A Formal Model of Client-Cloud Interaction

Károly Bósa, Roxana-Maria Holom, and Mircea Boris Vleju

Abstract In our former work, we have showed that cloud computing still requires lots of fundamental research. Among many other existing problems in cloud computing, we identified the lack of client orientation and lack of formal foundations as serious deficiencies. In this chapter, we give a summary on our research and discuss the architectures as well as the formal models of some software solutions with which we are going to address (a part of) these two problems in cloud computing.

The solution we propose is a novel and uniform client-cloud interaction approach by which cloud service owners, who may be different from the cloud providers, are able to fully control the usage of their services in the case of each user subscription. In this context, any cloud service can be invoked by distinct devices; therefore, the content must be adapted to various channels and end devices, in particular with respect to needs arising from mobile clients. For a quick and seamless integration between the cloud provider's identity management system and the system used by the client, we introduce the concept of a client-centric tool. An extension of the client-cloud interaction model enables client-to-client interaction (CTCI) in an almost direct way, so that the involvement of cloud services is transparent to the users.

In this chapter, we propose a formalization of this solution that incorporates the major advantages of *abstract state machines (ASMs)* and *ambient calculus* by specifying the algorithms of executable components (agents) in terms of ASMs and by describing their communication topology, locality, and mobility in the terms of ambient calculus.

1 Introduction

Nowadays "cloud computing" is one of the most often used buzzword in computing, and many providers (Amazon, Google, Microsoft, IBM, etc.) of cloud services

K. Bósa (✉) • R.-M. Holom • M.B. Vleju
Christian Doppler Laboratory for Client-Centric Cloud Computing, Johannes Kepler University Linz, Softwarepark 21, 4232 Hagenberg, Austria
e-mail: k.bosa@cdcc.faw.jku.at; r.holom@cdcc.faw.jku.at; b.vleju@cdcc.faw.jku.at

© Springer International Publishing Switzerland 2015
B. Thalheim et al. (eds.), *Correct Software in Web Applications and Web Services*, Texts & Monographs in Symbolic Computation,
DOI 10.1007/978-3-319-17112-8_4

(infrastructure as a Service (IaaS), software as a service (SaaS), platform as a service (PaaS), data as a service (DaaS), etc.) emphasize the many benefits of outsourcing applications into a (private or public) cloud. In other words, it is suggested that cloud computing represents a mature technology that is ready to be massively used. But it is our conviction that cloud computing still requires lots of fundamental research [99].

In particular, most of the offerings in cloud computing are provider centric. For instance, a client (or tenant) may rent a certain piece of infrastructure, load and execute a piece of software on it, pay for the use, and leave the cloud without leaving permanent traces. Certainly, there are many computing-intensive applications, for example, Web crawling, image processing, machine learning, etc., that fit well into such a scenario. However, if we think of a multiuser database application, its usefulness decreases significantly.

Among many other problems in cloud computing, we identified the lack of client orientation as a serious problem that needs to be addressed in research.[1] This subsumes the problems of identity of tenants, access rights, adaptivity to the needs of clients, and more. For instance, in many cases (e.g., multiuser database applications), it would be indispensable to keep knowledge of users and their rights in the authority of the client instead of in the cloud. The immediate consequence of such an approach is that cloud applications should become hybrid and distributed, as parts of data and software will reside on premise, while others reside off-site in the cloud.

There is another serious lack of formal foundations in cloud computing starting from the simple fact that key notions such as service are not defined. Therefore, in our research, we deal with the challenges in cloud computing that require solutions with more stronger client-side orientation, and we also address the fundamental research question on how a uniform formal model for clouds must look like without any bias to particular languages and technology.

In this chapter, we present the high-level formal models of our cloud-related algorithms and software solutions which have strong client-side aspects and which are integrated into a single cloud service architecture. We split the specification into three parts.

The first part lays out a cloud service infrastructure that provides a transparent and uniform way to interact with its clients (service owners and end users). This includes the following:

- The end users are able to access and combine the available functions of cloud services.
- The owners of the cloud services, who may be different from the cloud providers and who may possess exclusive access to certain cloud resources (service functionalities or data) which is shared among their end users, are able to define

[1]We define the term *client* as being a small and medium enterprise (SME) that contracts and uses any cloud service. Similarly, we refer to a *user* as an identity within the SME using a cloud-based service.

some special kind of action schemas called *service plots*, which may be specific for end-user subscriptions, respectively. By these action schemas, clients are able to restrict not only which are the permitted actions belonging to certain cloud resources (e.g., services) for certain end users, but they are also able to specify and tune precisely which are the permitted combinations of these actions to perform certain tasks.
- It is also described how we extended the formal model of the proposed cloud system with a *client-to-client interaction (CTCI)* mechanism via a cloud architecture. The discussed solution for transparent use of services is a kind of switching service, where registered cloud users communicate with each other, and the only role the cloud plays is to switch resources from one client to another.

With respect to identity management in a cloud-based approach, a problem arises in maintaining identity data across the providers. The second component of our system provides a client-focused identity meta-system based on the concept of ASMs, which allows a client to maintain a private identity directory while offering individual users automatic authentication to cloud-based services. In this part, we introduce the concept of an *identity management machine (IdMM)*, an ASM-based, client-centric, single sign-on tool, which deals with the client-side aspects of identity and access management in cloud computing.

The last but not least, the third part of our specification uses ASM ground models for presenting in a rigorous way the proposed *Web application (WA)*, which tries to solve the problem of adaptivity by including aspects regarding content adaptation and displaying. We chose to use ASMs since they permit to design and analyze asynchronous multiple-agent distributed systems, as the cloud architecture we deal with; moreover, thanks to refinement mechanism, we show how ground models can be refined to pseudo-code-like descriptions. A future refinement can possibly bring to the implementation. ASM method also provides a high-level notation that permits to concisely describe complex systems and that can be easily understood by all the stakeholders [25].

As it is shown in Sect. 3, these novel interaction solutions are dynamically reconfigurable, and they can either take place on a cloud or can be easily shifted to the client side and wrapped into a middleware software which may be located in the authority of one or more clients. The latter case can also provide enhanced privacy protection, since the cloud provider does not necessarily know the (real) identities of the end users.

In order to achieve this required flexibility of the model, the component formalization was done in terms of ambient abstract state machines [26, 31], which inherently makes possible to create such dynamically variable architectures. This method incorporates the major advantages of the *abstract state machines (ASMs)* [26, 64] and of *ambient calculus* [38, 61]. Namely, one can describe formal models of complex dynamically reconfigurable distributed (or cloud) systems including mobile components in two abstraction layers such that while the algorithms of executable components (agents) are specified in terms of ASMs, their communication topology, locality, and mobility are described with the terms of ambient calculus in our method. The abstracted formal architecture presented in this chapter is also going to be served as a framework which can be enriched with other novel client-centric mechanisms in the future.

The rest of the chapter is organized as follows: Section 2 gives on overview on the basic notions of cloud computing and on access control techniques for cloud. Section 3 informally summarizes the components of our integrated cloud architecture. Section 4 introduces the applied formal approaches and gives a short overview on ambient calculus and ambient ASM as well as defines some nonbasic ambient capability actions which are applied in the latter sections. Section 5 describes our high-level ambient ASM model of our cloud service architecture which is equipped with service plots and client-to-client interaction via cloud. Section 6 discusses the specification of client-centric identity and access management for cloud services. Section 7 deals with the ASM ground models of the software components for cloud service adaptivity. Section 8 highlights some aspects of verification of the formal model of our integrated cloud system, which we are going to perform in the near future. Finally, Sect. 9 discusses the related work, and Sect. 10 concludes this chapter.

2 Cloud Computing and Access Control Techniques for Cloud

In this chapter, we give a short overview on the basic notions of cloud computing and on various cloud-related access control techniques in order to make this chapter more self-containing as well as to facilitate its understanding for a wider audience.

2.1 Cloud Computing

Cloud computing appeared years ago as a buzzword in computing technology, being related to other existing technologies, like grid computing and seen as a scalable external data center [114]. Even though people are referring to cloud computing since some time and many providers (Amazon, Rackspace, Google, Microsoft, IBM, etc.) are suggesting that it’s a mature technology, we recognize cloud computing as still an evolving paradigm [80].

The *National Institute of Standards and Technology (NIST)* is mentioning important aspects of cloud computing in their definition: “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models” [80].

Further on, they mention the cloud computing characteristics: *on-demand self-service*, a consumer can automatically provision computing capabilities without the need for human interaction; *broad network access*, services are available over the network and accessed through a variety of mechanisms (e.g., mobile phones, tablets, laptops, and workstations); *resource pooling*, a provider's capability to use a multi-tenant model for pooling computing resources in order to handle multiple clients; *rapid elasticity*, the possibility to rapidly scale (in and out) resources; and *measured service*, providers are using resources monitoring in order to optimize their services and support the pay-per-use business model. Specific measurements are used, fitting the type of service (e.g., storage, processing, bandwidth, and active user accounts).

Cloud providers offer their services (as a utility) conforming to the following models: *software as a service (SaaS)*, *platform as a service (PaaS)*, and *infrastructure as a service (IaaS)*. SaaS is a service model where a client uses applications offered by the provider that run on a cloud infrastructure. Examples of this service comprise Google Apps for Business [57] and Microsoft Office 365 [81]. PaaS offers the client the possibility to deploy on a cloud infrastructure his or her applications created using predefined languages, libraries, services, and tools that are supported by the cloud provider. Microsoft Azure [15] and Google App Engine [60] are two examples of such services. IaaS is a service model, which provides the client fundamental computing resources, like processing, storage, and networks. In this case, the client can also deploy operating systems, which in the case of PaaS is not possible. An example of such a service model is the Amazon Elastic Compute Cloud (Amazon EC2) [10].

There are also different deployment models of a cloud: private cloud (internal data centers designed for a single organization), community cloud (used by several organizations with shared interests), public cloud (made available for the general public), and hybrid cloud (a combination of the all previous models: some parts of the infrastructure are usable for the general public, while others are ready to use only for some organizations).

Some of the disadvantages that come together with the adoption of cloud services are the following [46, 52]: loss of governance, provider lock-in, isolation failure, and security and privacy issues (insecure interfaces and APIs, malicious insiders, shared technology issues, data loss or leakage, account or service hijacking, unknown risk profile).

2.2 Access Management in Cloud Computing

A detailed description of identity and access management in cloud computing can be found in [109]. The author describes four distinct identity and access management scenarios within the domain of cloud computing. The first scenario entails a traditional model where the client maintains a local identity and access management system which is mirrored on the cloud provider. Such a model is useful only in IaaS where the client has the ability to install custom software on his or her virtual machine. The second scenario entails a trusted model where the identity information is shared to a cloud provider via the use of previously agreed-upon identity federation protocols. In an identity provider scenario, the client either acts or uses an external identity provider which makes use of common identity federation protocols (such as OpenID, OAuth, SAML) to offer the identity-related information to cloud services. Finally, the author of [109] describes a fourth model where the client makes use of a cloud provider's identity and access management system. This system of *all in the cloud* means that the client has no control over the data and must subscribe to a provider's identity and access management system. Using multiple services across multiple providers further increases the task of maintaining the correct identity information data.

The authors of [63] provide a review of identity and access management in cloud computing. They provide a description of identity and access management as well as a description of existing tools for identity and access management. They then provide a review of the existing identity and access management tools in cloud computing. This review is extended by [83] where the authors better described the technologies and protocols used in identity and access management emphasizing the protocols for identity federation. For small and medium enterprises, [79] describes the challenges when adopting cloud computing services. The authors provide a detailed description of identity and access management in general, also emphasizing the existing identity federation and open standards.

While the authors of [63, 79, 83] emphasize the use of identity federation and existing open standards, it must be noted that most cloud providers tend to use the "all in the cloud" approach when it comes to their identity and access management systems. As such, most providers will offer their own identity system, often not providing the option of using identity federation protocols such as OpenID or OAuth. When such protocols are offered, the providers often act just as identity providers and not relaying parties [50].

3 Overview of the Client-Cloud Interaction Software System

Roughly the formal model of cloud systems employed by us can be regarded as a pool of resources equipped with some infrastructure services. Depending whether these abstract resources represent only physical hardware and virtual resources or the entire computing platforms, the model can be an abstraction of IaaS or PaaS, respectively. The basic hardware (and software) infrastructure is owned by the cloud provider, whereas the software running on the resources may be either rented or owned by some tenants of the cloud (service owners). We assume that these software products may in turn be offered as *services* (denoted by $S_1 \dots S_i$ in Fig. 1) and thus used by some groups of cloud users.

Accordingly, we apply a loose definition of the term service cloud here, where an entity who is different from the cloud provider and who has disposal of the access rights to some hardware or software resources running on the cloud may become

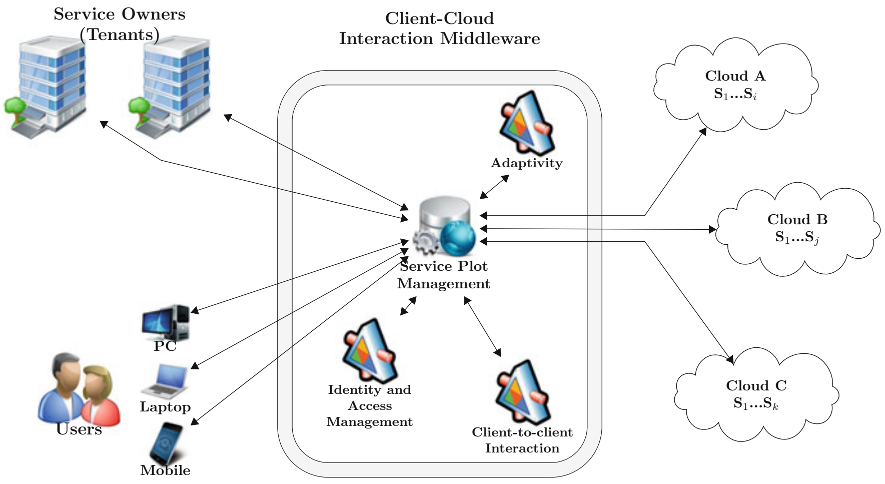


Fig. 1 Architecture

a cloud service provider. Thus, from this aspect, the model can be regarded as an abstraction of a mixture of SaaS and of IaaS (or a mixture of SaaS and PaaS).

We make a distinction between cloud *service owners* (or tenants) and cloud *users* (or end users); see Fig. 1. Users are registered in the cloud, and they subscribe to and use some (software) services available in the cloud. Service owners are usually *small and medium enterprises (SMEs)* that contract with cloud providers, rent some cloud resources, and/or bring and deploy their own resources (e.g., software service instances, data, etc.) on the cloud which they share among their end users. Of course, service owners can also act as normal users, which means they can use services provided by other tenants.

A developed prototype of this cloud service system has been deployed for testing purposes on Windows virtual machines under OpenStack cloud infrastructure software on an IBM CloudBurst server machine.

Due to the applied ambient concept, the relocation of the system component is trivial, and we can apply our model according to different scenarios (see Fig. 2a, b). In our developed prototype and in our case studies (see Sect. 3.4), all our novel software solutions discussed in this chapter are integrated into a compound software component on the client side called the *client-cloud interaction middleware*, which takes place among the end users, the service owners, and the cloud(s) in order to manage the interactions between them; see Fig. 1. In this middleware-based architecture, the various components (e.g., service plot management, client-to-client interaction feature, identity and access management, and content adaptivity) are only loosely coupled with each other; any of them can be eliminated and the others still remain functional.

One of the advantages of the employed middleware-based configuration is that the same instance of the infrastructure services is able to extend the functionalities

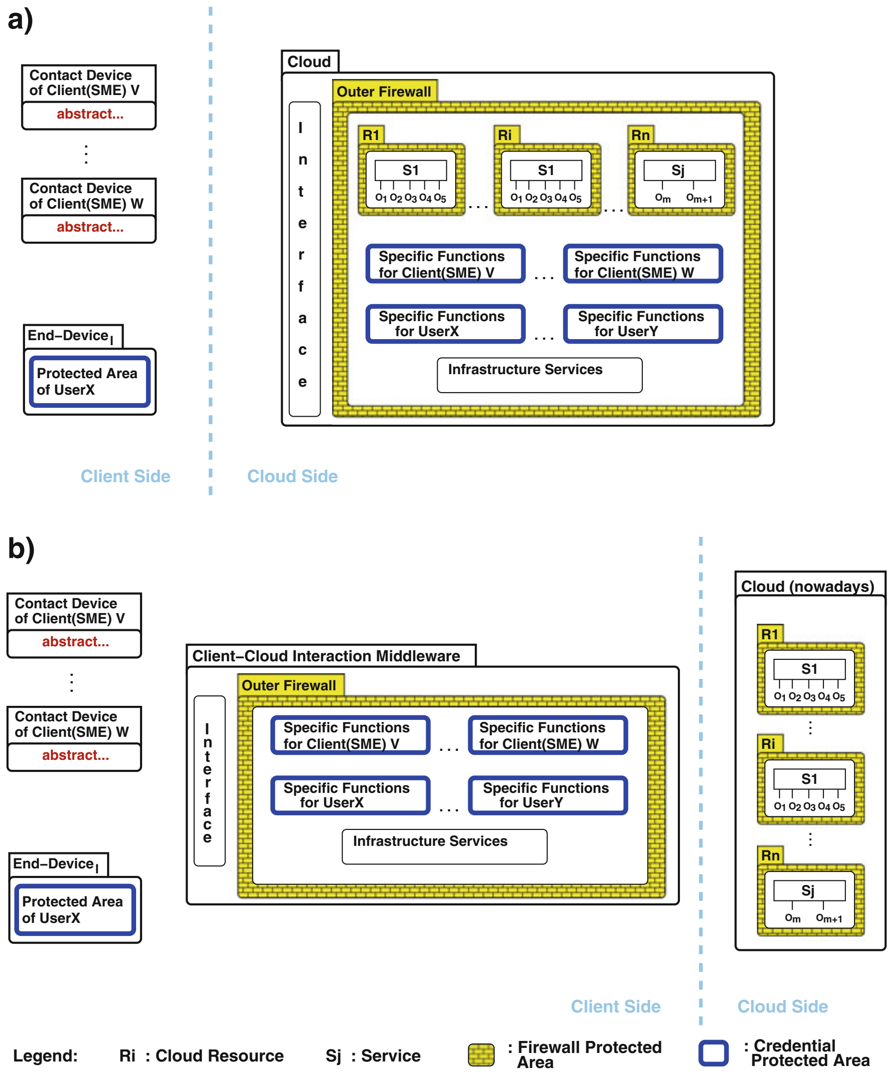

a)
Contact Device of Client(SME) V
abstract...
Contact Device of Client(SME) W
abstract...
End–Device$_l$
Protected Area of UserX
Cloud
Interface
Outer Firewall
R1
S1
o_1 o_2 o_3 o_4 o_5
Ri
S1
o_1 o_2 o_3 o_4 o_5
Rn
Sj
o_m o_{m+1}
Specific Functions for Client(SME) V
Specific Functions for Client(SME) W
Specific Functions for UserX
Specific Functions for UserY
Infrastructure Services
Client Side
Cloud Side
b)
Contact Device of Client(SME) V
abstract...
Contact Device of Client(SME) W
abstract...
End–Device$_l$
Protected Area of UserX
Client–Cloud Interaction Middleware
Interface
Outer Firewall
Specific Functions for Client(SME) V
Specific Functions for Client(SME) W
Specific Functions for UserX
Specific Functions for UserY
Infrastructure Services
Cloud (nowadays)
R1
S1
o_1 o_2 o_3 o_4 o_5
Ri
S1
o_1 o_2 o_3 o_4 o_5
Rn
Sj
o_m o_{m+1}
Client Side
Cloud Side
Legend:
Ri : Cloud Resource
Sj : Service
: Firewall Protected Area
: Credential Protected Area

Fig. 2 Application of our model according to different scenarios. (**a**) Scenario I. (**b**) Scenario II

of and to provide a transparent and uniform access to more than one cloud. It should be noticed that the end users cannot bypass and omit the middleware and access to the cloud services, since they do not have direct access to the cloud(s) (they have credentials only to the middleware, and the middleware assigns permanently or temporarily their credentials to cloud service accounts).

In the following, we give an overview of various components of our integrated cloud service architecture, whose formal specifications are discussed in this chapter.

3.1 A Cloud Architecture Equipped with Service Plots and a Client-to-Client Interaction Feature

There are two major problems which we address with the discussed cloud architecture model [30]: on the one hand, how to provide a transparent and uniform way to interact with the end users, such that it allows the users to access and combine the available functions of cloud services which may belong to various cloud service owners (or tenants), and on the other hand, how to give full control into the hands of the tenants over the usage of the cloud resources which they own or rent on the cloud.

In our model, we assume that service instances are always equipped with *service operations* denoted by unique identifiers $o_1,\ldots, o_n$; see Fig. 2a. These service operations actually compose the interface of a service instance, and they are exported from a service to be used by other systems or directly by users.

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

The received service plots, which may be composed individually for each subscribing user by service owners, are collected with other cloud functions available for this particular user in a kind of personal user area by the cloud; see Fig. 2a. Later, when the subscribed user sends a service request, it is checked whether the requested service operations are allowed by any service plot. If a requested operation is permitted, then it is triggered to perform; otherwise, it is blocked as long as a plot may allow to trigger it in the future. Each triggered operation request is authorized to enter into the user area of the corresponding service owner to whom the requested service operation belongs. Here, a scheduler mechanism assigns to the request a one-off access to a cloud resource on which an instance of the corresponding service runs. Then the service operation request is forwarded to this resource, where the request is processed by an instance of the service whose operation was requested. Finally, the outcome of the performed operation returns to the area of the initiator user, where the outcome is either stored or sent further to a given end-user device.

In this way, the service owners have direct influence on the service usage of particular users via the provided service plots. If a user subscribes to more than one service, he or she may have access to more than one plot. These plots are independent from each other and they can be applied concurrently. If a service owner makes available more than one service for a user, the owner has the choice either to provide independent plots for the user or to combine some functions of various services into a common service plot. This conceptual solution shows a transparent and uniform way how to provide an advanced access control mechanism for cloud services without giving up the flexibility of heterogeneous cloud access to these services.

Regarding our proposed cloud service model, 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.), since these systems rule the market at present and they will have impact on the cloud business in the near future as well. Since due to the applied ambient concept the relocation of the system component is trivial, we can apply our model according to different scenarios. For instance, all our novel methods including our client-cloud interaction solution can be shifted to the client side and wrapped into a middleware software which takes place between the end users and cloud in order to control their interactions; see Fig. 2b (this scenario was implemented in the mentioned software prototype). Note that the specified communication topology among the distributed system components remains the same in both proposed scenarios.

Thinking further this second scenario, we can envisage a new (cloud) service whose customers are enterprises that take over the role of the service owners in this service, such that they can fully control how their employees can access and use third-party clouds. Namely, assuming that these enterprises own or pay for various cloud services, they can provide customized service plots for their employees via such a client-cloud interaction controller service to restrict end-user accesses to the available functions of these cloud services.

3.1.1 Client-to-Client Interaction

We also extended the model of our cloud service architecture with a *client-to-client interaction (CTCI)* mechanism via a cloud architecture [29]. Our envisioned cloud feature can be regarded as a special kind of services we call *channels*, via which registered cloud users can interact with each other in an almost direct way and, what is more, they are able to share available cloud resources among each other as well.

Some use cases, which may claim the need of such CTCI functions, can be, for instance, disseminating large or frequently updated data whose direct transmitting meets some limitations or connecting devices of the same user (in the latter case, an additional challenge can be during a particular interpretation of the modeled CTCI functions, how to wrap and transport local area protocols, like *upnp* via the cloud). See an overview of the formal model of CTCI in Sect. 5.3.

3.2 Client-Centric Identity and Access Management in Cloud Computing

The adoption of cloud-based services offers many advantages for small and medium enterprises. However, a cloud-based approach also entails certain disadvantages. The papers [7, 34, 46, 99] outline some of these disadvantages: loss of control, contracting issues, provider lock-in, and other security and privacy issues. Such issues imply an extra level of trust between a client and a cloud provider. With respect to identity management, a loss of control implies that the client must trust the cloud provider with sometimes critical or important identity information such as credit card information. A potential client would prefer such data to be stored on premise and only be offered to a service on demand. The vendor lock-in issue might be mitigated by the adoption of services across multiple providers. In this scenario, a problem arises in maintaining identity data across the providers. Changing the surname, for example, entails changing this property for each service in particular (especially if the service provider does not adopt open standards). Our research is focused on providing a client-centric identity meta-system which allows a client to maintain a private identity directory while offering individual users automatic authentication to cloud-based services via a single sign-on, privacy-enhanced service.

In the paper [115], we have introduced the concept of an identity management machine (IdMM), an ASM-based, client-centric, single sign-on tool for small and medium enterprises that want to adopt or migrate to cloud-based services. This concept has been further refined in the paper [116] where we described the architecture of the IdMM. As mentioned in the paper [116], the IdMM is composed of six agents: the core agent (comprising the rules described in the paper [115]), the client agent (managing the interaction with the client's directory), the cloud agent (used for the interaction with a cloud service), the user agent (handling the interaction with an individual user), the protocol agent (used for protocol-based authentication), and the provisioning agent (managing user provisioning, password resets, and user de-provisioning). Apart from the core agent, each agent is defined by further refinement of the abstract functions presented in the paper [115]. This process is still an ongoing task, with the provisioning and protocol agents still left at an abstract level. In the paper [117], we have described the IdMM$_{Client}$ agent by further refinement of the client-side abstract functions. We also gave an example of the IdMM$_{Client}$'s interaction with an ApacheDS LDAP directory [11].

The IdMM makes an abstraction of the protocols used by both the client and cloud provider for their identity management systems via the use of the abstract functions presented in the paper [115]. Such functions leave the organizational and implementation aspects of the identity management systems directly into the hands of the end parties. To describe these functions, we must first consider the interaction between a client and a cloud provider with respect to identity management, authentication, and authorization. We consider three distinct cases of client-to-cloud interaction: the direct case, the obfuscated case, and the protocol-based case.

3.2.1 Direct Client-to-Cloud Interaction

As showed by the authors of the papers [7, 34, 50], one of the greatest issues surrounding identity management for cloud providers is the need of the cloud provider to control the customer experience. Many providers make use of their own custom-designed identity systems to which a client must subscribe. This means that the client has no choice but to use the cloud provider identity system. While this may be an inconvenience from a privacy point of view, the real problem lies in managing the client's information across multiple providers. A simple change, such as changing a user's address, entails changing the value on every single provider the client uses. Any change made on the client side must also be made on the cloud via the synchronization of attributes. Concurrently, any changes made by the provider (such as the addition, replacement, or removal of an attribute) must be reflected in the client's directory system.

3.2.2 Obfuscated Client-to-Cloud Interaction

While the direct client-to-cloud interaction allows for an efficient use of cloud services, it does suffer from a lack of privacy. Since all information about a client is stored on the provider's infrastructure, there is an increased risk that through data leakage or unauthorized access, that information could fall into the wrong hands. We mitigate this threat by introducing the concept of obfuscated identities.[2]

We consider an identity to be real if information contained corresponds to the identity's owner and is visible to any external entity. An obfuscated identity has its information obfuscated. Depending on the method of obfuscation, the information is either undecipherable or can only be deciphered by the owner of the identity. We also consider a third kind of identity, a *partially obfuscated identity*. Such an identity contains a mixture of real and obfuscated data. An example for the use of real identities can be found in the usage of online stores where the information must be accurate in order to process the payment and shipping. By contrast, one could use obfuscated identities for a free online storage service. If, for example, the storage service requires an age restriction, then one could use a partially obfuscated identity where only the date of birth is real.

The usage of obfuscated and partially obfuscated identities is dependent on the cloud service. As mentioned, some services do require real identities. Even if the service can be used with obfuscated identities, we leave the matter in the hands of the client. The client can choose whether or not to use obfuscation in such cases.

[2]For the purpose of this paper, we consider the definition of an identity as explained in paper [112].

For partially obfuscated identities, we also allow the client to choose what real data attributes are sent to a service.

3.2.3 Protocol-Based Client-to-Cloud Interaction

In recent years, there has been a drive to improve interoperability between cloud providers mostly to prevent vendor lock-in. From an identity management perspective, the result has been the adoption of some open-based protocols to facilitate both cloud interoperability as well as identity access management. Protocols such as OAuth or OpenID represent an important tool for a client-centric identity management system. They allow the provider to focus on the requirements of the service while allowing access via the standard implementation of these protocols. From the client's perspective, such protocols allow for an easier integration across multiple cloud providers. It must be noted however that such protocols do suffer from a variety of security and privacy issues, as described in [50, 88].

3.3 Cloud Content Adaptivity

This subsection is giving an overview of the problem of adaptivity to different services, devices, preferences, and environments. By adaptivity we understand that all the services provided by the cloud to the client should be adapted on the fly to the different contexts mentioned above.

The previously specified problem (initially presented in the paper [43]) is described in Fig. 3. As an example, we use a database manager application, which is deployed on the cloud. The user should be able to use the application (we can

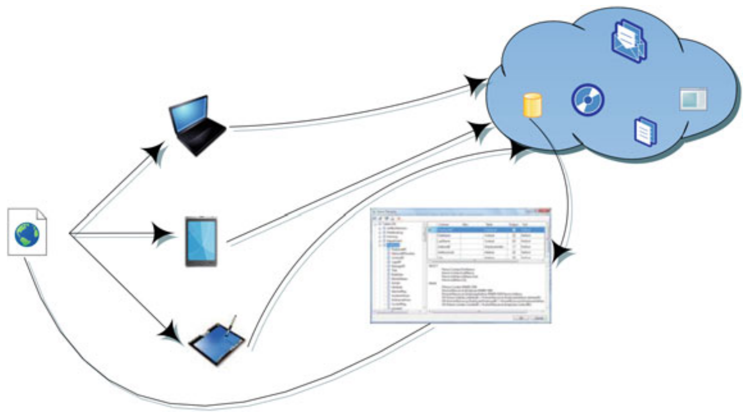


Fig. 3 Problem description

think of an application as of a list of services available in the cloud for that user) from any device he or she logs in without having to install any other application. How can we make the cloud services available on all devices? We need a general application that will adapt to different end devices on the fly. The implementation of a *Web application (WA)* comes as a solution to the client-cloud interaction, because it transfers data among different software components, which execute on different devices, using only the browser [85]. However, WAs do not have a precise definition or a precise model to follow, because they are related to various standards and implementation frameworks. To tackle this problem, we propose the conception of a formal model prior to code development that would include a rigorous analysis. Using formal modeling and verification, we want to guarantee reliability properties in order to ensure that, e.g., the client will receive the same (or as similar as possible) output independently of the device he or she is using. We show how requirements can be captured with ASM ground models and how the refinement method can be applied in order to link the ground models to pseudo-code. Together with the refinement method, the ASM ground models are generating a documentation which can be used for inspection, reuse, and maintenance.

We decided to use ASMs, instead of other modeling methods (e.g., *Unified Modeling Language (UML)*, *finite-state machine (FSM)*), for realizing the design of the client-cloud interaction system, because with them, we can prove that the system's development is correct and reliable (we can check if the WA under development behaves as expected). ASM method [25]:

- offers the possibility to design and analyze both procedural single-agent and asynchronous multiple-agent distributed systems (as the cloud framework we deal with). In ASMs, an action can be replaced in a refinement step by multiple parallel actions [22], which means that by going from the current state to the next state, the set of rules are executed simultaneously [23]. Because of these attributes, the ASM method was chosen over the FSM.
- creates a high-level modeling at the level of abstraction (allowing to describe complex systems, which can be easily understood by all the stakeholders) and links the descriptions in a chain of coherent system models (that can possibly bring to the implementation) using stepwise refinement. The former characteristic improves upon the loose character of UML description, and the second one also fills in a breach in UML [25].
- supports rigorous model validation and verification.

3.4 Cloudification Case Studies

In order to show and prove the feasibility and viability of our approach for controlling client-cloud interaction and managing access control, we have accomplished two cloudification case studies.

In the first case study, we took a stand-alone application called visual SQL [45, 66, 111], whose source code was available for us. Visual SQL is a software tool which provides a graphical database description language to facilitate intelligent conceptual diagramming of database queries. We adapted this software to an existing cloud architecture and made it interoperable with the prototype of our *client-cloud interaction middleware* software such that it provides access control for this cloudified application. This work shows how software applications which were originally designed for single usage on a local desktop use can be adapted to a cloud infrastructure and equipped with a sophisticated access control mechanism for multiple users.

The second case study was related to Office 365 which is a subscription-based cloud service provided by Microsoft and which integrates other Microsoft online services together such as Exchange Online, SharePoint Online, Lync Online, Web Apps, and Office Professional Plus. We combined Office 365 with our client-cloud interaction middleware such that its functionality was extended with an access control mechanism, which is more sophisticated than its own. Namely, Office 365 provides only a limited authority for customers/admin to control their users.

This advanced access control which is achieved by the application of the client-cloud interaction middleware cannot be bypassed by the end users, since they have direct access only to the middleware, but not to the purchased Office 365 service package of any customer (company) of client-cloud interaction middleware. For instance, customers of client-cloud interaction middleware can restrict which end user can make a copy from e-mails received in his or her Office 365 mailbox, who can attach document stored in Office 365 to e-mails, or who can send e-mails to external e-mail addresses using Office 365 mail service. We should emphasize that this work was carried out in the way that we discovered and used only the publicly available programming interfaces and tools for Office 365 and we did not get any extra support from Microsoft.

An implemented prototype in the cases of both case studies, respectively, was deployed for testing purposes on Windows virtual machines under OpenStack cloud infrastructure software on an IBM CloudBurst server machine as well. It is beyond the scope of this chapter to describe these case studies and their formal specification, but we refer to [103], which discusses the first case study and its formal model in details.

4 Preliminaries

In this section, we give a short summary on the applied formal approach as well as ambient calculus and ambient ASM in order to facilitate the understanding of the latter sections.

4.1 The Applied Formal Approach

For our research, we searched for a software engineering method by which both the algorithms of the concurrent system components and the dynamic topology of complex distributed and cloud systems can be formally described. As a first step, we investigated the following two formal approaches:

- One of the most outstanding methods for formal modeling of distributed components of (mobile) network applications is a calculus of mobile agents called *ambient calculus* [38, 61]. This concept is simple, succinct, and sufficient enough to efficiently describe locality as well as phenomena related to mobility.
  Additionally, besides mobility, ambient calculus inherently supports the reasoning of high-level abstraction of many security considerations which can be defined by formulating the ability or inability of various entities to cross certain physical or virtual barriers [37] (e.g., certain combinations of some ambient expressions can be interpreted as firewalls that filter traffic according to some security policies, others can be regarded as abstractions of certain access controls or of ciphertext decodings, etc.). But one of its main drawbacks is that the ambient calculus is not capable to treat the algorithmic specification of the executable agents which appear in its ambient constructs.
- For this latter purpose, an obvious candidate would be the mathematically well-funded and efficient software engineering method called *abstract state machines (ASMs)* [25, 64]. ASMs have already demonstrated their usefulness and their viability in many fields, for example, in the formal definition of sequential [65] and parallel algorithms [19, 20], giving comprehensive high-level definition and analysis (verification and validation) of Java programming language and of the Java virtual machine [108], and modeling Web services [9]. However, there is a limitation in ASMs to describe dynamically changing hierarchical structures of components in distributed systems and to express some system properties which depends on their distribution in space (e.g., local deadlock).

In [26], the ambient concept (notion of “nestable” environments where computation can happen) is introduced into the ASM method. In that article, it is also shown how to encode the mobile ambients of ambient calculus in terms of *ambient ASM* rules. Since one of the main goals of [26] is to reveal the inherent opportunities of the new ambient concept introduced into ASMs, the presented definitions for moving ambients are unfortunately incomplete.

In [31], we extended and completed the ASM rules mentioned above, such that they fully capture the ambient calculus. By this, a new method is created in terms of ASM rules, in which one is able to describe formal models of distributed systems including mobile components in two different abstraction layers (see Fig. 4). This means that while the algorithms and local interactions of executable agents are given in terms of ASMs, the long-term interactions and movements of system components via various administrative domains are specified with the terms of ambient calculus

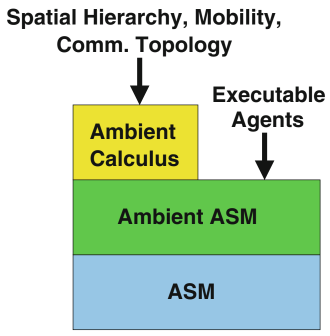


Fig. 4 Abstraction layers in the chosen formal approach

in our approach. This novel method makes possible in a given model that:

- the agents may not only be arbitrarily placed and linked initially, but the composed structure is rearranged from time to time by the programs of agents residing in it;
- the behavior of agents is influenced by their current spatial locations and communication topology;
- one can express visibility and access conditions on ASM agents (or on some administrative boundaries) explicitly.

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

4.2 Ambient Calculus

The ambient calculus [38] was inspired by the π-calculus [82], but it focuses primarily on the concept of locality and process mobility across well-defined boundaries instead of channel mobility as π-calculus. The concept of *ambient* stands in the center of the calculus; see a summary of the definition of ambient calculus in Table 1.

The ambient calculus includes only the mobility and communication primitives depicted in Table 1(A).[3] 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).

[3]Name restriction creates a new (unique) name n within a scope P. One must be careful with the term $!(\nu\ n)P$, because it provides a fresh value for each replica, so $(\nu\ n)!P \neq !(\nu\ n)P$.

Table 1 Definition of ambient calculus

(**A**) The mobility and communication primitives		(**B**) Reduction (operational semantics)
P, Q, R::=	Processes	$P \equiv P', Q \equiv Q', P \longrightarrow Q \Longrightarrow P' \longrightarrow Q'$
$P \mid Q$	Parallel composition	
$n[\ P\ ]$	Ambient	$P \longrightarrow Q \Longrightarrow P \mid R \longrightarrow Q \mid R$
$(\nu\ n)P$	Restriction of name n within P [3]	
0	Inactivity (**skip** process)	$P \longrightarrow Q \Longrightarrow n[\ P\ ] \longrightarrow n[\ Q\ ]$
$!P$	Replication of P	
$M.P$	(Capability) action M then P	$P \longrightarrow Q \Longrightarrow (\nu\ n)P \longrightarrow (\nu\ n)Q$
$(x).P$	Input action (the input value is	
	Bound to x in P)	$n[\textsc{In}\ m.P\ \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$
$\langle a \rangle$	Async output action	
$M_1.\ \ldots\ .M_k$	A path formation on actions	$m[\ n[\ \textsc{Out}\ m.p\ \mid Q]\mid R] \longrightarrow n[P \mid Q] \mid m[R]$
M::=	Capabilities	$\textsc{Open}\ n.P \mid n[\ Q\ ] \longrightarrow P \mid Q$
$\textsc{In}\ n$	Entry capability (to enter n)	
$\textsc{Out}\ n$	Exit capability (to exit n)	$(x).P \mid \langle a \rangle \longrightarrow P(x/a)$
$\textsc{Open}\ n$	Open capability	
	(To dissolve n's boundary)	

(**C**) Structural congruence (operational semantics)	
$P \equiv P$	$P \equiv Q \Longrightarrow Q \equiv P$
$P \equiv Q, Q \equiv R \Longrightarrow P \equiv R$	$P \equiv Q \Longrightarrow P \mid R \equiv Q \mid R$
$P \equiv Q \Longrightarrow n[\ P\ ] \equiv n[\ Q\ ]$	$P \equiv Q \Longrightarrow !P \equiv !Q$
$P \equiv Q \Longrightarrow (\nu\ n)P \equiv (\nu\ n)Q$	$P \equiv Q \Longrightarrow M.P \equiv M.Q$
$P \equiv Q \Longrightarrow (x).P \equiv (x).Q$	$P \mid Q \equiv Q \mid P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$!P \equiv P \mid !P$
$(\nu\ n)(\nu\ m)P \equiv (\nu\ m)(\nu\ n)P$	$(\nu\ n)(P \mid Q) \equiv P \mid (\nu\ n)Q \quad \texttt{if}\ n \notin fn(P)$
$(\nu\ n)(m[\ P\ ]) \equiv m[\ (\nu\ n)P\ ] \quad \texttt{if}\ n \neq m$	$P \mid 0 \equiv P$
$!0 \equiv 0$	$(\nu\ n)0 \equiv 0$

An ambient is defined as a bounded place where computation happens. An ambient is written as $n[P]$, where n is its name, which can be used to control access (entry, exit, communication, etc.), and a process P is running inside its *body* (P may be running even if n is moving). Ambient names may not be unique. Ambients can be embedded into each other such that they can form a hierarchical tree structure. An ambient body is interpreted as the parallel composition of its elements (its local ambients and its local agents) and can be written as follows:

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

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). An action defined in the calculus can precede a process P. P cannot start to execute until the preceding actions are performed. Those actions that are able to control the movements of ambients in the hierarchy or to dissolve ambient boundaries are restricted by capabilities. By using capabilities, an ambient can allow some processes to perform certain operations without publishing its true name to them (see the entry, exit, and open in Table 1). In case of the modeling of a real-life system, communication of (ambient) names should be rather rare, since knowing the name of an ambient gives a lot of control over it. Instead, it should be common to exchange restricted capabilities to control interactions between ambients (from a capability, the ambient name cannot be retrieved).

4.3 Ambient ASM

The core idea of ambient ASM [26] is to introduce an implicit parameter *curamb* to each location expressing a context for evaluation of terms and execution of machines. Analogously to conventional implicit object-oriented parametrization (e.g., $\mathit{this}.f(x) = f(x)$), the *dot term* $s.t$ is introduced, where s is a term standing for an ambient expression, 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 a given argument do not depend on any ambient expression.

An ASM is an ambient ASM if it can be obtained from a basic ASM [25] by allowing for every given machine P also a machine of the following form:

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

where execution of P is performed in the ambient *exp* (*exp* is bound to *curamb*). Additionally, the notation $n[\ P\ ]$ introduced by Cardelli and Gordon [38] is also defined in 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 [26].

4.3.1 Moving Ambients

In [26], an ASM machine called MobileAgentsManager is described as well, which gives a natural formulation for the reduction of the three basic capabilities (Entry, Exit, and Open) of ambient calculus in terms of ambient ASM rules. For this machine, an ambient tree hierarchy is always specified initially in a dynamic-derived function called *curAmbProc*. The machine MOBILEAGENTSMANAGER transforms the current value of *curAmbProc* according to the capability actions given in *curAmbProc*.

In [31], we presented an extended version of the ASM machine MOBILEAGENTS-MANAGER, which fully captures the calculus of mobile agents[4] and which is also able to interpret (in the corresponding ambient contexts) the agents' algorithms given in terms of ASM syntax in *curAmbProc*. This updated ASM machine is called EXTENDEDMOBILEAMBIENTMANAGER.

One of consequences of this is that one is able to describe formal models of distributed systems including mobile components in the mentioned two abstraction layers and apply some experimental validation on these models. This means that:

- a particular system model can be defined as part of the value of *curAmbProc* in terms of ambient calculus terms (like what we did in Sect. 5.2) and in terms of ASMs;
- either models of some external instruments or some (user) actions (see in Sect. 5.1) can additionally be added to *curAmbProc* in parallel composition with the given model (they will act the role of the environment of the system in a particular case).

The value of *curAmbProc* will serve as input for the ASM machine EXTENDEDMOBILEAMBIENTMANAGER, which will transform the given value of *curAmbProc* (by performing the possible reduction steps and by parsing the specified ASMs) in order to check how the defined model interacts with its environment and how it behaves in a particular situation.

4.4 Definitions

As Cardelli and Gordon showed in [38], the ambient calculus with the three basic capabilities (ENTRY, EXIT, and OPEN) is powerful enough to be Turing complete. But for facilitating the specification of such a compound formal model as a model of a cloud infrastructure, we defined some new *nonbasic capability* actions encoded in terms of the three basic capabilities.

All the new ambient calculus capabilities defined in this section either directly appear in the presented cloud model in Sect. 5.2 or serve as a basis and were utilized for the definitions of one or more subsequent capabilities presented in this section as well. Table 2 summarizes the definitions of these nonbasic capabilities.

[4]Besides the three basic capabilities, the reductions of name restriction, input action, and asynchronous output action as well as the structural congruence for replication are also defined in terms of ambient ASM rules.

Table 2 A summary of the definitions of some nonbasic capabilities

Names	New reduction relations (based on the definitions)	Definitions of the new capabilities
1. Renaming	$n[\ n\ \textsc{Be}\ m.P \mid Q\ ] \longrightarrow^{*} m[\ P \mid Q\ ]$	$n\ \textsc{Be}\ m.P \equiv (\nu\ s)(s[\ \textsc{Out}\ n \mid m[\ \textsc{Open}\ n.\textsc{Out}\ s.P\ ]\ ] \mid \textsc{In}\ s.\textsc{In}\ m)$
2. Seeing	$n[\ ] \mid \textsc{See}\ n.P \longrightarrow^{*} n[\ ] \mid P$	$\textsc{See}\ n.P \equiv (\nu\ r, s)(\ r[\ \textsc{In}\ n.\textsc{Out}\ n.r\ \textsc{Be}\ s.P\ ] \mid \textsc{Open}\ s\ )$
3. Wrapping	$n[\ m\ \textsc{Wrap}\ n.P\ ] \longrightarrow^{*} m[\ n[\ P\ ]\ ]$	$m\ \textsc{Wrap}\ n.P \equiv$ $(\nu\ s,\ r)(\ s[\ \textsc{Out}\ n.\textsc{See}\ n.s\ \textsc{Be}\ m.r[\ \textsc{In}\ n\ ]\ ] \mid \textsc{In}\ s.\textsc{Open}\ r.P\ )$
4. Allowing code	$\textsc{Allow}\ \mathit{Key}.P \mid \mathit{Key}[\ Q\ ] \longrightarrow^{*} P \mid Q$	$\textsc{Allow}\ \mathit{Key}.P \equiv \textsc{Open}\ \mathit{Key}.P$
5. Drawing in (an Ambient)	$m[\ Q \mid \textsc{Allow}\ \mathit{Key}\ ] \mid n[\ n\ \textsc{DrawIn}_{key}\ m.P\ ]$ $\longrightarrow^{*} n[\ Q \mid P\ ]$	$n\ \textsc{DrawIn}_{key}\ m.P \equiv\ \ \mathit{Key}[\ \textsc{Out}\ n.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \textsc{Allow}\ m.P$
6. Drawing in Then Release a Lock	$m[\ Q \mid \textsc{Allow}\ \mathit{Key}\ ] \mid n[\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ \mathit{lock}.P\ ] \longrightarrow^{*} \mathit{lock}[\ n[\ Q \mid P\ ]\ ]$	$n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ \mathit{lock}.P \equiv\ \ \mathit{Key}[\ \textsc{Out}\ n.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \textsc{See}\ m.\mathit{lock}\ \textsc{Wrap}\ n.\textsc{Allow}\ m.P$
7. Concurrent Server Process	$m[\ Q \mid \textsc{Allow}\ \mathit{Key}\ ] \mid \textsc{Server}^{n}_{key}\ m.P \longrightarrow^{*} \textsc{Server}^{n}_{key}\ m.P \mid n^{uniq}_{k}[\ Q \mid P\ ]$	$\textsc{Server}^{n}_{key}\ m.P \equiv (\nu\ \mathit{next})(\mathit{next}[\ ] \mid !(\nu\ n)(\textsc{Open}\ \mathit{next}.n[n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ \mathit{next}.P])\ )$

4.4.1 Applied Notations

In the rest of this chapter, the term $P \longrightarrow^{*} Q$ denotes multiple reductions. In addition, $P \overset{asm}{\longrightarrow}^{*} Q$ denotes one or more steps of some ASM agents. We also apply the following abbreviations:

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

4.4.2 Nonbasic Capabilities

Below we give an informal description of each nonbasic capability in Table 2. It is beyond the scope of the chapter to present detailed explanations and reductions of their ambient calculus-based definitions, but we refer to our former works [30] and [28] for more details.

1. **Renaming** This capability is applied to rename an ambient comprising this capability. Such a capability was already given in [38], but our definition differs from Cardelli's definition. In the original definition, the ambient m was not enclosed into another, name-restricted ambient (it is called s in our definition), so after it has left ambient n, n may enter into another ambient called m (if more than one m exist as sibling of n).
2. **Seeing** This operation was defined in [38], and it is used to detect the presence of a given ambient.
3. **Wrapping** Its aim is to pack an ambient comprising this capability into another ambient.
4. **Allowing Code** This capability is just a basic OPEN capability action. It is applied if an ambient allows/accepts an ambient construct (which may be a bunch of foreign codes) contained by the body of one of its sub-ambients (which may be sent from a foreign location). The name of the sub-ambient can be applied for identifying its content, since its name may be known only by some trusted parties.
5. **Draw in (an Ambient)** The aim of this capability is to draw in a particular ambient (identified by its name) into another ambient (which contains this capability) and then to dissolve this captured ambient in order to access its content. For achieving this, a mechanism (contained by the ambient *key*) is applied which can be regarded as an abstraction of a kind of protocol identified by *key*. The ambient *key* enters into one of the available target ambients which should accept its content in order to be led into the initiator ambient.
6. **Draw in then Release a Lock** This capability is very similar to the previous one, but after m has been captured by n (and before m is dissolved), n is wrapped by

another ambient. The new outer ambient is usually employed as release for a lock.[5]

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

5 The Specification of the Cloud Service Architecture Based on Ambient ASM

As was explained at the end of Sect. 4.3.1, the model of our cloud architecture is given as part of the value of the dynamic-derived function called *curAmbProc*, which serves as input for the ASM machine EXTENDEDMOBILEAMBIENTMANAGER [31].

$curAmbProc := root[\ Cloud \mid Client_1 \mid \ldots \mid Client_n\ ]$[6]

In this section, we focus on the system configuration scenario, where the model of our new software solutions is integrated with a cloud model (see Fig. 2a); however, due to the applied ambient concept, the relocation of the new infrastructure services into a middleware component within the model (see Fig. 2b) cannot be a problem.

In the formal model discussed in this section, we assume that there are some standardized public ambient names, which are known by all contributors. We distinguish the following kinds of public names: addresses (e.g., $cloud$, $client_1$, …, $client_n$), message types (e.g., $reg(istration)$, $request$, $subs(cription)$, $returnValue$, etc.), and parts of some common protocols (e.g., $lock$, msg, $intf$, $access$, out, o_1, …, o_s, op). All other ambient names are nonpublic in the model which follows.

In this section, we leave the end devices on the client-side abstract, but we define some user actions with which the cloud service architecture is able to interact.

[5]In ambient calculus, the capability OPEN $n.P$ is usually used to encode locks [38]. Such a lock can be released with an ambient like $n[\ Q\ ]$ whose name corresponds with the target ambient of the OPEN capability.

[6]The ambient called $root$ is a special ambient which is required for the ASM definition of ambient calculus; see [26] and [31].

5.1 User Actions

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

$MsgFrame \equiv msg[\ \textsc{In}\ cloud.\textsc{Allow}\ intf.content\ ]$
where *content* can be
$RegMsg \equiv reg[\ \textsc{Allow}\ CID.\langle UID_x \rangle\ ]$

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

$RequestMsg \equiv request[\ \textsc{In}\ UID_x\ |$
$\quad o_i[\ \textsc{Allow}\ op.\langle client_k, args_i \rangle\ ]\ |$
$\quad \vdots$
$\quad o_j[\ \textsc{Allow}\ op.\langle client_k, args_j \rangle\ ]\ ]$

$AddClMsg \equiv addCl[\ \textsc{In}\ UID_x\ |\ \textsc{Allow}\ CID.\langle client_k, path_l, UID_{(\text{on } client_l)} \rangle\ ]$

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

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

$ShareInfoMsg \equiv share[\ \textsc{In}\ CHID_i\ |\ \textsc{Allow}\ CID.\langle sndr, rcvr, info \rangle\ ]$

$ShareSvcMsg \equiv share[\ \textsc{In}\ CHID_i\ |\ \textsc{Allow}\ CID.\langle sndr, rcvr, info, o_i, argsP, argsF \rangle\ ]$

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

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

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

The rest of the message types is used by the CTCI functions; see Sect. 5.3. With *AddClMsg*, a user can register a new possible target (client) device or location for the outcomes of the requests initiated by him or her. Such a message should contain the chosen identifier $client_k$ of the new device, the address $path_l$ of the device, and the user identifier $UID_{(\text{on } client_l)}$ used on the given target device.

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

5.2 The Formal Model of the Cloud Architecture

The basic structure of the defined cloud model, which is based on the simplified *infrastructure as a service (IaaS)* specification given in [31], is the following:

$$
\begin{array}{l}
cloud \equiv (\nu\, fw,\, q,\, rescr_1, \ldots rescr_m) cloud[\\
\quad interface \mid \\
\quad fw\,[\; rescr_1[\; service_1 \;] \mid \ldots \mid rescr_l[\; service_1 \;] \mid rescr_{l+1}[\; service_2 \;] \mid \ldots \mid rescr_m[\; service_n \;] \mid \\
\qquad q[\; !\textsc{Open}\; msg \mid BasicCloudfunctions \mid CTCI functions \mid \\
\qquad\quad UID_x[\; userIntf \;] \mid \ldots \mid UID_y[\; userIntf \;] \mid \\
\qquad\quad UID_v^{owner}[\; ownerIntf \;] \mid \ldots \mid UID_w^{owner}[\; ownerIntf \;] \\
]\,]\,] \\
\textbf{where} \\
\quad interface \equiv \textsc{Server}^{n}_{intf}\; msg.\textsc{In}\; fw.\textsc{In}\; q.n\; \textsc{be}\; msg
\end{array}
$$

In the cloud definition above, the names of the ambients *fw*, *q*, and $rescr_1, \ldots rescr_m$ are bound by name restriction. The consequence of this is that the names of these ambients are known only within the cloud service system, and therefore the contents of their body are completely hidden and not accessible at all from outside of the cloud. So each of them can be regarded as an abstraction of a firewall protection.

The ambient expression represented by *interface* "pulls in" into the area protected by the ambients *fw* and *q* any ambient construct which is encompassed by the message frame *msg*. The purpose of the restricted ambients *fw* and *q* is to prevent any malicious content which may cut loose in the body of *q* after a message frame (*msg*) has been broken (by OPEN *msg*) to leave the cloud together with some sensitive information. For more details, we refer to [30].

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

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

It is beyond the scope of this chapter to describe all parts of this model in detail (e.g., the structure of service instances $service_i$, the functions of a service owner area *ownerIntf*, and the ASM agents in *BasicCloudfunctions*). For the specification of these components, we refer to [30].

5.2.1 User Access Layers

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

$$
\begin{aligned}
&\mathit{userIntf} \equiv \\
&\quad !\mathrm{ALLOW}\ \mathit{request} \mid !\mathrm{ALLOW}\ \mathit{newPlot} \mid \mathit{clientRegServer} \mid !\mathrm{ALLOW}\ \mathit{returnValue} \mid \\
&\quad \mathrm{PLOT}_{SID_i} \mid \ldots \mid \mathrm{PLOT}_{SID_j} \mid \\
&\quad \mathit{sortingOutput} \mid \mathit{client}_1[\ \mathit{posting}_{\mathit{client}_1}\ ] \mid \ldots \mid \mathit{client}_k[\ \mathit{posting}_{\mathit{client}_k}\ ] \\
&\textbf{where} \\
&\quad \mathit{sortingOutput} \equiv\ !(o, \mathit{client}, a).\mathit{output}[\ \mathrm{IN}\ \mathit{client}.\mathrm{ALLOW}\ \mathit{CID} \mid \langle o, \mathit{client}, a\rangle\ ]\ ] \\
&\quad \mathit{clientRegServer} \equiv \mathrm{SERVER}^n_{CID}\ \mathit{addCl}.(\mathit{client}, \mathit{path}, \mathit{UID}).(n\ \mathrm{BE}\ \mathit{client} \mid \mathit{posting}_{\mathit{client}}\ ) \\
&\quad \mathit{posting}_{\mathit{client}_i} \equiv \mathrm{SERVER}^n_{CID}\ \mathit{output}.(o, \mathit{client}, a). \\
&\qquad \mathrm{OUT}\ \mathit{client}_i.\mathit{forwardTo}_{\mathit{client}_i}.\mathit{returnValue}[\ \mathrm{IN}\ \mathit{UID}_{(\text{on } \mathit{client}_i)} \mid \langle o, \mathit{client}, a\rangle\ ]\ ] \\
&\quad \mathit{forwardTo}_{\mathit{client}_i} \equiv n\ \mathrm{BE}\ \mathit{outgoingMsg}.\mathrm{OUT}\ \mathit{UID}_x.\mathit{leavingCloud}.\mathit{path}_i \\
&\quad \mathit{leavingCloud} \equiv \mathrm{OUT}\ q.\mathrm{OUT}\ \mathit{fw}.\mathrm{OUT}\ \mathit{cloud}.\mathit{outgoingMsg}\ \mathrm{BE}\ \mathit{msg}
\end{aligned}
$$

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

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

sortingOutput distributes every service operation output among the communication endpoints in an ambient called *output*. The mechanism $posting_{client_i}$, which resides in each communication endpoint, is responsible to wrap each output of service operations which reaches the corresponding endpoint again into an ambient *returnValue* and to forward it to the specified user $UID_{(\text{on } client_i)}$ on the corresponding device $client_i$. It is beyond the scope of this chapter to present a reduction on how a simple request is processed in our model, but we refer to [30] for more details.

5.2.2 Service Plots

According to [96], a plot is a high-level specification of an action scheme, i.e., it captures possible combination of actions (e.g., service operations) in order to perform a certain task. As it has been mentioned in Sect. 3.1, in our model, service plots are provided by the service owners who have disposal of access rights of the services and they are placed into the user areas during the service subscriptions. The purpose of their usage is to determine the permitted combination of service operations which can be used by the user with his/her valid subscriptions.

For an algebraic formalization of plots, *Kleene algebras with tests (KATs)* [70] have been applied in [94, 95]. Then a plot is an algebraic expression that is composed out of elements of a carrier-set K containing two elementary operations 0, 1 and elementary processes (or propositional atoms), of binary operators $\cdot$ and +, and of unary operators * and $^-$, the last one being only applicable to propositions.

For our purposes, we employ a simplified definition of service plots where we leave the propositional ground of KATs. This means that in a service plot, the elementary processes (e.g., $o_i,\ o_j \in K$) correspond to *slots* for operation requests such that each slot can accept and permit a request only for a particular service operation, $\cdot$ denotes a sequence (e.g., $o_i \cdot o_j$ or $o_i o_j$), + denotes a choice (e.g., $o_i + o_j$), and * denotes iteration (e.g., o_i^*). In addition, we can simulate parallelism by equating $o_i \mid o_j$ with $o_i o_j + o_j o_i$; so in fact, we are interested in interleaved, concurrent service operations when we talk of *parallel composition* of plots.

It is beyond the scope of the chapter to discuss service plots in detail; for more information, see [30].

5.3 *Client-to-Client Interaction Feature*

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

$$
\begin{array}{l}
\textit{CTCIfunctions} \equiv \\
\quad CHID_1[\ \textit{channelIntf}\ ] \mid \ldots \mid CHID_l[\ \textit{channelIntf}\ ] \mid \\
\quad \textsc{Server}^n_{CID}\ \textit{addCh}.(\textit{UID}, \textit{cname}).\textsc{ChMgr}(n, \textit{UID}_x, \textit{cname}) \mid \\
\quad \textsc{Server}^n_{CID}\ \textit{subsToCh}.(\textit{UID},\textit{cname},\textit{uname},\textit{client},\textit{pymt}). \\
\qquad \textsc{ChSubsMgr}(n, \textit{UID}, \textit{cname}, \textit{uname}, \textit{client}, \textit{pymt}) \\
\textbf{where} \\
\quad \textit{channelIntf} \equiv \textsc{Server}^n_{CID}\ \textit{share}. \\
\qquad ((\textit{sndr}, \textit{rcvr}, \textit{info}).\langle \text{sndr,rcvr},\textit{info}, \textbf{undef}, \textbf{undef}, \textbf{undef}\rangle \mid \\
\qquad (\textit{sndr}, \textit{rcvr}, \textit{info}, o, \textit{argsP}, \textit{argsF}).\textsc{SharingMgr}(n, \textit{sndr}, \textit{rcvr}, \textit{info}, o, \textit{argsP}, \textit{argsF}))
\end{array}
$$

The specification above contains three ASM agents called CHMGR, CHMGR, and SHARINGMGR, whose complete ASM definitions are given in [29], but it is beyond the scope of this chapter to describe them here.

Every cloud user can create and own some channels by sending the message *AddChMsg*[7] to the cloud, where an instance of the ASM agent CHMGR, which is equipped with a server mechanism, processes such a request and creates a new ambient with unique names for the requested channel; see Sect. 5.3.1.

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

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

5.3.1 Establishing a New Channel

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

First, the agent checks whether the given *UID* has already been registered on the cloud and whether the given name *cname* has not been used as a name of an existing channel yet. If it is the case, the agent generates a new and unique identifier denoted by *CHID* for the new channel; and it inserts into an abstract database a new entry with all the details of the new channel which are the channel identifier, the channel name, and the identifier of the owner.

Then it creates an ambient called *CHID* with the terms denoted by *channelIntf* in its body which encode the functions of the new channel. By the abstract tree

[7]User actions *AddChMsg*, *SubsToChMsg*, *ShareInfoMsg*, and *ShareSvcMsg* are defined in Sect. 5.

manipulation operation called NEWAMBIENTCONSTRUCT[8] introduced in [31], this generated ambient construct is placed into the ambient tree hierarchy as sibling of the agent.

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

5.3.2 Subscribing to a Channel

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

First, the agent checks whether the given *UID* and *cname* have already been registered on the cloud and whether the given *uname* has not been used as a name of a member of the channel yet. If it is the case, the agent informs the owner of the channel about the new subscription, who responses with a set of access rights to the channel that he or she composed based on the information given in the subscription.

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

5.3.3 Sharing Information via a Channel

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

[8]This is the only way how an ASM agent can make changes in the ambient tree hierarchy contained by dynamic-derived function *curAmbProc* [31].

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

The agent first generates a new and unique operation identifier for the service operation *o* (if *o* is not equal to **undef**). This new identifier which is stored in the nullary location function *shOp* will be announced to the channel member(s) specified in *rcvr*. In the next step, the agent checks whether the *sndr* is a registered member of the channel. Then if the given value of *rcvr* is equal to “*,” the agent broadcasts the corresponding message(s) to all members of the channel. Otherwise, if the value of *rcvr* corresponds to the name of a particular member of the channel, the agent sends the corresponding message(s) only to him or her.

In the case of the processing of *ShareInfoMsg*, the agent sends to the member(s) specified in *rcvr* the message $sharedM_{content_1}$ (for its definition, see Table 3), which contains the sender *sndr* and the shared information *info*.

In the case of the processing of *ShareSvcMsg*, two ambient constructs are created by NEWAMBIENTCONSTRUCT. The first one is the message $sharedM_{content_2}$ (for its definition, see Table 3) and it is sent to the member(s) specified in *rcvr*. It contains the sender *sndr*, the new operation identifier *shOp*, the list of public arguments *argsP*, and the informal description of the shared operation denoted by *info*.

The second ambient construct is the plot $\textsc{Plot}_{shOp}$ (for its definition, see Table 3) enclosed by the ambient *newPlot* and equipped with some additional ambient actions (see the underlined capabilities in the definition of *sharedPlot* in Table 3) which move the entire construct into the user area of the channel member(s) specified in *rcvr*, where the plot will be accepted by the term !ALLOW *newPlot*.

Table 3 Definitions of ambient constructs used by the ASM agent SHARINGMGR

$sharedM_{content_i}$	$\equiv returnValue[\ \underline{\textsc{Out}\ n.\textsc{Out}\ myChId.\textsc{In}\ UID}.\langle cname, client, content_i\rangle\ ]$
$content_1$	$\equiv$ {“sender:” *sndr*, “content:” *info*}
$content_2$	$\equiv$ {“sender:” *sndr*, “operation:” *shOp*, “arguments:” *argsP*, “description:” *info*}
$sharedPlot$	$\equiv newPlot[\ \underline{\textsc{Out}\ n.\textsc{Out}\ myChId.\textsc{In}\ UID}\ \mid \textsc{Plot}_{shOp}\ ]$
$\textsc{Plot}_{shOp}$	$\equiv \textsc{Server}^{s}_{op}\ shOp.trigger_o$
$trigger_o$	$\equiv (\nu\ tmp)$ $(client, argsP).(\textsc{Out}\ UID.\textsc{In}\ UID_{sndr}.s\ \textsc{Be}\ request\ \mid$ $o[\ \textsc{Allow}\ op.\langle tmp, (argsP \setminus argsF) + argsF\rangle\ ]\ \mid$ $tmp[\ \textsc{Allow}\ output\ \mid CID[\ (o, c, a).\textsc{Out}\ UID_{sndr}.$ $\textsc{In}\ UID.tmp\ \textsc{Be}\ returnValue.\langle shOp, client, a\rangle\ ]\ ]\ )$

The execution of the shared service operation *shOp* can be requested in a usual *RequestMsg* as normal service operations. The $\textsc{Plot}_{shOp}$ is a plot, which can accept service operation requests for *shOp* several times. It is a special plot, because instead of triggering the execution of *shOp* as in the case of a normal operation, a normal plot does (see [30]); it converts the original request to another request for operation *o* by applying the term $trigger_o$. This means that it substitutes the operation identifier *o* for *shOp*, it completes its argument list with *argsF*, and it forwards the request for *o* to the user area of the user *sndr* who actually has right to trigger the execution of the operation *o*.

To the new request, the name-restricted ambient *tmp* is attached (see its definition within the definition of $trigger_o$ in Table 3), whose purpose is similar to the communication endpoints of registered clients. Namely, it is placed into the user area of *sndr* temporarily and it is responsible to forward the outcome of this particular request from the user area of *sndr* to the user area of the user who initiated the request. It is beyond the scope of this chapter to present a reduction on how a particular request for a shared operation is processed in our model, but we refer to [29] for more details.

6 The Specification of the Identity Management Machine

The identity management machine (IdMM), first presented in the paper [115], is a privacy-enhanced client-centric identity meta-system based on the concept of abstract state machines. The IdMM provides a “proxy” between a client and cloud-based identity management solutions “translating” the protocols used by the client to manage identities to a set of protocols used by cloud providers. The IdMM can then authenticate a user to a given cloud service as well as manage any private identity-related data stored on the cloud.

As mentioned in Sect. 3.2, the IdMM is composed of six agents. Figure 5 details the overall architecture of the system. The six agents, first introduced in the paper [116], define the various interactions needed for the authentication, de-authentication, and attribute synchronization for cloud services, based on the abstract functions described in Fig. 6. In this section, we will give an overview for of the agents: the core, client, cloud, and user agents. We will then detail a proof-of-concept implementation previously presented in [119]. Since the specification of the agents is still an ongoing task, we will end this section with a description of future work on the IdMM.

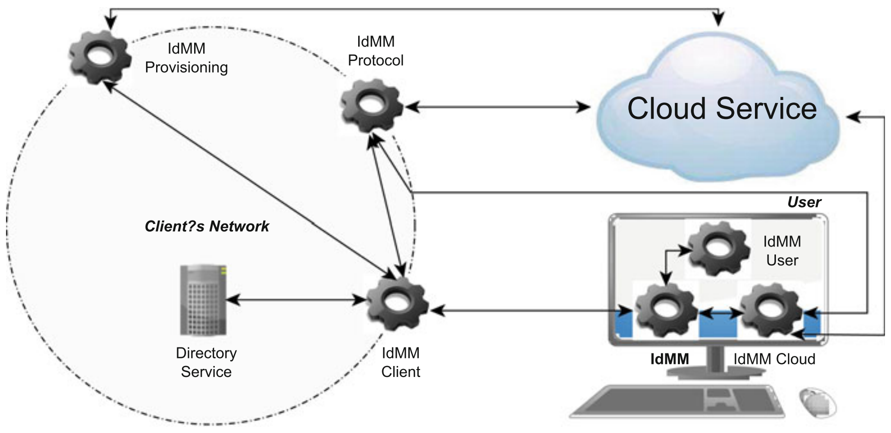
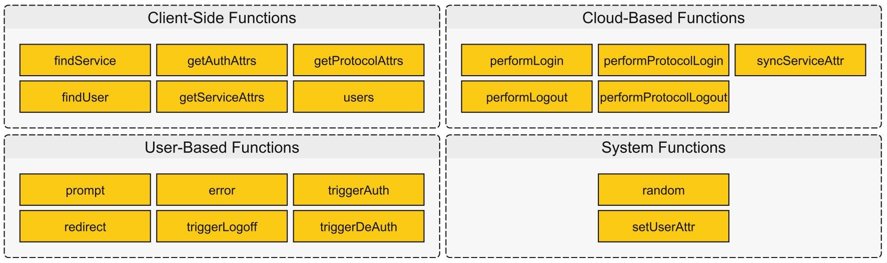


Fig. 5 IdMM architecture



Client-Side Functions

- findService
- getAuthAttrs
- getProtocolAttrs
- findUser
- getServiceAttrs
- users

Cloud-Based Functions

- performLogin
- performProtocolLogin
- syncServiceAttr
- performLogout
- performProtocolLogout

User-Based Functions

- prompt
- error
- triggerAuth
- redirect
- triggerLogoff
- triggerDeAuth

System Functions

- random
- setUserAttr

Fig. 6 Abstract functions

6.1 The IdMM Core Agent

The core agent for our client-centric solution is described in the paper [115]. The agent has nine states (Fig. 7). The initial state of the IdMM is *UserLogin*. While in this state, the user is prompted to input his or her credentials and is subsequently authenticated to the machine. If a user cannot be authenticated, the machine halts with the appropriate error. When a user wants to log out, the event triggered will set the state to *UserLogout*. This will log the user out of the machine, set the state to *UserLogoutService* which logs the user out of every single service he or she was connected to, and halt the execution of the machine. The termination of the machine occurs in the *halt* state. If an error exists, the appropriate message will be displayed to the user.

When the user wants to use a specific service, he or she will specify the service's uniform resource identifier (URI). The event triggered by this action will set the state to *ServiceLogin*. In the *ServiceLogin* state, the machine attempts to find the matching service given the URI. If no services can be found, an error message is triggered. If a service is found, the state will be set to *AuthorizeLogin*. In case the

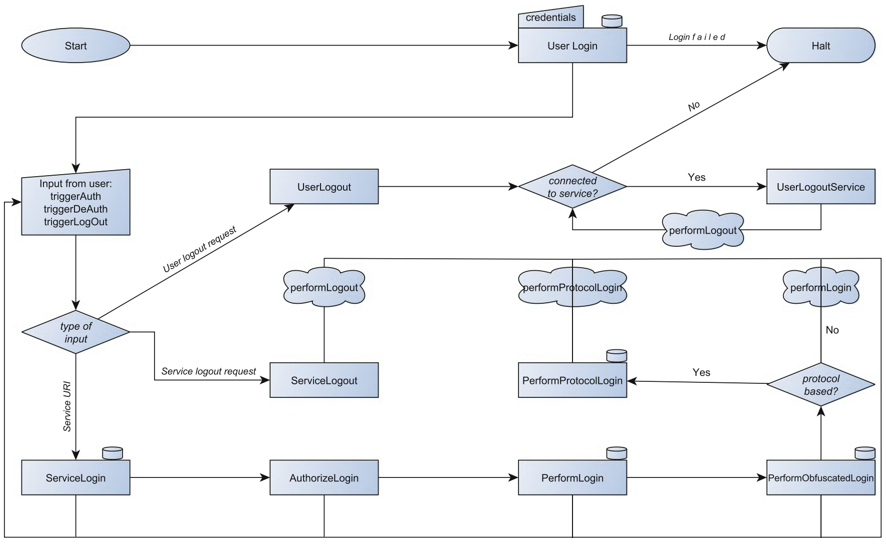


Fig. 7 IdMM flow

user is already connected to the service, he or she will be redirected to the given URI. The authorization is done by the *AuthorizeLogin* state. If the user has the appropriate access rights, the state is set to *PerformLogin* where the machine checks the type of identity required and sets the state to *PerformObfuscatedLogin*. If the service supports protocol-based authentication, then *PerformObfuscatedLogin* will switch to *PerformProtocolLogin*, which will perform the authentication based on a given protocol. In case the service does not support protocol-based authentication, the machine will perform the authentication with the given identity. The machine switches to the *ServiceLogout* state when a log-out event is triggered. In this state, the IdMM searches for the matching service and performs the log-out.

To describe the rules for each state, we make use of abstract functions (Fig. 6). The client-side functions, cloud-based functions, and user-based functions will be further refined by describing the IdMM$_{Client}$, IdMM$_{Cloud}$, IdMM$_{Protocol}$, and IdMM$_{User}$ agents. To ease this process, we keep the underlying communication layers abstract.[9] The system functions include two important functions: *random* and *setUserAttr*. In order to handle an obfuscated identity, we use the function *users* to select a list of the obfuscated identities contained in the client's directory. We would then choose a random identity from this list and use it to authenticate the required cloud service. At present, we propose that the function *random* be restricted to only this requirement. More research can be conducted in this topic to further enhance

[9]In the paper [119], for example, we have included the IdMM$_{Client}$ agent as part of a Tomcat server. The communication with the core agent is done via the Google Web Toolkit RPC framework.

privacy. For instance, we might choose not to include identities that have been previously used with the given service. If the service requires a partially obfuscated identity, we use the function *setUserAttr* to replace some of the obfuscated values in the identity with real values that correspond to the current user.

6.2 The IdMM Client Agent

The IdMM$_{Client}$ agent is responsible with the interaction with the client's directory service. It retrieves any relevant information from the directory service and converts it to the IdMM's data structures. To achieve this, we have further refined the client-side functions presented in Fig. 6. Apart for the *getAuthAttrs* function, the refinement involves introducing a new submachine for each client-side function.[10]

The refinement of the client-side functions is described in the paper [117]. To allow a seamless adoption of the IdMM, we make an abstraction with respect to the client's directory. The refinement of the client-side functions yielded an interface (*client interaction interface*) of 20 directory-dependent functions (Fig. 8). In addition to the interface, the refinement of the client-side functions also yielded a list of parameters for each of the submachines. These parameters include information about the client's directory (location, authentication mechanism, credentials) as well as information about the structure of the client's directory. In the paper [117], we illustrated how to implement a client interaction interface for an ApacheDS server.[11]

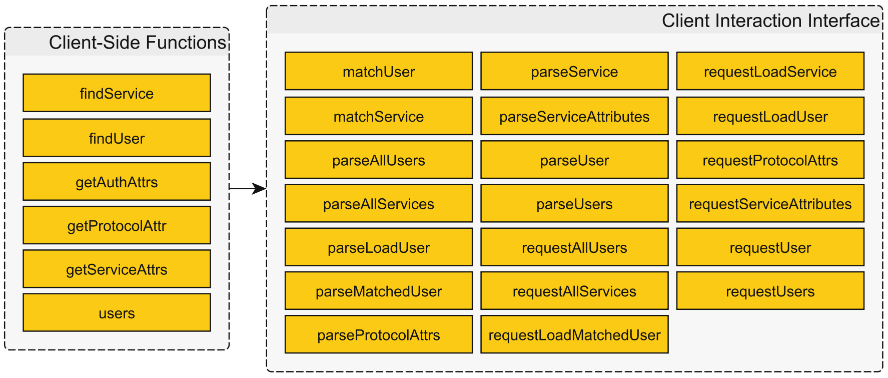


Fig. 8 Client-side function refinement

[10]For the purpose of this paper, we will not detail each submachine. The refinement and description of the client side functions are described in the paper [117].

[11]ApacheDS [11] is a Java-based embeddable directory server certified by the Open Group as LDAPv3 (Lightweight Directory Access Protocol) [101] compatible.

We used the Novell LDAP Classes for Java API [84] to facilitate the interaction with the ApacheDS server.

6.3 The IdMM Cloud Agent

The IdMM$_{Cloud}$ agent is responsible for the interaction with any cloud service. Its purpose is to authenticate or de-authenticate a user to/from a given cloud service and, when needed, to perform attribute synchronization (see Sect. 3.2.1). To achieve this goal, the cloud-based abstract functions presented in Fig. 6 must be further refined. At present, we have the specifications for only the direct and obfuscated client-to-cloud interaction scenarios presented in Sect. 3.2. Our current work entails the understating and specification of the protocol interaction as detailed in Sect. 10. As such, the refinement of the functions *performLogin*, *performLogout*, and *syncServiceAttr* is achieved by introducing the IdMM$_{Cloud}$ submachine as presented in the paper [120].

Since the direct interaction scenario takes into consideration the fact that different cloud providers will have different authentication systems, we have introduced the concept of a cloud plug-in (Fig. 9). This interface represents a set of functions that will be used for authentication, de-authentication, and attribute synchronization on various cloud services. For each cloud service used, an implementation of this interface must be provided. We consider two distinct ways for authentication, de-authentication, and attribute synchronization. In case the service provider already offers an API for this purpose, we simply use the functions *makeAPIAuthentication*, *makeAPIDeAuthentication*, and *makeAPISync*. If the service provider does not have such an implementation, we use a custom-created one marked by the *generic* functions. For example, we use the functions *requestAuthParameters* and *parseAuthParameters* to obtain all parameters used for the authentication process. We then use the functions *requestGenericAuthentication* and *parseGenericAuthentication* to make the actual authentication process.

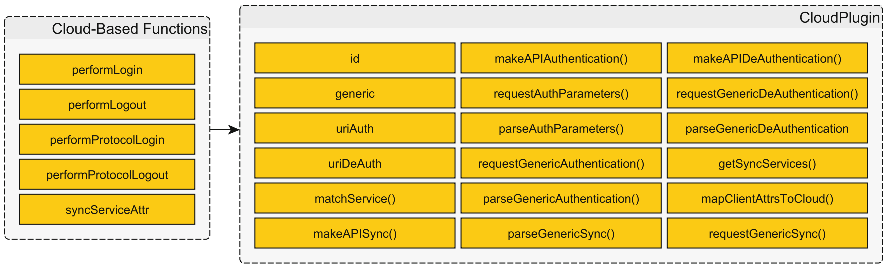


Fig. 9 Cloud plug-in

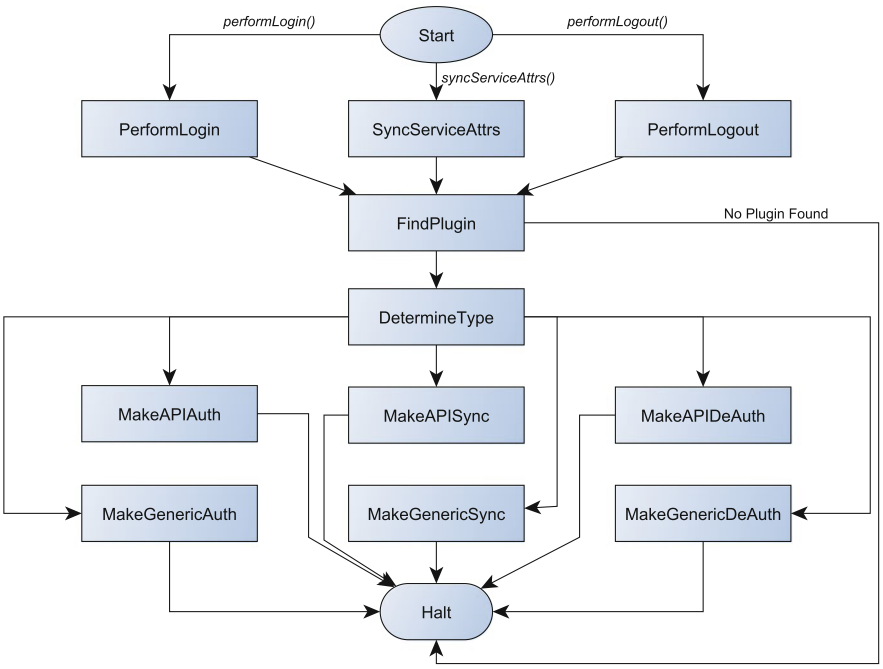


Fig. 10 IdMM$_{Cloud}$ flow

The rules for the IdMM$_{Cloud}$ submachine are presented in the paper [120]. The initial state of the machine depends on the function being called. For the function *performLogin*, the initial state will be *PerformLogin* (Fig. 10). Similarly, the initial states will be *PerformLogout* and *SyncServiceAttrs* for the functions *performLogout* and *syncServiceAttrs*. While in these states, the appropriate values of the machine's dynamic frame are set and the state is set to *FindPlugin*. In this state, the machine attempts to find a plug-in that matches the provided service. If no plug-in is found, the machine halts. When a plug-in is found, the state of the machine is set to *DetermineType*. The rules for this state determine which of the *MakeAPI-Auth*, *MakeAPIDeAuth*, *MakeAPISync*, *MakeGenericSync*, *MakeGenericAuth*, or *MakeGenericDeAuth* states will be chosen based on the type of the request and whether the service offers an API. While in the aforementioned states, the actual authentication, de-authentication, or attribute synchronization processes take place via the use of the *CloudPlugin* functions presented in Fig. 9.

6.4 The IdMM User Agent

The IdMM$_{User}$ agent is responsible for the interaction with the user. It handles inputs from the user and displays the appropriate messages. This goal is achieved by implementing the user-based functions presented in Fig. 6. Since the implementation of these functions is software dependent, we opted to keep the functions abstract.

The function *prompt* is used to prompt the user into typing his or her credentials and returns a list of attributes representing the credentials. The function *error* is used to notify the user of an error that occurred during the authentication process. The function *redirect* is used to redirect the user to a specific URI, specified as a parameter. The event triggered by the user entering a URI will call the function *triggerAuth* which changes the state of the IdMM core agent to *ServiceLogin*. When the user wants to de-authenticate from a service, the underlying event will call the function *triggerDeAuth*, causing the state of the core agent to change to *ServiceLogout*. The de-authentication from the IdMM is done via the *triggerLogoff* function, which sets the state to *UserLogout*. During the implementation phase described in the paper [119], we became aware that some Web-based applications do require the user to enter dynamically generated data for the sole purpose of disallowing automated requests. Most often, this is achieved via captcha messages. As such, we were forced to introduce a new user-based function, *String captcha*(*Object message*), to allow the user to input captcha messages.

6.5 A Proof-of-Concept Implementation

In the paper [119], we described a proof-of-concept implementation for the IdMM focusing primarily on software-as-a-service (SaaS) platforms. The implementation follows the architecture described in Fig. 5. We used an ApacheDS [11] directory service running in a cloud-based environment on a private virtual machine. The implementation for the IdMM$_{Client}$ agent is running as part of an Apache Tomcat [12] server. The communication between the IdMM$_{Client}$ agent and the ApacheDS service is done via the use of the Novell LDAP Classes for Java [84]. The client interaction parameters described in Sect. 6.2 are stored in a *.properties* file (as opposed to an XML file as described in the paper [117]). The Tomcat server hosting the IdMM$_{Client}$ agent is running on a public virtual machine in the same cloud-based environment as the ApacheDS server.

The IdMM$_{Core}$, IdMM$_{Cloud}$, and IdMM$_{User}$ agents are running as part of a Google Chrome browser extension. Since the specification of Google Chrome extensions [58] entails the use of JavaScript, as opposed to the usage of Java for the IdMM$_{Client}$, we opted to use the Google Web Toolkit framework [59]. By using this framework, we can restrict the programming language of the implementation to only Java, leaving the compilation of the code to JavaScript down to the GWT framework. The communication between the IdMM$_{Core}$ agent and the IdMM$_{Client}$ agent (for the execution of the client-side function presented in Fig. 6) is achieved by using GWT's own RPC framework. The credentials returned by the user-based function *prompt()* as well as the address of the IdMM$_{Client}$ are stored in encrypted format in the browsed local storage environment. The user may modify the values via the use of the extension's option page. All communication between the client's directory and the IdMM$_{Client}$ agent, IdMM$_{Core}$, and IdMM$_{Cloud}$ agents is encrypted. Any privacy-related data stored in the dynamic frame is encrypted as well.

A demo of the implementation can be found in [118].

7 Cloud Content Adaptivity Specification

The adaptivity solution, first presented in [43], uses a WA wrapped inside a middleware system to adjust cloud services to client's device. This research includes the creation of ASM ground models in a way to reflect the requirements and to serve as a basis for implementation. The next subsection introduces the system architecture of the proposed solution.

7.1 The System Architecture

The system architecture illustrated in Fig. 11 shows how the cloud, the middleware, and the client (here represented by his or her devices) are interacting. The client's point of connection is a WA which should adapt itself, on the fly, to different devices (smartphones, tablets, laptops, and desktop computers) and different browsers. The interaction between the client and the cloud is done through a middleware software. The WA represents the front-end and the middleware the back end. After a successful log-in, the application will display a list of cloud services corresponding to the user's credentials. By selecting a service, a request is sent to the middleware, which will forward it to the cloud and wait for the corresponding answer. Meanwhile, a device profile will be created, using the third-party tools and frameworks.

For detecting the device properties, the *Modernizr* framework is used by creating JavaScript tests. These tests will be executed on the client side and the corresponding cookie variable will be updated, sending the information to the server side, where the session and the local database will also be updated. When the features cannot be detected using *Modernizr*, then a device detection database tool can be used (e.g., Wireless Universal Resource FiLe (WURFL)). If the user uses the cloud services from the same device several times, then the device information can be read from the local database, without accessing the device detection database or writing again the JavaScript tests. Even when the information is locally available, the session is still used, because of optimization purposes. Another reason for saving the information locally is also the client-client interaction. One client could choose to send the output

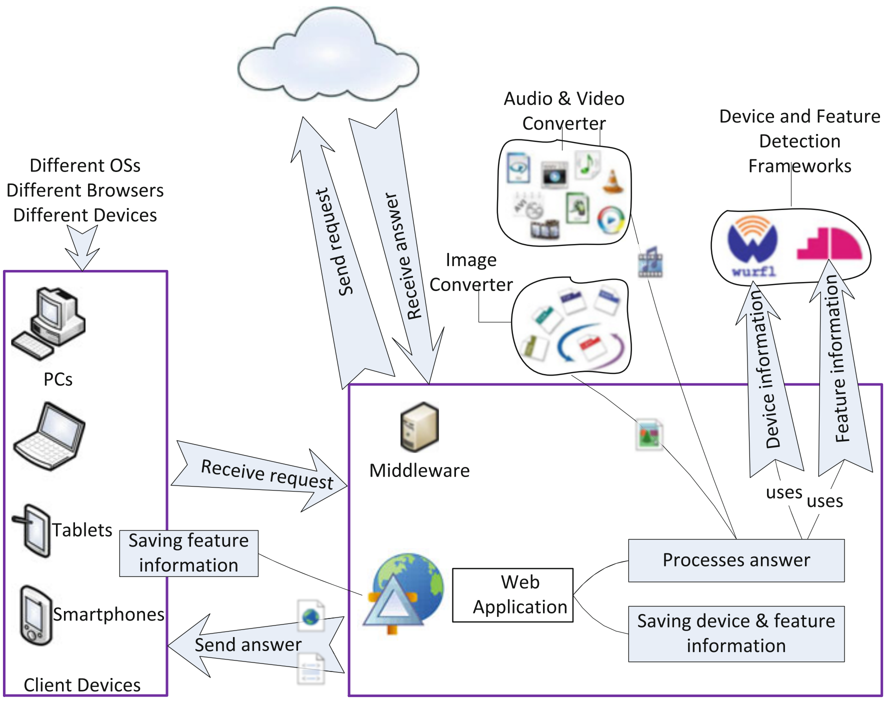


Fig. 11 System architecture

of a service from cloud to another client, which is logged in from another device, which means that sending the answer from the cloud not only to the device for which we have the profile in the session but also to another device for which we can read the information from the database.

The answer that arrives from the cloud is parsed and processed using the device information. When images or videos are involved, their format is checked, and if it is not accepted by the device, then third-party tools are used to transform the media to the corresponding format. When the adaptation is finished, the message is sent back to the device. In case some JavaScript tests exist, they will be executed on the client side for getting the new information; afterward, the cookie is updated. When the loading finishes, the message is displayed on the device.

The middleware acts like the virtual provider described in [9]. In our case, we leave out for now the sending and receiving part and concentrate only on the adaptation.

7.2 ASM Ground Models

In order to reflect the WA intended behavior, the ASM method was used for building the ground models. The main components of the system were modeled such that the refinement of their abstract models would be a description of the future implementation.

Further on, the models including only aspects with respect to content adaptation and displaying are presented. Most of the work is done on the server side using the information discovered on the client side.

7.2.1 Display Output Agent

Figure 12 illustrates the ASM ground model that describes the client's device activity. There are three states through which the agent goes, *Waiting for message* (the initial state), *Execute client tests*, and *Displaying the message* (the final state).

The agent waits in the initial state until a message comes from the server. The messages sent by the middleware to the client are saved in a queue. When a message becomes available in the queue, the agent executes the macro *Decrypt message*. After the decryption of the message, the flow goes further with the condition *Client tests available*. If JavaScript tests (which verify the device properties) are available, then the agent goes to the state *Execute client tests*. This is a durative action (there is an interval of time between starting and ending the execution) which is executed internally by the browser. When the JavaScript execution finishes, the agent retrieves the new device information and updates the cookie, in this way communicating to the server the new values of the device properties. The flow takes the same route, as when no client tests would exist. The guard *Extra resources* verifies if extra

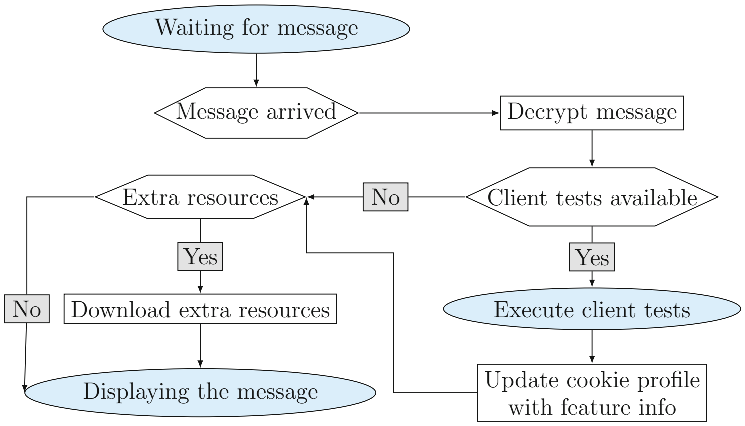


Fig. 12 Display output ASM

images and/or videos are necessary. In a positive case, the system downloads them (the abstract macro *Download extra resources* does this action). The agent's state changes, reaching the final one, *Displaying the message*.

7.2.2 Receive Request Agent

Figure 13 displays the ASM ground model for receiving the client's request. The algorithmic idea consists in having only the initial and the final states: *Waiting for requests* and *Waiting for answers from Cloud*. Again, a queue is used to store the requests sent by the client. When a client's request arrives, it is then forwarded to the cloud (this is fulfilled by the *Send requests to Cloud* macro), and in parallel, the agent verifies if the session or the cookie contains information regarding the device which sent the request.

If the condition *Device info available in the session/cookie* is not met, then we are dealing with the first request sent during that session. Using the guard *Device profile available on the server*, the agent verifies if this is the first time that the user logs in from the corresponding device. If this is the case, then the agent retrieves the device details from the local database by executing the *Retrieve the local device profile* macro. After retrieving the device details, another macro is executed to *Update session/cookie with device info*. The information is saved on the session and on the cookie to ease the communication between client and server agents and to optimize the process regarding the verification of device properties. Whenever a new JavaScript test is executed, the cookie and the session variables are updated using the test's result. The state is changed and the agent waits for answers from the cloud.

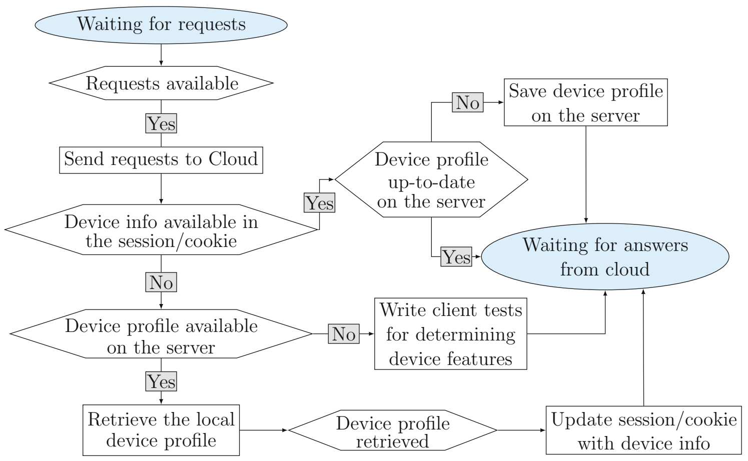

Waiting for requests
Requests available
Yes
Send requests to Cloud
Device info available in the session/cookie
No
Device profile available on the server
Yes
Retrieve the local device profile
Device profile retrieved
Update session/cookie with device info
No
Write client tests for determining device features
Yes
Device profile up-to-date on the server
No
Save device profile on the server
Yes
Waiting for answers from cloud

Fig. 13 Receive request ASM

If the guard *Device profile available on the server* is not satisfied, then the client's tests for retrieving the values of all device properties that are necessary for code adaptation are written. The macro *Write client tests for determining device features* is responsible for the previous action. In the same time, the state is changed to *Waiting for answers from Cloud*.

By writing JavaScript code together with *Modernizr*, the client's tests are created in order to determine the device and browser capabilities. The tests will be written on server side and executed in background on the client side. After the execution finishes, the answer is sent back to the server. Information regarding JavaScript interpretation using ASMs is provided by Börger et al. [27]. Using the information provided in that article, the macro *Update session/cookie profile with feature info*, which can be seen by the reader in Fig. 12, deals with the description of the client's tests. Starting from the basic client's tests, new tests could be inserted if, while parsing the answer sent by the cloud, new device properties are involved.

Another way to follow is when the information is already available in the session, which means that the request in case is not the first one coming from the client. Using the guard *Device profile up-to-date on the server*, the agent verifies if the local database already contains the device information. If no modification is necessary, the final state is reached. In the contrary case, when device properties have to be created or updated, the macro *Save device profile on the server* is executed. At this point, the agent also reaches the final state.

7.2.3 Receive and Process Answer Agent

The algorithm presented in the ground model from Fig. 14 describes what the agent does with the answer sent by the cloud in order to adapt it to the device. The agent finds itself in four different control states: *Waiting for answers from Cloud*, *Filter and adapt content*, *Message format transformed*, and *Send answer to client*. The starting control state is *Waiting for answers from Cloud* and the final state is *Send answer to client*.

The agent waits for the messages having the state set to *Waiting for answers from Cloud*. The messages sent by the cloud are saved in a queue which is checked by the guard *Message available*. Whenever a message is available, a verification is done to see if its format is supported by the browser installed on the device (this check is contained by the guard *Message format supported*). If the message is not written in Hypertext Markup Language (HTML) or Extensible Hypertext Markup Language (XHTML), then the macro *Transform the message format (html/xhtml)* is executed and the agent's state is changed to *Message format transformed*. If the message's format can be displayed in the browser, we go further on with checking if the device details are available, using the guard *Device information available*. What is this guard actually doing? It checks to see if the device profile is available in the session. If the device information is not already available, then the flow goes on with the next condition without adapting the HTML code. In this case, the page will be

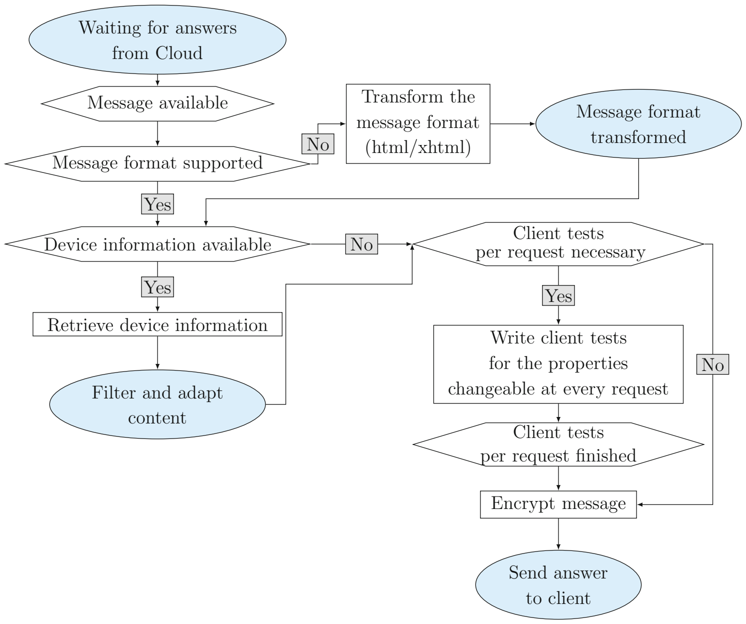

Waiting for answers from Cloud
Message available
Message format supported
No
Transform the message format (html/xhtml)
Message format transformed
Yes
Device information available
No
Client tests per request necessary
Yes
Retrieve device information
Filter and adapt content
Yes
Write client tests for the properties changeable at every request
No
Client tests per request finished
Encrypt message
Send answer to client

Fig. 14 Receive and process answer ASM

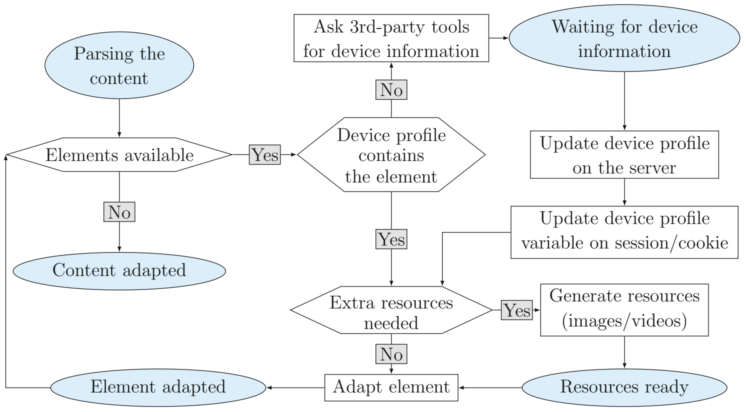

Parsing the content
Ask 3rd-party tools for device information
Waiting for device information
No
Elements available
Yes
Device profile contains the element
Update device profile on the server
No
Content adapted
Yes
Update device profile variable on session/cookie
Extra resources needed
Yes
Generate resources (images/videos)
No
Element adapted
Adapt element
Resources ready

Fig. 15 Filter and adapt content ASM

displayed on the device in its original state, containing only the adjustments done automatically by the browser.

If the device information is available in the session, then using the macro *Retrieve device information*, the corresponding information is retrieved and the agent's state is set to *Filter and adapt content*. This action is extended in the model displayed by Fig. 15. When the adaptation finishes, the agent reaches the same condition as the one reached when the device information is not available.

The guard *Client tests per request necessary* checks if an extra JavaScript code has to be inserted in order to test the device capabilities which can change by every request. An example is the Global Positioning System (GPS) location property. GPS information is something that could change each time the user makes a new request, because he or she might change his or her position (we can imagine that the user is moving, being in a train or in a car, when accessing the cloud). If there are no tests per request required, the agent has one more step to do before going to the final state, which is *Encrypt message*. When this macro is executed, the control state *Send answer to client* is reached. The macro *Write client tests for the properties changeable at every request* is executed if the corresponding tests are necessary, and the guard *Client tests per requests finished* verifies if this execution finished and if the agent is ready for encryption. The same flow as for the "No" branch follows; the macro *Encrypt message* is executed and the final state is reached.

7.2.4 Filter and Adapt Content Agent

For adapting the content coming from the cloud, a general HTML/XHTML parser has to be designed. As a basis for this, the paper [56] can be used. In [56], a multi-agent ASM formal model is described, which shows the browser behavior and how its components interact.

Figure 15 displays the algorithm of the *Filter and adapt content* model, which was mentioned as a control state in the previous ground model (Fig. 14). In this ASM ground model, the agent can reach five control states: *Parsing the content*, which is also the initial state, *Waiting for device information*, *Resources ready*, *Element adapted*, and the final state *Content adapted*.

The adaptation algorithm starts with the parsing of the HTML content of the answer which came from the cloud. Each HTML element is identified, while parsing the content and the guard *Elements available* verifies if the algorithm still has to check and adapt other HTML elements. If this is the case, then the agent goes to the next condition *Device profile contains the element*. If the device profile saved in the session does not contain information with respect to the chosen element, then a device detection database third-party tool is used. A query containing the specific element is sent to the device detection database. When the agent reaches the macro *Ask 3rd-party tools for device information*, the state changes to *Waiting for device information*. The agent waits until this durative action is executed by the device detection database. When the information is returned to the agent, two macros are executed in parallel: *Update device profile on the server* and *Update session/cookie with device information*. By updating the information in the session and in the local database, we will not have to query again the third-party tool regarding the same HTML element for the possible future requests. Nearby these two actions, the guard *Extra resources needed* is also fired.

One of the existing device detection databases can be used, without creating an own database, because this would cost a lot of time, mainly because of the database maintenance.

The agent executes the *Extra resources needed* guard for verifying if the element's (e.g., an image or a video) format is supported by the browser running on the current device. In case the format is not applicable, the macro *Generate resources (images/videos)* is executed and the control state is updated to *Resources ready*. The next step is the same as the one when no extra resources are needed.

The *Adapt element* macro does the job of changing the HTML element corresponding to the device profile. If, for example, a very complex list should be displayed on a smaller device, on which the user should scroll too much for seeing the information, then the list should be changed by eliminating the not-so-important information and make it expandable, such that the user could easier read the information and reach faster the information which is interesting for him or her. By detecting the device properties, the developer discovers which types of elements are suitable for the device and can also change the format correspondingly. Reaching the execution of this last macro, the state corresponding to each element is set to *Element adapted*. Changing the state of each element is like a flag which is set on each element. The agent goes to the final state, *Content adapted*, when all elements reached the state *Element adapted*.

7.3 The Use of ASM Ground Models

In the previous section, the WA's constraints/assumptions were expressed by the creation of ASM ground models. The fact that the requirements change based on the device that is querying the cloud services requires the development of a flexible and extendable piece of software by using successive refinement steps that adapt the abstract models to the changing requirements.

For reaching the compilable code, the next phase after the creation of the ground models is the definition of the macros by using pseudo-code-like descriptions. Typically, there are necessary more steps to reach the code, because the design phase starts with a not-so-precise model and gradually more details are introduced with each step, taking into account the requirements of the system [23]. Below is an exemplification of how the *Display Output Agent* ground model from Fig. 12 can be refined by describing the agent's signature and the ASM macros.

Below follows the agent's signature. The *ctl_state*={*Waiting for message, Execute client tests, Displaying the message*} variable represents the agent's control states and has as initial value the state ***Waiting for message***. The queue *messages*(*self*) contains the messages sent by the middleware; its initial value is the ***empty set***. The *cookie*(*deviceProfile*) $\in$ *COOKIE* variable stores the device profile, where the set *COOKIE* contains *key* $\times$ *value* elements (initial value = ***undef***). *headPage*: *messages*(*self*) $\rightarrow$ *html* is a function that returns the first HTML text sent by the middleware. Its initial value is ***undef***. *Modernizr* is a set that contains the features (and their corresponding values) tested by using the Modernizr framework. Using the agent's signature, the macros can be defined as follows.

```
ClientDisplayOutputMacros =
  if MessageArrived then
      DecryptMessage
      if ClientTestsAvailable($headPage($
        $messages(self)))$ then
          $ctl\_state := executeClientTests$
          UpdateCookie
      end if
      if ExtraResources then
          DownloadExtraResources
      end if
      $ctl\_state := displayingTheMessage$
  end if
```

Where

```
MessageArrived = $(messages(self) \neq empty)$
DecryptMessage - abstract
ClientTestsAvailable$(page)$ =
        $(\exists t \in htmlTags\ tagName(t) =$ “SCRIPT” and
            ContainsModernizrTests$(t))$
ContainsModernizrTests$(scriptTag)$ =
  if $hasAttribute(scriptTag, src)$ then
      let $url = valueOfAttribute(scriptTag, src)$ in
        $\exists c \in contentOfFile(url)\ c =$ “Modernizr.addTest”
  else
      $\exists c \in contentOfTag(scriptTag)$
          $c =$ “Modernizr.addTest”
  end if
ExtraResources = $(extraResources \neq empty)$
DownloadExtraResources - abstract
UpdateCookie =
  if $ctl\_state = executeClientTests$ then
      for all $f \in self.Modernizr$ do
          insert $f$ into $self.cookie(deviceProfile)$
      end for
  end if
```

ClientTestsAvailable macro verifies if there are Modernizr tests defined in the JavaScript part of the page. There are two possibilities to declare the client's tests, directly in the SCRIPT section of the page and through another JavaScript file. The *“Modernizr.addTest”* method (made available by the Modernizr framework) can be used either in the content part of the SCRIPT tag or inside the JavaScript file.

UpdateCookie macro parses all the features ($key \times value$ pairs) tested using the Modernizr framework and creates the agent's cookie.

Following the example given above, each of the ground models defined in Sect. 7.2 could be refined to pseudo-code and afterward translated to CoreASM.

8 The Problem of Verification

Although our work is focused only on the specification of some novel solutions concerning client-cloud interaction at present, the validation and the verification of our (ground) models have been planned as our future work. Hence, we devote the following section to this subject.

In the context of software systems, formal verification can be regarded as a decision problem [105]. Namely, if it is given a programming language L and a formal language F suitable to express properties of programs in L (e.g., completeness, soundness, compactness, liveness, and safety and halting problems), the problem of verifying L programs against F properties can be considered as follows [105]:

- Given a program $\Pi \in L$ and a property $\varphi \in F$, decide whether for every input $\mathcal{I}$ appropriate for Π, Π on input $\mathcal{I}$ satisfies φ.

The decidability of this problem depends on the expressiveness of both L and F. Although there exists some artificial subset of formal languages which are extended with some (first-order) logics and for which the above problem is always decidable (e.g., sequential nullary ASMs conjugated with a first-order branching temporal logic [105]), their computational power and expressiveness are limited.

Verification in process calculi (e.g., ambient calculus) and in state transition systems is often done by checking bisimulation equivalence [104], where it is investigated whether the associating systems behave in the same way in the sense that one system simulates the other and vice versa. A more complex process representing an implementation is shown to be bisimilar to a simpler process representing a specification. The simpler process should be so clear that it can be regarded as satisfying correctness requirements in an intuitive sense, not with rigorous mathematical proof.

This procedure may be not reliable or not applicable at all for many cases in general (intuitive checking of correctness of *formal ground model* is often nontrivial and a very error-prone method). Hence, for verifying correctness properties for algorithms specified by these formal languages, some (temporal) logics are required, such that some model checking verification can be applied. There are model checking algorithms for process calculi with calculus style specification logic [17, 49], but such a logic often is very complicated to describe or understand, and sometimes results are not obtained in reasonable amount of time even for the proofs of very simple correctness requirements.

In some other cases, the aim of the coupling of a formal language and such a special logic is to address only the verification of specific aspects or features in the specified systems. For instance, *Cloud Calculus* [67] applies ambient calculus to specify topology of cloud systems and firewall security rules such that it is able to verify whether the global security policy after the virtual machine migration is consistently preserved with respect to the initial one.

In the case of the formal model of our integrated system discussed in this chapter, which is given in terms of our two-abstraction-layer approach mentioned above, verification can be reduced to pure ASM verification, since the definitions which are given in terms of ambient calculus can be replaced with ambient ASM rules which in turn can be translated to normal ASMs.

Although ASMs can be seen merely as a specification formalism, strictly speaking, ASMs constitute a computation model on structures. The program of an ASM is like the program of a Turing machine, a description of how to modify the current configuration of a machine in order to obtain a possible successor configuration. Since ASMs are computationally complete—they can calculate all computable functions—the problem given above is, in its full generality, undecidable.

The advantage of ASMs is that they are close to logic, which makes the overall design easily amenable to well-understood mathematical techniques. Essentially, the mathematical foundation of ASMs supports the formal verification of dynamic systems designed by means of ASMs.

For the verification of the properties of ASMs, mechanical theorem proving (e.g., Isabelle [24, 89], KIV [93], PVS [51, 54], etc.) as well as model checking systems [106, 121] can be applied. For these, some logics were also developed [107], which are tailored for ASMs in terms of an atomic predicate for function updates.

After the correctness of an ASM (ground) model is proved, we can apply stepwise refinement method [22] to extend the model and to prove in each step that the new model preserves the same properties, which the predecessor model possessed. The design of complex systems is typically organized as a series of refinement steps, where the final goal is to minimize the gap in reasoning between the refined formal model and a particular implementation.

In Sects. 8.1 and 8.2 below, it is presented shortly which correctness properties for Web application are assumed and how could ASM ground models help, respectively, and which solutions are available for the client- and server-side adaptation.

8.1 Correctness of Web Application

WAs do not have a precise definition or a precise model to follow. This happens because they are related to different standards (which can be incomplete or bad documented) and implementation frameworks [27]. They are using a huge number of diverse technologies and there is not much knowledge about how to measure or ensure the quality properties [85]. Because of this, it is hard to prove the correctness of WAs.

Correctness is an important aspect in providing good-quality WAs. The quality properties are separated in two sections: external and internal qualities. Correctness is one of the external qualities and is part of "Web Content" properties' category.

If we are referring to the functionality part, we can state that a WA is correct if it behaves according to its specification [69] (based on the requirements, we can decide if the information available in a Web page is valid or not [3]). The first step in creating a correct WA is to make a rigorous analysis to precisely state and analyze the similarities and differences among the various devices and browsers. The development of a WA is an entire process, which after the analysis phase continues with stating a list of precisely formulated properties and then creating the WA (ground) models to hold the properties. The implementation can be described as refinements of those initial abstract models. From this, we could build a way to certify the properties of a WA, by checking the obtained model against the given properties. We can classify the correctness properties into two interesting groups[27]:

- **Correctness properties** for **session** and **state management**. The session state should not be affected when, for example, the user navigates away from the page and afterward he or she returns. Replicated parts of the state should be consistent, equivalent between client side and server side, and the state should persist when the client changes (e.g., from desktop to mobile).
- **Application correctness properties**. These properties deal with the dependence of the WA intended behavior on the programming and execution infrastructure (e.g., browser, connection, plug-ins). In order to achieve a precise analysis, a rigorous high-level description is mandatory.

From the fact that no strict way of proving correctness and completeness of the requirements/design exists results a problem regarding the transition from informal to mathematical representations. In [122], the use of ASM ground models (defined as an analysis of dynamic properties of a system using pseudo-code-like descriptions over abstract data structures) is recommended. A characteristic of ground models is the direct correspondence between the interpretation of the system requirements to be modeled and their abstract state machine representation, which simplifies the transition's problem (mentioned previously).

By building an abstract model from the requirements, which satisfies the **CoCoCo-properties**—consistency, correctness, and completeness [23]—we can check whether the WA satisfies the requirements [21]. To fulfill these properties, it means to directly solve the following problems: communication, verification, and validation. Using ASMs for ground models, we can satisfy the properties mentioned before. The simplicity and generality of the ASM language solves the communication problem. The verification problem can be solved by applying standard (pseudo-code) inspection and reasoning. The validation of ASM models can be realized by simulating the ASM runs using the existent tools [25] (e.g., ASM Workbench [40], .NET-executable AsmL engine [16]). CoreASM[12] and ASMETA

[12]http://sourceforge.net/projects/coreasm/.

(ASM mETAmodeling)[13] are two examples of the tools allowing different forms of model analysis for the ASM macros mentioned in the ground models.

In order to reason about correctness of programs and increase their quality, formal verification techniques can be used. By applying model checking to ASMs, one could assure the correctness and quality of software specifications. Farahbod et al. [53] specifies how ASM specifications written in CoreASM can be automatically transformed into Promela specifications, which, afterward, can be verified using the SPIN model checker. AsmetaSMV[13] is a tool that automatically translates ASM specifications written in AsmetaL[55] (the textual notation for ASM models in ASMETA) into models of the NuSMV model checker, and so it allows the verification of computation tree logic (CTL) and linear temporal logic (LTL) formulae.

8.2 Correctness with Respect to Adaptivity

Adaptivity to different devices and user preferences is essential for developing a correct WA. A clear condition for a correct WA says that the application should behave corresponding to the specification. If a WA is not adaptive, then it might not function or display correctly on some devices, which means that it would not reflect the specification. The vast number of types of devices and their browsers are big issues for the content and presentation adaptation. The formal model has to describe the particularities of each device and how to change the content and the presentation correspondingly. This could be done by using the client- and server-side adaptation techniques, which require to query the device for its features.

For each of the two main content adaptation techniques which were mentioned before, server-side and content-side adaptation, there already exist frameworks or third-party tools that could be used during the development of an application. The server-side adaptation is realized with the help of the device detection databases, which are available both in the open-source format (e.g., OpenDDR [86]) and in the commercial format (e.g., WURFL [100], DeviceAtlas [1]). A significant difference between the open source and the commercial solutions is how often the databases are being updated. Commercial databases are more reliable, because most of them are daily updated. This is an important issue to check when a device detection database is considered, because an out-of-date database could result in the delivery of erroneous data to the devices. For the client-side adaptation, the "Modernizr" JavaScript framework [14] could be used. The problem of missing browser functionalities could be solved by using replacement code done in JavaScript, the so-called polyfills. Small visual layouts could be also solved on the client side. A big advantage of the server-side adaptation is the loading time, everything that loads on the server will load faster [48]. When significant changes have to be done, the

[13]http://asmeta.sourceforge.net/.

server offers a high level of control in fine-tuning. Both of the abovementioned techniques are presenting limitations in discovering all the properties of a device. On client side, the physical nature of the device cannot be determined (e.g., OS version, model, maximum HTML size), but on server side, it is not possible to determine the real-time information (e.g., GPS coordinates, device orientation). On server side, the user could cause problems in detecting the device by changing the user agent (UA). The UA is a string available in the HTML header of a Web page and it is used to query the device detection databases. A similar problem could happen on client side, even without the intervention of the user—many browsers return false positives for certain query tests.

9 Related Work

It is beyond the scope of this chapter to discuss the vast literature of formal modeling mobile systems and *service-oriented architectures (SOAs)*, but we refer to some surveys on these fields [32, 37, 96].

However, if we would like to put our work in context, we should mention first of all [110], in which one of the first examples is presented for representing various kinds of published services as a pool of resources, like it is in our model.

The application of the concept of mobile ambients in the development of distributed SOAs is not a novel idea. One of the first research works that investigated and analyzed location-based services whose availability is related to the surrounding physical environment of the user is [73]. This work has not considered mobility yet, but it introduces the notion of service domains. These domains refer to geographical boundaries which are associated with a set of services that is available for the user within the boundary. In other words, Loke et al. [73] describes a new kind of location-based service discovery architecture.

In the software development community, a UML-based modeling approach called *service-oriented architecture modeling language (SoaML)* [102] has started to become increasingly popular for modeling service-oriented architectures. Moreover, SoaML has an extension called Ambient-SoaML [4, 6] which combines SoaML with the concept of mobile ambients for modeling service-oriented mobile applications.

Another modeling technique is *cloud modeling language (CloudML)* [33, 42], which aims at facilitating the specification of provisioning, deployment, monitoring, and adaptation concerns of multi-cloud systems at design time. Furthermore, CloudML also contains the models@run-time environment for enacting the provisioning, deployment, and adaptation of these systems, as well as for monitoring their status at run time.

Ambient-PRISMA [5] is an aspect-oriented software architectural approach for modeling and developing distributed and mobile applications, which is also extended with the ambient concept. Ambients appear in the (UML-like) metamodel of PRISMA as some special kind of connectors that model the notion of location and offer mobility services to the components. Mobility of architectural elements is supported by reconfiguring the software architecture. Although SoaML, Ambient-SoaML, CloudML, and Ambient-PRISMA are advanced model-driven engineering techniques, which have more or less been integrated with the software development practice, they are definitely not formal methods and they have no relation to any mathematically rigorous formal specification and verification techniques unlike our approach.

Another research similar to ours is *Cloud Calculus* [67], which is built upon ambient calculus for capturing the dynamic topology of cloud computing systems. Cloud Calculus is effective to verify whether global security policies are preserved after virtual machine migrations, but it is a very specific tool which is not applicable for giving the formal specification of functionalities of cloud/distributed systems.

There exists also some research works which apply ambient logic [35, 39] tailored for ambient calculus to formally verify various security protocols (e.g., authentication and key agreement protocol (AKA) [125]). Furthermore, [41] presents a general algorithm on how the processes expressed by ambient calculus can be model checked against formulas of the ambient logic.

In [26], the ambient concept (notion of "nestable" environments where computation can happen) is introduced into the ASM method, such that the definition of ambient ASM is based upon the semantics of ASM without any changes. But ambient ASM, on which our model is based, is not the only research which aims to build in a concept of mobile ambients to the ASM method. In [113], 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 as refined and versatile as ambient ASM.

Our approach according to which service instances must be always equipped with unique operations such that they compose the interface of a service instance was originally applied among others in the definition of *Abstract State Services (AS^{2}s)*. AS^{2}s were introduced first in [75] and were extended and described in detail in [76, 77]. The theory of AS^{2}s integrates a customized ASM thesis for database transformations [98] as well. In an AS^2, there are views on some hidden database layer that are equipped with service operations denoted by unique identifiers. 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.

For an algebraic formalization of plots, *Kleene algebras with tests (KATs)* [70] have been applied in [78]. Prior to this work, the formalization of algebraic plots was founded in [94, 95]. In [18], the idea of ASM-based plot expressions was outlined. Then in [97], it was described in detail how KAT expressions in plots can be replaced with *assignment free* ASMs, which have more expressive power.

In [78], a formal high-level specification of service cloud is given. This work is similar to ours in some aspects. Namely, it applies the language-independent AS^{2}s with algebraic plots for representing services. But it principally focuses on service specification, service discovery, service composition, and orchestration of service-based processes; and it does not apply any formal approach to define either static or dynamically changing structures of distributed system components.

Workflow-based solutions have often been applied in case of Web service orchestration for controlling execution of some combination of activities. The *Business Process Execution Language (BPEL)* [68, 74] is a typical representative of this approach, where a workflow can be specified such that the given Web services can be run sequentially, in parallel or even iterated. But in contrast to algebraic plot-based solutions equipped with the concept of service operations, services appear as indivisible components without being interleaved in a BPEL orchestration.

With respect to identity management, there have been several attempts to define a client-centric approach [115]. The author of the paper [2] describes a privacy-enhanced user-centric identity management system allowing users to select their credentials when responding to authentication requests. It introduces "*a category-based privacy preference management for user-centric identity management*" using a CardSpace compatible selector for Java and extended privacy utility functions for P3PLite and PREP languages. The advantage of such a system is that it allows users to select the specific attributes that will eventually be sent to a relying party. Such a system works well for enhancing privacy; however, it fails to address the extra overhead inflicted on the user. As the paper [92] shows, a typical user would tend to ignore obvious security and privacy indicators. For composite services, the authors of the paper [124] describe a universal identity management model focused on anonymous credentials. The model "*provides the delegation of anonymous credentials and combines identity meta-system to support easy-to-use, consistent experience and transparent security*."

From a client-centric perspective, Microsoft introduced an identity management framework (CardSpace) aimed at reducing the reliance on passwords for Internet user authentication while improving the privacy of information. The identity meta-system, introduced with Windows Vista and Internet Explorer 7, makes use of an "*open*" XML-based framework allowing portability to other browsers via customized plug-ins. However, CardSpace does suffer from some known privacy and security issues, mentioned in the papers [8, 87]. The concept of a client-centric identity meta-system is thoroughly defined in the paper [36]. The framework proposed here is used for the protection of privacy and the avoidance of unnecessary propagation of identity information while at the same time facilitating exchange of specific information needed by Internet systems to personalize and control access to services. By defining abstract services, the framework facilitates the interoperation of the different meta-system components.

Passwords managers can, in our opinion, reside within the topic of automatic authentication for individual users. Projects such as KeePass[90] and LastPass [72] do offer similar functionalities to the IdMM. However, both fall short in two key criteria. Neither of them is truly automatic, since some input is required by the user upon authentication to a service, and while both work well with individual users, they cannot be adapted for SMEs.

The content adaptation topic presented major interest for the mobile device direction. Different solutions for cross-platform mobile development are available, like cross-platform compilers/applications and building HTML5 or HTML5 hybrid applications [44]. Native mobile applications are not applicable to our problem, because we do not want to create a corresponding application for every mobile operating system; our scope is developing a general application which respects an ASM specification. Cremin [47] explains shortly the different mobile Web content adaptation techniques. Each technique has its own advantages and disadvantages; one should choose the technique that better suits the project, after analyzing the requirements. In our case, the hybrid approach suits the best, so we want to achieve something similar to what [91] presents. Still, our solution would like to use the device detection database as few times as possible, only when the information regarding the properties cannot be retrieved on the client side. The content of a Web page is adapted on the server side corresponding to the properties detected on the client side. Another technique is responsive design, which is combining the Cascading Style Sheets (CSS) media queries with the flexible images and is using the flexible grid technique to scale the page [47]. An example of how to use responsive design for creating a mobile application is presented in [62]. Regardless of the adaptation technique used, the previous works do not apply a formal method in order to prove the correctness of their system.

10 Conclusions

In this chapter among others, we described a high-level formal model of a cloud service architecture in terms of a novel formal approach. The applied method is able to incorporate the major advantages of the ASMs and of ambient calculus. Namely, by this, one is capable to specify in the same formal model of a distributed system both the long-range mobility via several boundaries in a dynamically changing spatial hierarchy and the algorithms of executable components. Since this cloud model is the first nontrivial model which is based on this method, our work also revealed how viable is in practice our two abstraction layers formal approach.

Our cloud model applies a new client-cloud interaction solution 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.

The ASM formal models we presented can allow model analysis at early stages of system design, by applying different validation techniques on them, like simulation or scenario construction, through the use of one of the existing ASM tools (e.g., CoreASM, ASMETA). This, together with the verification (model checking of properties) of our models, is part of our future work.

Client-To-Client Interaction Besides the cloud service architecture model, we also discussed a high-level formal definitions of some novel client-to-client interaction features, by which not only information but cloud service functions can be also shared among the cloud users. Our approach is general enough to manage a situation in which a shared version of a cloud service is shared again several times by several users.

Furthermore, if we shift the client-to-client functionality to client side and wrap into a middleware as it is proposed in Sect. 3 and depicted on Fig. 2b, then no traces of the user activities belonging to the shared services will be left on the cloud. The reason for this is because all the service operations which are shared via a channel are used on behalf of its initial distributor. This consideration can lead one step into the direction of anonym usage of cloud services. The consequence of this is that if a cloud user who has contracts with some service providers completely or partially shares some services via a channel, then he or she should be aware of the fact that all generated costs caused by the usage of these shared services will be allocated to him or her.

Identity Management Machine The current specification of the IdMM, presented in Sect. 6, is thus far limited to the IdMM$_{Core}$, IdMM$_{Client}$, IdMM$_{User}$, and IdMM$_{Cloud}$ agents, allowing for an automatic authentication tool for cloud-based services in the direct and obfuscated interaction scenarios. Since the specification of the IdMM$_{Protocol}$ and IdMM$_{Provisioning}$ is currently an ongoing task, we intend to study the various open protocols used in identity management (such as LDAP) or in authentication and authorization (such as OpenID and OpenAuth). With this study complete, we will then refine the cloud-based functions regarding the authentication via the formally mentioned protocols. Our plan is to first allow IdMM to use external identity providers to authenticate to services while IdMM only takes care of automating the process. With this complete, we can then specify the IdMM$_{Protocol}$ agent which will act as a stand-alone identity provider communicating with the client's directory via the IdMM$_{Client}$ agent.

With the specification of the IdMM$_{Protocol}$ agent complete, we can then detail the IdMM$_{Provisioning}$ agent which will allow for an easier management of the identities and access rights stored both on the client and the cloud provider's systems. The agent will be responsible for the creation, modification, and deletion of identities on the client's directory as well as account creation, synchronization, and deletion on the necessary cloud services. The system will also be responsible for periodical passwords resets on cloud services.

The final step in our research is to apply an access management component to the IdMM. The specification of the IdMM core agent allows for the enforcement of access rights via the rules for the *AuthorizeLogin* state. Further research in the realm of access rights management is required before a full specification of the access management component is achieved.

Cloud Content Adaptivity In Sects. 7.2 and 7.3, the readers can discover how a content adaptation system, created for the interaction between the client's devices and the cloud, can be represented using ASM ground models and how these models are refined to reach implementation phases. We identified four different agents, which are executing the algorithms defined by the corresponding ground models. These models are designed using the ASM formal modeling method, which gives us the possibility of validating and verifying the system.

Further work includes the refinement of all ASM ground models presented in Sect. 7.2, as we did in Sect. 7.3, which means that we will go on with writing the ASM macros, refine them, and finally reach the development phase. The refinement of the ASM macros will lead to the system prototype's implementation. Future research could include the specification of the adaptation rules for each HTML element corresponding to different categories. It is not mandatory to have a different rule for every device; we could also define a rule per group/category. Kumar and Kumar [71] and Yang et al. [123] are also dealing with HTML adaptation, by using the DOM content extraction and rule repositories, respectively, by adopting description logics for the creation of a hierarchical ontology. In our case, the rules will be specified in terms of ASMs.

Acknowledgements This research has been supported by the Christian Doppler Society.

References

1. Afilias Technologies Ltd: Mobile device detection solution—deviceatlas. https://deviceatlas.com/ (2013)
2. Ahn, G.J., Ko, M., Shehab, M.: Privacy-enhanced user-centric identity management. In: IEEE International Conference on Communications, 2009. ICC '09, pp. 1–5 (2009). doi:10.1109/ICC.2009.5199363
3. Alalfi, M.H., Cordy, J.R., Dean, T.R.: Modeling methods for web application verification and testing: state of the art. Softw. Test. Verif. Reliab. **19**(4), 265–296 (2009). doi:10.1002/stvr.v19:4. http://dx.doi.org/10.1002/stvr.v19:4
4. Ali, N., Babar, M.: Modeling service oriented architectures of mobile applications by extending soaml with ambients. In: 35th Euromicro Conference on Software Engineering and Advanced Applications, 2009. SEAA '09, pp. 442–449 (2009). doi:10.1109/SEAA.2009.25
5. Ali, N., Ramos, I., Solis, C.: Ambient-prisma: ambients in mobile aspect-oriented software architecture. J. Syst. Softw. **83**(6), 937–958 (2010). doi:http://dx.doi.org/10.1016/j.jss.2009.12.009. http://www.sciencedirect.com/science/article/pii/S0164121209003161 [Software Architecture and Mobility]
6. Ali, N., Chen, F., Solis, C.: Modeling support for mobile ambients in service oriented architecture. In: IEEE First International Conference on Mobile Services (MS), 2012, pp. 1–8 (2012). doi:10.1109/MobServ.2012.18
7. Alpár, G., Hoepman, J.H., Siljee, J.: The identity crisis, security, privacy and usability issues in identity management. CoRR abs/1101.0427 (2011)
8. Alrodhan, W., Mitchell, C.: Addressing privacy issues in cardspace. In: Third International Symposium on Information Assurance and Security, 2007. IAS 2007, pp. 285–291 (2007). doi:10.1109/IAS.2007.12
9. Altenhofen, M., Börger, E., Lemcke, J.: An abstract model for process mediation. In: Proceedings of the 7th International Conference on Formal Methods and Software Engineering, ICFEM'05, pp. 81–95. Springer, Berlin/Heidelberg (2005). doi:10.1007/11576280_7. http://dx.doi.org/10.1007/11576280_7
10. Amazon Web Services: Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/ (2014)
11. Apache Software Foundation: Apache directory. http://directory.apache.org/apacheds/ (2013)
12. Apache Software Foundation: Apache tomcat. http://tomcat.apache.org/ (2013)

13. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010). Lecture Notes in Computer Science, vol. 5977, pp. 61–74. Springer, Heidelberg (2010)
14. Ateș, F., Irish, P., Sexton, A., Seddon, R., Farkas, A.: Modernizr: the feature detection library for html5/css3. http://modernizr.com/ (2013)
15. Azure, M.: Azure: Microsoft's cloud platform. https://azure.microsoft.com/ (2014)
16. Barnett, M., Schulte, W., Tillmann, N.: Using asml for runtime verification. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) Abstract State Machines 2003. Lecture Notes in Computer Science, vol. 2589, pp. 407–407. Springer, Berlin/Heidelberg (2003). doi:10.1007/3-540-36498-6_24. http://dx.doi.org/10.1007/3-540-36498-6_24
17. Beste, F.: The model prover: a sequent-calculus based modal π-calculus model checker tool for finite control π-calculus agents. Master's thesis, Department of Computer Science, Uppsala University (1998). ftp://ftp.docs.uu.se/pub/mwb/x4.ps.gz
18. Binemann-Zdanowicz, A., Thalheim, B.: Modeling information services on the basis of ASM semantics. In: Proceedings of the Abstract State Machines 10th International Conference on Advances in Theory and Practice, ASM'03, pp. 408–410. Springer, Berlin/Heidelberg (2003). http://dl.acm.org/citation.cfm?id=1754749.1754777
19. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Logic **4**, 578–651 (2003). doi:http://doi.acm.org/10.1145/937555.937561. http://doi.acm.org/10.1145/937555.937561
20. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: correction and extension. ACM Trans. Comput. Logic **9**, 19:1–19:32 (2008). doi:http://doi.acm.org/10.1145/1352582.1352587. http://doi.acm.org/10.1145/1352582.1352587
21. Bolis, F., Gargantini, A., Guarnieri, M., Magri, E., Musto, L.: Model-driven testing for web applications using abstract state machines. In: Grossniklaus, M., Wimmer, M. (eds.) Current Trends in Web Engineering. Lecture Notes in Computer Science, vol. 7703, pp. 71–78. Springer, Berlin/Heidelberg (2012). doi:10.1007/978-3-642-35623-0_7. http://dx.doi.org/10.1007/978-3-642-35623-0_7
22. Börger, E.: The asm refinement method. Form. Asp. Comput. **15**(2–3), 237–257 (2003). doi:10.1007/s00165-003-0012-7. http://dx.doi.org/10.1007/s00165-003-0012-7
23. Börger, E.: Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. Form. Asp. Comput. **19**(2), 225–241 (2007). doi:10.1007/s00165-006-0019-y. http://dx.doi.org/10.1007/s00165-006-0019-y
24. Börger, E., Rosenzweig, D.: The wam—definition and compiler correctness. In: Beierle, C., Plümer, L. (eds.) Logic Programming: Formal Methods and Practical Applications. Studies in Computer Science and Artificial Intelligence, vol. 11, pp. 20–90. North-Holland, Amsterdam (1995)
25. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, New York (2003)
26. Börger, E., Cisternino, A., Gervasi, V.: Ambient abstract state machines with applications. J. Comput. Syst. Sci. (Special Issue in honor of Amir Pnueli) **78**(3), 939–959 (2012). doi:10.1016/j.jcss.2011.08.004. http://dx.doi.org/10.1016/j.jcss.2011.08.004
27. Börger, E., Cisternino, A., Gervasi, V.: Contribution to a rigorous analysis of web application frameworks. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) Integrated Formal Methods. Lecture Notes in Computer Science, vol. 7321, pp. 1–20. Springer, Berlin/Heidelberg (2012). doi:10.1007/978-3-642-30729-4_1. http://dx.doi.org/10.1007/978-3-642-30729-4_1
28. Bósa, K.: A formal model of a cloud service architecture in terms of ambient asm. Technical report, Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC), Johannes Kepler University Linz, Hagenberg (2012)

29. Bósa, K.: An ambient asm model for client-to-client interaction via cloud computing. In: Proceedings of the 8th International Conference on Software and Data Technologies (ICSOFT), Reykjavik, Iceland, pp. 459–470. SciTePress (2013). doi:10.5220/0004490904590470. http://www.icsoft.org/. (Best Paper Award)
30. Bósa, K.: An Ambient ASM Model for Cloud Architectures. Acta Cybernetica (2014). Submitted
31. Bósa, K.: Formal modeling of mobile computing systems based on ambient abstract state machines. Semant. Data Knowl. Bases **7693**, 18–49 (2013). doi:10.1007/978-3-642-36008-4_2. http://dx.doi.org/10.1007/978-3-642-36008-4_2
32. Boudol, G., Castellani, I., Hennessy, M., Kiehn, A.: A theory of processes with localities. Form. Asp. Comput. **6**, 165–200 (1994). doi:10.1007/BF01221098. http://dx.doi.org/10.1007/BF01221098
33. Brandtzaeg, E., Parastoo, M., Mosser, S.: Towards a domain-specific language to deploy applications in the clouds. In: Cloud Computing 2012: 3rd International Conference on Cloud Computing, Grids, and Virtualization, pp. 213–218. IARIA (2012)
34. Brunette, G., Mogull, R.: Security guidance for critical areas of focus in cloud computing V2. 1. http://goo.gl/PxAeP (2009)
35. Caires, L., Cardelli, L.: A spatial logic for concurrency (part I). Inf. Comput. **186**(2), 194–235 (2003)
36. Cameron, K., Posch, R., Rannenberg, K.: Proposal for a common identity framework: a user-centric identity metasystem. http://goo.gl/3q2sF (2008)
37. Cardelli, L.: Mobility and security. In: Bauer, F.L., Steinbrüggen, R. (eds.) Foundations of Secure Computation Proceedings of the NATO Advanced Study Institute. Lecture Notes for Marktoberdorf Summer School 1999 (A summary of several Ambient Calculus papers), pp. 3–37. IOS Press, Amsterdam (1999)
38. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. **240**(1), 177–213 (2000)
39. Cardelli, L., Gordon, A.D.: Anytime, anywhere: modal logics for mobile ambients. In: In POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 365–377. ACM (2000)
40. Castillo, G.: The asm workbench: a tool environment for computer-aided analysis and validation of abstract state machine models. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 2031, pp. 578–581. Springer, Berlin/Heidelberg (2001). doi:10.1007/3-540-45319-9_40. http://dx.doi.org/10.1007/3-540-45319-9_40
41. Charatonik, W., Gordon, A., Talbot, J.M.: Finite-control mobile ambients. In: Métayer, D.L. (ed.) Programming Languages and Systems. Lecture Notes in Computer Science, vol. 2305, pp. 295–313. Springer, Berlin/Heidelberg (2002). doi:10.1007/3-540-45927-8_21. http://dx.doi.org/10.1007/3-540-45927-8_21
42. Chauvel, F., Ferry, N., Morin, B., Rossini, A., Solberg, A.: Models@Runtime to support the iterative and continuous design of autonomous reasoners. In: Bencomo, N., France, R., Götz, S., Rumpe, B. (eds.) MRT 2013: 8th International Workshop on Models@run.time at MODELS 2013: ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems. CEUR Workshop Proceedings (2013)
43. Chelemen, R.M.: Modeling a web application for cloud content adaptation with asms. In: International Conference on Cloud Computing and Big Data (CloudCom-Asia), 2013, pp. 44–51 (2013). doi:10.1109/CLOUDCOM-ASIA.2013.76
44. Chipperfield, R.: An introduction to cross-platform mobile development technologies. http://www.codeproject.com/Articles/388811/An-introduction-to-cross-platform-mobile-developme (2012)
45. Christian-Albrechts-Universität zu Kiel: Visual programming of databases - visual sql. http://www.informatik.uni-kiel.de/en/is/miscellaneous/visualsql (2008)
46. Cloud Security Alliance: Top threats to cloud computing. http://goo.gl/wLd7m (2010)
47. Cremin, R.: Mobile web content adaptation techniques. http://mobiforge.com/starting/story/mobile-web-content-adaptation-techniques (2011)

48. Cremin, R., Passani, L.: Server-side device detection: history, benefits and how-to. http://mobile.smashingmagazine.com/2012/09/24/server-side-device-detection-history-benefits-how-to/ (2012)
49. Dam, M.: Model checking mobile processes. In: Best, E. (ed.) CONCUR'93, 4th International Conference on Concurrency Theory. Lecture Notes in Computer Science, vol. 715, pp. 22–36. Swedish Institute of Computer Science/Springer, Kista/Berlin/Heidelberg (1993). Full version in Research Report R94:01
50. Dhamija, R., Dusseault, L.: The seven flaws of identity management: usability and security challenges. IEEE Secur. Priv. **6**(2), 24 –29 (2008). doi:10.1109/MSP.2008.49
51. Dold, A.: A formal representation of abstract state machines using pvs. Technical report, University Ulm (1998)
52. ENISA: Cloud computing. benefits, risks and recommendations for information security. Technical report, The European Network and Information Security Agency. http://www.enisa.europa.eu/activities/risk-management/files/deliverables/cloud-computing-risk-assessment (2009)
53. Farahbod, R., Glässer, U., Ma, G.: Model checking coreasm specifications. In: Proceedings of the 14th International Abstract State Machines Workshop (ASM'07) (2007)
54. Gargantini, A., Riccobene, E.: Encoding abstract state machines in pvs. In: Gurevich, Y., Kutter, P., Odersky, M., Thiele, L. (eds.) Abstract State Machines: Theory and Applications. Lecture Notes in Computer Science, vol. 1912, pp. 303–322. Springer, Heidelberg (2000)
55. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univers. Comput. Sci. **14**(12), 1949–1983 (2008)
56. Gervasi, V.: An asm model of concurrency in a web browser. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 79–93. Springer, Berlin/Heidelberg (2012). doi:10.1007/978-3-642-30885-7_6. http://dx.doi.org/10.1007/978-3-642-30885-7_6
57. Google: Google apps for business. http://www.google.com/enterprise/apps/business/ (2014)
58. Google Chrome: What are extensions?. http://developer.chrome.com/extensions/index.html (2013)
59. Google Developers: Google web toolkit. https://developers.google.com/web-toolkit/ (2013)
60. Google Developers: Google app engine: platform as a service . https://developers.google.com/appengine/ (2014)
61. Gordon, A.D., Cardelli, L.: Equational properties of mobile ambients. Math. Struct. Comp. Sci. **13**, 371–408 (2003). doi:10.1017/S0960129502003742. http://dl.acm.org/citation.cfm?id=966815.966816
62. Grigsby, J.: Responsive design for apps. http://blog.cloudfour.com/responsive-design-for-apps-part-1/ (2013)
63. Gunjan, K., Sahoo, G., Tiwari, R.K.: Identity management in cloud computing - a review. Int. J. Bus. Forecast. Market. Intell. **1**(4) (2012). http://www.ijert.org
64. Gurevich, Y.: Evolving algebra 1993: Lipari guide. In: International Conference on Functional Programming, pp. 9–36. Oxford University Press, New York (1994)
65. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Trans. Comput. Logic **1**, 77–111 (2000). doi:http://doi.acm.org/10.1145/343369.343384. http://doi.acm.org/10.1145/343369.343384
66. Jaakkola, H., Thalheim, B.: Visual SQL – high-quality ER-based query treatment. In: Jeusfeld, M.A., Pastor, O. (eds.) Conceptual modeling for novel application domains. In: Proceedings of ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, 13 October 2003. Lecture Notes in Computer Science, vol. 2814, pp. 129–139. Springer, Heidelberg (2003)
67. Jarraya, Y., Eghtesadi, A., Debbabi, M., Zhang, Y., Pourzandi, M.: Cloud calculus: security verification in elastic cloud computing platform. In: Smari, W.W., Fox, G.C. (eds.) CTS, pp. 447–454. IEEE, Denver (2012). http://dblp.uni-trier.de/db/conf/cts/cts2012.html#JarrayaEDZP12

68. Juric, M.B.: Business Process Execution Language for Web Services BPEL and BPEL4WS, 2nd edn. Packt Publishing, Birmingham (2006)
69. Kappel, G.: Chapter I: Web applications. University Lecture. http://is.uni-paderborn.de/fileadmin/Informatik/AG-Engels/Lehre/WS1213/WE/slides/WE-2012-01.pdf (2012)
70. Kozen, D.: Kleene algebra with tests. Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
71. Kumar, V., Kumar, A.: Client device based content adaptation using rule base. J. Comput. Sci. **7**(12), 1908–1913 (2011)
72. LastPass: Lastpass—the last password you have to remember. https://lastpass.com/ (2013)
73. Loke, S.W., Krishnaswamy, S., Naing, T.T.: Service domains for ambient services: concept and experimentation. Mob. Netw. Appl. **10**(4), 395–404 (2005). http://dl.acm.org/citation.cfm?id=1160162.1160165
74. Louridas, P.: Orchestrating web services with BPEL. IEEE Softw. **25**(2), 85–87 (2008). http://doi.ieeecomputersociety.org/10.1109/MS.2008.42
75. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: Abstract state services. In: Object-Oriented and Entity-Relationship Modelling/International Conference on Conceptual Modeling/The Entity Relationship Approach, pp. 406–415 (2008). doi:10.1007/978-3-540-87991-6_48
76. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: Composing personalised services on top of abstract state services. In: Delcambre, L., Kaschek, R.H., Mayr, H.C. (eds.) The Evolution of Conceptual Modeling, no. 08181 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl (2008). http://drops.dagstuhl.de/opus/volltexte/2008/1597
77. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: A theory of data-intensive software services. Serv. Orient. Comput. Appl. **3**(4), 263–283 (2009)
78. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: A formal model for the interoperability of service clouds. Service Oriented Computing and Applications **6**(3), 189–205 (2012). doi:10.1007/s11761-012-0101-7. http://cdcc.faw.jku.at/publications/kdschewe/schewe2012FMCloud.pdf
79. Mather, T., Kumaraswamy, S., Latif, S.: Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance. O'Reilly Media, Inc., Sebastopol (2009)
80. Mell, P., Grance, T.: The nist definition of cloud computing. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (2011)
81. Microsoft: Office 365. http://office.microsoft.com/en-us/ (2014)
82. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Inform. Comput. **100**(1), 1–77 (1992). doi:10.1016/0890-5401(92)90008-4. http://dx.doi.org/10.1016/0890-5401(92)90008-4
83. Nida, P., Dhiman, H., Hussain, S.: A survey on identity and access management in cloud computing. Int. J. Eng. Res. Technol. **3**(4) (2014). http://www.ijert.org/
84. Novell: Ldap classes for java. http://www.novell.com/developer/ndk/ldap_classes_for_java.html (2013)
85. Offutt, J.: Web software applications quality attributes. In: Quality Engineering in Software Technology (CONQUEST 2002), pp. 187–198 (2002). http://www.cs.gmu.edu/~offutt/rsrch/papers/conquest02.pdf
86. OpenDDR LLC: Openddr - the best open and completely free device description repository with access apis available worldwide. http://www.openddr.org/ (2013)
87. Oppliger, R., Gajek, S., Hauser, R.: Security of microsoft's identity metasystem and cardspace. In: ITG-GI Conference Communication in Distributed Systems (KiVS), pp. 1–12 (2007)
88. Prodromou, E.: Openid privacy concerns. http://goo.gl/WIDYx (2007)
89. Pusch, C.: Verification of compiler correctness for the wam. In: von Wright, J., Grundy, J., Harrison, J. (eds.) Theorem Proving in Higher Order Logics (TPHOLs'96). Lecture Notes in Computer Science, vol. 1125, pp. 347–362. Springer, Berlin (1996)
90. Reichl, D.: Keepass password safe. http://www.keepass.info/contact.html (2013)

91. Reiger, B., Rieger, S.: Adaptation: why responsive design actually begins on the server. http://www.slideshare.net/yiibu/adaptation-why-responsive-design-actually-begins-on-the-server (2012)
92. Schechter, S., Dhamija, R., Ozment, A., Fischer, I.: The emperor's new security indicators. In: IEEE Symposium on Security and Privacy, 2007. SP '07, pp. 51–65 (2007). doi:10.1109/SP.2007.35
93. Schellhorn, G.: Verifikation Abstrakter Zustandsmaschinen. Ph.D. thesis, University Ulm (1999)
94. Schewe, K.D., Thalheim, B.: Reasoning about web information systems using story algebras. In: Advances in Databases and Information Systems, ADBIS, pp. 54–66 (2004)
95. Schewe, K.D., Thalheim, B.: Conceptual modelling of web information systems. Data Knowl. Eng. **54**(2), 147–188 (2005)
96. Schewe, K.D., Thalheim, B.: Personalisation of web information systems: a term rewriting approach. Data Knowl. Eng. **62**(1), 101–117 (2007). doi:DOI:10.1016/j.datak.2006.07.007. http://www.sciencedirect.com/science/article/pii/S0169023X06001406
97. Schewe, K.D., Thalheim, B.: Term rewriting for web information systems: termination and Church-Rosser property. In: Proceedings of the 8th International Conference on Web Information Systems Engineering, WISE'07, pp. 261–272. Springer, Berlin/Heidelberg (2007). http://dl.acm.org/citation.cfm?id=1781374.1781404
98. Schewe, K.D., Wang, Q.: A customised ASM thesis for database transformations. Acta Cybern. **19**(4), 765–805 (2010). http://dl.acm.org/citation.cfm?id=1945572.1945579
99. Schewe, K.D., Bosa, K., Lampesberger, H., Ma, J., Rady, M., Vleju, B.: Challenges in cloud computing. Scal. Comput. Pract. Exp. **12**(4), 385–390 (2011)
100. ScientiaMobile, Inc: Wurfl - mobile device database by scientiamobile. http://wurfl.sourceforge.net/ (2013)
101. Sermersheim, J.: Lightweight directory access protocol (ldap): the protocol. RFC. http://tools.ietf.org/html/rfc4511 (2006)
102. Service Oriented Architecture Modeling Language (SoaML): Specification for the UML Profile and Metamodel for Services (UPMS) Revised Submission. OMG document: ptc/2009-04-01 (2009)
103. Shaarawy, M.: Cloudification of visual sql. Master's thesis, Johannes Kepler University Linz (2013)
104. Song, H., Compton, K.J.: Verifying π-calculus processes by promela translation. Technical report, Department of Electrical Engineering and Computer Science University of Michigan, Ann Arbor (2003)
105. Spielmann, M.: Abstract state machines: verification problems and complexity. Ph.D. thesis, RWTH Aachen (2000)
106. Spielmann, M.: Model checking abstract state machines and beyond. In: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications, ASM '00, pp. 323–340. Springer, London (2000). http://dl.acm.org/citation.cfm?id=647752.734545
107. Stärk, R.F., Nanchen, S.: A logic for abstract state machines. J. Univers. Comput. Sci. **7**(11), 980–1005 (2001)
108. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Heidelberg (2001)
109. Sturrus, E.: Identity and access management in a cloud computing environment. Master's thesis, Econometric Institute, Erasmus School of Economics, Erasmus University Rotterdam (2011). http://thesis.eur.nl/pub/10422/MA-5%20IENE%20Sturrus_294763.pdf
110. Tanaka, Y.: Meme Media and Meme Market Architectures: Knowledge Media for Editing, Distributing, and Managing Intellectual Resources. Wiley, New York (2003). http://books.google.at/books?id=tezm83Wiqv8C
111. Thalheim, B.: Visual SQL: towards ER-based object-relational database querying. In: Proceedings of the 27th International Conference on Conceptual Modeling, ER '08, pp. 520–521. Springer, Berlin/Heidelberg (2008). doi:10.1007/978-3-540-87877-3_41. http://dx.doi.org/10.1007/978-3-540-87877-3_41

112. The Open Group Identity Management Work Area: Identity management. http://goo.gl/ssPTu (2004)
113. Valente, M., Bigonha, R., Loureiro, A., Maia, M.: Abstractions for mobile computation in ASM. In: Graham, P., Maheswaran, M. (eds.) Proceedings of the International Conference on Internet Computing, IC 2000, 26–29 June, pp. 165–172. CSREA Press, Las Vegas (2000)
114. Venters, W., Whitley, E.A.: A critical review of cloud computing: researching desires and realities. J. Inf. Tech. **27**(3), 179–197 (2012)
115. Vleju, M.B.: A client-centric asm-based approach to identity management in cloud computing. In: Advances in Conceptual Modeling. Lecture Notes in Computer Science, vol. 7518, pp. 34–43. Springer, Berlin/Heidelberg (2012). doi:10.1007/978-3-642-33999-8_5. http://dx.doi.org/10.1007/978-3-642-33999-8_5
116. Vleju, M.B.: A client-centric identity management tool for small and medium enterprises using cloud services. In: 4th Workshop on Software Services, pp. 15–19. Bled, Slovenia (2012). http://www.cloudconference.eu/
117. Vleju, M.B.: Interaction of the idmm with a client-side identity management component. Technical report, Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC), Johannes Kepler University Linz, Hagenberg (2012)
118. Vleju, M.B.: IdMM demo. http://youtu.be/DoM36D0ydkA (2013)
119. Vleju, M.B.: A practical implementation of a client-centric identity management tool for cloud computing. In: EUROCAST-Computer Aided Systems Theory, Gran Canaria (2013)
120. Vleju, M.B.: Automatic authentication to cloud-based services. J. Univers. Comput. Sci. **20**(3), 385–405 (2014). http://www.jucs.org/jucs_20_3/automatic_authentication_to_cloud
121. Winter, K.: Model checking for abstract state machines. Ph.D. thesis, Technical University of Berlin (2001)
122. Yaghoubi Shahir, H., Farahbod, R., Glässer, U.: Refactoring abstract state machine models. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 345–348. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30885-7_28. http://dx.doi.org/10.1007/978-3-642-30885-7_28
123. Yang, S.H., Zhang, J., Huang, A., Tsai, J.P., Yu, P.: A context-driven content adaptation planner for improving mobile internet accessibility. In: IEEE International Conference on Web Services, 2008. ICWS '08, pp. 88–95 (2008). doi:10.1109/ICWS.2008.31. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4670163
124. Zhang, Y., Chen, J.L.: Universal identity management model based on anonymous credentials. In: IEEE International Conference on Services Computing (SCC), pp. 305–312 (2010). doi:10.1109/SCC.2010.46
125. Zhang, X., Li, X., Luo, W.: Aka protocol and its formal analysis and verification using ambient calculus and logics. In: International Conference on Networking and Digital Society, vol. 1, pp. 194–197 (2009). doi:http://doi.ieeecomputersociety.org/10.1109/ICNDS.2009.54