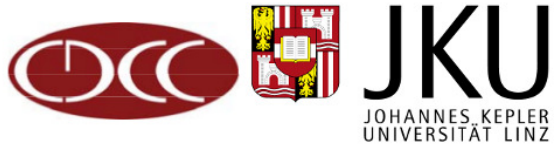


An Ambient ASM Specification of a Client-Centric Cloud Interaction Architecture

Károly Bósa[1] Klaus-Dieter Schewe[1,2]

[1]Christian Doppler Laboratory for Client-Centric Cloud Computing
Johannes-Kepler-University Linz, Austria
k.bosa|kd.schewe@cdcc.faw.jku.at

[2] Software Competence Center Hagenberg, Austria
kd.schewe@scch.at

October 16, 2012

The Problem

Engagement of a client in cloud computing means

- To shift data and functionality to a remote location – the cloud – which is maintained by the cloud provider
- Advantages:
  - Easy scalability, maintenance, virtuality, etc.
  - Pay per use
- Disadvantages:
  - The provider, communication and other third parties involved must be trusted
  - Risks of lock-in, security/privacy breeches, unavailability, inacceptable performance, etc.

The lack of trust (for various reasons) is the major obstacle in cloud computing

There is enough practical evidence that the risks are real

Cloud Service Architecture

Introduce a (client-owned) "layer" between the client and the cloud to handle the critical issues

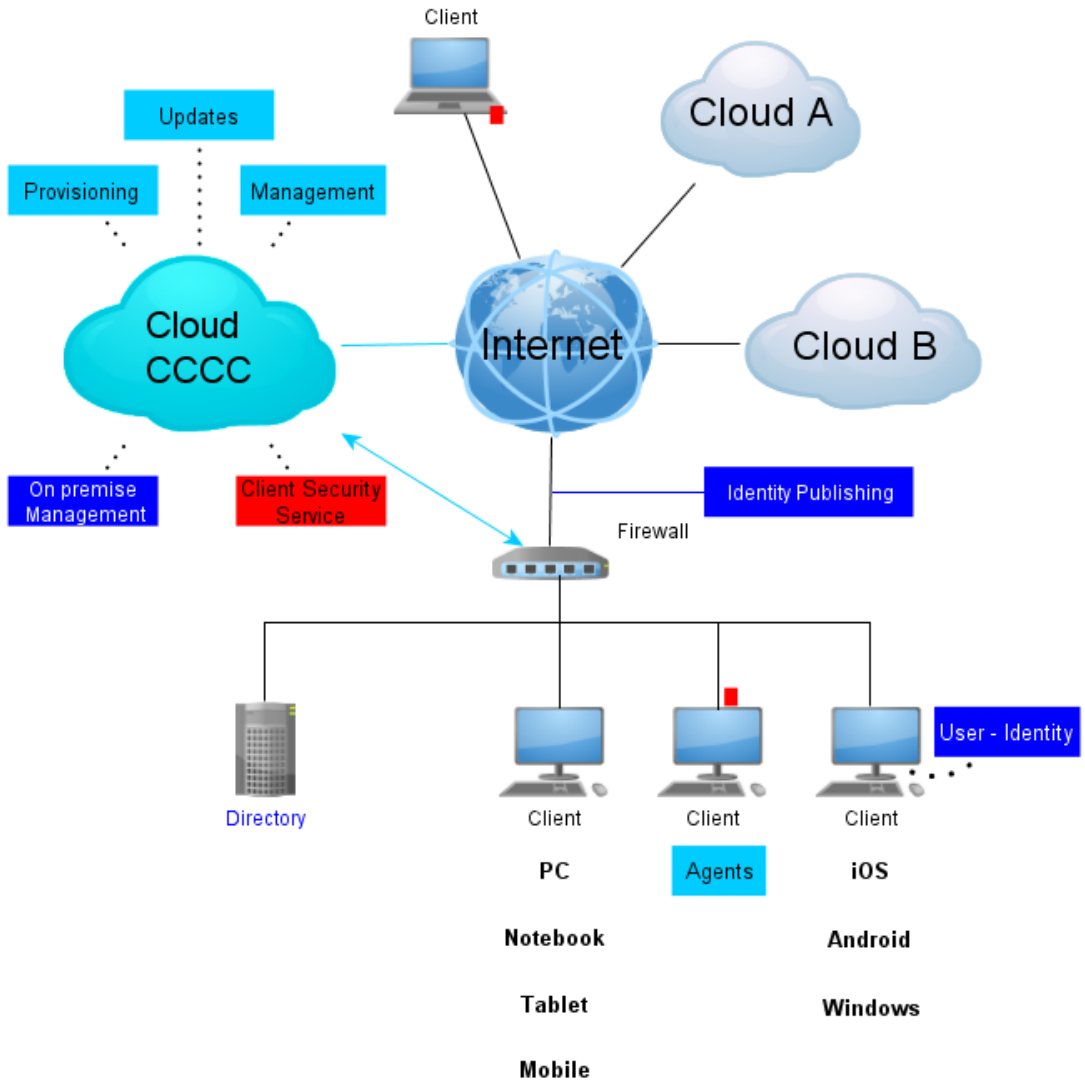
Our model gives a formal specification of a cloud service architecture in terms of ambient ASM:

- Spatial locations, mobility and some security considerations (e.g.: accessibility of certain resources) are described by a dynamically changing hierarchy of ambient constructs,
- Algorithmic functionalities are defined by ASM agents (which reside in various locations in the ambient hierarchy).

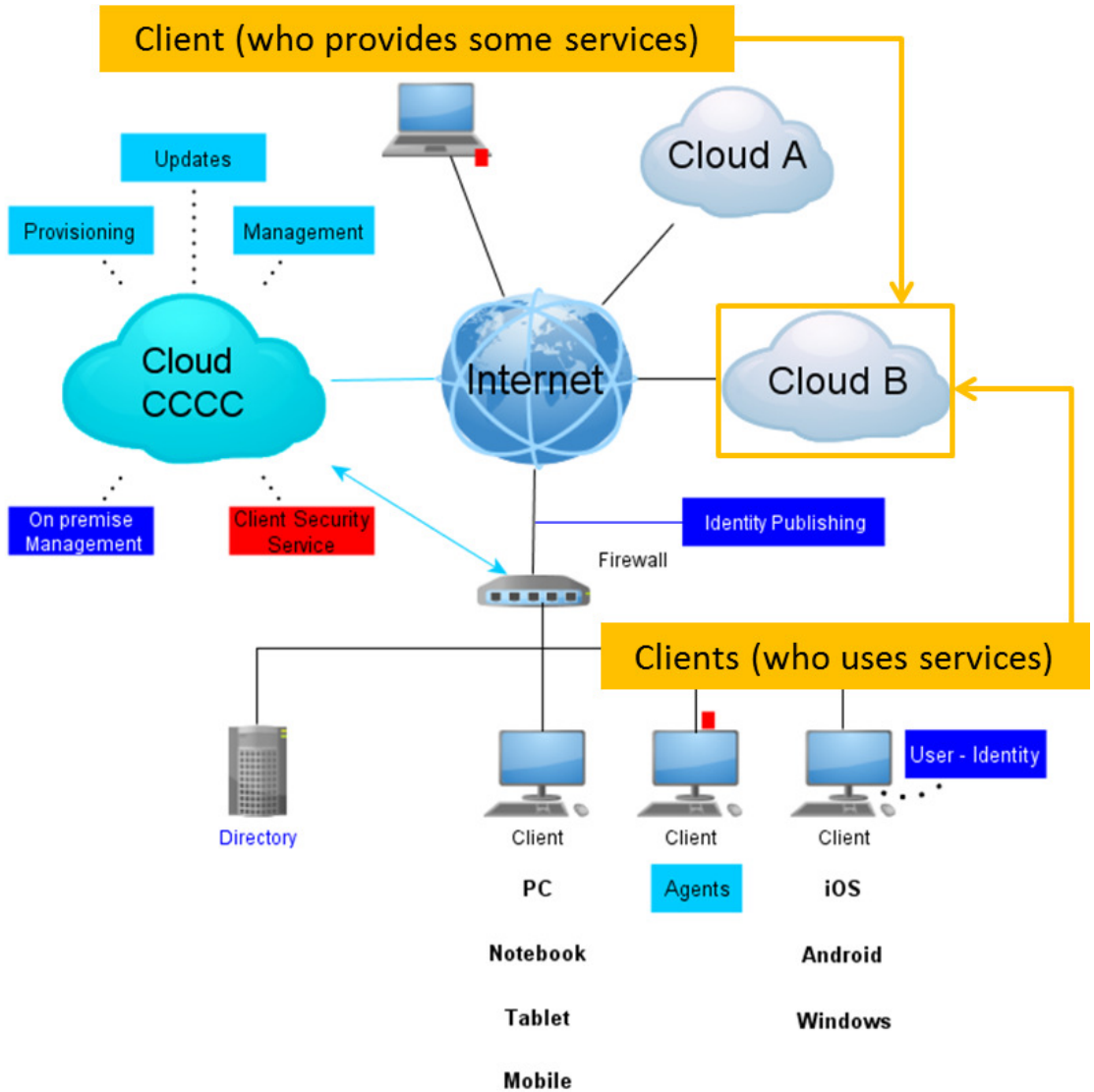
Our formal model provides the following new features:

- The cloud is able to adapt its services to heterogeneous client devices.
- The cloud architecture makes possible interaction between users and service owners (different from the cloud provider), such that each service owner controls the subscriptions and usages of her services.
- This is basic formal model which can be extended (e.g.: with identity and access control managements).

Putting Our Work into Context



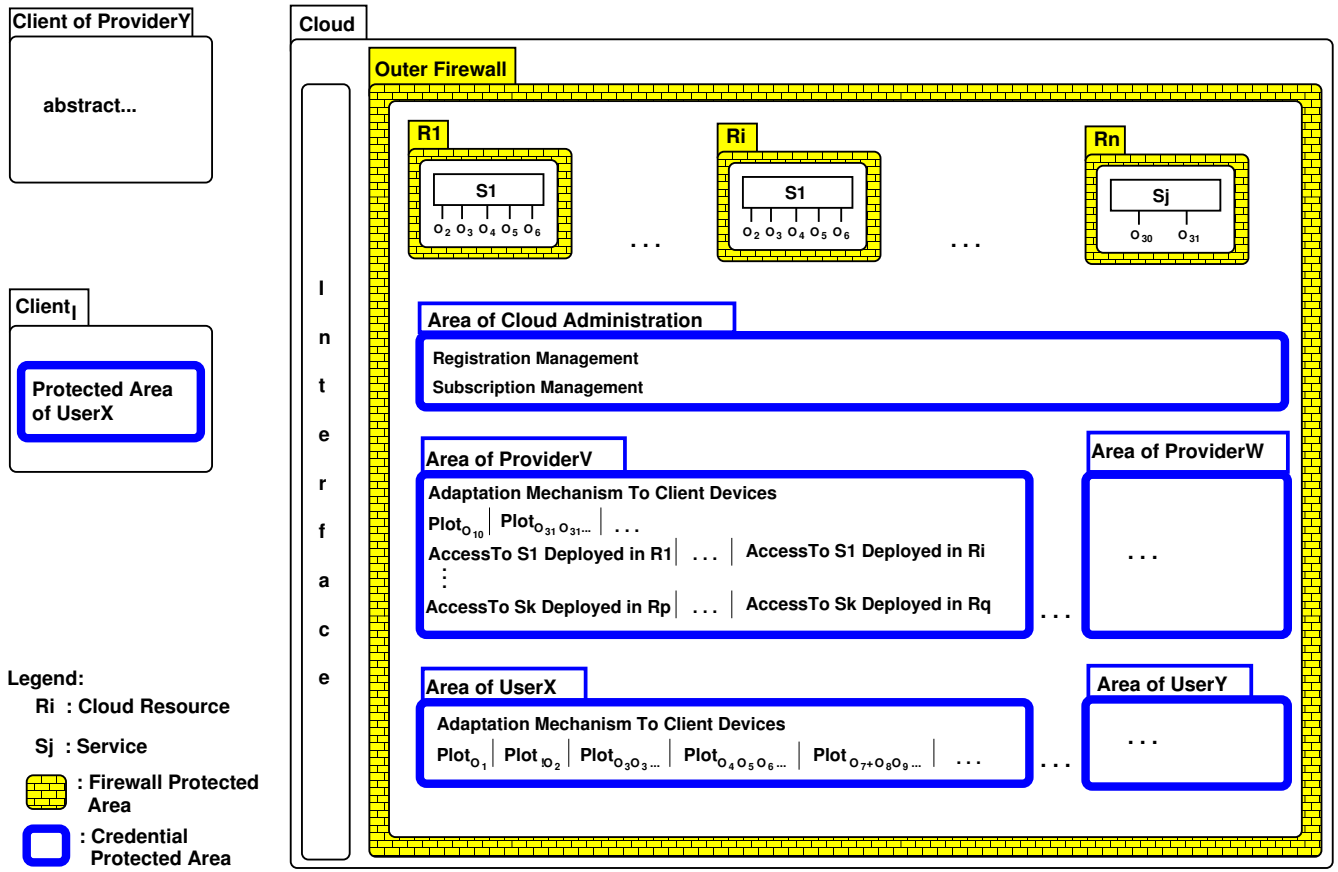
Putting Our Work into Context



Outline of the Talk

- A general and informal overview on our formal cloud model.
- A summary on the theoretical background based on ambient calculus:
  - Ambient calculus,
  - Definitions of some new non-basic ambient capabilities and
  - An abstraction of a multi-threaded server functionality.
- Our formal model of a cloud service architecture.
- Example: Processing of a Particular User Request in our model.
- Conclusions.

Overview 1/5: High-Level Structure of the Model



- For representing a service instance like SERVICE_i we adopt the formal model of *Abstract State Services (AS^{2}s)*.

2.1 Abstract State Services

- We first address the problem of service specification
- The key question is:

What is a *service*?

- Perusing the abundant literature on SOC we find that quite often no definition of service is given at all
- Instead, work deals with models that are intuitively accepted being "services", but no attempt is made to define services in general and without reference to a particular language (such as XML)
- We will develop the model of *Abstract State Services* (AS^2s) by means of postulates (in analogy to the definition of sequential / parallel algorithms in the seminal work by Y. Gurevich)

Initial Considerations

- Services have to deal with data, so it is no harm to consider first databases
- In particular, concentrate on the conceptual layer describing a database schema in an abstract way, and an external layer made out of views
- We complete this architecture by adding operations on both the conceptual and the external layer
- The former ones are handled as database transactions; the latter ones provide the means with which users can interact with a database
- This will lead us to a *database postulate* and an *extended view postulate*

The Database Layer

- Consider each database computation as a sequence of abstract states
- Each state represents the database at a certain point in time plus maybe additional data that is necessary for the computation, e.g. transaction tables, log files, etc.
- In order to capture the semantics of transactions we distinguish between a wide-step transition relation and small step transition relations
  - A transition in the former one marks the execution of a transaction, so the wide-step transition relation defines infinite sequences of transactions
  - Without loss of generality we can assume a serial execution, while of course interleaving is used for the implementation.
- Each transaction itself corresponds to a finite sequence of states resulting from the small step transition relation, which should then be subject to the postulates for database transformations

The Database Postulate

Postulate 1 (database postulate). A ***database system*** DBS consists of a set $\mathcal{S}$ of states, together with a subset $\mathcal{I} \subseteq \mathcal{S}$ of initial states, a wide-step transition relation $\tau \subseteq \mathcal{S} \times \mathcal{S}$, and a set $\mathcal{T}$ of transactions, each of which is associated with a small-step transition relation $\tau_t \subseteq \mathcal{S} \times \mathcal{S}$ $(t \in \mathcal{T})$ satisfying the postulates of a database transformation over $\mathcal{S}$.

- Write $DBS = (\mathcal{S}, \mathcal{I}, \tau, \{\tau_t \mid t \in \mathcal{T}\})$

- A *run* of a database system DBS is an infinite sequence $S_0, S_1, \ldots$ of states $S_i \in \mathcal{S}$ starting with an initial state $S_0 \in \mathcal{I}$ such that for all $i \in \mathbb{N}$ $(S_i, S_{i+1}) \in \tau$ holds, and there is a transaction $t_i \in \mathcal{T}$ with a finite run $S_i = S_i^0, \ldots, S_i^k = S_{i+1}$ such that $(S_i^j, S_i^{j+1}) \in \tau_{t_i}$ holds for all $j = 0, \ldots, k-1$.

Example

Let us consider a (very simplified) flight booking system:

- At its core it may use a database storing data about flights and bookings
  - relation FLIGHT with attributes flight_no, departure_date, departure_time, origin, and destination for the available flights
  - relation SEAT with attributes flight_no, departure_date, class, and number for the available seats per class in a flight
  - relation BOOKING with attributes booking_ref, flight_no, departure_date, class, customer_id for the already made bookings
  - more relations to capture customer data, status of bookings, etc.
- A state of the DBS would contain an instance of the relational database schema

Example / 2

- A booking transaction would change the state by adding further tuples to the booking relation
  - the number of seats booked for each class of each flight must not exceed the number of available seats
  - the booking transaction itself proceeds stepwise, and each step also changes the database, i.e. the state
  - a booking may be issued by a customer after receiving an answer to a query, e.g. asking for flight itineraries from a specified origin airport to a destination airport within a specified timeframe
    - the answer to such a query would be a set of itineraries, and each itinerary would be specified by a set of flight tuples stored in the database
    - thus, the state in which the booking transaction is started should also contain the set of itineraries, which is a view on top of the relational database

The View Layer

- If views are considered as part of states of a DBS, then transactions also affect them
- Views in general are expressed by queries, i.e. read-only database transformations
- Therefore, we can assume that a view on a database state $S_i \in \mathcal{S}$ is given by a finite run $S_0^v, \ldots, S_\ell^v$ of some database transformation v with $S_0^v = S_i$ and $S_i \subseteq S_\ell^v$
- Here we exploit that we can write a state of a database system as a set (later)
- Extend a database system by views: let each state $S \in \mathcal{S}$ to be composed as $S = S_d \cup V_1 \cup \cdots \cup V_k$ such that each $S_d \cup V_j$ is a view on S_d
- Then each wide-step state transition becomes a parallel composition of a transaction and an operation that "switches views on and off"

The Extended View Postulate

Postulate 2 (extended view postulate). An ***Abstract State Service*** **(AS^2)** consists of a database system DBS, in which each state $S \in \mathcal{S}$ is a finite composition $S_d \cup V_1 \cup \cdots \cup V_k$, and a finite set $\mathcal{V}$ of (extended) views.

Each view $v \in \mathcal{V}$ is associated with a database transformation such that for each state $S \in \mathcal{S}$ there are views $v_1, \ldots, v_k \in \mathcal{V}$ with finite runs $S_0^j, \ldots, S_{n_j}^j$ of v_j $(j = 1, \ldots, k)$, starting with $S_0^j = S_d$ and terminating with $S_{n_j}^j = S_d \cup V_j$.

Each view $v \in \mathcal{V}$ is further associated with a finite set $\mathcal{O}_v$ of (service) operations $o_1, \ldots, o_n$ such that for each $i \in \{1, \ldots, n\}$ and each $S \in \mathcal{S}$ there is a unique state $S' \in \mathcal{S}$ with $(S, S') \in \tau$.

Furthermore, if $S = S_d \cup V_1 \cup \cdots \cup V_k$ with V_i defined by v_i and o is an operation associated with v_k, then $S' = S'_d \cup V'_1 \cup \cdots \cup V'_m$ with $m \geq k - 1$, and V'_i for $1 \leq i \leq k - 1$ is still defined by v_i.

Notes on AS^{2}s

- In an AS^2 we have view-extended database states, and each service operation associated with a view induces a transaction on the database, and may change or delete the view it is associated with, and activate other views
- Therefore, talk of views that are *open* and those that are *closed*
- The service operations and the view generating queries are actually what is exported from the database system
  - what is exported can be very limited such as simple aggregation functions, in which case most of the data in the database would be hidden

Notes on AS^{2}s / 2

- the other extreme would be to export the complete database and define operations that take a query text as input and then process the query

- both extremes (and anything between them) are supported by the definition of AS^{2}s

- The abstract handling of service operations that induce transactions avoids the view update problem, which has to be taken into account when dealing with concrete specifications for AS^{2}s

Example (cont.)

- The booking operation in the previous example is a service operation
- It is associated with a view that produces a list of itineraries for given search criteria such as origin and destination, preferred class, departure time frame, etc.
- The induced transaction on the DBS updates the BOOKING relation
- Initial states for this database transformation can be any consistent database plus any set of open views
- The successor state (for the wide-step transition relation τ) would contain the updated database and the same set of views except the one containing the list of itineraries
- The latter one could be replaced by a view that simply contains a confirmation message for the selected and booked itinerary

2.3 Complete Languages for Abstract State Services

First get a variant of Abstract State Machines capturing all database transformations

An *Abstract Database Transformation Machine* (ADTM) $\mathcal{M}$ over signature Σ as in Postulate 4 and with a background as in Postulate 5 consists of

- a set $\mathcal{S}_\mathcal{M}$ of states over Σ satisfying the requirements in Postulate 4 and closed under isomorphisms,
- non-empty subsets $\mathcal{I}_\mathcal{M} \subseteq \mathcal{S}_\mathcal{M}$ of initial states, and $\mathcal{F}_\mathcal{M} \subseteq \mathcal{S}_\mathcal{M}$ of final states, both also closed under isomorphisms,
- a closed ADTM-rule $r_\mathcal{M}$ over Σ, and
- a binary relation $\tau_\mathcal{M}$ over $\mathcal{S}_\mathcal{M}$ determined by $r_\mathcal{M}$ such that we have

$$\{S_{i+1} \mid (S_i, S_{i+1}) \in \tau_\mathcal{M}\} = \{S_i + \Delta \mid \Delta \in \Delta(r_\mathcal{M}, S_i)\}$$

ADTM Rules

The set $\mathcal{R}$ of *ADTM-rules* over a signature $\Sigma = \Sigma_{db} \cup \Sigma_a \cup \{f_1, \ldots, f_\ell\}$ is defined as follows:

- If $t_0, \ldots, t_n$ are terms over Σ, and f is an n-ary function symbol in Σ, then $f(t_1, \ldots, t_n) := t_0$ is a rule r in $\mathcal{R}$ called *assignment rule*

- If φ is a Boolean term and $r' \in \mathcal{R}$ is an ADTM-rule, then **if** φ **then** r' **endif** is a rule r in $\mathcal{R}$ called *conditional rule*

- If φ is a Boolean term with only database variables $fr(\varphi) = \{x_1, \ldots, x_k\}$ and $r' \in \mathcal{R}$ is an ADTM-rule, then **forall** $x_1, \ldots, x_k$ **with** φ **do** r' **enddo** is a rule r in $\mathcal{R}$ called *forall rule*

- If $r_1, \ldots, r_n$ are rules in $\mathcal{R}$, then also the rule r defined as **par** $r_1 \| \ldots \| r_n$ **par** is a rule in $\mathcal{R}$, called *parallel rule*

ADTM Rules (cont.)

- If φ is a Boolean term with only database variables $fr(\varphi) = \{x_1, \ldots, x_k\}$ and $r' \in \mathcal{R}$ is an ADTM-rule, then **choose** $x_1, \ldots, x_k$ **with** φ **do** r' **enddo** is a rule r in $\mathcal{R}$ called *choice rule*

- If $r_1, \ldots, r_n$ are rules in $\mathcal{R}$, then also the rule r defined as **seq** r_1 ; ... ; r_n **seq** is a rule in $\mathcal{R}$, called *sequence rule*

- If $r' \in \mathcal{R}$ is an ADTM-rule and ϑ is a location function that assigns location operators ϱ to terms γ with $var(\gamma) \subseteq fr(r')$, then **let** $\vartheta(\gamma) = \varrho$ **in** r' **endlet** defines another ADTM-rule $r \in \mathcal{R}$ called *let rule*

The definition of sets of update (multi)sets $\Delta(r, S)$ (and $\ddot{\Delta}(r, S)$, resp.) is straightforward

Capturing the Database Layer with ADTMs

- Adapt ADTMs to specify the database layer by
  - a background class specifying additional base types, each associated with a base domain, constructor symbols and function symbols associated with these constructors,
  - a signature comprising function symbols for the database and algorithmic parts of states, and for the bridge functions,
  - a set of initial states for the database system,
  - a set of transactions, each of which will be defined by an ADTM-rule, and
  - a set of auxiliary ADTM-rules

DBS Specifications

- On these grounds the definition of database system specifications is straightforward:

- A *database system specification* DBSS over a background specification $\mathcal{BS}$ consists of

  - a signature Σ over $\mathcal{BS}$ fulfilling the requests of the Abstract State Postulate,

  - a set $\mathcal{I}$ of states over Σ called initial states of DBSS that is closed under isomorphisms,

  - a finite set $\mathcal{T}$ of parameterised transactions, each of which is defined by an ADTM rule with free variables equal to the parameters, and

  - a finite set $\mathcal{A}$ of auxiliary rules defined in the same way as $\mathcal{T}$

Capturing the View Layer with ADTMs

- On top of such specification of a database system we define the view layer by a set of extended views. Each view is defined by
  - a signature defined similarly to the signature for the underlying database system,
  - a defining query that is defined by another ADTM-rule possibly using auxiliary ADTM-rules, and
  - a set of operations that are specified similar to transactions, but in addition include details on how to handle views.
- While such a definition captures all AS^{2}s, it does not exploit declarative query languages
- Therefore, in a second step we extend the language by adding "syntactic sugar", e.g. declarative query expressions taken from a complete fixed-point query language (such as XIQL – not handled here)

Extended View Specifications

- If we extend the signature by adding database function symbols and bridge functions to obtain the extended signature $\Sigma_{ext} = \Sigma'_{db} \cup \Sigma_a \cup \{f_1, \ldots, f_k\}$, the added function symbols, i.e. $\Sigma'_{db} - \Sigma_{db} \cup \{f_{n+1}, \ldots, f_k\}$ define a *view signature*, denoted as Σ_v.
- An ADTM-rule r_v over the extended schema Σ_{ext} will be called a *query* over Σ, iff the input database is preserved, i.e. the following two conditions must be satisfied:
  - For all state pairs (S, S') produced by r_v, i.e. there is a finite run $S_0, \ldots, S_\ell$ of r_v with initial state $S_0 = S$ and final state $S_\ell = S'$ such that the restrictions of S and S' to Σ coincide
  - For all state pairs (S_1, S'_1) and (S_2, S'_2) produced by r_v such that the restrictions of S_1 and S_2 to Σ coincide we have $S'_1 = S'_2$
- If DBSS is a database system specification with signature Σ, then a *view* v over DBSS is defined by a view signature Σ_v over Σ and a defining query r_v over $\Sigma \cup \Sigma_v$

Overview 2/5: Service Plots

o_i	one-off usage of a single operation
$o_i o_j \ldots o_k$	one-off usage of a sequence of operations
$o_i + o_j + \ldots + o_k$	one-off usage of a choice among some operations
o_i^*	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[1]

- A *plot* is a high-level specification of an action scheme.
- For an algebraic formalization of plots in case of AS^{2}s it is possible to exploit *Kleene algebras with tests (KATs)*.
- Each service operation has a unique name (for instance, in one interpretation each service operation has the full name like *serviceId.operationId*).

[1] $o_i \mid o_j$ is simulated with $o_i o_j + o_j o_i$ in KATs, so in fact we are interested in interleaved services operations, when we talk of parallel plots.

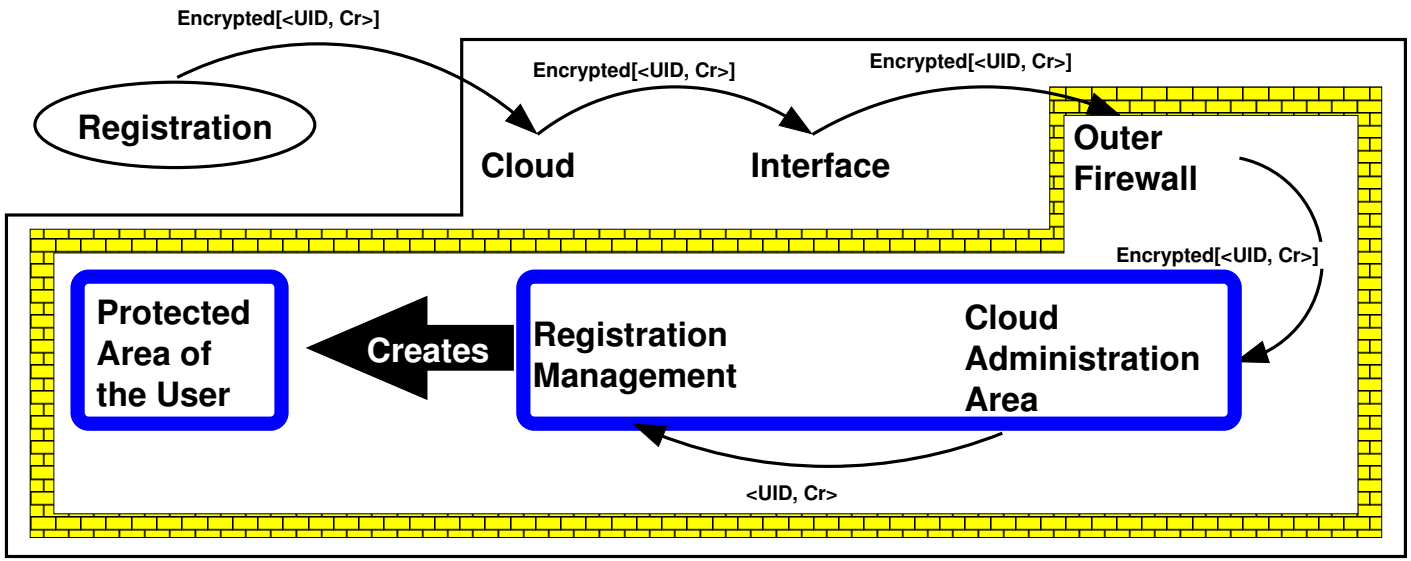
Specification of Service Operations

- *Selection conditions* are Boolean terms that can be evaluated on structures over Σ_v and thus define substructures
- If φ is a selection condition, we permit the use of *restriction terms* $t[\varphi]$
- A *v-rule* over view v with selection condition φ is given by a parametrised ADTM-rule without assignments, but with the possibility to
  - use restriction terms
  - open views by means of rules $open(v')$ for $v' \neq v$, and
  - close the view v using the rule $close(v)$

Specification of Service Operations / 2

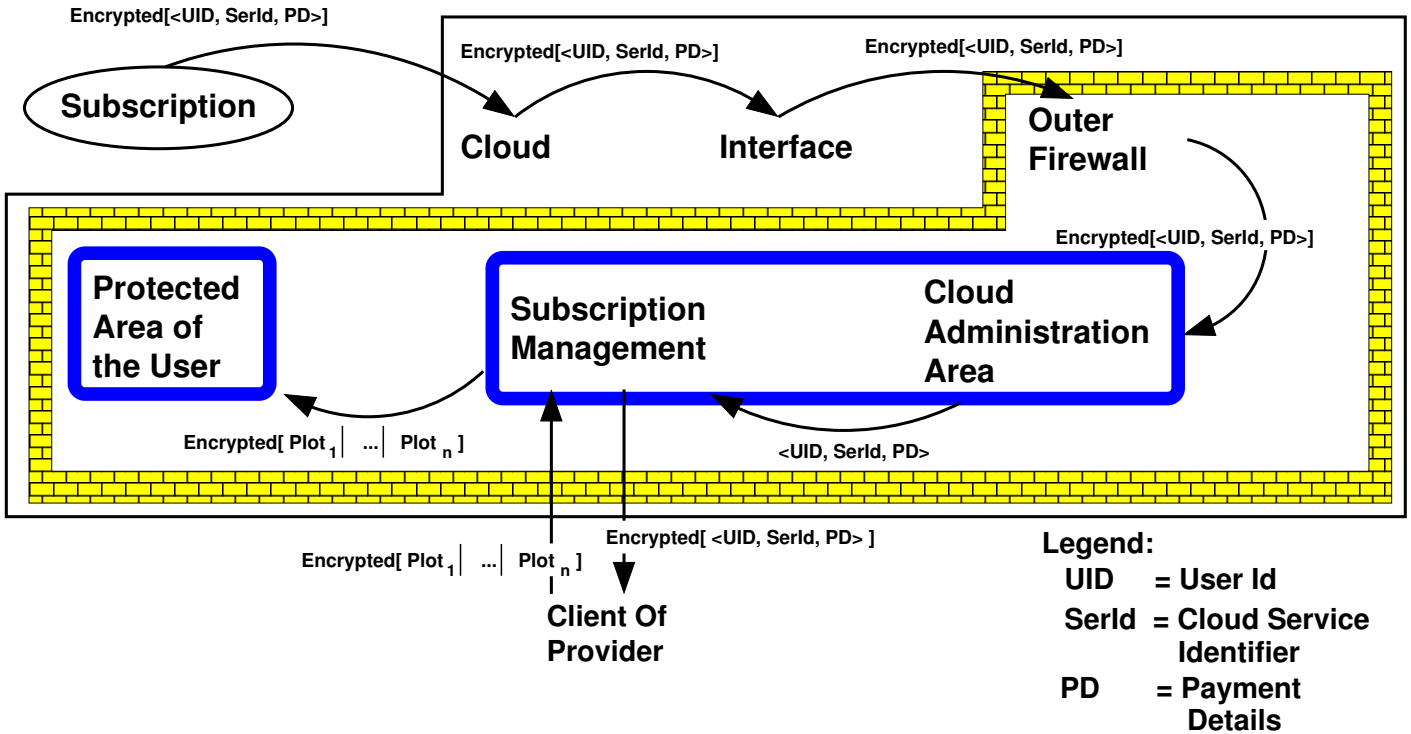
- Opening a view v means to initiate the functions in the corresponding view signature Σ_v
- Closing it can be expressed simply by letting all functions in Σ_v be totally undefined
- The use of restriction terms can be replaced by using conditional rules with the term φ
- An *extended view* over a database system specification DBSS consists of a view v over DBSS and a set $\mathcal{O}_v$ of v-rules over v
- An *Abstract State Service Specification* (A3S) $\mathcal{AS}$ consists of a database system specification DBSS and a set $\mathcal{V}$ of extended views over DBSS

Overview 3/5: User Registration



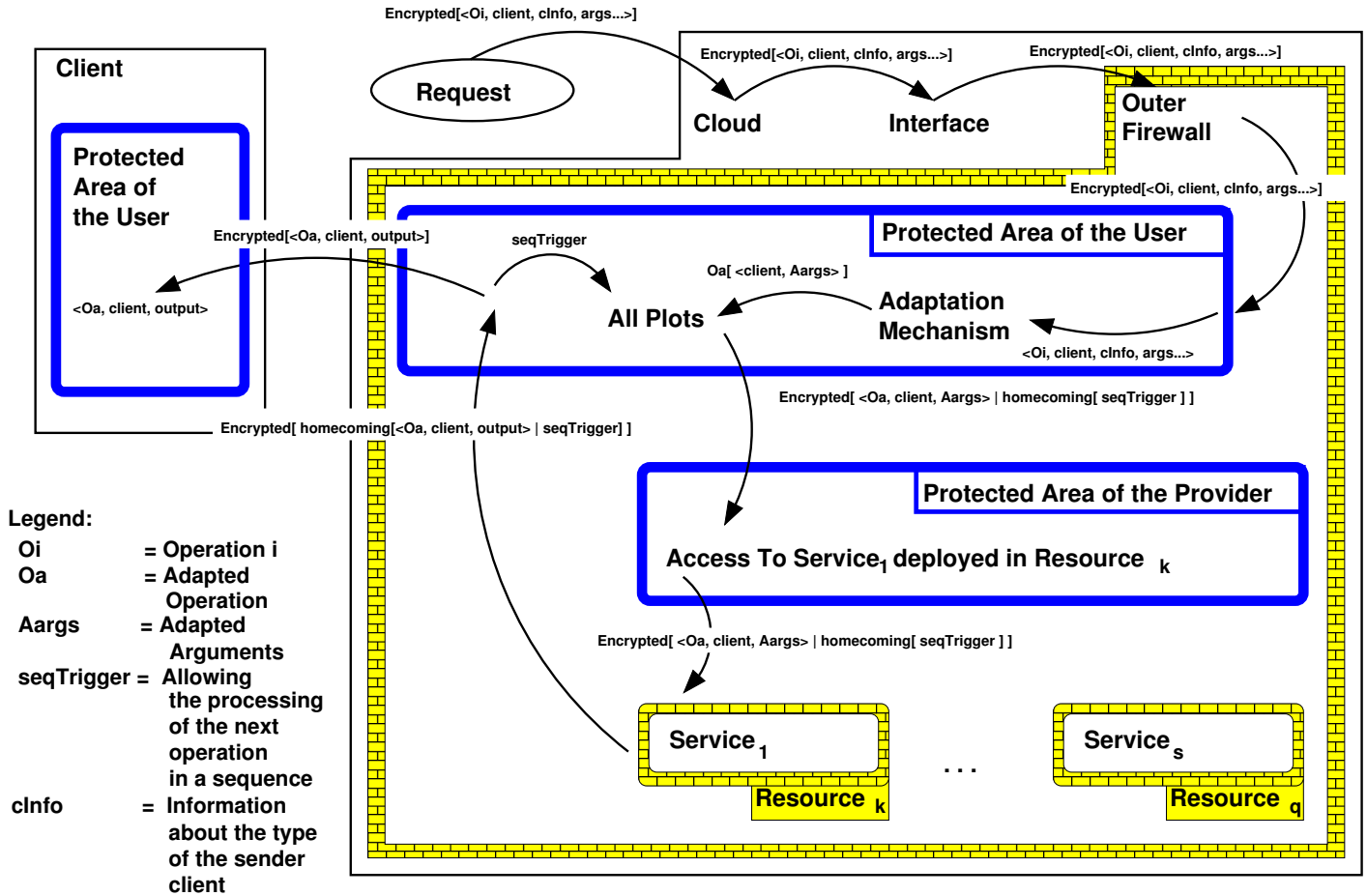
Legend:
UID = User Id
Cr = Credential

Overview 4/5: Subscription to a Service



- Adaptation rules (applied by the adaptation mechanism) may be provided by the service provider as well (this is the current interpretation in Maria's implementation).

Overview 5/5: A User Request



Ambient Calculus Summary 1/4

The concept of *ambient* has the following main characteristics:

- 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 it).

Ambient Calculus Summary 2/4

$P, Q, R ::=$	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)

- 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).
- A reduction relation $P \longrightarrow Q$ describes the evolution of a term P into a new term Q (and $\longrightarrow^*$ denotes a reflexive and transitive reduction relation).

Ambient Calculus Summary 3/4

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:

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

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

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

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.

Ambient Calculus Summary 4/4

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

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

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

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

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

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

Combining Ambient Calculus with ASMs. The simple key idea due to E. Börger is that the processes are to be specified by ASMs

Definitions of Non-Basic Capabilities 1/8

- n BE $m.P \equiv$ //It differs from Cardelli's definition.
  $(\nu\ s)(s[\ \text{OUT}\ n \mid m[\ \text{OPEN}\ n.\text{OUT}\ s.P\ ]\ ] \mid \text{IN}\ s.\text{IN}\ m)$

$$
\begin{array}{rl}
\boxed{n[\ n\ \text{BE}\ m.P \mid Q\ ]} & \\
= & (\nu\ s)(n[\ s[\ \underline{\text{OUT}\ n} \mid m[\ \text{OPEN}\ n.\text{OUT}\ s.P\ ]\ ] \mid \text{IN}\ s.\text{IN}\ m \mid Q\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(s[\ m[\ \text{OPEN}\ n.\text{OUT}\ s.P\ ]\ ] \mid n[\ \underline{\text{IN}\ s.\text{IN}\ m} \mid Q\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(s[\ m[\ \underline{\text{OPEN}\ n}.\text{OUT}\ s.P \mid n[\ Q\ ]\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(s[\ m[\ \underline{\text{OUT}\ s}.P \mid Q\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ s)(m[\ P \mid Q\ ] \mid s[\ ]\ ) \\
\approx & \boxed{m[\ P \mid Q\ ]}
\end{array}
$$

Definitions of Non-Basic Capabilities 2/8

- See $n.P \equiv$
  $(\nu\ r, s)(\ r[\ \text{In}\ n.\text{Out}\ n.r\ \text{Be}\ s.P\ ]\ |\ \text{Open}\ s\ )$

$\boxed{n[\ ]\ |\ \text{See}\ n.P}$

$$
\begin{array}{rl}
= & (\nu\ r, s)(\ n[\ ]\ |\ r[\ \underline{\text{In}\ n}.\text{Out}\ n.r\ \text{Be}\ s.P\ ]\ |\ \text{Open}\ s\ ) \\
\longrightarrow^* & (\nu\ r, s)(\ n[\ r[\ \underline{\text{Out}\ n}.r\ \text{Be}\ s.P\ ]\ ]\ |\ \text{Open}\ s\ ) \\
\longrightarrow^* & (\nu\ r, s)(\ n[\ ]\ |\ r[\ \underline{r\ \text{Be}\ s}.P\ ]\ |\ \text{Open}\ s\ ) \\
\longrightarrow^* & (\nu\ s)(\ n[\ ]\ |\ s[\ P\ ]\ |\ \underline{\text{Open}\ s}\ ) \\
\longrightarrow^* & \boxed{n[\ ]\ |\ P}
\end{array}
$$

Definitions of Non-Basic Capabilities 3/8

- m WRAP $n.P \equiv$
  $(\nu\ s, r)(\ s[\ \text{OUT}\ n.\text{SEE}\ n.s\ \text{BE}\ m.r[\ \text{IN}\ n\ ]\ ]\ |\ \text{IN}\ s.\text{OPEN}\ r.P\ )$

$\boxed{n[\ m\ \text{WRAP}\ n.P\ ]}$

$$
\begin{aligned}
&= && (\nu\ s, r)(\ n[\ s[\ \underline{\text{OUT}\ n}.\text{SEE}\ n.s\ \text{BE}\ m.r[\ \text{IN}\ n\ ]\ ]\ | \\
& && \text{IN}\ s.\text{OPEN}\ r.P\ ]\ ) \\
&\longrightarrow^* && (\nu\ s, r)(\ s[\ \text{SEE}\ n.s\ \text{BE}\ m.r[\ \text{IN}\ n\ ]\ ]\ |\ n[\ \underline{\text{IN}\ s}.\text{OPEN}\ r.P\ ]\ ) \\
&\longrightarrow^* && (\nu\ s, r)(\ s[\ \underline{\text{SEE}\ n.s}\ \text{BE}\ m.r[\ \text{IN}\ n\ ]\ |\ n[\ \text{OPEN}\ r.P\ ]\ ]\ ) \\
&\longrightarrow^* && (\nu\ s, r)(\ s[\ \underline{s\ \text{BE}\ m}.r[\ \text{IN}\ n\ ]\ |\ n[\ \text{OPEN}\ r.P\ ]\ ]\ ) \\
&\longrightarrow^* && (\nu\ r)(\ m[\ r[\ \underline{\text{IN}\ n}\ ]\ |\ n[\ \text{OPEN}\ r.P\ ]\ ]\ ) \\
&\longrightarrow^* && (\nu\ r)(\ m[\ n[\ r[\ ]\ |\ \underline{\text{OPEN}\ r}.P\ ]\ ]\ ) \\
&\longrightarrow^* && \boxed{m[\ n[\ P\ ]\ ]}
\end{aligned}
$$

Definitions of Non-Basic Capabilities 4/8

- ALLOW *key* ≡ OPEN *key*
- *n* DRAWIN$_{key}$ *m*.*P* ≡
  key[OUT *n*.IN *m*.IN *n*] | OPEN *m*.*P*

$$
\begin{array}{rl}
\boxed{m[\ Q \mid \text{ALLOW } key\ ] \mid n[\ n\ \text{DRAWIN}_{key}\ m.P\ ]} & \\
= & m[\ Q \mid \text{ALLOW } key\ ] \mid n[\ key[\ \underline{\text{OUT } n}.\text{IN } m.\text{IN } n\ ] \mid \\
& \text{OPEN } m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \text{ALLOW } key\ ] \mid key[\ \underline{\text{IN } m}.\text{IN } n\ ] \mid n[\ \text{OPEN } m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\text{ALLOW } key} \mid key[\ \text{IN } n\ ]\ ] \mid n[\ \text{OPEN } m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\text{IN } n}\ ] \mid n[\ \text{OPEN } m.P\ ] \\
\longrightarrow^* & n[\ m[\ Q\ ] \mid \underline{\text{OPEN } m}.P\ ] \\
\longrightarrow^* & \boxed{n[\ Q \mid P\ ]}
\end{array}
$$

Definitions of Non-Basic Capabilities 5/8

- n DrawIn$_{key}$ m ThenRelease $lock.P \equiv$
  $key[$ Out n.In m.In n $] \mid$ See $m.lock$ Wrap n.Open $m.P$

$$
\begin{array}{rl}
\boxed{m[\ Q \mid \text{Allow } key\ ] \mid n[\ \text{DrawIn}_{key}\ m\ \text{ThenRelease } lock.P\ ]} & \\
= & m[\ Q \mid \text{Allow } key\ ] \mid n[\ key[\ \underline{\text{Out } n}.\text{In } m.\text{In } n\ ] \mid \\
 & \text{See } m.lock\ \text{Wrap } n.\text{Open } m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \text{Allow } key\ ] \mid key[\ \underline{\text{In } m}.\text{In } n\ ] \mid \\
 & n[\ \text{See } m.lock\ \text{Wrap } n.\text{Open } m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\text{Allow } key} \mid key[\ \text{In } n\ ]\ ] \mid \\
 & n[\ \text{See } m.lock\ \text{Wrap } n.\text{Open } m.P\ ] \\
\longrightarrow^* & m[\ Q \mid \underline{\text{In } n}\ ] \mid n[\ \text{See } m.lock\ \text{Wrap } n.\text{Open } m.P\ ] \\
\longrightarrow^* & n[\ m[\ Q\ ] \mid \underline{\text{See } m.lock}\ \text{Wrap } n.\text{Open } m.P\ ] \\
\longrightarrow^* & n[\ m[\ Q\ ] \mid \underline{lock\ \text{Wrap } n}.\text{Open } m.P\ ] \\
\longrightarrow^* & lock[\ n[\ m[\ Q\ ] \mid \underline{\text{Open } m}.P\ ]\ ] \\
\longrightarrow^* & \boxed{lock[\ n[\ Q \mid P\ ]\ ]} \qquad // \ldots \mid \text{Open } lock.R^2
\end{array}
$$

[2] The capability Open is used to encode locks. Such a lock can be released with an ambient whose name corresponds with the target of the Open.

Definitions of Non-Basic Capabilities 6/8

- $\textsc{Server}^{n}_{key}\ m.P \equiv$
  $(\nu\ next)(next[\ ]\ |$
  $!(\nu\ n)(\textsc{Open}\ next.n[$
  $n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ )$

$$\boxed{\overbrace{m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ]}^{k\ \text{pieces of}\ m} \mid \textsc{Server}^{n}_{key}\ m.P}$$

$$\begin{aligned}
= \quad & m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
& (\nu\ next)(next[\ ] \mid \underline{!\ (\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ )\ ) \\
\longrightarrow^* \quad & m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
& (\nu\ next)(next[\ ] \mid !(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \mid \\
& \quad \underline{\textsc{Open}\ next}.n_1^{unique}[\ n_1^{unique}\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \\
\longrightarrow^* \quad & m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
& (\nu\ next)(!(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \mid \\
& \quad n_1^{unique}[\ \underline{n_1^{unique}\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P}\ ]\ ) \\
\longrightarrow^* \quad & \overbrace{m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ]}^{k-1\ \text{pieces of}\ m} \mid \\
& (\nu\ next)(\underline{!\ (\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \mid \\
& \quad next[\ n_1^{unique}[\ Q_i \mid P\ ]\ ]\ )
\end{aligned}$$

Definitions of Non-Basic Capabilities 6/8 (cont.)

$$
\begin{aligned}
&\longrightarrow^* \quad \overbrace{m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ]}^{k-1 \text{ pieces of } m} \mid \\
&\qquad (\nu\ next)(\ !\ \underline{(\nu\ n)}(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \mid \\
&\qquad\quad next[\ n_1^{unique}[\ Q_i \mid P\ ]\ ]\ ) \\
&\longrightarrow^* \quad m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
&\qquad (\nu\ next)(!(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \mid \\
&\qquad\quad \underline{\textsc{Open}\ next}.n_2^{unique}[\ n_2^{unique}\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ] \mid \\
&\qquad\quad next[\ n_1^{unique}[\ Q_i \mid P\ ]\ ]\ ) \\
&\longrightarrow^* \quad m[\ Q_1 \mid \textsc{Allow}\ key\ ] \mid \ldots \mid m[\ Q_k \mid \textsc{Allow}\ key\ ] \mid \\
&\qquad (\nu\ next)(!(\nu\ n)(\textsc{Open}\ next.n[\ n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]\ ) \mid \\
&\qquad\quad n_2^{unique}[\ \overline{n_2^{unique}\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P}\ ] \mid \\
&\qquad\quad n_1^{unique}[\ Q_i \mid P\ ]\ ) \\
&\longrightarrow^* \quad \ldots \\
&\longrightarrow^* \quad \boxed{\textsc{Server}_{key}^{n}\ m.P \mid \underbrace{n_1^{unique}[\ Q_i \mid P\ ] \mid \ldots \mid n_k^{unique}[\ Q_j \mid P\ ]}_{\text{each } m \text{ is captured by a replica of } n}}
\end{aligned}
$$

Definitions of Non-Basic Capabilities 7/8

- $\text{RNDAssign}^{n}_{key}\ m.(P_1 + \ldots + P_r) \equiv$
  $(\nu\ next)(next[\ ] \mid$
  $!(\nu\ n)(\text{Open}\ next.n[$
  $n\ \text{DrawIn}_{key}\ m\ \text{ThenRelease}\ next.P_1\ ]\ ) \mid$
  $\vdots$
  $\mid\ !(\nu\ n)(\text{Open}\ next.n[$
  $n\ \text{DrawIn}_{key}\ m\ \text{ThenRelease}\ next.P_r\ ]\ )\ )$[3]

[3] It is very similar to the capability $\text{Server}^{n}_{key}\ m.P$.

Definitions of Non-Basic Capabilities 8/8

- n CHOICE$_{key}$ $m_1.P_1 + \ldots + m_k.P_k \equiv$
  $(\nu\ trap, r, t)($
  $trap[$ OUT $n.key[$ OPEN r.OUT $trap$.IN n] |
  IN $m_1.r[$ IN $key.m_1$ BE t.OPEN $t.P_1$] |
  .
  .
  .
  IN $m_k.r[$ IN $key.m_k$ BE t.OPEN $t.P_k$]] |
  OPEN $t.t[$]

$$\boxed{m_2[\ Q \mid \text{ALLOW } key\ ] \mid n[\ n \text{ CHOICE}_{key}\ m_1.P_1 + m_2.P_2\ ]}$$

$$
\begin{aligned}
= \quad & (\nu\ trap, r, t)(\ m_2[\ Q \mid \text{ALLOW } key\ ] \mid \\
& \quad n[\ \ trap[\ \underline{\text{OUT } n}.key[\ \text{OPEN } r.\text{OUT } trap.\text{IN } n\ ] \mid \\
& \qquad \text{IN } m_1.r[\ \text{IN } key.m_1 \text{ BE } t.\text{OPEN } t.P_1\ ] \mid \\
& \qquad \text{IN } m_2.r[\ \text{IN } key.m_2 \text{ BE } t.\text{OPEN } t.P_2\ ]\ ]\ \mid \\
& \quad \text{OPEN } t.t[\ ]\ ]\ ) \\
\longrightarrow^* \quad & (\nu\ trap, r, t)(\ m_2[\ Q \mid \text{ALLOW } key\ ] \mid \\
& \quad trap[\ key[\ \text{OPEN } r.\text{OUT } trap.\text{IN } n\ ] \mid \\
& \qquad \text{IN } m_1.r[\ \text{IN } key.m_1 \text{ BE } t.\text{OPEN } t.P_1\ ] \mid \\
& \qquad \underline{\text{IN } m_2}.r[\ \text{IN } key.m_2 \text{ BE } t.\text{OPEN } t.P_2\ ]\ ]\ \mid \\
& \quad n[\ \text{OPEN } t.t[\ ]\ ]\ ) \\
\longrightarrow^* \quad & (\nu\ trap, r, t)(\ m_2[\ Q \mid \text{ALLOW } key \mid \\
& \quad trap[\ key[\ \text{OPEN } r.\text{OUT } trap.\text{IN } n\ ] \mid \\
& \qquad \text{IN } m_1.r[\ \text{IN } key.m_1 \text{ BE } t.\text{OPEN } t.P_1\ ] \mid \\
& \qquad r[\ \underline{\text{IN } key}.m_2 \text{ BE } t.\text{OPEN } t.P_2\ ]\ ]\ ]\ \mid \\
& \quad n[\ \text{OPEN } t.t[\ ]\ ]\ )
\end{aligned}
$$

Definitions of Non-Basic Capabilities 8/8 (cont.)

$$
\begin{array}{ll}
\longrightarrow^* & (\nu\ trap,\ r,\ t)(\ m_2[\ Q \mid \textsc{Allow}\ key \mid \\
& \quad trap[\ key[\ \underline{\textsc{Open}\ r}.\textsc{Out}\ trap.\textsc{In}\ n \mid r[\ m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ ]\ \mid \\
& \quad\quad \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ \mid \\
& \quad n[\ \textsc{Open}\ t.t[\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ trap,\ r,\ t)(\ m_2[\ Q \mid \textsc{Allow}\ key \mid \\
& \quad trap[\ key[\ \underline{\textsc{Out}\ trap}.\textsc{In}\ n \mid m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ \mid \\
& \quad\quad \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ \mid \\
& \quad n[\ \textsc{Open}\ t.t[\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ trap,\ r,\ t)(\ m_2[\ Q \mid \underline{\textsc{Allow}\ key}\ \mid \\
& \quad key[\ \textsc{In}\ n \mid m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2\ ]\ \mid \\
& \quad trap[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ \mid \\
& \quad n[\ \textsc{Open}\ t.t[\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ trap,\ r,\ t)(\ m_2[\ Q \mid \underline{\textsc{In}\ n} \mid \underline{m_2\ \textsc{Be}\ t.\textsc{Open}\ t.P_2} \mid trap[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ \mid \\
& \quad n[\ \textsc{Open}\ t.t[\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ trap,\ r,\ t) \\
& \quad (\ n[\ \underline{\textsc{Open}\ t}.t[\ ]\ \mid t[\ Q \mid \textsc{Open}\ t.P_2 \mid trap[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ trap,\ r,\ t) \\
& \quad (\ n[\ t[\ ]\ \mid Q \mid \underline{\textsc{Open}\ t.P_2} \mid trap[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ ) \\
\longrightarrow^* & (\nu\ trap,\ r,\ t) \\
& \quad (\ n[\ Q \mid P_2 \mid trap[\ \textsc{In}\ m_1.r[\ \textsc{In}\ key.m_1\ \textsc{Be}\ t.\textsc{Open}\ t.P_1\ ]\ ]\ ]\ ) \\
\approx & \quad \boxed{n[\ Q \mid P_2\ ]}^4
\end{array}
$$

[4] We assume that the bound names *trap*, *r* and *t* do not occur in $P_1, \ldots, P_k$.

System Specification

$$curAmbProc := root[\ Cloud \mid Client_1 \mid \ldots \mid Client_n\ ]^5.$$

- We assume that there are some standardized public ambient names, which are known by all contributors. We distinguish the following kinds of public names: addresses (e.g.: *cloud*, $client_1$, ..., $client_n$), message types (e.g.: *reg*(*istration*), *request*, *subs*(*cription*), *output*) and parts of some common protocols (e.g.: *lock*, *msg*, *intf*, *access*, *out*, o_1, ..., o_s, *op*). All other ambient names are non-public in the model.

[5] The ambient called *root* is a special ambient which is required for the ASM definition of ambient calculus

User Actions

$$RegistrationMsg \equiv msg[\ \textsc{In}\ cloud.\textsc{Allow}\ intf.reg[\ \textsc{Allow}\ cloudId.\langle UID_x \rangle\ ]\ ]$$

$$SubscriptionMsg \equiv msg[\ \textsc{In}\ cloud.\textsc{Allow}\ intf.subs[\ \textsc{Allow}\ cloudId.\langle UID_x, SID_i, payment \rangle\ ]\ ]$$

$$\begin{aligned} RequestMsg \equiv msg[\ &\textsc{In}\ cloud.\textsc{Allow}\ intf.request[\ \textsc{In}\ UID_x\ \mid \\ &\langle \text{“}o_i'', client_k, cInfo_k, args_i \rangle\ \mid \\ &\vdots \\ &\langle \text{“}o_j'', client_k, cInfo_k, args_j \rangle\ ]\ ] \end{aligned}$$

The Cloud Service Architecture

$$
\begin{array}{l}
Cloud \equiv (\nu\ k,\ q,\ rescr_1, \ldots rescr_m)cloud[\\
\quad interface \mid \\
\quad k\ [\ rescr_1[\ service_1\ ] \mid \ldots \mid rescr_l[\ service_1\ ] \mid \\
\qquad rescr_{l+1}[\ service_2\ ] \mid \ldots \mid rescr_m[\ service_n\ ] \mid \\
\qquad q[\ !\textsc{Open}\ msg \mid \\
\qquad\quad \textsc{Server}^n_{cloudId}\ reg.(UID).\textsc{RegMgr} \mid \\
\qquad\quad \textsc{Server}^n_{cloudId}\ subs.(UID,\ SID,\ payment).\textsc{SubsMgr} \mid \\
\qquad\quad UID_x[\ userIntf\ ] \mid \ldots \mid UID_y[\ userIntf\ ] \mid \\
\qquad\quad UID_v^{owner}[\ ownerIntf\ ] \mid \ldots \mid UID_w^{owner}[\ ownerIntf\ ] \\
]\ ]\ ] \\
\textbf{where} \\
\quad interface \equiv !intf[\ \textsc{in}\ msg.\textsc{In}\ k.\textsc{In}\ q\ ]
\end{array}
$$

User Access Layers

$userIntf \equiv$
$\quad !\textsc{Allow}\ request \mid !\textsc{Allow}\ newPlot \mid !\textsc{Allow}\ returnValue \mid$
$\quad !(op, client, cInfo, args).\textsc{Adapter}(op, client, cInfo, args) \mid$
$\quad postingOutput \mid plot_{service_1} \mid \ldots \mid plot_{service_r}$
where
$\quad postingOutput \equiv$
$\quad\quad !(o, client, a).msg[\ \textsc{Out}\ UID_x.\textsc{Out}\ q.\textsc{Out}\ k.$
$\quad\quad \textsc{Out}\ cloud.\textsc{In}\ client.output[\ \textsc{Allow}\ UserId_x.\langle o, a\rangle\ ]\ ]$

User Access Layers with Service Ownership

$$
\begin{aligned}
&\mathit{ownerIntf} \equiv \\
&\quad \mathit{userIntf} \mid \\
&\quad \textsc{RndAssign}^{n}_{access}\ \mathit{SID}_1.(\textsc{AccessTo}\ \mathit{resrc}_1 + \ldots + \textsc{AccessTo}\ \mathit{resrc}_l) \mid \\
&\quad \vdots \\
&\quad \textsc{RndAssign}^{n}_{access}\ \mathit{SID}_r.(\textsc{AccessTo}\ \mathit{resrc}_s + \ldots + \textsc{AccessTo}\ \mathit{resrc}_t) \mid \\
&\textbf{where} \\
&\quad \textsc{AccessTo}\ \mathit{resrc}_h \equiv \\
&\qquad \textsc{Out}\ \mathit{UID}_v^{owner}.\textsc{Out}\ q.\textsc{In}\ \mathit{resrc}_h.n\ \textsc{Be}\ \mathit{SID}_i.\textsc{Allow}\ \mathit{access}
\end{aligned}
$$

Service Instances

$service_i \equiv (\nu\ lock)(\textsc{Server}^{n}_{access}\ SID_i.($
$(o, client, args).lock[\ \textsc{Service}_i(o, client, args)\ ]\ |$
$\textsc{Open}\ lock.(o, client, a).sendBack\ )\ )$

where

$sendBack \equiv \textsc{Out}\ resrc_h.\textsc{In}\ q.out[$
$\textsc{In}\ returnValue.\textsc{Out}\ n.\langle o, client, a\rangle\ ]$
$\textsc{Service}_i(o, client, args) \equiv$ an Abstract State Service with
$ctr\_state : \{RunningState, \ldots, EndState\}\ \ldots$

ASM Agents 1/3: User Registration Agent

RegMgr = RegistrationManager(*UID*, *userCr*)
RegistrationManager(*UID*) ≡
  ctr_state : {*RunningState*, *EndState*}
  initially *ctr_state* := *RunningState*

  if *ctr_state* = *RunningState* **then**
    if *ids*(*UID*) = **undef then**
      storeNewUser(*UID*)
      let *UIConstruct* = *getAccessLayerForUser*(*UID*) **in**
        NewAmbientConstruct(*UIConstruct*)
        ctr_state := *EndState*
where
  UIConstruct ≡ *UID*[Out *n* | *userIntf*$^{(without\ \ plots)}$]

ASM Agents 2/3: Service Subscription Agent

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

  if *ctr_state* = *RunningState* **then**
    if *ids*(*UID*) ≠ **undef then**
      let *owner* = *getServiceOwner*(*serviceID*)) **in**
        let *plot* = getPlotFromOwner(*owner*, *UID*, *SID*, *payment*) **in**
          NewAmbientConstruct(*newPlot*[Out *n*.In *UID* | *plot*])
          ctr_state := *EndState*

ASM Agents 3/3: Adapter Agent

ADAPTER(op, $client$, $cInfo$, $args$) $\equiv$
  ctr_state : $\{RunningState, EndState\}$
  initially $ctr\_state := RunningState$

  let ($newOp$, $newArgs$) = $adaptToClient($ $cInfo$, op, $args$ $)$ **in**
    NEWAMBIENTCONSTRUCT($newOp$[ALLOW $op.\langle client, newArgs\rangle$])
    $ctr\_state := EndState$

Service Plots 1/2

$slot_{o_i}^{next} \equiv (\nu\ n)(n[\ n\ \textsc{DrawIn}_{op}\ o_i.trigger_{o_i}^{next}\ ]\ )$

$slot_{o_i+\ldots+o_j}^{next} \equiv (\nu\ n)(n[\ n\ \textsc{Choice}_{op}\ o_i.trigger_{o_i}^{next}+\ldots+o_j.trigger_{o_j}^{next}\ ]\ )$

where

$trigger_{o_i}^{next} \equiv$
$(\nu\ lock)((client,\ args).\textsc{Open}\ lock.\langle \text{“o}_\text{i}'', client, args\rangle\ |$
$lock[\ \textsc{RequestFor}\ SID_j.homecoming^{next}\ ]\ )$

$\textsc{RequestFor}\ SID_j \equiv$
$\textsc{Out}\ UID_x.\textsc{In}\ UID_v^{owner}.n^{unique}\ \textsc{Be}\ SID_j.\textsc{Allow}\ access$

$homecoming^{next} \equiv \textsc{In}\ UID_x.returnValue[\ \textsc{Open}\ out\ |\ next\ ]$

- *next* denotes an ambient construct (called *sequence trigger*) which unlocks the subsequent slot in a sequential plot, when *homecoming* returns to the user area with the output of the current operation.
- If there is no any subsequent slot, *next* is equal to 0 (inactivity).

Service Plots 2/2

$$plot_{o_i} \equiv slot^0_{o_i}$$

$$plot_{(o_i)^*} \equiv (\nu\ seq)(\textsc{allow}\ seq \mid !seq[\ slot^{\textsc{Allow}seq}_{o_i}\ ]\ )$$

$$plot_{(o_i o_j \ldots o_k)^*} \equiv (\nu\ seq, next)(\textsc{Allow}\ seq \mid !seq[\ slot^{\textsc{Allow}\ next}_{o_i} \mid next[\ slot^{\textsc{Allow}\ next}_{o_j} \mid next[\ \ldots next[\ slot^{\textsc{Allow}\ seq}_{o_j}\ ]\ldots]\ ]\ ]\ )$$

$$plot_{((o_i+o_j)o_k o_l)^*} \equiv (\nu\ seq, next)(\textsc{Allow}\ seq \mid !seq[\ slot^{\textsc{Allow}\ next}_{o_i+o_j} \mid next[\ slot^{\textsc{Allow}\ next}_{o_k} \mid next[\ slot^{\textsc{Allow}\ seq}_{o_l}\ ]\ ]\ ]\ )$$

Example 1/6: Cloud Receives a Request

$$
\begin{aligned}
&\mathit{Cloud} \mid \mathit{RequestMsg} \\
&\quad = \quad (\nu\ k,\ q,\ \mathit{rescr}_1, \ldots \mathit{rescr}_m)\mathit{cloud}[\\
&\qquad \overbrace{!}^{(2nd)}\ \mathit{intf}[\ \overbrace{\text{IN}\ \mathit{msg}}^{(3rd)}.\overbrace{\text{IN}\ k.\text{IN}\ q}^{(5th)}\ ]\ \mid \\
&\qquad k\ [\ \mathit{rescr}_1[\ \mathit{service}_1\ ] \mid \ldots \mid \mathit{rescr}_i[\ \mathit{service}_1\ ] \mid \\
&\qquad\quad \mathit{rescr}_{i+1}[\ \mathit{service}_2\ ] \mid \ldots \mid \mathit{rescr}_m[\ \mathit{service}_n\ ] \mid \\
&\qquad\quad q[\ \overbrace{!\ \text{OPEN}\ \mathit{msg}}^{(6th)}\ \mid \\
&\qquad\qquad \text{SERVER}^n_{\mathit{cloudId}}\ \mathit{reg}.(\mathit{UID}).\text{REGMGR}\ \mid \\
&\qquad\qquad \text{SERVER}^n_{\mathit{cloudId}}\ \mathit{subs}.(\mathit{UID},\mathit{SID},\mathit{payment}).\text{SUBSMGR}\ \mid \\
&\qquad\qquad \mathit{UID}_x[\ \mathit{userIntf}\ ] \mid \ldots \mid \mathit{UID}_y[\ \mathit{userIntf}\ ] \mid \\
&\qquad\qquad \mathit{UID}_v^{\mathit{owner}}[\ \mathit{ownerIntf}\ ] \mid \ldots \mid \mathit{UID}_w^{\mathit{owner}}[\ \mathit{ownerIntf}\ ] \\
&\qquad]\ ]\ ]\ \mid \\
&\qquad \mathit{msg}[\ \overbrace{\text{IN}\ \mathit{cloud}}^{(1st)}.\overbrace{\text{ALLOW}\ \mathit{intf}}^{(4th)}.\mathit{request}[\ \overbrace{\text{IN}\ \mathit{UID}_x}^{(7th)}\ \mid \\
&\qquad\quad \langle \text{“}o_2'', \mathit{client}_1, \mathit{cInfo}_1, \mathit{args}_2 \rangle\ \mid \\
&\qquad\quad \langle \text{“}o_3'', \mathit{client}_1, \mathit{cInfo}_1, \mathit{args}_3 \rangle\ \mid \\
&\qquad\quad \langle \text{“}o_1'', \mathit{client}_1, \mathit{cInfo}_1, \mathit{args}_1 \rangle\ ]\ ]
\end{aligned}
$$

Example 2/6: Adaptation

$$\langle \text{“}o_2'', client_1, cInfo_1, args_2 \rangle$$
$$\langle \text{“}o_3'', client_1, cInfo_1, args_3 \rangle$$
$$\langle \text{“}o_1'', client_1, cInfo_1, args_1 \rangle$$

Each operation request is processed by an instance of the agent ADAPTER which may replace the given operation name with one or more other operations (let us call them adapted operations (e.g.: o_1^a, o_2^a and o_3^a), according to the provided client information $cInfo_1$ and some internal adaptation rules (which may be unique for the user). The arguments may be modified/converted as well. The agent ADAPTER provides the adapted operations for the plots in the following format:

$$o_2^a[\ \textsc{Allow}\ op.\langle client_1, args_2^a \rangle\ ]$$
$$o_3^a[\ \textsc{Allow}\ op.\langle client_1, args_3^a \rangle\ ]$$
$$o_1^a[\ \textsc{Allow}\ op.\langle client_1, args_1^a \rangle\ ]$$

Then if a service operation (e.g.: o_1^a) is allowed by any plot it will be triggered.

Example 3/6: The Plot $plot_{(o_1 o_2 o_3)^*}$ *Triggers* o_1

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

$\vdots$

$UID_x[\ \overbrace{!\ \underline{\textsc{Allow}\ request}}^{(8th)}\ |\ !\textsc{Allow}\ newPlot\ |\ !\textsc{Allow}\ returnValue\ |\ postingOutput\ |$
$!(op, client, cInfo, args).\textsc{Adapter}(op, client, cInfo, args)\ |$
$(\nu\ seq, next)(\overbrace{\underline{\textsc{Allow}\ seq}}^{(11th)}\ |\ \overbrace{\underline{!}}^{9th}\ seq[\ \overbrace{\underline{(\nu\ n)}}^{(10th)}(n[\ \overbrace{\underline{n\ \textsc{DrawIn}_{op}\ o_1}}^{(12th)}\ |$
$\overbrace{\underline{(client,\ args)}}^{(13th)}.\overbrace{\underline{\textsc{Open}\ lock}}^{(14th)}.\langle \text{“}o_1'', client, args\rangle\ |\ lock[\ \overbrace{\underline{\textsc{RequestFor}\ SID_1}}^{(15th)}.homecoming^{\textsc{Allow}\ next}\ ]\ ]\ )\ |$
$next[\ \overbrace{\underline{(\nu\ n)}}^{(10th)}(n[\ n\ \textsc{DrawIn}_{op}\ o_2\ |$
$(client, args).\textsc{Open}\ lock.\langle \text{“}o_2'', client, args\rangle\ |\ lock[\ \textsc{RequestFor}\ SID_1.homecoming^{\textsc{Allow}\ next}\ ]\ ]\ )\ |$
$next[\ \overbrace{\underline{(\nu\ n)}}^{(10th)}(n[\ n\ \textsc{DrawIn}_{op}\ o_3\ |$
$(client, args).\textsc{Open}\ lock.\langle \text{“}o_3'', client, args\rangle\ |\ lock[\ \textsc{RequestFor}\ SID_1.homecoming^{\textsc{Allow}\ seq}\ ]\ ]\ )$
$]\ ]\ ]\ |$
$request[$

$o_2[\ \textsc{Allow}\ op.\langle client_1, args_2\rangle\ ]\ |\ o_3[\ \textsc{Allow}\ op.\langle client_1, args_3\rangle\ ]\ |\ o_1[\ \overbrace{\underline{\textsc{Allow}\ op}}^{(12th)}.\langle client_1, args_1\rangle\ ]\ ]\ )\ ]\ |$

$\vdots$

$]$

Example 4/6: Scheduling the Request

- **The triggered o_1 has the form:**

  $n_i^{unique}[\ \langle \text{“}o_1\text{''}, client_1, args_1 \rangle \mid \textsc{RequestFor}\ SID_1.homecoming^{\textsc{Allow}\ next}\ ]$

  where

  $\textsc{RequestFor}\ SID_j \equiv \underline{\textsc{Out}\ UID_x.\textsc{In}\ UID_v^{owner}.n_i^{unique}\ \textsc{Be}\ SID_j}.\textsc{Allow}\ access$

  $homecoming^{next} \equiv \textsc{In}\ UID_x.returnValue[\ \textsc{Open}\ out \mid next\ ]$

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

$\vdots$

$UID_v^{owner}[\ userIntf \mid \overbrace{\underbrace{\textsc{RndAssign}^n_{access}\ SID_1.(\textsc{AccessTo}resrc_k + \ldots + \textsc{AccessTo}resrc_l)}_{(16th)} \mid}^{(16th)}$

$SID_1[\ \langle \text{“}o_1\text{''}, client_1, args_1 \rangle \mid \overbrace{\underline{\textsc{Allow}\ access}}.homecoming^{\textsc{Allow}\ next}\ ]\ ] \mid$

$\vdots$

$]$

- **The scheduled o_1 has the form:**

  $n_j^{unique}[\ \langle \text{“}o_1\text{''}, client_1, args_1 \rangle \mid homecoming^{\textsc{Allow}\ next} \mid \textsc{AccessTo}\ resrc_h\ ]$

  where

  $\textsc{AccessTo}\ resrc_h \equiv \underline{\textsc{Out}\ UID_v^{owner}.\textsc{Out}\ q.\textsc{In}\ resrc_h.n_j^{unique}\ \textsc{Be}\ SID_i}.\textsc{Allow}\ access$

  $homecoming^{next} \equiv \textsc{In}\ UID_x.returnValue[\ \textsc{Open}\ out \mid next\ ]$

Example 5/6: Performing the Request

$\longrightarrow^*$ $(\nu\ k,\ q,\ resc r_1, \ldots resc r_m)cloud[$

$\vdots$

$resrc_h[\ \overbrace{\textsc{Server}^{n}_{access}\ SID_1}^{(17th)}.(\overbrace{(o,\ client,\ args)}^{(18th)}.lock[\ \textsc{Service}_1(o,\ client,\ args)\ ]\ |$

$\underbrace{\overbrace{\textsc{Open}\ lock}^{(19th)}.(o,\ client,\ a).\textsc{Out}\ resrc_h.\textsc{In}\ q.out[\ \textsc{In}\ returnValue.\textsc{Out}\ n.\langle o,\ client,\ a\rangle\ ]\ )}_{(17th)}\ |$

$SID_1[\ \langle \text{“}o_1\text{”},\ client_1,\ args_1\rangle\ |\ homecoming^{\textsc{Allow}\ next}\ |\ \overbrace{\textsc{Allow}\ access}^{(17th)}\ ]\ ]\ |$

$\vdots$

$]$

$\longrightarrow^*$ $(\nu\ k,\ q,\ resc r_1, \ldots resc r_m)cloud[$

$\vdots$

$resrc_h[\ \textsc{Server}^{n}_{access}\ SID_1.((o,\ client,\ args).lock[\ \textsc{Service}_1(o,\ client,\ args)\ ]\ |$

$\textsc{Open}\ lock.(o,\ client,\ a).\textsc{Out}\ resrc_h.\textsc{In}\ q.out[\ \textsc{In}\ returnValue.\textsc{Out}\ n.\langle o,\ client,\ a\rangle\ ]\ )\ |$

$n_f^{unique}[\ homecoming^{\textsc{Allow}\ next}\ |\ \overbrace{\underline{\textsc{Service}_{1(RunningState)}(\text{“}o_1\text{”},\ client_1,\ args_1)}}^{(20th)}\ |$

$(o,\ client,\ a).\textsc{Out}\ resrc_h.\textsc{In}\ q.out[\ \textsc{In}\ returnValue.\textsc{Out}\ n_f^{unique}.\langle o,\ client,\ a\rangle\ ]\ ]\ ]\ |$

$\vdots$

$]$

Example 5/6: Performing the Request (cont.)

$\xrightarrow{asm}{}^{*}$ $(\nu\ k,\ q,\ rescr_1, \ldots rescr_m)cloud[$

$\vdots$

$resrc_h[\ \textsc{Server}^{n}_{access}\ SID_1.((o,\ client,\ args).lock[\ \textsc{Service}_1(o,\ client,\ args)\ ]\ |$

$\textsc{Open}\ lock.(o,\ client,\ a).\textsc{Out}\ resrc_h.\textsc{In}\ q.out[\ \textsc{In}\ returnValue.\textsc{Out}\ n.\langle o, client, a\rangle\ ]\ )\ |$

$n_f^{unique}[\ homecoming^{\textsc{Allow}\ next}\ |\ \cancel{\textsc{Service}_{1(EndState)}}\ |\ \langle\text{“}o_1\text{”},\ client_1,\ outcome_{o_1}\rangle\ |$

$\overbrace{(o,\ client,\ a)}^{(21st)}.\overbrace{\textsc{Out}\ resrc_h.\textsc{In}\ q}^{(22nd)}.out[\ \textsc{In}\ returnValue.\textsc{Out}\ n_f^{unique}.\langle o, client, a\rangle\ ]\ ]\ ]\ |$

$\vdots$

$]$

Example 6/6: Returning the Output

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

$\vdots$

$UID_x[\ !\textsc{Allow}\ request \mid !\textsc{Allow}\ newPlot \mid \overbrace{\underline{!\ \textsc{Allow}\ returnValue}}^{(27th)} \mid \overbrace{\underline{postingOutput}}^{(28th)} \mid$
$!(op, client, cInfo, args).\textsc{Adapter}(op, client, cInfo, args) \mid$
$plot_{(o_1 o_2 o_3)^*} \mid$

$next[\ n_{i'}^{unique}[\ \overbrace{\underline{n_{i'}^{unique}\ \textsc{DrawIn}_{op}\ o_2}}^{(30th)} \mid$
$(client, args).\textsc{Open}\ lock.\langle \text{“}o_2'', client, args\rangle \mid lock[\ \textsc{RequestFor}\ SID_1.homecoming^{\textsc{Allow}\ next}\ ]\ ] \mid$
$next[\ n_{i''}^{unique}[\ n_{i''}^{unique}\ \textsc{DrawIn}_{op}\ o_3 \mid$
$(client, args).\textsc{Open}\ lock.\langle \text{“}o_3'', client, args\rangle \mid lock[\ \textsc{RequestFor}\ SID_1.homecoming^{\textsc{Allow}\ seq}\ ]\ ]\ ]\ ] \mid$
$o_2[\ \textsc{Allow}\ op.\langle client_1, args_2\rangle\ ] \mid o_3[\ \textsc{Allow}\ op.\langle client_1, args_3\rangle\ ]\ ] \mid$

$n_f^{unique}[\ \overbrace{\underline{\textsc{In}\ UID_x}}^{(23th)}.returnValue[$
$\overbrace{\underline{\textsc{Open}\ out}}^{(25th)} \mid \overbrace{\underline{\textsc{Allow}\ seq_i^{unique}}}^{(29th)}\ ] \mid out[\ \overbrace{\underline{\textsc{In}\ returnValue}}^{(24th)}.\overbrace{\underline{\textsc{Out}\ n_f^{unique}}}^{(26th)}.\langle \text{“}o_1'',\ client_1,\ outcome_{o_1}\rangle\ ]\ ] \mid$

$\vdots$

$]$

What are Clouds?

- The common understanding of the notion of "cloud" is that it is some kind of service pool, from which services can be extracted and used
- One of the key problems is to discover the services that are needed for a particular application by means of a search engine
- It is crucial that the service operations including the view defining queries that are made available through some cloud are provided with an adequate description:
  - a *functional* description of input and output types as well as pre- and post-conditions telling in technical terms, what the service operation will do
  - a *categorical* description by inter-related keywords telling what the service operation does by using common terminology of the application area
  - a *quality of service* (QoS) description of non-functional properties such as availability, response time, cost, etc.

Functional Description

- A functional description alone would be insufficient:
  - a flight booking service operation requires an itinerary to be selected, so the input type could be specified as $\{(flight\_no : STRING,\ day : DATE,\ departure : TIME,\ class : CHAR,\ price : DECIMAL)\}$
  - the input is a finite set of tuples, each of which defines a flight number, departure day and time, the booking class and the price
  - the output type could be similar with a status (confirmed, waitlisted, unavailable) added for each flight segment, i.e. we have the type $\{(flight\_no : STRING,\ day : DATE,\ departure : TIME,\ class : CHAR,\ price : DECIMAL,\ status : STRING)\}$
  - a precondition could simply be that the selected itinerary is meaningful, i.e. flight numbers exist for the corresponding date and time, and are compatible
  - a booking service for railway tickets would require the same types, so the functional description does not indicate exactly what kind of service is offered.

Categorical and QoS Description

- For the categorical description the terminology has to be specified
- This defines an ontology in the widest sense, i.e. we have to provide definitions of "concepts" and relationships between them, such that each offered service becomes an instantiation of one or several concepts in the terminology
- In this way we adopt the fundamental idea of the "semantic web"
- The QoS description is not needed for service discovery and merely useful to select among alternatives

3.1 Ontologies

- Ontologies (in the widest sense) can be exploited for service description:
  - a *terminological knowledge* layer (aka TBox in description logics) describing concepts and roles (or relationships) among them
  - this usually includes a subsumption hierarchy among concepts (and maybe also roles), and cardinality constraints
  - in addition, there is an *assertional knowledge* layer (aka ABox in description logics) describing individuals
  - services in a cloud constitute the ABox of an ontology, while the cloud itself is defined by the TBox
- We have to face the usual tradeoff between expressiveness and decidability

Description Logics

- Terminological and assertional knowledge could be defined in any logic
  - instead of TBox and ABox we could use the more classical notions of schema and instance, and exploit any kind of data model (e.g. XML)
  - a query language associated with the used data model (e.g. XQuery) could then be used to find the required services
- Description logics such as OWL or *DL-Lite* or more restricted (for sake of decidability)

Description Logics / 2

- Description logics use two important relationships, which due to the restrictions become decidable:
  - *Subsumption* is a binary relationship between concepts (denoted as $C_1 \sqsubseteq C_2$) guaranteering that all instances of the subsumed concept C_1 are also instances of the subsuming concept C_2
  - *Instantiation* defines a binary relationship between instances in the ABox and concepts in the TBox asserting that an element A of the ABox is an instance of a concept C in the TBox
  - Subsumption and instantiation together allow us to discover services that are more expressive than needed, but can be projected to a service just as required
- Concept and role names in the TBox could be subject to similarity search by a search engine
  - the search engine could produce services that are similar (with a certainty factor) to the ones required with respect to the categorical description, and match the functional description

A Sample Description Logic

- Look more closely into one particular description logic in the *DL-Lite* family
- For this assume that C_0 and R_0 represent not further specified specified sets of basic concepts and roles, respectively
- Then *concepts* C and *roles* R are defined by the following grammar:

$$\begin{array}{lcl} R & = & R_0 \mid R_0^- \\ A & = & C_0 \mid \top \mid \geq m.R \text{ (with } m > 0) \\ C & = & A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists R.C \mid \forall R.C \end{array}$$

Terminologies

- A *terminology* (or TBox) is a finite set $\mathcal{T}$ of assertions of the form $C_1 \sqsubseteq C_2$ with concepts C_1 and C_2 as defined by the grammar above
- Each assertion $C_1 \sqsubseteq C_2$ in a terminology $\mathcal{T}$ is called a *subsumption axiom*
- The logic only permits subsumption between concepts, not between roles, though it is possible to define more complex terminologies that also permit role subsumption
- Shortcuts:
  - Write $C_1 \equiv C_2$ instead of $C_1 \sqsubseteq C_2 \sqsubseteq C_1$
  - $\bot$ is a shortcut for $\neg\top$
  - $\leq m.R$ is a shortcut for $\neg \geq m+1.R$

Semantics

- The semantics of a terminology is defined by its models
- A *structure* $\mathcal{S}$ for a terminology $\mathcal{T}$ consists of
  - a non-empty base set $\mathcal{O}$,
  - subsets $\mathcal{S}(C_0) \subseteq \mathcal{O}$ for all basic concepts C_0, and
  - subsets $\mathcal{S}(R_0) \subseteq \mathcal{O} \times \mathcal{O}$ for all basic roles R_0
- Extend the interpretation of basic concepts and roles and to all concepts and roles as defined by the grammar above

Semantics / 2

- For each concept C we define a subset $\mathcal{S}(C) \subseteq \mathcal{O}$, and for each role R we define a subset $\mathcal{S}(R) \subseteq \mathcal{O} \times \mathcal{O}$ as follows:

$$\mathcal{S}(R_0^-) = \{(y, x) \mid (x, y) \in \mathcal{S}(R_0)\}$$
$$\mathcal{S}(\top) = \mathcal{O}$$
$$\mathcal{S}(\geq m.R) = \{x \in \mathcal{O} \mid \#\{y \mid (x, y) \in \mathcal{S}(R)\} \geq m\}$$
$$\mathcal{S}(\neg C) = \mathcal{O} - \mathcal{S}(C)$$
$$\mathcal{S}(C_1 \sqcap C_2) = \mathcal{S}(C_1) \cap \mathcal{S}(C_2)$$
$$\mathcal{S}(C_1 \sqcup C_2) = \mathcal{S}(C_1) \cup \mathcal{S}(C_2)$$
$$\mathcal{S}(\exists R.C) = \{x \in \mathcal{O} \mid (x, y) \in \mathcal{S}(R) \text{ for some } y \in \mathcal{S}(C)\}$$
$$\mathcal{S}(\forall R.C) = \{x \in \mathcal{O} \mid (x, y) \in \mathcal{S}(R) \Rightarrow y \in \mathcal{S}(C) \text{ for all } y\}$$

- A *model* (or ABox) for a terminology $\mathcal{T}$ is a structure $\mathcal{S}$, such that $\mathcal{S}(C_1) \subseteq \mathcal{S}(C_2)$ holds for all assertions $C_1 \sqsubseteq C_2$ in $\mathcal{T}$

Example

The general part of a service ontology could be defined by a terminology as follows:

$$\text{Service} \sqsubseteq \exists \text{name.Identifier} \sqcap \leq 1.\text{name} \sqcap \exists \text{address.URL} \sqcap$$
$$\exists \text{offered\_by.Provider} \sqcap \leq 1.\text{address} \sqcap \leq 1.\text{offered\_by}$$
$$\sqcap \exists \text{defining.Query} \sqcap \leq 1.\text{defining} \sqcap \exists \text{offers.Operation}$$
$$\text{Operation} \sqsubseteq \exists \text{associated\_with.Query} \sqcap \leq 1.\text{associated\_with}$$
$$\text{Data\_Service} \equiv \text{Query} \sqcap \geq 1.\text{defining}^{-}$$
$$\text{Functional\_Service} \equiv \text{Operation} \sqcap \geq 1.\text{offers}^{-}$$
$$\text{Service\_Operation} \equiv \text{Data\_Service} \sqcup \text{Functional\_Service}$$
$$\text{Service\_Operation} \sqsubseteq \exists \text{input.Type} \sqcap \leq 1.\text{input}$$
$$\exists \text{output.Type} \sqcap \leq 1.\text{output}$$
$$\text{Type} \sqsubseteq \exists \text{name.Identifier} \sqcap \leq 1.\text{name} \sqcap \exists \text{format.Format}$$

3 Algebraic Plots on Abstract State Services

- A plot is a high-level specification of an action scheme, i.e. it specifies possible sequences of service operations
- For an algebraic formalisation of plots in Web Information Systems (WISs) it was possible to exploit Kleene algebras with tests (KATs)
- Then a plot is an algebraic expression that is composed out of elementary operations including 0, 1, and propositional atoms, binary operators $\cdot$ and $+$, and unary operators * and $^-$, the latter one being only applicable to propositions

The set of ***process expressions*** of an AS^2 is the smallest set $\mathcal{P}$ containing all elementary processes that is closed under sequential composition $\cdot$, parallel composition $\|$, choice $+$, and iteration *. That is, whenever $p, q \in \mathcal{P}$ hold, then also pq, $p\|q$, $p+q$ and p^* are process expressions in $\mathcal{P}$.

The ***plot*** of an AS^2 is a process expression in $\mathcal{P}$.

Plots (continued)

- For the definition of plots for AS^{2}s the service operations give rise to ***elementary processes*** of the form

$$\varphi(\vec{x})\ op[\vec{z}](\vec{y})\ \psi(\vec{x},\vec{y},\vec{z}),$$

  in which

  - op is the name of a service operation
  - $\vec{z}$ denotes input for op selected from the view v with $op \in Op_v$
  - $\vec{y}$ denotes additional input from the user
  - φ and ψ are first-order formulae denoting pre- and postconditions, respectively

- Furthermore, simple formulae $\chi(\vec{x})$ – again interpreted as tests checking their validity – also constitute elementary processes

Examples

- For a flight booking service we may have the following simple plot:

$$\text{get\_itineraries}[](d)\ \text{select\_itinerary}[i]()\ \text{personal\_data}[](t)\ \text{confirm\_flight}[](y)\ \text{pay\_flight}[](c)$$

- The following expression represents another plot for accommodation booking:

$$\text{get\_hotels}[](d)\ \text{select\_hotel}[h]()\ \text{select\_room}[r]()\ \text{personal\_data}[](t)\ \text{confirm\_hotel}[](y)\ \text{pay\_accommodation}[](c)$$

- The following expression represents another plot for conference registration:

$$\text{personal\_data}[](t)\ (\text{papers}[]() \parallel \text{discount}[](d'))$$

4 Mediators for Abstract State Services

- We want to capture the plot of a composed AS^2, i.e. the plot of an application yet to be constructed
- Such mediators specify service operations to be searched for to solve a problem in a service-oriented way
- Relax the definition of a plot in such a way that service operations do not have to come from the same AS^2
- Use prefixes to indicate the corresponding AS^2, so we obtain

$$\varphi(\vec{x})\ X : op[\vec{z}](\vec{y})\ \psi(\vec{x}, \vec{y}, \vec{z}),$$

in which X denotes a ***service slot***

A ***service mediator*** is a process expression with service slots.

Furthermore, each service operation is associated with input- and output-types, pre- and postconditions, and a concept in a service terminology.

Example

- Specify a service mediator for a conference trip application by combining conference registration, flight booking, and accommodation booking
- Replicative entry of customer data should be avoided, and confirmation of selection as well as payment should be unified in single local operations

$$
\begin{array}{l}
L : \text{personal\_data}[](t)\ (X : \text{papers}[]() \parallel X : \text{discount}[](d') \\
\quad (Y : \text{get\_itineraries}[](d)\ Y : \text{select\_itinerary}[i]() \parallel \\
\quad\quad Z : \text{get\_hotels}[](d)\ Z : \text{select\_hotel}[h]()\ Z : \text{select\_room}[r]())) \\
L : \text{confirm}[](y)\ (Y : \text{confirm\_flight}[](y) \parallel Z : \text{confirm\_hotel}[](y)) \\
L : \text{pay}[](c)\ (Y : \text{pay\_flight}[](c) \parallel Z : \text{pay\_hotel}[](c))
\end{array}
$$

- The three slots X, Y and Z refer to the three services for conference registration, flight booking, and accommodation booking, respectively, while the slot L refers to local operations
- For confirmation and payment the input parameters y and c are simply pushed through to the two booking services

5 Matching Services for Service Slots

- We need exact criteria to decide, when a service matches a service slot in a service mediator
- For all service operations in a mediator associated with a slot X we must find matching service operations in the same AS^2
  - The matching of service operations has to be based on their functional and categorical description
  - The placeholder in the mediator must be replaceable by matching service operations
  - Functionally, the input for the service operation as defined by the mediator must be accepted by the matching service operation, while the output of the matching service operation must be suitable to continue with other operations as defined by the mediator

Matching Criteria

- So we need supertypes and subtypes of the specified input- and output-types, respectively, in the mediator, as well as a weakening of the precondition and a strengthening of the postcondition
- Categorically, the matching service operation must satisfy all the properties of the concept in the terminology that is associated with the placeholder operation

- We also have to ensure that the projection of the mediator to a particular slot X results in a subplot of the plot of the matching AS[2]

A ***subplot*** of a plot p is a process expression q such that there exists another process expression r such that $p = q + r$ holds in the equational theory of process expressions.

The ***projection*** of a mediator m is a process expression p_X such that $p_X = \pi_X(m)$ holds in the equational theory of process expressions, where $\pi_X(m)$ results from m by replacing all placeholders $Y : o$ with $Y \neq X$ and all conditions that are irrelevant for X by 1.

Matching Criteria (continued)

- Requiring that the projection of a mediator should result in a subplot of a matching service is too simple, as the order of service operations may differ, and certain service operations may be redundant

- We call such redundant service operations phantoms:

  - If for a condition $\varphi(\vec{x})$ appearing in a process expression p the equation $\varphi(\vec{x}) = \varphi(\vec{x})op[\vec{y}](\vec{z})$ holds, then $op[\vec{y}](\vec{z})$ is called a ***phantom*** of p

  - Whenever $p = q$ holds in the equational theory of process expressions, and $op[\vec{y}](\vec{z})$ is a phantom of p with respect to condition $\varphi(\vec{x})$, we may replace $\varphi(\vec{x})$ by $\varphi(\vec{x})op[\vec{y}](\vec{z})$ in q

  - Each process expression resulting from such replacements is called an ***enrichment of*** p ***by phantoms***

- Thus, we must consider projections of enrichments by phantoms

Matching Definition

An AS^2 $\mathcal{A}$ ***matches*** a service slot X in a service mediator m iff the following two conditions hold:

1. For each service operation $X : o$ in m there exists a service operation op provided by $\mathcal{A}$ such that
   - the input-type I_{op} of op is a supertype of the input-type I_o of o,
   - the output-type O_{op} of op is a subtype of the output-type O_o of o,
   - $pre_o \Rightarrow pre_{op}$ holds for the preconditions pre_o and pre_{op} of o and op, respectively,
   - $post_{op} \Rightarrow post_o$ holds for the postconditions $post_o$ and $post_{op}$ of o and op, respectively, and
   - the concept C_o associated with o in the service terminology subsumes the concept C_{op} associated with op.
2. There exists an enrichment m_X of m by phantoms such that building the projection of m and replacing all service operations $X : o$ by matching service operations op from $\mathcal{A}$ results in a subplot of the plot of $\mathcal{A}$.

Example

- Look the previous service mediator we can assume that the local operation personal_data[](t) has the postcondition $person(t)$, and this is invariant under the service operations for itinerary and hotel selection
- We can further assume that in both booking services the service operation personal_data[](t) is a phantom for $person(t)$
- Thus, the mediator can enriched by phantoms, which results in:

$$
\begin{array}{l}
L : \text{personal\_data}[](t)\ (X : \text{papers}[]()\ \|\ X : \text{discount}[](d') \\
\quad (Y : \text{get\_itineraries}[](d)\ Y : \text{select\_itinerary}[i]()\ Y : \text{personal\_data}[](t)\ \| \\
\qquad Z : \text{get\_hotels}[](d)\ Z : \text{select\_hotel}[h]()\ Z : \text{select\_room}[r]()) \\
\qquad\quad Z : \text{personal\_data}[](t) \\
L : \text{confirm}[](y)\ (Y : \text{confirm\_flight}[](y)\ \|\ Z : \text{confirm\_hotel}[](y)) \\
L : \text{pay}[](c)\ (Y : \text{pay\_flight}[](c)\ \|\ Z : \text{pay\_hotel}[](c))
\end{array}
$$

- The projection of this process expression to the services X, Y and Z, respectively, results exactly in the three plots in our previous example

Conclusions

Our model applies a new client-cloud interaction approach based on algebraic plots by which service owners are able to fully control the usages of their services in the case of each subscription, respectively.

- One of the major questions can be whether it is adaptable to nowadays leading cloud architectures and solutions (e.g.: Amazon S3, Microsoft Azure, IBM SmartCloud, etc.)?
- In the model both the client side and the services are still abstract and all our novel methods either can be wrapped into a single cloud service or can be shifted to the client side as well.

Adaptivity can be approached by defining appropriate ambients

Extensions regarding Identification, Authorisation, Authentication, Monitoring, etc. can be seen as refinements of the generic architecture