopt the abstract model of *Abstract State Services (AS^{2}s)* [9, 10], which is based on views on some hidden database layer that are equipped with service operations $o_1,\ldots,\ o_n$. These service operations are actually what is exported from the 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.

One of the credential protected areas in Figure 1 is the *Area of Cloud Administration* which contains some functionalities regarding user registration (see Section 2.2) and service subscription (see Section 2.3).

Another type of credential protected components in Figure 1 are *Area of UserX,. . . , Area of UserY*. For each cloud user such a restricted area is created after her registration. Later when the user subscribes to some cloud services, some *service plots* specified by the owners of the services will be placed into this user area. These plots determine which service operations of a particular service and how (e.g.: in which order) can be used by the user (see Section 2.1). This practically means if a service operation sent in a user request is permitted by a service plot, the request will be forwarded to the area of the owner of the service associated with the operation. We must emphasize that these plots belong to the service owner and they are not readable and writable by the user, but she is able to use them up via some cloud functionalities (like in Linux systems when a user is able to execute an application, e.g.: `passwd`, with the rights of the superuser to access/modify certain system settings, e.g.: `/etc/passwd`).

A user area may also comprises an adaptation mechanism, which is able to adapt cloud offers to various types of user limitations at the client side. This practically means that a requested service operation sent by the user is replaced with one or more other service operations according to some *adaptation rules* (and the sent arguments may also be converted). The particular emphasis is on different operating systems such as iOS, Windows and Android, their mobile analogs, and devices such as PCs, notebooks, tablets and mobile phones. In this way, the cloud becomes adaptive to the various client devices and softwares.

The credential protected *Area of ProviderV,. . . , Area of ProviderW* are for service owners. Since service owners are treated in the same way as users in our model, their restricted areas may contain service plots and adaptation mechanism as normal user areas. But areas of service owners are also extended with some additional functionalities. A user becomes service owner if it rents some cloud resources and deploy some service instances of one or more services.

This means a service owner has disposal of access rights to some cloud resources (with certain condition). These access information are stored in the protected area of the service owner. We must emphasize, a service owner is not able to read or write the stored access rights (they belong to the cloud provider), but she is able to use them up via some provided cloud functionalities. If an user request is forwarded by service plot to the area of a service owner, a scheduling mechanism (which may be controlled by the service owner) assigns an access right of a cloud resource to it. Then the user request is forwarded to a service instance which is deployed on the chosen cloud resource.

The model assumes that there exist two kinds of clients at least. A “normal” client is on which each user can execute a cloud client application (currently abstract) and where the users have a credential protected area, where the outputs received from the cloud services can be stored (such that only the corresponding user can access to them).

Every instance of the other client type always belongs to a service owner and it serves as a contact point for the service owner (e.g.: the *Client of ProviderY* depicted on Figure 1). Such a client may be used as normal cloud client, but it can also act as a sever for the cloud itself in some cases. Namely, the cloud may send a request regarding a subscription to a particular cloud service belonging to the service owner and this machine responses with a service plot which specifies how the service can be used by a particular user, see Section 2.3. This kind of client is abstract in the current model.

2.1 The Basic Types of Service Plots

As was mentioned earlier, *service plots* are provided by the owners of the services and they are placed into the user areas during the service subscription processes. These plots determine which service operations of a particular service and how (e.g.: in which order) can be used by the user.

A plot can be imagined as a structure which contains some slots for service operations, such that each operation can fit only into a particular slot (like in the case of keys and a key holes). Some of these slots are available, while some of them is not unlocked yet. If an operation request arrives and a plot provides a free slot which can be matched with the requested operation, the corresponding operation is triggered by the plot. If such an unlocked slot is not available for it, the operation is blocked and waits for a matching free slot. The possible plots and their combinations are summarized in the following:

o_i	one-off usage of a single operation
$o_i o_j o_k$	one-off usage of a sequence of operations
$o_i + o_j$	one-off usage of a choice among some operations
$!o_i$	multiple concurrent usage of a single operation/plot
$o_i \ldots$	multiple sequential usage of a single operation/plot
$o_i \mid o_j$	operations/plots enumerated next to each other separated by parallel composition may are performed in parallel

The simplest plot contains a single slot for one single operation and it allows a user to apply the operation once. A plot can consist of a sequence of service operation slots. In this case a slot is unlocked (and the corresponding operation can be triggered by a user) if and only if the output of the previous operation is available in the user area. The sequential plot can be used only at once. A plot can comprise choices among two or more operations. In this case a user can choose one from the enumerated operations, but after the chosen operation was triggered, the other operations cannot be applied anymore (the slots for these operations are locked forever).

Of course, without any chance of repetition of certain actions these plots would be not so useful. If a plot is preceded with an exclamation mark it can be triggered many times. If a plot is followed by three dots it can be performed several times as well, but a new instance of such a plot can be triggered if and only if the execution of the previous instance is finished (and all the outputs of the previous instance have arrived at the user area).

If a user subscribes to more than one service, she may have access to more than one plot. These plots are independent from each other and they can be applied in parallel (these plots are given in parallel compositions in the model). If a service owner provides more than one service for one user, she can have the choice to provide an independent plot for her each service or she combine some operations of various services into some common service plots.

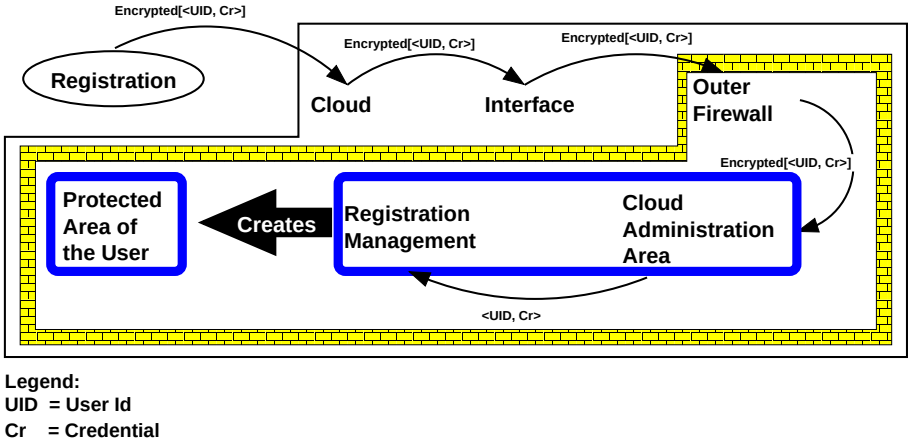

Figure 2: The Scenario of a User Registration in the Model

2.2 The User Registration

In this section we present a high level overview on how a new user can register to the cloud according to our formal model, see Figure 2. A new user sends (at least) the following information to the cloud in a message encrypted by the public key of the cloud: her *user identifier (UID)* and her credential. Sending a credential to a third-party is a weak point of the current model, but since the abstraction of the security mechanism of the model are going to be refined significantly (with a sophisticated identity and access control managements) in the near future, it is just fine at the moment.

The message reaches the cloud where the interface leads through the outer firewall and places it in the cloud administration area. Here the message content is decrypted, the personal data of the user are stored and then an area protected by the credential of the user is created for the user.

2.3 The Subscription to Cloud Services

In this section, we give an overview on how an user can subscribe to a particular cloud service, see Figure 3. The user sends the following information to the cloud in a *subscription* message (the message is encrypted with the public key of the cloud):

- her UID in the cloud,
- the public identifier of a service which she is going to use and
- the details of some payment (amount, chosen service options, etc).

The message arrives at the cloud administration area in the same way as a registration message and then it is decrypted. The subscription management checks first whether the user has already been registered in the cloud (by checking the given UID), then searches for the owner of the requested service from a database (using the service identifier) and sends all the information received from the user to the official client of the corresponding service owner (the sent

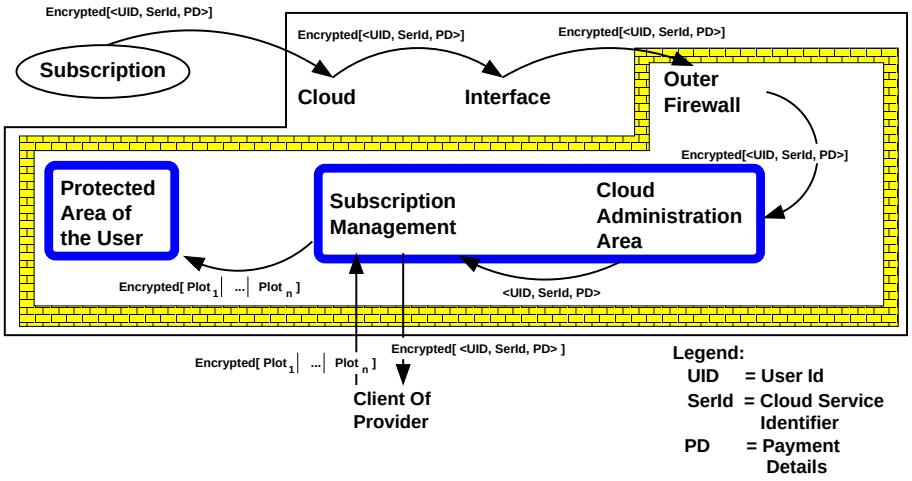

Figure 3: The Scenario of a Subscription to a Service in the Model

information is encrypted with the public key of the service owner; this part is abstract in the model).

The (client of the) service owner responds with a *new plot* message (the message is encrypted with the public key of the cloud; this part is abstract in the model). This message contains a service plot, which determines and restricts how the user can use the requested cloud service. The message is received by the subscription management, which encrypts the service plot with the public key of the corresponding/target user and sends it to the credential protected user area of this user, where the plot will be stored.

2.4 The Processing of User Requests

The following section gives a high level overview on how a typical user request is processed and performed in our cloud service system model, see Figure 4. An user can request the cloud to perform one or more service operations (which may belong to various services) in a message encrypted with its own key/passphrase. Such a message includes the following information associated with each requested operation:

- the full name of the service operation (the full name of an operation is always unique within a cloud, e.g.: $serviceId.operationId$),
- the address/id of the client from where the request was sent and to where the output should be delivered by the cloud,
- some information about the hardware and software components of the end device used by the user at the client side (it comprises everything which can be used by the adaptation mechanisms, e.g.:hardware and connection types and performance parameters, OS types, supported features, etc.) and
- an argument list for the requested operation.

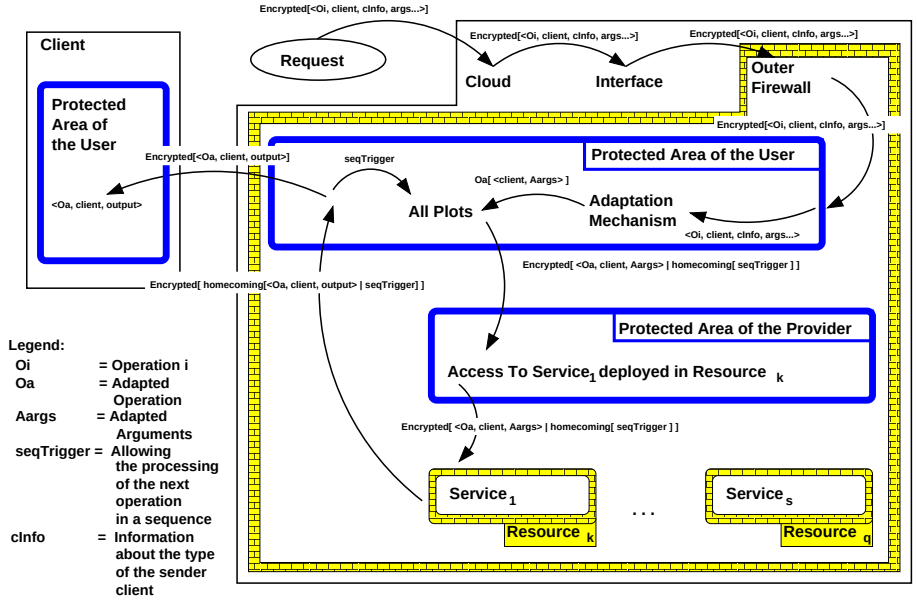

Figure 4: The Scenario of a User Request in the Model

The message pass through the outer firewall in the same way as a registration message or a subscription message and finally it arrives at the credential protected area of the corresponding user where it is decrypted.

Each message content is processed by an instance of the *adaptation agent* which takes the name of the operation and the information about the hardware and software framework of the client and may provide a new operation name and an adapted argument list according to some predefined adaptation rules[1]. The adapted operation and arguments will is sent to the plot area.

Then, if the adapted operation fits into a free slot of a plot, the operation is triggered. The term which describes such an operation always carries the following information: the full name of the (adapted) service operation, the address/id of the client, the (adapted) argument list for the requested operation and a special component called *homecoming*. The *homecoming* can be regarded as special container which travels together the triggered operation along the way to a cloud service and then carries back the output to the credential protected area of the user. Additionally, it may also comprise further terms, e.g.:

- the public key of the user with which the output is encrypted by the service instance and
- a special construct called *sequence trigger* which unlocks the next slot in a sequential plot, when it returns to the user area with the output of the current operation.

A service operation which is triggered by a service plot is encrypted with the public key of the corresponding service owner first. Then it is forwarded

[1]Although the current version of the model has not comprised yet, but the adaptation rules (which may be unique for the user) are provided by the service owners as well together with the service plots.

to the credential protected area of this service owner, where an access right of a cloud resource assigned to it. Then the operation is encrypted again with the public key of the chosen cloud resource and it is forwarded further to a service instance which resides on the cloud resource. One of the threads of this service instance then performs the triggered operation and provides an output. The output consists of three parts: the unique name of the performed service operation, the output of this operation and the address/id of the client to where the output should be sent back.

These information are placed into the *homecomming* component which returns to the protected area of the user in a message encrypted with the public key of the user. After it has arrived, the output (the name of the executed operation and its output) is forwarded to the given client in an output message which is encrypted with the public key of the user (so its content can be accessed on the client only with the corresponding private key).

If the *homecoming* component contains a sequence trigger, then this unlocks the next slot in the corresponding sequential plot (so it allows a user request to trigger an operation which matches into the next slot of the corresponding sequential plot).

3 State of the Art

Ambient ASM [3] is not the only research which aims to build in a concept of mobile ambients to the ASM method. In [6] some advantages of a simple ambient concept introduced into ASM are demonstrated. Although this work was also inspired by ambient calculus, it is by far not refined and versatile as ambient ASM.

3.1 Ambient Calculus

The ambient calculus [5] introduced by Luca Cardelli and Andrew Gordon captures the concepts of locality, mobility and ability to cross boundaries in concurrent systems. The concept of *ambient*, which has the following main characteristics, 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.).
- An ambient can be nested inside other ambients. Two or more ambients with the same name may reside as sibling of each other within the same parent.
- 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 the ambient).

The ambient calculus itself includes only the following mobility and communication primitives:

$P, Q, R ::=$	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)

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 in the following lines:

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

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

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:

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

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. The new name can be used to name ambients and to operate on ambients by name. The name restriction is transparent to reduction:

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

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

Furthermore, one must be careful with the term $!(\nu\ n)P$, because it provides 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 action 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 capabilities, an ambient can allow some processes to perform certain operations without publishing its true name to them (see the description of entry, exit and open capabilities 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 communicate restricted capabilities to controlled interactions between ambients (from a capability the ambient name cannot be retrieved).

Entry Capability. The capability action $\textsc{In}\ m$ 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:

$$n[\ \textsc{In}\ m.P \mid Q\ ] \mid m[\ R\ ] \longrightarrow m[\ n[\ P \mid Q\ ] \mid R\ ]$$

Exit Capability. The capability action $\textsc{Out}\ m$ 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:

$$m[\ n[\ \textsc{Out}\ m.p \mid Q\ ] \mid R\ ] \longrightarrow n[\ P \mid Q\ ] \mid m[\ R\ ]$$

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

$$\textsc{Open}\ n.P \mid n[\ Q\ ] \longrightarrow P \mid Q$$

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. $(\textsc{In}\ n.\textsc{In}\ m).P$ is interpreted as $\textsc{In}\ n.\textsc{In}\ m.P$ (P does not start to execute until the preceding capabilities are performed).

3.2 Ambient ASM

The core idea of ambient ASM [3] is to introduce an implicit parameter *curamb* expressing a context for evaluation of terms and execution of machines. Analogously to conventional implicit object oriented parametrization (e.g., $this.f(x) = f(x)$), the *dot-term* $s.t$ is introduced, where s is a term standing for an ambient expression and t is a term of the form $f(t_1, \ldots, t_n)$ and f is a *location symbol*.

To each location an additional argument is added for the ambient *curamb* in which the location is evaluated. Moreover the basic ASM classification of functions and locations is extended with *ambient independent functions and locations* (i.e.: static or dynamic functions and location) whose values for given argument do not depend on any ambient expression.

An ASM is an ambient ASM if it can be obtained from a basic ASM [2] by allowing for every given machine P also a machine of the following form

$$\textbf{amb } exp \textbf{ in } P$$

where execution of P is performed in ambient exp (exp is bound to *curamb*). Additionally, the term $n[\ P\ ]$ introduced by [5] is also defined in the context of ambient ASMs as follows:

$$n[\ P\ ] = \textbf{amb } n \textbf{ in } P$$

The semantics of ambient ASMs is defined by transformation into basic ASMs in [3].

3.2.1 Moving Ambients

In [3] 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:

MobileAgentsManager $\equiv$
  choose $R \in \{\text{Entry}, \text{Exit}, \text{Open}\}$ **do** R

For this machine the 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 Entry, Exit and Open given in *curAmbProc*.

In [4], we extended the ASM machine given in [3], such that it fully captures the calculus of mobile agents and by which one is able to describe formal models (and to check certain properties) of concurrent systems including mobile components in two different abstraction layers: the spatial location of system components and their long range interaction and movements are specified in terms of moving ambients; while the algorithm of agents are defined in terms of ASM syntax.

4 Applied Notations and Definitions

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.:

$$\textsc{Listener}_{(ctr\_state:=RunningState,\ x:=a)}$$

or simply

$$\textsc{Listener}_{(RunningState,\ a)}$$

The latter case denotes that all the expressions enumerated in the subscript appear in the current state of the agent Listener.

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
$(\nu\ n_1, \ldots, n_m)P \equiv (\nu\ n_1) \ldots (\nu\ n_m)P$

4.1 Definitions of some Non-Basic Ambient Capabilities

As Cardelli and Gordon state in [5] the ambient calculus with the three basic capabilities (Entry, Exit and Open) described in Section 3 is powerful enough to be Turing-complete. But for expressing such a complex formal model as a model of cloud infrastructure we need to define and encode some new more complex non-basic capability actions in terms of the three basic capabilities. In this section, we define some capabilities which we used in the description of our cloud service system model.

Locks. In [5] the capability Open is used to encode locks:

$$\textsc{Acquire}\ lock.P \equiv \textsc{Open}\ n.P$$

Such a lock can be released with an ambient construct like $n[\ P\ ]$ where P may be equal to 0 (inactivity).

Renaming. It is used to rename an ambient comprising this capability:

$$n\ \textsc{Be}\ m.P \equiv (\nu\ s)(s[\ \textsc{Out}\ n \mid m[\ \textsc{Open}\ n.\textsc{Out}\ s.P\ ]\ ] \mid \textsc{In}\ s.\textsc{In}\ m)$$

The renaming capability was already used in [5], but our definition differs from Cardelli's definition. In the original definition, the ambient m was

not enclosed into another, name restricted ambient, so after it has left ambient n, n may enter into another ambient m (if more than one m exist as sibling of each other).

In our case we enclosed m into the name restricted ambient s (so the true name of s is unique). s leaves n together with m in its body. Since the name of s is unique, the ambient n cannot miss it, when it tries to enter into it. Then in the body of s, n can enter into the only m which is located in the body of s (according to the definition).

$$
\begin{array}{ll}
& n[\ n\ \textsc{Be}\ m.P \mid Q\ ] \\
= & (\nu\ s)(n[\ s[\ \underline{\textsc{Out}\ n}\ \mid m[\ \textsc{Open}\ n.\textsc{Out}\ s.P\ ]\ ]\ \mid \textsc{In}\ s.\textsc{In}\ m \mid Q\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(s[\ m[\ \textsc{Open}\ n.\textsc{Out}\ s.P\ ]\ ]\ \mid n[\ \underline{\textsc{In}\ s.\textsc{In}\ m}\ \mid Q\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(s[\ m[\ \underline{\textsc{Open}\ n}.\textsc{Out}\ s.P \mid n[\ Q\ ]\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(s[\ m[\ \underline{\textsc{Out}\ s}.P \mid Q\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(m[\ P \mid Q\ ]\ \mid s[\ ]\ ) \\
\approx & m[\ P \mid Q\ ]
\end{array}
$$

Since the name of s is not known by anybody anymore after the last step, it does not have influence for the further reductions.

Seeing. This operation was defined in [5] and it is used to detect the presence of a given ambient:

$$
\begin{array}{l}
\textsc{See}\ n.P \equiv \\
\quad (\nu\ r,\ s)(\ r[\ \textsc{In}\ n.\textsc{Out}\ n.r\ \textsc{Be}\ s.P\ ]\ \mid \textsc{Open}\ s\ )
\end{array}
$$

With this definition, P gets activated only if its r capsule can get back to the same place (and it is renamed to s). That is, P is not activated if it is caught in the movement of n and ends up somewhere else.

$$
\begin{array}{ll}
& n[\ ]\ \mid \textsc{See}\ n.P \\
= & (\nu\ r,\ s)(\ n[\ ]\ \mid r[\ \underline{\textsc{In}\ n}.\textsc{Out}\ n.r\ \textsc{Be}\ s.P\ ]\ \mid \textsc{Open}\ s\ ) \\
\longrightarrow^* & (\nu\ r,\ s)(\ n[\ r[\ \underline{\textsc{Out}\ n}.r\ \textsc{Be}\ s.P\ ]\ ]\ \mid \textsc{Open}\ s\ ) \\
\longrightarrow^* & (\nu\ r,\ s)(\ n[\ ]\ \mid r[\ \underline{r\ \textsc{Be}\ s}.P\ ]\ \mid \textsc{Open}\ s\ ) \\
\longrightarrow^* & (\nu\ s)(\ n[\ ]\ \mid s[\ P\ ]\ \mid \underline{\textsc{Open}\ s}\ ) \\
\longrightarrow^* & n[\ ]\ \mid P
\end{array}
$$

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

$$
\begin{array}{l}
m\ \textsc{Wrap}\ n.P \equiv \\
\quad (\nu\ s,\ r)(\ s[\ \textsc{Out}\ n.\textsc{See}\ n.s\ \textsc{Be}\ m.r[\ \textsc{In}\ n\ ]\ ]\ \mid \textsc{In}\ s.\textsc{Open}\ r.P\ )
\end{array}
$$

The name restricted ambient s leaves the ambient n first. Then n enters into s. If the presence of n is detected in s, s is renamed to the given

name m. The role of the ambient r is to trigger the execution of P only after n was wrapped into m.

$$
\begin{array}{ll}
n[\ m\ \textsc{Wrap}\ n.P\ ] & \\
\quad = & (\nu\ s, r)(\ n[\ s[\ \underline{\textsc{Out}\ n}.\textsc{See}\ n.s\ \textsc{Be}\ m.r[\ \textsc{In}\ n\ ]\ ]\ \mid \\
 & \quad \textsc{In}\ s.\textsc{Open}\ r.P\ ]\ ) \\
\longrightarrow^* & (\nu\ s, r)(\ s[\ \textsc{See}\ n.s\ \textsc{Be}\ m.r[\ \textsc{In}\ n\ ]\ ]\ \mid n[\ \underline{\textsc{In}\ s}.\textsc{Open}\ r.P\ ]\ ) \\
\longrightarrow^* & (\nu\ s, r)(\ s[\ \underline{\textsc{See}\ n.s}\ \textsc{Be}\ m.r[\ \textsc{In}\ n\ ]\ ]\ \mid n[\ \textsc{Open}\ r.P\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ s, r)(\ s[\ \underline{s\ \textsc{Be}\ m.r}[\ \textsc{In}\ n\ ]\ ]\ \mid n[\ \textsc{Open}\ r.P\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ r)(\ m[\ r[\ \underline{\textsc{In}\ n}\ ]\ ]\ \mid n[\ \textsc{Open}\ r.P\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ r)(\ m[\ n[\ r[\ ]\ ]\ \mid \underline{\textsc{Open}\ r}.P\ ]\ ]\ ) \\
\longrightarrow^* & m[\ n[\ P\ ]\ ]
\end{array}
$$

Remark. The name m must be revealed only after n enters to the wrapper ambient, since the wrapper ambient m is usually used to release a lock. So n must reach its destination before m is dissolved by a lock (this why capability See is used to detect the presence of n).

Accepting/Allowing Foreign Codes. This capability is just a basic Open capability action. It is applied if an ambient allows/accepts an ambient construct 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.

$$\textsc{Allow}\ key \equiv \textsc{Open}\ key$$

By the employment of this action the functionality contained by this particular sub-ambient appears directly in the body of the original ambient, so the accepted functionality will be able to perform some operation with this ambient (e.g.: moving it into or out from some other ambients, dissolving some sub-ambients its body, etc.).

Draw in an Ambient. The goal of this ambient capability is to draw in a particular ambient (identified by its name) into another ambient (which contains this capability) with the help of a functionality/protocol which is got accepted by a the captured ambient and then to dissolve this ambient in order to access to its content:

$$
\begin{array}{l}
n\ \textsc{DrawIn}_{key}\ m.P \equiv \\
\quad key[\ \textsc{Out}\ n.\textsc{In}\ m.\textsc{In}\ n\ ]\ \mid \textsc{Open}\ m.P
\end{array}
$$

In the definition n is the ambient which is going to capture of an ambient called m. The name of ambient key identifies the functionality/protocol, which this ambient comprises. The ambient key leaves n and it becomes blocked as long as an ambient m shows up as a sibling of key, then key enters into m (if there are more than one m, one of them will be chosen

non- deterministically). If m contains the capability ALLOW key, it accepts the functionalities contained by key which leads m into n, where m is dissolved.

$$
\begin{array}{ll}
m[\ Q \mid \textsc{Allow}\ key\ ] \mid n[\ n\ \textsc{DrawIn}_{key}\ m.P\ ] & \\
\quad = & m[\ Q \mid \textsc{Allow}\ key\ ] \mid n[\ key[\ \underline{\textsc{Out}\ n}.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \\
& \textsc{Open}\ m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \textsc{Allow}\ key\ ] \mid key[\ \underline{\textsc{In}\ m}.\textsc{In}\ n\ ] \mid n[\ \textsc{Open}\ m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\textsc{Allow}\ key} \mid key[\ \textsc{In}\ n\ ]\ ] \mid n[\ \textsc{Open}\ m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\textsc{In}\ n}\ ] \mid n[\ \textsc{Open}\ m.P\ ] \\
\longrightarrow^* & n[\ m[\ Q\ ] \mid \underline{\textsc{Open}\ m}.P\ ] \\
\longrightarrow^* & n[\ Q \mid P\ ]
\end{array}
$$

Draw in an Ambient then Release a Lock. This capability is very similar to the previous one, but after m is captured by n (and before m is dissolved), n is wrapped by another ambient. The new ambient is usually used to provide a release for a lock after m appears in the body of n. So in adder words the event of capturing an ambient m triggers a release for a lock.

$$
\begin{array}{l}
n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ lock.P \equiv \\
\quad key[\ \textsc{Out}\ n.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \textsc{See}\ m.lock\ \textsc{Wrap}\ n.\textsc{Open}\ m.P
\end{array}
$$

It is important to create the mentioned release before m is dissolved, otherwise the content of m which shows up in the body of n may move n away from the place where the lock should appear.

$$
\begin{array}{ll}
m[\ Q \mid \textsc{Allow}\ key\ ] \mid n[\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ lock.P\ ] & \\
\quad = & m[\ Q \mid \textsc{Allow}\ key\ ] \mid n[\ key[\ \underline{\textsc{Out}\ n}.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \\
& \textsc{See}\ m.lock\ \textsc{Wrap}\ n.\textsc{Open}\ m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \textsc{Allow}\ key\ ] \mid key[\ \underline{\textsc{In}\ m}.\textsc{In}\ n\ ] \mid \\
& n[\ \textsc{See}\ m.lock\ \textsc{Wrap}\ n.\textsc{Open}\ m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\textsc{Allow}\ key} \mid key[\ \textsc{In}\ n\ ]\ ] \mid \\
& n[\ \textsc{See}\ m.lock\ \textsc{Wrap}\ n.\textsc{Open}\ m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\textsc{In}\ n}\ ] \mid n[\ \textsc{See}\ m.lock\ \textsc{Wrap}\ n.\textsc{Open}\ m.P\ ] \\
\longrightarrow^* & n[\ m[\ Q\ ] \mid \underline{\textsc{See}\ m.lock}\ \textsc{Wrap}\ n.\textsc{Open}\ m.P\ ] \\
\longrightarrow^* & n[\ m[\ Q\ ] \mid \underline{lock\ \textsc{Wrap}\ n}.\textsc{Open}\ m.P\ ] \\
\longrightarrow^* & lock[\ n[\ m[\ Q\ ] \mid \underline{\textsc{Open}\ m}.P\ ]\ ] \\
\longrightarrow^* & lock[\ n[\ Q \mid P\ ]\ ] \qquad // \ldots \mid \textsc{Open}\ lock.R
\end{array}
$$

Encoding Alternative Choices The ambient construct discussed next can be regarded as a relatively complex capability by which we are able to choice and to capture one ambient from a pool which may contain ambients with different names. The structure of this choice construct is similar to the non-basic capability $n\ \textsc{DrawIn}_{key}\ m.P$ defined above. As before an ambient called n is prepared to pull in another ambient into its body by

some functionalities which is enclosed by the ambient *key* which leaves n, enters into one of the given target ambients and if it gets accepted it leads this target ambient into n.

In the definition the terms $m_1,\ldots,m_k$ denote the different ambient names belonging to different choices. These are the names which can be accepted by n. Behind these ambient names in dot notations we can specify a process P_i which can be an ASM agent or an ambient construct. Such a process must be executed if an ambient with the preceding name is captured by n.

$$
\begin{array}{l}
n\ \textsc{Choice}_{key}\ m_1.P_1 + \ldots + m_k.P_k \equiv \\
\quad (\nu\ s,\ r,\ t)(\\
\qquad s[\\
\qquad\quad \textsc{Out}\ n.key[\ \textsc{Open}\ r.\textsc{Out}\ s.\textsc{In}\ n\ ]\ | \\
\qquad\quad \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ | \\
\qquad\quad \vdots \\
\qquad\quad \textsc{In}\ m_k.r[\ \textsc{In}\ key.m_k\ \textsc{Be}\ t.\textsc{Open}\ t.P_k\ ]\ ]\ | \\
\qquad \textsc{Open}\ t.t[\ ]
\end{array}
$$

Here we specified more than one entry capability with different ambient names in the body of *key*, but of course, only one of these prepared capability constructs will be used up finally (only one ambient will be chosen).

The major challenge is how to guarantee that the non-used ambient constructs will not have any influences for the latter reductions any more. According to our approach (which can be summarized roughly such that we lock the unused terms into some save place and throw away the key) we enclose these unused capabilities located in the body of the name restricted ambient s, whose name is not known anybody anymore after the choice has been done. The only reason why r and t are name restricted is that so they cannot accidently interact with any other ambient constructs defined in a hierarchy together with a choice capability.

In the following we present how the choice ambient construct works if we are able to accept an ambient which can be called either m_1 or m_2 and the choice m_2 is triggered. In this case P_2 will be performed finally.

$$
\begin{array}{l}
m_2[\ Q\ |\ \textsc{Allow}\ key\ ]\ | \\
n[\ n\ \textsc{ChooseFrom}_{key}\ m_1.P_1 + m_2.P_2\ ] \\
\quad = \quad (\nu\ s,\ r,\ t) \\
\qquad\quad (\ m_2[\ Q\ |\ \textsc{Allow}\ key\ ]\ | \\
\qquad\quad n[\\
\qquad\qquad s[\ \underline{\textsc{Out}\ n}.key[\ \textsc{Open}\ r.\textsc{Out}\ s.\textsc{In}\ n\ ]\ | \\
\qquad\qquad\quad \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ | \\
\qquad\qquad\quad \textsc{In}\ m_2.r[\ \textsc{In}\ key.m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ ]\ | \\
\qquad\qquad \textsc{Open}\ t.t[\ ]\ ]\ )
\end{array}
$$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ m_2[\ Q \mid \textsc{Allow}\ key\ ] \mid$
$s[\ key[\ \textsc{Open}\ r.\textsc{Out}\ s.\textsc{In}\ n\ ] \mid$
$\textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ] \mid$
$\underline{\textsc{In}\ m_2}.r[\ \textsc{In}\ key.m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ ] \mid$
$n[\ \textsc{Open}\ t.t[\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ m_2[\ Q \mid \textsc{Allow}\ key \mid$
$s[\ key[\ \textsc{Open}\ r.\textsc{Out}\ s.\textsc{In}\ n\ ] \mid$
$\textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ] \mid$
$r[\ \underline{\textsc{In}\ key.m_2}\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ ]\ ] \mid$
$n[\ \textsc{Open}\ t.t[\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ m_2[\ Q \mid \textsc{Allow}\ key \mid$
$s[\ key[\ \underline{\textsc{Open}\ r}.\textsc{Out}\ s.\textsc{In}\ n \mid r[\ m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ ]\ ] \mid$
$\textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ] \mid$
$n[\ \textsc{Open}\ t.t[\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ m_2[\ Q \mid \textsc{Allow}\ key \mid$
$s[\ key[\ \underline{\textsc{Out}\ s}.\textsc{In}\ n \mid m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ] \mid$
$\textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ] \mid$
$n[\ \textsc{Open}\ t.t[\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ m_2[\ Q \mid \underline{\textsc{Allow}\ key} \mid$
$key[\ \textsc{In}\ n \mid m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ] \mid$
$s[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ] \mid$
$n[\ \textsc{Open}\ t.t[\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ m_2[\ Q \mid \underline{\textsc{In}\ n} \mid \underline{m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2} \mid$
$s[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ] \mid$
$n[\ \textsc{Open}\ t.t[\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r, t)$
$(\ n[\ \underline{\textsc{Open}\ t}.t[\ ] \mid$
$t[\ Q \mid \textsc{Open}\ t.P_2 \mid$
$s[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r)$
$(\ n[\ t[\ ] \mid Q \mid \underline{\textsc{Open}\ t}.P_2 \mid$
$s[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ )$

$\longrightarrow^*$ $(\nu\ s, r)$
$(\ n[\ Q \mid P_2 \mid$
$s[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ )$

After the last step, the message content Q and a functionality assigned to the captured message type, encoded by process P_2 show up in the body of started and the content of the ambient s remains hidden and unaccessible forever.

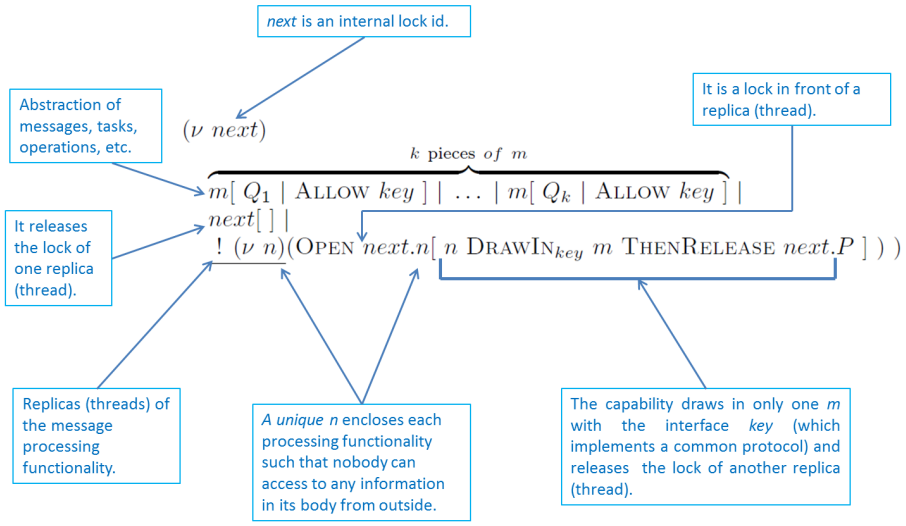

Figure 5: An Ambient Construct for Processing Numerous Ambients Called m Concurrently

4.2 Concurrent Processing of Messages

The ambient construct which is discussed in this section can be regarded (among others) as an abstraction of a multi-threaded server application. It is able to capture and dissolve several ambients (with the same name) in parallel.

The slightly modified versions of this construct are applied in several components of the specification, started from the service plots, through the scheduling of the service operations and in the deployed service instances.

In the example depicted in Figure 5 we assume that k pieces of ambients with the name m are present. Each of them is going to be captured and accepted by an independent replica of the ambient n (the replication of n is denoted by the exclamation mark). 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 is generated for each replica of n. One of the consequences of this is that nobody knows 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).

Every replica is blocked by a lock encoded by the capability Open $next$. A single release encoded by the empty ambient $next$ is provided for such a lock, so the content of one of the replicas of n can be started. This means the capability n DrawIn$_{key}$ m ThenRelease $next.P$ (defined in Section 4.1) pulls in one of the ambients m (it is chosen non-deterministically) into n (of course, each m must allow the mechanism which leads it into the body of a replica of n, see Section 4.1). Then it packs the ambient n into a new instance of the ambient $next$ which encodes a release for a lock of another replica of n. So before the previously captured ambient m is dissolved an it content is processed another replica of n may have stood ready for accepting and processing another instance of m.

The ambient name $next$ is name restricted, because it must be an unique and

unknown name in its environment in order to prevent any accidental interfere with other ambient constructs (if our described mechanism is part of a more complex ambient hierarchy).

In the following reduction, we present the outlined scenario of the capturing of k pieces of m by some replicas of n in details.

$$
\begin{array}{ll}
 & (\nu\ next) \\
 & \overbrace{m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ \ldots\ \mid m[\ Q_k \mid \textsc{Allow}\ key\ ]}^{k \text{ pieces } of\ m} \mid \\
 & next[\ ] \mid \\
 & \underline{!(\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ ) \\
\longrightarrow^* & (\nu\ next) \\
 & \quad (m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
 & \quad next[\ ] \mid \\
 & \quad !(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad \underline{\textsc{Open}\ next}.n_1[\ n_1\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \\
\longrightarrow^* & (\nu\ next) \\
 & \quad (m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
 & \quad !(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad n_1[\ \underline{n_1\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P}\ ]\ ) \\
 & \\
\longrightarrow^* & (\nu\ next) \\
 & \quad \overbrace{(m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ \ldots\ \mid m[\ Q_k \mid \textsc{Allow}\ key\ ]}^{k-1 \text{ pieces } of\ m} \mid \\
 & \quad \underline{!(\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad \underline{next}[\ n_1[\ Q_i \mid P\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ next) \\
 & \quad (m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
 & \quad !(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad \underline{\textsc{Open}\ next}.n_2[\ n_2\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ] \\
 & \quad next[\ n_1[\ Q_i \mid P\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ next) \\
 & \quad (m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
 & \quad !(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad n_2[\ \underline{n_2\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P}\ ] \mid \\
 & \quad n_1[\ Q_i \mid P\ ]\ ) \\
 & \\
\longrightarrow^* & (\nu\ next) \\
 & \quad \overbrace{(m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ \ldots\ \mid m[\ Q_k \mid \textsc{Allow}\ key\ ]}^{k-2 \text{ pieces } of\ m} \mid \\
 & \quad \underline{!(\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad \underline{next}[\ n_2[\ Q_j \mid P\ ]\ ]\ \mid \\
 & \quad n_1[\ Q_i \mid P\ ]\ ) \\
\longrightarrow^* & \ldots \\
\longrightarrow^* & (\nu\ next) \\
 & \quad (\underline{!(\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ \mid \\
 & \quad \underbrace{\underline{next}[\ n_k[\ Q_l \mid P\ ]\ ] \mid n_{k-1}[\ Q_1 \mid P\ ] \mid \ \ldots\ \mid n_1[\ Q_k \mid P\ ]\ )}_{\text{each pieces of m is processed in a unique n}}
\end{array}
$$

After the last step in the reduction above k instances of the ambient m is under processing in k pieces of independent replicas of the ambient n. Furthermore a release (encoded by the wrapper ambient *next*) is provided for the lock of a new replica of n, such that the mechanism is prepared to accept newly incoming m ambients.

5 The System Specification

In this section we give the formal model of a cloud service architecture in terms of ambient ASM. In the presented model, we assume that there are some standardized public ambient names, which are known by all contributors[3]. We distinguish the following kinds of public ambients: addresses (e.g.: *cloud*, $client_1, \ldots, client_n$), message types (e.g.: *reg*(*istration*), *request*, *subs*(*cription*), *output*) and elements of some (public) protocols (e.g.: *lock*, *msg*, *intf*, *access*, *task*, *out*, $o_1, \ldots, o_s$, *op*). All other ambient names are non-public in the model which follows:

$curAmbProc :=$
  root[// The ambient tree hierarchy[4]
    Cloud |
    $Client_1 | \ldots | Client_n$
  ]

5.1 A Simplified Client

In the current model we specify only a simplified client, which can be accessed by more than one user:

//Client spec.
$Client_i \equiv$
  $client_i$[!OPEN *msg* |
    $userCr_x$[CLOUDCLIENT | !ALLOW *output*] |
    ⋮
    $userCr_z$ [CLOUDCLIENT | !ALLOW *output*]
  ]
where
  CLOUDCLIENT ≡ a client application for the cloud (abstract)

Each user who uses a particular client has a restricted area protected by her credential (denoted by ambients $userCr_x, \ldots, userCr_z$) on the client, where the outputs of the service operations received from the cloud can be stored and accessed. Furthermore, each user can execute an ASM agent (it is abstract in the current model), via which she can send messages to the cloud, see below.

[3] In ambient calculus, the name of an ambient is used to control access. So the true names of most of the ambients would be guarded very closely, and only specific capabilities would be handed out about how to use the name (the name of an ambient can never be retrieved from a capability).

[4] The ambient called *root* is a special ambient which is required for the ASM specification of the three basic capability actions [3]

5.2 The User Actions

A user can perform three actions: registration, subscription to a service and requesting some service operations via some given service operations). These user actions can be triggered by sending the following three messages to the ambient *Cloud*.

$$RegistrationMsg \equiv (\nu\ l)msg[\ \text{In}\ cloud.\text{Open}\ intf.l[\ \text{In}\ cloudCr.reg[\ \text{Out}\ l.\langle userX, userCr_x\rangle\ ]\ ]\ ]^{5}$$

$$SubscriptionMsg \equiv (\nu\ l)msg[\ \text{In}\ cloud.\text{Open}\ intf.l[\ \text{In}\ cloudCr.subs[\ \text{Out}\ l.\langle userX, serviceId_1, payment_x\rangle\ ]\ ]\ ]$$

$$\begin{aligned} RequestMsg \equiv (\nu\ l)msg[\ &\text{In}\ cloud.\text{Open}\ intf.l[\\ &\text{In}\ userCr_x.request[\ \text{Out}\ l\ |\\ &\quad \langle \text{“}o_i'', \text{“}client'', cInfo, args\ldots\rangle\ |\\ &\quad \vdots\\ &\quad |\ \langle \text{“}o_k'', \text{“}client'', cInfo, args\ldots\rangle\ ]\ ]\ ] \end{aligned}$$

These messages carry those information which was already enumerated in Section 2.2, in Section 2.3 and in Section 2.4. In Appendix B a particular example is presented how such a user request is processed and performed by the modeled functionalities of the cloud service system.

The content of all these messages above is enclosed by an ambient whose name is bound by name restriction (e.g.: see the ambient l in the definitions of the messages above). Consequently the content of these messages is totally hidden and inaccessible from outside as long as message body leaves the restricted ambient. Such a construct can be regarded as an abstraction of an encryption technique, like passphrase-based encryption or the public- and private-key cryptography. For instance, the public key can be represented by an entry action, e.g.: In *cloudCr* in the case of *RegistrationMsg*; and an ambient can be employed as a private key, e.g.: the ambient *cloudCr* in the specification of our cloud service system (see Section 5.3).

In such cases, the information are embedded into a name restricted ambient, such that this protected content leaves the surrounding restricted ambient if and only if the entire construct arrives at the corresponding ambient (which represents a passphrase or a private key), e.g.:

$$(\nu\ l)l[\ \overbrace{\text{In}\ key}^{(1st)}.access[\ \overbrace{\text{Out}\ l}^{(2nd)}.ProtectedContent\ ]\ ]\ |\ key[\ \overbrace{!\text{Open}\ access}^{(3rd)}\ ]^{6}$$

In other words the encrypted information can be retrieved if and only if the corresponding passphrase or private key is present.

5.3 The Specification of the Cloud Service Architecture

The cloud service architecture is defined as follows:

[5] $userCr_x$ is the credential of user X.

[6] The numbers over the ambient actions denote their execution order.

$Cloud \equiv (\nu\ k,\ q,\ resource_1, \ldots resource_m) cloud[$
$\quad interface \mid$
$\quad k\,[\, resource_1[\, service_1\,] \mid \ldots \mid resource_i[\, service_1\,] \mid$ //a service may have
//several instances
$\quad\quad resource_{i+1}[\, service_2\,] \mid \ldots \mid resource_m[\, service_n\,] \mid$
$\quad\quad q[\,$!Open $msg \mid$
$\quad\quad\quad cloudCr[$
$\quad\quad\quad\quad$!Open $reg.(UID,\ userCr)$.RegMgr $\mid$
$\quad\quad\quad\quad$!Open $subs.(UID,\ serviceID,\ payment)$.SubsMgr $\,] \mid$
$\quad\quad\quad userCr_x[\, userInterface\,] \mid \ldots \mid userCr_y[\, userInterface\,]$
$\quad\quad\quad ownerCr_v\ [\, ownerInterface\,] \mid \ldots \mid ownerCr_w\ [\, ownerInterface\,]$
$\quad\quad]\,]\,]$
where
$\quad interface \equiv !intf[\,$ in msg.In k.In $q\,]$

In this definition, some ambients are bound by name restriction. This means that the names of these ambients are known only within the cloud service system, and therefore their contents are completely hidden and not accessible at all from outside of the cloud. These are:

- The ambients $resource_1, \ldots, resource_m$, which represent restricted computational resources of the cloud. Within each cloud resource some service instances can be deployed (which owned by a service owner), see Section 5.3.3;
- the ambient k provides a restricted environment for the entire content of the cloud; and
- the ambient q (for quarantine) provides another restricted area within k, where all the messages arrived from outside of the cloud are placed for processing by the mechanism of $interface$.

The purpose of the ambients k and q is to prevent any malicious content which may cut loose in the body of q after a message frame (msg) is broken open (by Open msg) to leave the cloud together with some sensitive information. Since q is bound by name restriction, an ambient which came from outside of the cloud with any malicious content in its body cannot leave q. Furthermore since k is bound by name restriction as well, any malicious code which appears in the body of q (after a message frame has been dissolved) cannot take q out of k.

The ambient expression represented by $interface$ "pulls in" into the area protected by the ambients k and q any ambient construct which is encompassed by an ambient called msg (by a message frame).

The ambient $cloudCr$ in q denotes the credential protected area of the cloud administration. This contains the functionalities of the user registration and the service subscription. Both mechanisms are replicated, which means a new replica processes each registration and subscription message, such that:

- First the lock of a replica (e.g.: Open reg or Open $subs$) of the corresponding mechanism is released if a registration message called reg or a subscription message called $subs$ arrives at the cloud administration area.

- Then the content of such message contained by an asynchronous output action is captured by an input action and an instance of corresponding ASM agent (either RegMgr or SubsMgr) is started with the received information as arguments. For more information about these two agents, see Section 5.4.2 and Section 5.4.3.

The ambients $userCr_x, \ldots, userCr_y$ denote the credential protected areas of the registered users, while the ambients $ownerCr_v, \ldots, ownerCr_w$ denote the credential protected areas of the service owners. These ambients comprise the personal *access layers* for users and service owners (for details see Section 5.3.1 and Section 5.3.2 below).

In Appendix B a particular example is presented how such a user request is processed and performed by the modeled functionalities of the cloud service system.

5.3.1 The User Access Layers

A user access layer contained by the credential protected area of each users is defined as follows:

$userInterface \equiv$
!Open *request* |
!(*op*, *client*, $cInfo$, *args* ...).Adapter(*op*, *client*, $cInfo$, *args* ...) |
$allPlots$[
!Open $newPlot$ | !Open $returnValue$ |
//Composing output messages
!(ν *l*)((*o*, *client*, *a*).*msg*[Out $allPlots$.Out $userCr_x$.Out *q*.Out *k*.
Out *cloud*.In *client*.*l*[In $userCr_x$.*output*[Out *l*.$\langle o, a \rangle$]]]) |
//Plot choices for triggering service operations
$Plot_{(o_1)} \mid Plot_{(!o_2 o_3)} \mid \ldots \mid Plot_r$] //We assume that each op
//appears in one plot at most!

A user area which comprises the construct $userInterface$ is not necessarily an abstraction of a physical or virtual location like a user directory. It can also be regarded as a pool of functionalities which are permitted to perform only if the key/passphrase of the corresponding user is given. These functionalities are the following:

- The ambient construct !Open *request* accepts every *request* message received from the user.
- The adaptation mechanism captures the information contained by a user request and starts a new instance of the ASM agent Adapter with these information as arguments, see Section 5.4.4. The agent Adapter may convert the service operation name and its arguments according to some adaptation rules and provides a service operation request which is adapted to the end device of the user at the client side. This adapted operation request is sent to the area of the plots denoted by the ambient $allPlots$.
- The ambient construct !Open $newPlot$ accepts new service plots yielded by service owners.

- The ambient construct !OPEN $returnValue$ accepts service operation outputs received from service instances.
- The ambient construct which started with the expression $!(\nu\ l)(\ (o, client, a).msg[\ldots$ captures the output of every service operation and in a message encrypted with the public key of the user it sends it to the client whose address/id is given as a part of the output.
- The service plots are responsible for triggering the service operations requested by the user in the way as it is ordered by the corresponding service owners. For the formal specifications of some typical plots see Appendix A.

5.3.2 The User Access Layers Extended with Service Ownership

An access layer contained by the credential protected area of each service owner can be regarded as normal user access layer extended with some extra functionalities. It is defined as follows:

$ownerInterface$[7] $\equiv$
  $userInterface$ |
  //each service may have more than one instances deployed on various resources
  Ser_1[
    $(\nu\ next)(next[\ ]$ |
      $!(\nu\ n)$OPEN $next.n$[
        n DRAWIN$_{access}$ s_1 THENRELEASE $next$.ACCESSTO $resource_i$] |
      ⋮
      $!(\nu\ n)$OPEN $next.n$[
        n DRAWIN$_{access}$ s_1 THENRELEASE $next$.ACCESSTO $resource_j$])] |
  $Ser_2[\ldots]$ |
  ⋮
  $Ser_n[\ldots]$ |
where
  ACCESSTO $resource_i \equiv$
    OUT Ser_j.OUT $ownerCr_v$.OUT q.IN $resource_i.n$ BE $task$.ALLOW $serviceId_j$
  REQUESTFOR $Ser_q.P \equiv$
    OUT $ownerCr_v$.IN $ownerCr_w$.IN $Ser_q.n$ BE s_q.ALLOW $access$

A user area which comprises the construct $ownerInterface$ (similarly to $userInterface$) can also be regarded as a pool of functionalities which are permitted to perform only if the key/passphrase of the corresponding service owner is given.

As was mentioned before the construct $ownerInterface$ contains all the functionalities of a normal user access layer called $userInterface$. It also contains an ambient for each services owned by the service owner. The purpose of

[7] Instead of iteration(!) of $access$, the ambient Ser_i can comprise fixed number of ambient $access$ (access to cloud resources) or an ASM scheduler can be applied here.

such ambients is to to maintain the cloud resource accesses assigned to the services. The mechanisms located within these ambients are based on the ambient construct described in Section 4.2. This means:

- one of the replicas of these mechanisms captures a triggered service operation belonging to the corresponding service (a service operation arrived at $ownerInterface$ is always enclosed by an ambient which uniquely determines its service, ,e.g.: s_1,..., s_n, see Appendix B);
- then the replica provides a release for the lock of another such a replica (so another triggered operation can be accepted and processed immediately);
- finally, the replicated mechanism forwards the captured service operation to an instance of the service deployed on a particular cloud resource.

The enumerated and replicated mechanisms mentioned above contain access rights to various resources, so the captured operations are forwarded to different service instances (residing on different resources) depending on which replica is unlocked by the last release (so it can be regarded as a simple non-deterministic scheduler).

5.3.3 The Service Instances

The service instances deployed on cloud resources apply the ambient construct described in Section 4.2 for concurrent processing of incoming service operations sent by the user. Each request is processed by an independent replica of the service instance (these replicas can be regarded as abstraction of threads).

$$
\begin{array}{l}
service_i \equiv \\
\quad (\nu\ next)(next[\ ]\ | \\
\qquad !(\nu\ n) \\
\qquad\quad (\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{serviceId_i}\ task\ \textsc{ThenRelease}\ next\ | \\
\qquad\qquad (o,\ client,\ args\ldots).lock[\ \textsc{Service}_i(o,\ client,\ args\ldots)\ ]\ | \\
\qquad\qquad \textsc{Open}\ lock.(o,\ client,\ a).\textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
\qquad\qquad\quad \textsc{In}\ returnValue.\textsc{Out}\ n.\langle o, client, a\rangle\ ]\ ]\ )\ )
\end{array}
$$

A requested service operation is processed in the following way:

- a replica captures a service operation first (which is enclosed into an ambient called *task*, after it arrived at the resource, see Appendix B);
- a release is provided for the lock of another replica next (so another service operation request can be accepted and processed in parallel with the current one;
- then the information provided by the request (the full name of the operation, the address/id of the client to where the output should be delivered and the argument list of the operation) is captured by the input action $(o,\ client,\ args\ldots)$;
- the ambient *lock* provides a release for the lock encoded by Open *lock* and activates the mechanism which waits for the output of the service operation and carries back to the corresponding user area; and

- finally, an instance of the ASM agent SERVICE$_i$ is started with the captured information provided by the requested operation as arguments.

5.4 ASM Agents within the Cloud

5.4.1 About Cloud Services

For representing the services we adopt the abstract model of *Abstract State Services (AS^{2}s)* [9, 10], which is based on views on some hidden database layer that are equipped with service operations $o_1, \ldots, o_n$. These service operations are actually what is exported from the 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.

SERVICE$_i$(o, *client*, *args*...) ≡ //It is an Abstract State Service [9]
  ctr_state : {*RunningState*, ..., *EndState*}
  ...

In our model services are located in cloud resources and they are equipped with server functionalities for accepting triggered service operations (see Section 5.3.3). Additionally, we assume they are always parameterized and they expect parameter lists which consist of the following at least:

- the full name of the operation,
- the address/id of the client to where the output should be delivered, and
- the argument list of the operation itself.

The output of the operation is provided as a tuple (operation name, client address/id, output) in terms of an asynchronous output action, which is returned to the user area of the requester.

Remark. In the example presented in Appendix B, we assume that every service has at least the following two control states: *RunningState* and *EndState*.

5.4.2 The Cloud User Registration Agent

REGISTRATIONMANAGER is a parameterized ASM agent, which expects a UID and a credential as arguments. This agent is located in the cloud administration area (within the ambient *cloudCr*).

REGMGR = REGISTRATIONMANAGER(UID, *userCr*)[8]
REGISTRATIONMANAGER(UID, *userCr*) ≡
  ctr_state : {*RunningState*, *EndState*}
  initially *ctr_state* := *RunningState*

[8]The only purpose of the usage of the terms CLOUDMGR and SUBSMGR is to make the definition of the cloud more compact more compact.

```
  if ctr_state = RunningState then
    if getCr(UID) = undef
      storeNewUser(UID, userCr)
      let UIConstruct = getAccessLayerForUser( userCr ) in
        NewAmbientConstruct( UIConstruct )
        ctr_state := EndState
where
  UIConstruct ≡ userCr[
    Out CloudCr | !Open request |
    !(op, client, cInfo, args...).Adapter(op, client, cInfo, args...) |
    allPlots[
      !Open newPlot | !Open returnValue |
      //Composing output messages
      !(ν l)( (o, client, a).msg[ Out allPlots.Out userCr.Out q.Out k.
        Out cloud.In client.l[ In userCr_x.output[ Out l.⟨o, a⟩ ] ] ] ) |
      //No plots yet!!!
    ] ]
```

The agent first checks whether a credential has already been associated with the given use id by the function *getCr*. If it is not the case, then the user has not been registered yet and it associates and stores the given credentials and UID by applying the abstract macro storeNewUser.

Then it calls the abstract derived function *getAccessLayerForUser*, which generates the ambient construct *UIConstruct*. This newly created ambient construct is always made from a predefined template construct which is extended with the given user credential *userCr* (so *UIConstruct* is the same for all the new users except the applied user credentials in it).

The construct *UIConstruct* is placed into the ambient tree hierarchy as sibling of the agent RegistrationManager (by calling the abstract tree manipulation operation called NewAmbientConstruct [4]). Although such a construct is always created as a sibling of the agent CloudManager, but it leaves the ambient *cloudCr* as a first step (see the underlined moving action in the definition of *UIConstruct* above) and yields a credential protected personal access layer for the newly registered user.

5.4.3 The Cloud Service Subscription Agent

SubscriptionManager is a parameterized ASM agent, which expects a UID, the public identifier of a service which is intended to use and the details of some payment (amount, chosen service options, etc.) as arguments. This agent is located in the cloud administration area (within the ambient *cloudCr*).

```
SubsMgr = SubscriptionManager(UID, serviceID, payment )
SubscriptionManager(UID, serviceID, payment ) ≡
  ctr_state : {RunningState, EndState}
  initially ctr_state := RunningState

  if ctr_state = RunningState then
```

```
let cr = getCr(UID) in // =userCr_x
  if cr ≠ undef then
    let prov = getServiceOwner( serviceID ) ) in
      let plotExpr = GETPLOTFROMSOWNER( prov,
                                        UID,
                                        serviceID,
                                        payment
                                      )
      in
        NEWAMBIENTCONSTRUCT(
          (ν l)l[ OUT CloudCr.IN cr.newPlot[
            OUT l.IN allPlots | plotExpr ] ]
        )
        ctr_state := EndState
```

First the agent checks whether the user has already registered (whether the function *getCr* returns with an associated user credential). If this is the case, then the agent checks who is the owner of the requested service with the function *getServiceOwner*.

By applying the abstract ASM macro GETPLOTFROMOWNER the agent sends its arguments to the announced client of the corresponding provider, which responses with a service plot of the requested service for the user according to the given information. The client agent forwards the plot to the credential protected area of the user in a message encrypted with the public key of the user.

Remark. Since the model is going to be extended with an identity management in the near future, the currently applied unique UID will be replaced by a user identifier provided by this mentioned identity management.

5.4.4 The Adapter Agent

ADAPTER is a parameterized ASM agent, which expects a name of a service operation, the address/id of a client, some information concerning the end device used by the user at the client side and the argument list of the requested operation as arguments. The instances of this agent are located in all credential protected user areas.

```
ADAPTER(op, client, cInfo, args...) ≡
  ctr_state : {RunningState, EndState}
  initially ctr_state := RunningState

  let (newOp, newArgs...) = adaptToClient( cInfo, op, args... )
   in
    NEWAMBIENTCONSTRUCT(
        newOp[ IN allPlots.ALLOW op.⟨client, newArgs...⟩ ] )
    ctr_state := EndState
```

Based on the given information concerning the client device the agent may convert the name and the arguments of the operation according to some adaptation rules provided by the service owner and may generates a new operation request adapted to the end device of the user. The adapted request is forwarded to the plots assigned to the user (they are located within the ambient $allPlots$).

Appendix

A Formal Models of some Plots

In this section we give the definition of some typical plots in terms of ambient calculus. The described plots are the following:

o_1**:** a plot containing a single operation (with one-off access), see Appendix A.1.

$!o_1$**:** a plot containing a single operation (with multiple accesses), see Appendix A.2.

$o_1 \ldots$**:** a plot containing a sequence of a single operation, see Appendix A.3.

$o_1 o_2 o_3 \ldots$**:** a plot containing a sequence of operations, see Appendix A.4.

$o_1 + o_2 \ldots$**:** a plot containing a sequence of choices, see Appendix A.5.

Each plot (independently from its type) expects a service operation request in the following form:

$$o_i[\ \textsc{Allow}\ op.\langle \text{“}client''_j, args \ldots\rangle\ ]$$

where o_i is the identifier (the full name) of a service operation, $client_j$ is the address/id of the client to where the outcome of the operation must be sent and $args \ldots$ denotes the arguments of the operation.

Furthermore, a requested service operation is always triggered in the same way by each plot. This means that a triggered service operation is always given in the following form by a plot:

n_{unique}[$\langle$"$\mathtt{o_i}$", "$client''_j, args \ldots\rangle$ | RequestFor $Ser_k.homecoming$]
where
RequestFor $Ser_k \equiv$ Out $allPlots$.Out $userCr_x$.
In $ownerCr_v$.In $Ser_k.n_{unique}$ Be s_k.Allow $access$
$homecoming \equiv$ In $userCr_x$.In $allPlots.returnValue$[Open out | ...]

n_{unique} is always a fresh and unique name in the case of each triggered operation (provided by a replicated name restriction action $!(\nu\ n)$) within a plot), so nobody can access to its body from outside. The expression $\langle$"$\mathtt{o_i}$", "$client''_j, args \ldots\rangle$ is the description of the operation itself. The ambient construct RequestFor is responsible to move the triggered operation to the area of the corresponding service owner and to request an access to a particular service instance for the operation. The *homecoming* expression carries back the outcome of the operation to the area of the user (it may contain additional information, e.g.: sequence trigger, see Section 2)

A.1 A Plot Containing a Single Operation (with One-Off Access)

This plot is based on the ambient construct described in Section 4.2, but since this plot can be used only once it does not apply any replication and locks. It simple captures an ambient whose name corresponds to the name of the expected operation and triggers this operation in the form described above.

$Plot_{(o_1)} \equiv$
$(\nu\ n)$
n[n DrawIn$_{op}$ o_1 |
($client$, $args \ldots$).Open $lock.\langle$"$\mathtt{o_i}''$, $client, args \ldots\rangle$ |
$lock$[RequestFor $Ser_1.homecoming$]]
where
RequestFor $Ser_i \equiv$
Out $allPlots$.Out $userCr_x$.In $ownerCr_v$.In $Ser_i.n$ Be s_i.Allow $access$
$homecoming \equiv$ In $userCr_x$.In $allPlots$.returnValue[Open out]

A.2 A Plot Containing a Single Operation (with Multiple Accesses)

This plot applies the ambient construct described in Section 4.2. This means one of the replicas of the mechanism captures an ambient whose name corresponds to the name of the expected operation, but before the operation is triggered a release is provided for the lock of another replica (so another instance of the same operation can be accepted and triggered immediately). An operation is triggered by this plot in the form which is described at the beginning of Appendix A.

$Plot_{(!o_1)} \equiv$
  $(\nu\ next)$
    $(!(\nu\ n)$
      OPEN $next.\ n[\ n$ DRAWIN$_{op}$ o_1 THENRELEASE $next\ |$
        $(client, args \ldots).$OPEN $lock.\langle$"$\mathtt{o}_\mathtt{i}'', client, args \ldots\rangle\ |$
        $lock[$ REQUESTFOR $Ser_1.homecoming\ ]\ ]\ |$
     $next[\ ]\ )$
where
  REQUESTFOR $Ser_i \equiv$
    OUT $allPlots.$OUT $userCr_x.$IN $ownerCr_v.$IN $Ser_i.n$ BE $s_i.$ALLOW $access$
  $homecoming \equiv$ IN $userCr_x.$IN $allPlots.$returnValue[OPEN out]

A.3 A Plot Containing a Sequence of a Single Operation

This plot is based on the ambient construct described in Section 4.2, but a replicas of the mechanism never provides a release for the lock of another replica, after it captures an ambient whose name corresponds to the name of the expected operation. The reason for this is because the next instance of the operation can be accepted and triggered if and only if the output of the previous operation is available. So this plot places the mentioned release into the *homecoming* component of a triggered operation as a sequence trigger, see Section 2.

$Plot_{(o_1 \ldots)} \equiv$
  $(\nu\ next)($
    $!(\nu\ n)$
      OPEN $next.n[\ n$ DRAWIN$_{op}$ $o_1\ |$
        $(client, args \ldots).$OPEN $lock.\langle$"$\mathtt{o}_\mathtt{i}'', client, args \ldots\rangle\ |$
        $lock[$ REQUESTFOR $Ser_1.homecoming\ ]\ ]\ |$
    $next[\ ]\ )$
where
  REQUESTFOR $Ser_i \equiv$
    OUT $allPlots.$OUT $userCr_x.$IN $ownerCr_v.$IN $Ser_i.n$ BE $s_i.$ALLOW $access$
  $homecoming \equiv$ IN $userCr_x.$IN $allPlots.$returnValue[OPEN $out\ |\ next[\ ]\ ]$

An operation is triggered by this plot in the form which is described at the beginning of Appendix A. The term $next[\ ]$ located in the ambient $returnValue$ is a sequence trigger (see Section 2), which permits the triggering of the operation again after the outcome of the previous instance of the operation has returned.

A.4 A Plot Containing a Sequence of Operations

The following plot is a sequential plot, which can trigger three operations, respectively. An operation is allowed to trigger, if and only if the output of the previous operation arrived. If the output of the last operation is available, then the triggering of another instance of the sequence is permitted again.

The plot consists of a replicated structure, which contains thee pieces of the slightly modified version of the construct described in Section 4.2 (each piece belongs to a service operation). The ambient construct of the first operation is blocked by the usual lock (Open *next*). But second one is enclosed by a name restricted ambient called *seq* and within this ambient another ambient also called *seq* is located additionally, which encloses the third construct belonging to the third operation (so the ambients *seq* compose a nested structure).

The first operation can be triggered if its lock is released by *next*[ ], but the ambients *seq* separates the second and the third constructs from the received operations requested by a user, so they prevent the triggering of the latter operations. For allowing the next operation in the sequence, each triggered operation carries an ambient capability Open *seq* in its *homecoming* part as a sequence trigger (see Section 2). The only exception is the last operation in the sequence which applies the construct *next*[ ] as a sequence trigger to permit a new instance of the entire sequence by allowing the first operation again.

$Plot_{(o_1 o_2 o_3 \ldots)} \equiv$
  $(\nu\ next)$
    (!$(\nu\ seq, n)$
      (Open $next.n$[n DrawIn$_{op}$ o_1 |
        $(client, args \ldots)$.Open $lock.\langle$“$\mathtt{o}_1'', client, args \ldots\rangle$ |
        $lock$[RequestFor $Ser_1.homecoming_{seq}$]] |
      seq[$(\nu\ n)$
          n[n DrawIn$_{op}$ o_2 |
            $(client, args \ldots)$.Open $lock.\langle$“$\mathtt{o}_2'', client, args \ldots\rangle$ |
            $lock$[RequestFor $Ser_1.homecoming_{seq}$]] |
        seq[$(\nu\ n)$
            n[n DrawIn$_{op}$ o_3 |
              $(client, args \ldots)$.Open $lock.\langle$“$\mathtt{o}_1'', client, args \ldots\rangle$ |
              $lock$[RequestFor $Ser_2.homecoming_{next}$]]]]) |
    $next$[ ])
where
  RequestFor $Ser_i \equiv$
    Out $allPlots$.Out $userCr_x$.In $ownerCr_v$.In $Ser_i.n$ Be s_i.Allow $access$
  $homecoming_{seq} \equiv$ In $userCr_x$.In $allPlots$.returnValue[Open out |
    Open seq]
  $homecoming_{next} \equiv$ In $userCr_x$.In $allPlots$.returnValue[Open out | $next$[ ]]

An operation is always triggered by this plot in the form which is described at the beginning of Appendix A. The terms Open *seq* and *next*[ ] located in the different instances of ambient *returnValue* in this plot are sequence triggers. The first one permits the triggering of the next operation in the sequence after the outcome of the previous operation has returned. The second one allows the triggering of the plot from the beginning (started with the first operation in the sequence) again, after the outcome of the last operation of the previous sequence has returned.

A.5 A Plot Containing a Sequence of Choices

The last plot example is based on the choice construct defined in Section 4.1. The plot can accept and trigger either the operation o_1 or the operation o_2. If one of the operations is triggered the choice cannot be applied again as long as the output of the triggered operation arrives back.

$Plot_{(o_1+o_2\ldots)} \equiv$
  $(\nu\ next)($
    $!(\nu\ n,\ lock_1,\ lock_2\ )$
      Open $next.n[\ n$ Choice$_{op}$ $o_1.AcceptOp_1 + o_2.AcceptOp_2\ |$
        $lock_1[$ RequestFor $Ser_k.homecoming\ ]\ |$
        $lock_2[$ RequestFor $Ser_l.homecoming\ ]\ ]\ |$
    $next[\ ]\ )$
where
  $AcceptOp_i \equiv (client,\ args\ldots).$Open $lock_i.\langle$“$\mathtt{o}_\mathtt{i}''$, $client, args\ldots\rangle$
  RequestFor $Ser_i \equiv$
    Out $allPlots.$Out $userCr_x.$In $ownerCr_v.$In $Ser_i.n$ Be $s_i.$Allow $access$
  $homecoming \equiv$ In $userCr_x.$In $allPlots.$returnValue[Open $out\ |\ next[\ ]\ ]$

An operation is always triggered by this plot in the form which is described at the beginning of Appendix A. The term $next[\ ]$ located in the ambient $returnValue$ is a sequence trigger (see Section 2), which permits the choice between the two operations again after the outcome of the previously chose operation has returned.

B An Example for Processing a Particular User Request

In our example the user X sends the following request message to the cloud (the message is encrypted with the public key of the user which is encoded by In $userCr_x$):

$msg[$ In $cloud.$Open $intf.l_{unique}[$ In $userCr_x.request[$
  Out $l_{unique}\ |\ \langle$“o_1'', “$client_1''$, $cInfo, args\ldots\rangle\ ]\ ]\ ]$

The name l_{unique} is a fresh and unique name in the case of each message (provided by a replicated name restriction action $!(\nu\ l)$), so nobody can access to the message content from outside. After the message has reached the credential protected user area of the user X in the cloud, it has been decoded and its message frame has been dissolved. Then the message content is available in the following format:

$$\langle\text{“}o_1'', \text{“}client_1'', cInfo, args\ldots\rangle$$

This information is processed by an instance of the agent Adapter which replaces the given operation name o_1 with one or more other operations (let us call them adapted operations, e.g.: o_1^a) according to the provided client information $cInfo$ and some internal adaptation rules (which may be unique for the

user). The arguments $args\ldots$ may be modified/converted as well. The agent ADAPTER provides the adapted operation for the plots in the following format:

$$o_1^a[\ \underline{\text{IN } allPlots}.\text{ALLOW } op.\langle \text{“}client_1'', args^a \ldots\rangle\ ]$$

First this ambient construct enters into the ambient $allPlots$ (see the underline ambient construct IN allPlots). Then if the service operation o_1^a is allowed by any plot it will be triggered. Each kind of plots triggers a service operation in the following form:

$n_{unique}[\ \langle \text{“}\mathtt{o_1^{a''}}, \text{“}client_1'', args^a \ldots\rangle \mid \text{REQUESTFOR } Ser_1.homecoming\ ]$
where
REQUESTFOR $Ser_1 \equiv \underline{\text{OUT } allPlots.\text{OUT } userCr_x.}$
$\underline{\text{IN } ownerCr_v.\text{IN } Ser_1.n_{unique} \text{ BE } s_1.\text{ALLOW } access}$
$homecoming \equiv \text{IN } userCr_x.\text{IN } allPlots.returnValue[\ \text{OPEN } out \mid \ldots]$

The name of n_{unique} is unique in the case of each triggered service operation (provided by a replicated name restriction action $!(\nu\ n)$), so nobody can access to its body from outside. The underlined ambient construct in “REQUESTFOR Ser_1“ moves this triggered operation (in the body of n_{unique}) to the corresponding service administration located within the corresponding credential protected service owner area. After it has arrived at this service administration, it renames the unique n_{unique} to the s_1 (which is an internal identifier of the service) by performing the capability n_{unique} BE s_1.

The construct denoted by *homecoming* will travel together with the triggered operation along the way to a cloud service instance deployed on a cloud resource. Then after the triggered operation was performed by a service instance it returns to the credential protected user area of user X (with the output of the service operation). It contains the access information (IN $userCr_x$.IN $allPlots$) by which it can enter into the user area of user X.

homecoming also contains the ambient $returnValue$, which carries the output of the service operation. The ambient $returnValue$ may also contain some *sequence trigger* (an ambient expression which is able to permit the triggering of the next service operation in a sequential plot) depending on the type of the employed plot (for more details see Section 2 and Appendix A).

The triggered service operation (located in the ambient s_1) sent from a user area will be assigned to a service instance of the corresponding service:

$s_1[\ \langle \text{“}\mathtt{o_1^{a''}}, \text{“}client_1'', args^a \ldots\rangle \mid \text{ALLOW } access.homecoming\ ] \mid$
$n'_{unique}[\ \underline{n'_{unique} \text{ DRAWIN}_{access}\ s_1 \text{ THENRELEASE } next.\text{ACCESSTO } resource_1}\ ]$
$\longrightarrow^*\ next[\ n'_{unique}[$
$\langle \text{“}\mathtt{o_1^{a''}}, \text{“}client_1'', args^a \ldots\rangle \mid homecoming \mid \text{ACCESSTO } resource_1\ ]\ ]$
where
$homecoming \equiv \text{IN } userCr_x.\text{IN } allPlots.returnValue[\ \text{OPEN } out \mid \ldots]$
ACCESSTO $resource_1 \equiv \underline{\text{OUT } Ser_1.\text{OUT } ownerCr_v.\text{OUT } q.}$
$\underline{\text{IN } resource_1.n'_{unique} \text{ BE } task.\text{ALLOW } serviceId_1}$

The name of n'_{unique} is unique in the case of each service instance access assignment (provided by a replicated name restriction action $!(\nu\ n)$), so nobody

can access to its body from outside. After the ambient *next* has been dissolved (it releases a lock which permits an access assignment to another triggered service operation), the underlined ambient construct in "AccessTo $resource_1$" moves the triggered operation (in the body of n'_{unique}) to the corresponding service instance. After it has arrived at the service instance, it renames the unique n'_{unique} to *task*, which is a public ambient name (part of a standardized protocol).

The triggered service operation (located in the ambient *task*) is performed by one of the threads of the corresponding service instance:

$$
\begin{array}{ll}
& task[\ \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''}, \text{“}client''_1, args^a \ldots\rangle \mid homecoming \mid \textsc{Allow}\ serviceId_1\ ]\ \mid \\
& n''_{unique}[\ \underline{n''_{unique}\ \textsc{DrawIn}_{serviceId_1}\ task\ \textsc{ThenRelease}\ next}\ \mid \\
& \quad \underline{(o,\ client,\ args\ldots).lock}[\ \textsc{Service}_1(o,\ client,\ args\ldots)\ ]\ \mid \\
& \quad \textsc{Open}\ lock.(o,\ client,\ a).\textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
& \qquad \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle o, client, a\rangle\ ]\ ] \\
\longrightarrow^* & \quad next[\ n''_{unique}[\ \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''}, \text{“}client''_1, args^a \ldots\rangle \mid homecoming \mid \\
& \qquad \underline{(o,\ client,\ args\ldots).lock}[\ \textsc{Service}_1(o,\ client,\ args\ldots)\ ]\ \mid \\
& \qquad \textsc{Open}\ lock.(o,\ client,\ a).\textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
& \qquad\quad \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle o, client, a\rangle\ ]\ ]\ ] \\
\longrightarrow^* & \quad next[\ n''_{unique}[\ homecoming \mid \\
& \qquad lock[\ \textsc{Service}_1(\text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ args^a\ldots)\ ]\ \mid \\
& \qquad \underline{\textsc{Open}\ lock}.(o,\ client,\ a).\textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
& \qquad\quad \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle o, client, a\rangle\ ]\ ]\ ] \\
\longrightarrow^* & \quad next[\ n''_{unique}[\ homecoming \mid \\
& \qquad \underline{\textsc{Service}_{1(RunningState)}(\text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ args^a\ldots)}\ \mid \\
& \qquad \underline{(o,\ client,\ a).}\textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
& \qquad\quad \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle o, client, a\rangle\ ]\ ]\ ] \\
\xrightarrow{asm}{}^* & \quad next[\ n''_{unique}[\ homecoming \mid \\
& \qquad \cancel{\textsc{Service}_{1(EndState)}}\ \mid \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\ \mid \\
& \qquad \underline{(o,\ client,\ a).}\textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
& \qquad\quad \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle o, client, a\rangle\ ]\ ]\ ] \\
\longrightarrow^* & \quad next[\ n''_{unique}[\ homecoming \mid \textsc{Out}\ resource_1.\textsc{In}\ q.out[\\
& \qquad\quad \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\ ]\ ]\ ]
\end{array}
$$

where

$homecoming \equiv \textsc{In}\ userCr_x.\textsc{In}\ allPlots.returnValue[\ \textsc{Open}\ out \mid \ldots]$

The name of n''_{unique} which carries the output is unique (provided by a replicated name restriction action $!(\nu\ n)$), so nobody can access to its body from outside. As soon as the enclosing ambient *next* is dissolved (it releases a lock which permits the execution of another thread of the service), the ambient n''_{unique} returns to the ambient q (by performing the capabilities "Out $resource_1$.In q"). Then it enters to the corresponding credential protected user area with the capability "In $userCr_x$" comprised by the *homecoming* part:

$$
\begin{array}{l}
n''_{unique}[\ \underline{\textsc{In}\ userCr_x.\textsc{In}\ allPlots.returnValue}[\ \textsc{Open}\ out \mid \ldots]\ \mid \\
\quad out[\ \textsc{In}\ returnValue.\textsc{Out}\ n''_{unique}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\ ]\ ]\ \mid \\
userCr_x[\ \ldots \mid allPlots[\ !\textsc{Open}\ returnValue \mid \ldots]\ ]
\end{array}
$$

$$
\begin{array}{ll}
\longrightarrow^* & userCr_x[\,\ldots \mid allPlots[\,!\textsc{Open}\ returnValue \mid \ldots \mid \\
& \quad n''_{unique}[\,returnValue[\,\textsc{Open}\ out \mid \ldots] \mid \\
& \quad\quad out[\,\underline{\textsc{In}\ returnValue}.\textsc{Out}\ n''_{unique}. \\
& \quad\quad\quad \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]\,]\,] \\
\longrightarrow^* & userCr_x[\,\ldots \mid allPlots[\,!\textsc{Open}\ returnValue \mid \ldots \mid \\
& \quad n''_{unique}[\,returnValue[\,\underline{\textsc{Open}\ out} \mid \ldots \mid \\
& \quad\quad out[\,\textsc{Out}\ n''_{unique}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]\,]\,]\,] \\
\longrightarrow^* & userCr_x[\,\ldots \mid allPlots[\,!\textsc{Open}\ returnValue \mid \ldots \mid \\
& \quad n''_{unique}[\,returnValue[\,\ldots \mid \\
& \quad\quad \underline{\textsc{Out}\ n''_{unique}}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]\,]\,] \\
\longrightarrow^* & userCr_x[\,\ldots \mid allPlots[\,\underline{!\textsc{Open}\ returnValue} \mid \ldots \mid \\
& \quad \cancel{n''_{unique}[\,]} \mid returnValue[\,\ldots \mid \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]\,] \\
\longrightarrow^* & userCr_x[\,\ldots \mid allPlots[\,!\textsc{Open}\ returnValue \mid \ldots \mid \\
& \quad \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]
\end{array}
$$

In the last step the ambient $returnValue$ is dissolved and the outcome of the operation is available within the ambient $allPlots$ (if $returnValue$ contained a sequence trigger, then it permits the processing of the next operation in the corresponding sequential plot). The name of the ambient n''_{unique} is not known by anybody anymore, so it does not have influence for the latter reductions.

The outcome $\langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle$ is sent back to the given client in an output message (the content of this message is encrypted with the public key of the user which is encoded by $\textsc{In}\ userCr_x$) by an ambient construct located in ambient $allPlots$ within the ambient $userCr_x$:

$$
\begin{array}{ll}
& cloud[\,\ldots \mid k[\,\ldots \mid q[\,\ldots \mid userCr_x[\,\ldots \mid allPlots[\,\ldots \mid \\
& \quad \underline{!(\nu\ l)(\ (o,\ client,\ a).msg[\,\textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.} \\
& \quad\quad \underline{\textsc{Out}\ cloud.\textsc{In}\ client.l[} \\
& \quad\quad\quad \textsc{In}\ userCr_x.output[\,\textsc{Out}\ l.\langle o,a\rangle\,]\,]\,]\,)\ \mid \\
& \quad \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]\,]\,]\,]\,] \mid \\
& client_1[\,\ldots \mid !\textsc{Open}\ msg \mid userCr_x[\,\textsc{CloudClient} \mid !\textsc{Allow}\ output\,]\,] \\
\longrightarrow^* & \quad cloud[\,\ldots \mid k[\,\ldots \mid q[\,\ldots \mid userCr_x[\,\ldots \mid allPlots[\,\ldots \mid \\
& \quad\quad !(\nu\ l)(\ (o,\ client,\ a).msg[\,\textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k. \\
& \quad\quad\quad \textsc{Out}\ cloud.\textsc{In}\ client.l[\\
& \quad\quad\quad\quad \textsc{In}\ userCr_x.output[\,\textsc{Out}\ l.\langle o,a\rangle\,]\,]\,]\,)\ \mid \\
& \quad\quad \underline{(o,\ client,\ a).msg[\,\textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.} \\
& \quad\quad\quad \underline{\textsc{Out}\ cloud.\textsc{In}\ client.l'_{unique}[} \\
& \quad\quad\quad\quad \textsc{In}\ userCr_x.output[\,\textsc{Out}\ l'_{unique}.\langle o,a\rangle\,]\,]\,]\ \mid \\
& \quad\quad \langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ \text{“}client''_1,\ outcome_{o_1^a}\rangle\,]\,]\,]\,]\,]\,] \mid \\
& \quad client_1[\,\ldots \mid !\textsc{Open}\ msg \mid userCr_x[\,\textsc{CloudClient} \mid !\textsc{Allow}\ output\,]\,] \\
\longrightarrow^* & \quad cloud[\,\ldots \mid k[\,\ldots \mid q[\,\ldots \mid userCr_x[\,\ldots \mid allPlots[\,\ldots \mid \\
& \quad\quad !(\nu\ l)(\ (o,\ client,\ a).msg[\,\textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k. \\
& \quad\quad\quad \textsc{Out}\ cloud.\textsc{In}\ client.l[\\
& \quad\quad\quad\quad \textsc{In}\ userCr_x.output[\,\textsc{Out}\ l.\langle o,a\rangle\,]\,]\,]\,)\ \mid \\
& \quad\quad msg[\,\underline{\textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.} \\
& \quad\quad\quad \underline{\textsc{Out}\ cloud.\textsc{In}\ \text{“}client''_1.l'_{unique}[} \\
& \quad\quad\quad\quad \textsc{In}\ userCr_x.output[\,\textsc{Out}\ l'_{unique}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}}\text{''},\ outcome_{o_1^a}\rangle\,]\,]\,]\,]\,]\,]\,]\,]\ \mid \\
& \quad client_1[\,\ldots \mid !\textsc{Open}\ msg \mid userCr_x[\,\textsc{CloudClient} \mid !\textsc{Allow}\ output\,]\,]
\end{array}
$$

$\longrightarrow^*$ $cloud[\ \ldots \mid k[\ \ldots \mid q[\ \ldots \mid userCr_x[\ \ldots \mid allPlots[\ \ldots \mid$
$!(\nu\ l)(\ (o, client, a).msg[\ \textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.$
$\textsc{Out}\ cloud.\textsc{In}\ client.l[$
$\textsc{In}\ userCr_x.output[\ \textsc{Out}\ l.\langle o, a\rangle\ ]\ ]\ ]\ )\ ]\ ]\ ]\ ]\ ]\ \mid$
$client_1[\ \ldots \mid \underline{!\textsc{Open}\ msg} \mid userCr_x[\ \textsc{CloudClient} \mid !\textsc{Allow}\ output\ ] \mid$
$msg[\ l'_{unique}[$
$\textsc{In}\ userCr_x.output[\ \textsc{Out}\ l'_{unique}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}\prime\prime},\ outcome_{o_1^a}\rangle\ ]\ ]\ ]\ ]$

$\longrightarrow^*$ $cloud[\ \ldots \mid k[\ \ldots \mid q[\ \ldots \mid userCr_x[\ \ldots \mid allPlots[\ \ldots \mid$
$!(\nu\ l)(\ (o, client, a).msg[\ \textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.$
$\textsc{Out}\ cloud.\textsc{In}\ client.l[$
$\textsc{In}\ userCr_x.output[\ \textsc{Out}\ l.\langle o, a\rangle\ ]\ ]\ ]\ )\ ]\ ]\ ]\ ]\ ]\ \mid$
$client_1[\ \ldots \mid !\textsc{Open}\ msg \mid userCr_x[\ \textsc{CloudClient} \mid !\textsc{Allow}\ output\ ] \mid$
$l'_{unique}[\ \underline{\textsc{In}\ userCr_x.output[}\ \textsc{Out}\ l'_{unique}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}\prime\prime},\ outcome_{o_1^a}\rangle\ ]\ ]\ ]$

$\longrightarrow^*$ $cloud[\ \ldots \mid k[\ \ldots \mid q[\ \ldots \mid userCr_x[\ \ldots \mid allPlots[\ \ldots \mid$
$!(\nu\ l)(\ (o, client, a).msg[\ \textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.$
$\textsc{Out}\ cloud.\textsc{In}\ client.l[$
$\textsc{In}\ userCr_x.output[\ \textsc{Out}\ l.\langle o, a\rangle\ ]\ ]\ ]\ )\ ]\ ]\ ]\ ]\ ]\ \mid$
$client_1[\ \ldots \mid !\textsc{Open}\ msg \mid userCr_x[\ \textsc{CloudClient} \mid !\textsc{Allow}\ output \mid$
$l'_{unique}[\ output[\ \underline{\textsc{Out}\ l'_{unique}}.\langle \text{“}\mathtt{o}_1^{\mathtt{a}\prime\prime},\ outcome_{o_1^a}\rangle\ ]\ ]\ ]$

$\longrightarrow^*$ $cloud[\ \ldots \mid k[\ \ldots \mid q[\ \ldots \mid \underline{userCr_x}[\ \ldots \mid allPlots[\ \ldots \mid$
$!(\nu\ l)(\ (o, client, a).msg[\ \textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.$
$\textsc{Out}\ cloud.\textsc{In}\ client.l[$
$\textsc{In}\ userCr_x.output[\ \textsc{Out}\ l.\langle o, a\rangle\ ]\ ]\ ]\ )\ ]\ ]\ ]\ ]\ ]\ \mid$
$client_1[\ \ldots \mid !\textsc{Open}\ msg \mid userCr_x[\ \textsc{CloudClient} \mid \underline{!\textsc{Allow}\ output} \mid$
$\cancel{l'_{unique}[\ ]} \mid output[\ \langle \text{“}\mathtt{o}_1^{\mathtt{a}\prime\prime},\ outcome_{o_1^a}\rangle\ ]\ ]\ ]$

$\longrightarrow^*$ $cloud[\ \ldots \mid k[\ \ldots \mid q[\ \ldots \mid userCr_x[\ \ldots \mid allPlots[\ \ldots \mid$
$!(\nu\ l)(\ (o, client, a).msg[\ \textsc{Out}\ allPlots.\textsc{Out}\ userCr_x.\textsc{Out}\ q.\textsc{Out}\ k.$
$\textsc{Out}\ cloud.\textsc{In}\ client.l[$
$\textsc{In}\ userCr_x.output[\ \textsc{Out}\ l.\langle o, a\rangle\ ]\ ]\ ]\ )\ ]\ ]\ ]\ ]\ ]\ \mid$
$client_1[\ \ldots \mid !\textsc{Open}\ msg \mid userCr_x[\ \textsc{CloudClient} \mid \underline{!\textsc{Allow}\ output} \mid$
$\langle \text{“}\mathtt{o}_1^{\mathtt{a}\prime\prime},\ outcome_{o_1^a}\rangle\ ]\ ]$

Remark. It is inherently also possible in the model that the output is stored in the user area and used up by some later service operations locally (as an argument).

References

[1] Andreas Blass and Yuri Gurevich. Abstract State Machines Capture Parallel Algorithms: Correction and Extension. volume 9, pages 19:1–19:32, New York, NY, USA, June 2008. ACM.

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

[3] Egon Börger, Antonio Cisternino, and Vincenzo Gervasi. Ambient Abstract State Machines with Applications. *J.CSS (Special Issue in honor of Amir Pnueli)*, 78(3):939–959, May 2012.

[4] Károly Bósa. Formal Modeling of Mobile Computing Systems Based on Ambient Abstract State Machines. In *Special Issue of LNCS on SDKB 2011*, 2012. To appear.

[5] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[6] Marco Tulio de Oliveira Valente, Roberto da Silva Bigonha, Antonio Alfredo Ferreira Loureiro, and Marcelo de Almeida Maia. Abstractions for Mobile Computation in ASM. In Peter Graham and Muthucumaru Maheswaran, editors, *Proceedings of the International Conference on Internet Computing, IC 2000, Las Vegas, Nevada, USA, June 26-29, 2000*, pages 165–172. CSREA Press, 2000.

[7] Andrew D. Gordon and Luca Cardelli. Equational Properties of Mobile Ambients. *Mathematical. Structures in Comp. Sci.*, 13:371–408, June 2003.

[8] Yuri Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. volume 1, pages 77–111, New York, NY, USA, July 2000. ACM.

[9] Hui Ma, Klaus dieter Schewe, Bernhard Thalheim, and Qing Wang. Abstract State Services. In *Object-Oriented and Entity-Relationship Modelling/International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 406–415, 2008.

[10] Hui Ma, Klaus-Dieter Schewe, Bernhard Thalheim, and Qing Wang. Composing Personalised Services on top of Abstract State Services. In Lois Delcambre, Roland H. Kaschek, and Heinrich C. Mayr, editors, *The Evolution of Conceptual Modeling*, number 08181 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.