DOI 10.1007/s00165-016-0371-5
BCS © 2016
Formal Aspects of Computing (2016) 28: 567–595

Formal Aspects
of Computing

ASM-based formal design of an adaptivity component for a Cloud system

Paolo Arcaini[1], Roxana-Maria Holom[2] and Elvinia Riccobene[3]

[1] Charles University in Prague, Faculty of Mathematics and Physics, Prague, Czech Republic
[2] Christian-Doppler Laboratory for Client-Centric Cloud Computing Hagenberg, Johannes Kepler University Linz, Linz, Austria
[3] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy

Abstract. The request of formal methods for the specification and analysis of distributed systems is nowadays increasing, especially when considering the development of Cloud systems and Web applications. This is due to the fact that modeling languages currently used in these areas have informal definitions and ambiguous semantics, and therefore their use may be unreliable. Thanks to their mathematical foundation, formal methods can guarantee rigorous system design, leading to precise models where requirements can be validated and properties can be assured, already at the early stages of the system development. In this paper, we present a rigorous engineering process for distributed systems, based on the Abstract State Machines (ASM) formal method. We rely on the foundational notions of ASM ground model and model refinement to obtain a precise model for a client-server application for Cloud systems. This application has been proposed to tackle the problem of making Cloud services usable to different end-devices by adapting on-the-fly the content coming from the Cloud to the different devices contexts. The ASM-based modeling process is supported by a number of validation and verification activities that have been exploited on the component under development to guarantee consistency, correctness, and reliability properties.

Keywords: Distributed systems, Cloud computing, Abstract State Machines, Modeling process, Model refinement, Validation, Verification

1. Introduction

In the context of service-oriented architecture, Cloud systems are emerging as an important trend. A typical architecture of a Cloud system presumes many different end-devices (desktop computers, laptops, tablets, smartphones, etc.), running different operating systems, owning distinct hardware characteristics (e.g., processor speed, size screen, resolution, etc.), using different browsers, connected to the Cloud, and asking for the same Cloud service. For a reliable Cloud system, one thing that must be guaranteed is that an end-user should be able to access the same Cloud service from any kind of device (s)he is using, and that (s)he will receive the same output independently of the used device.

Correspondence and offprint requests to: P. Arcaini, E-mail: arcaini@d3s.mff.cuni.cz

Engineering a solution to the problem of making Cloud services usable to different end-devices in presence of device fragmentation and variety of operating systems, is one of the objectives of the project—which the second author is involved in and this work is part of—presented in [SBL+11]. The long term goal of that project is to develop a Web service infrastructure that allows any kind of device an end-user is using to access the same Cloud service. A *middleware* application should act as an interface between clients' devices and Cloud services. The middleware makes use of various internal components to manage the interaction between the clients and the Cloud services. One of these components, the *adaptativity* one, is responsible for adapting the content coming from the Cloud to the different device profiles. Therefore, all the services available inside the Cloud are adapted on-the-fly to the different end-devices, and are accessible from any device without the need to install any extra tool.

In order to develop a correct and reliable solution for the Web service infrastructure, the use of formal modeling and verification was required. Languages and protocols currently used for developing Web applications have informal definitions and ambiguous semantics [GBC14]; their use is therefore difficult and unreliable. Differently, thanks to their mathematical foundation, formal methods can guarantee model preciseness and properties assurance [GBC14, MsYhSbJ10].

However, Cloud systems and Web service infrastructure pose many challenges to formal approaches. They are usually large systems having distributed setting, comprise many components of heterogeneous nature (physical devices, control software, user interfaces, etc.), and are characterized by the simultaneous operations of different components. These characteristics require a method that supports a distributed computational model, allows modeling a component in separation with the others, and must be able to express model compositionality and action parallelism. It has also to be enriched with techniques for model validation and verification of properties. Besides that, for its practical usage, the formal method must be endowed with a set of tools supporting the developer along the modeling and analysis activities.

We here concentrate only on the adaptivity component of the Web service infrastructure, and we take advantage of this client-server application to present *a system engineering method for the rigorous design of distributed systems*. The approach is based on the *Abstract State Machines* (ASMs) [BS03] formal method, an extension of Finite State Machines [Bör05].

We show how to use the ASMs as *modeling technique* supporting distributed computation, composition of models, and specification of parallel behaviors. The ASM modeling process is based on the concept of *ground model* representing a precise but concise high-level formalization of the system, and on the *refinement principle* that allows to capture all details of the system design through a sequence of refined models till the desired level of detail, possibly bringing to the implementation in a correct and traceable way. We present the *analysis techniques* that the method supports for model validation and verification and how to apply them along the development process to guarantee consistency, correctness, and reliability properties. In particular, we show how to manage the complexity of a distributed system by proceeding in a *modular way*, both during model refinement and model verification. We pose the theoretical bases of this modular approach that permits to design a component at a time, abstracting from the other system components suitably represented as *mock components*.

This paper is an extended version of the work in [HRWW16] where we already presented an ASM model for the adaptivity component. Starting from a preliminary solution presented in [Che13], in [HRWW16] we refined the sketched model to give a more detailed specification of the client and the middleware, leaving their communication abstract. Preliminary analysis techniques were applied to verify the requirements and guarantee application properties.

The work in [HRWW16] was aimed at presenting an improved version of the solution. It skipped a complete presentation of the rigorous design approach for distributed systems that we underline as the main contribution of this paper with respect to [HRWW16], although an enhancement is also from the application point of view. Indeed, the models of the client and the middleware have been improved, and the complete communication cycle between the client and the Cloud (mediated by the middleware) has been modeled. We have proceeded through refinement steps, and each step has been automatically proved correct using a logical solver. Model validation has been remade on the whole new system model. Temporal properties checking the correct behavior of the client and the middleware have been suitably updated, and some properties have been proved for the communication.

The paper is organized as follows. The ASM theory is presented in Sect. 2, and an ASM-based design process and tool support for validation and verification are presented in Sect. 3. Section 4 overviews the general architecture of a client-Cloud interaction middleware, and Sect. 5 presents the client-Cloud adaptivity component we consider. Section 6 describes the ASM formal specification of the adaptivity component. Sections 7 and 8 describe the validation and verification activities we have performed on the formal specification. Some related work is introduced in Sect. 9. Section 10 discusses some lessons learned and concludes the paper.

2. ASM theory

The *ASM method* is a system engineering method that guides the development of software and embedded hardware-software systems seamlessly from requirements capture to their implementation. Within a precise but simple conceptual framework, the ASM method allows a *modeling technique* which integrates dynamic (*operational*) and static (*declarative*) descriptions, and an *analysis technique* that combines *validation* (by simulation and testing) and *verification* methods at any desired level of detail.

ASMs have been successfully applied in different fields [BS03, SSB01] as: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, language design, verification of compilation schemes and compiler back-ends, etc. The method has a rigorous mathematical foundation, but a practitioner needs no special training to use it since ASMs can be correctly understood as pseudo-code or virtual machines working over abstract data structures.

We here introduce the essential concepts of the formalism, with particular emphasis on the multi-agent aspects, useful to understand our modeling of the client-Cloud adaptivity component. For a complete theoretical definition of the ASMs we refer the reader to [BS03].

For system specification, the ASM method builds upon three main concepts (further developed in the following sections):

- *ASM*, an extension of Finite State Machines where unstructured control states are replaced by states with arbitrary complex data;
- *ground model*, an ASM which is a precise but concise high-level system blueprint ("system contract"), serving as authoritative reference model for the design;
- *model refinement*, a general scheme for stepwise instantiations of model abstractions to concrete system elements, providing controllable links between the more and more detailed descriptions at the successive stages of system development.

2.1. Abstract state machines

Abstract State Machines (ASMs) are transition systems based on the concept of *state* representing the instantaneous configuration of the system under development, and *transition rules* describing the change of state. ASM *states* are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. ASM *transition rules* express how function interpretations are modified from one state to the next one, and therefore describe the system configuration changes. The basic form of a transition rule is the *guarded update*: "**if** *Condition* **then** *Updates*", where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed when *Condition* is true; f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms.

An ASM state S is represented by a set of couples (*location*, *value*). ASM *locations*, namely pairs (*function-name*, *list-of-parameter-values*), represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where loc is a location and v its new value.

Functions that never change during any run of the machine are *static*. Those updated by agent actions are *dynamic*, and distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine).

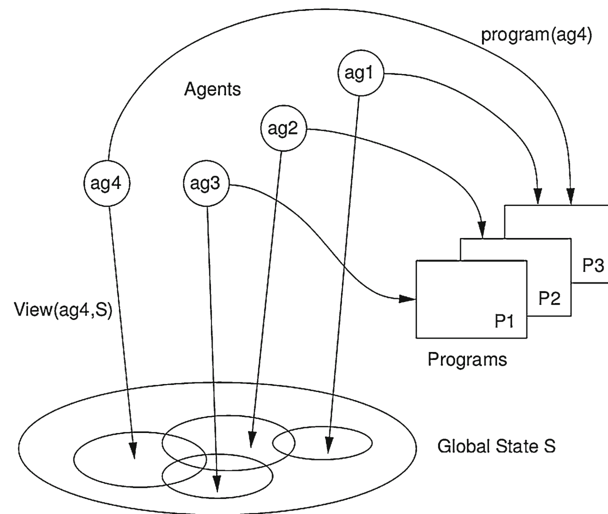

Fig. 1. Global state and partial view in a multi-agent ASM (taken from [BS03])

Besides `if-then`, there is a limited but powerful set of *rule constructors*: `par` for simultaneous parallel actions, `seq` for sequential actions, `choose` for nondeterminism (existential quantification), `forall` for unrestricted synchronous parallelism (universal quantification).

A *computation* of an ASM is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by simultaneously firing all the transition rules which are enabled in S_n. The (unique) *main rule* is a transition rule and represents the starting point of the computation. An ASM can have more than one *initial state*. It is possible to specify state *invariants*.

2.1.1. Multi agents ASM

ASMs allow to model any kind of computational paradigm, from a *single* agent executing simultaneous parallel actions, to distributed *multiple* agents interacting in a synchronous or asynchronous way.

Since a cloud system has a distributed setting, for our modeling purposes we exploit the computational model of a *multi-agent ASM* which is defined as a family of pairs (a, M_a), where a is an agent belonging to a (possibly dynamic) finite set *Agent*, and M_a is a machine specifying its behavior.

Each agent a has a "local" view, $View(a, S)$, of the *global state* S (see Fig. 1). The relation between global and local states is supported by the use of a special (reserved name) 0-ary function $self$ (of type $Agent$) to denote the agents which are executing the underlying "same" but differently instantiated ASM: the function $self$ is interpreted by each agent a as itself.

Agents can have shared view of a portion of a state, namely $View(a, S) \cap View(b, S)$ could be not empty for two different agents a and b. This is possible when *shared* functions are used to model communication among parties. Shared functions are defined as dynamic functions which are directly updatable by the rules of two different agents and can be read by both.[1] In the sequel we denote by $shared(M_a, M_b)$ the set of functions shared between two different agents a and b. Typically a protocol is needed to guarantee consistency of shared function updates.

Each agent a executes its local behavior (the machine M_a), and contributes to determine the next global state S'. However, the relation between local and global states, and therefore the definition of a run for a multi-agent ASM, depends on the synchronous or asynchronous nature of the agents.

[1] The concept of shared functions is also applicable between an agent and its environment, so a shared function can be updatable by the rules of the agent's machine and by the environment, and can be read by both.

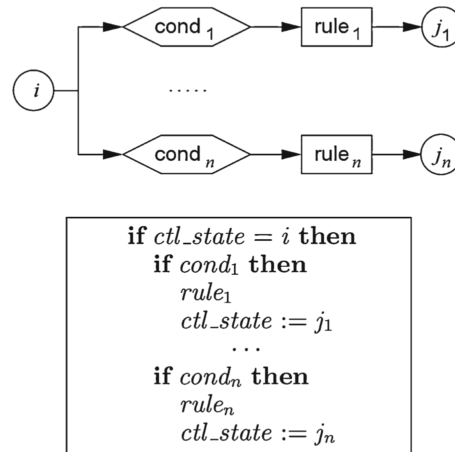

Fig. 2. Control state ASMs

A *multi-agent synchronous ASM* is defined as a set of agents which execute their own ASMs in parallel, synchronized using an implicit global system clock. Semantically a synchronous ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component. The sequence of events determining a run is the sequence of states forming the run of the underlying multi-agent synchronous ASM, where the global clock tick (a built-in signal which is supposed to be present in every event) plays the role of a step counter.

Definition 2.1 A multi-agent ASM with synchronous agents has *quasi-sequential runs*, namely a sequence of states where each state is obtained from the previous state by firing in parallel the rules of all agents.

In *asynchronous multi-agent ASM*, each agent reacts at its own speed without any global clock, and executes its ASM in its own local state. This makes it difficult to uniquely define a global state where agents' moves are executed and to establish an ordering of moves. However, a synchronization schema must be defined to avoid incomparability of agents' moves which may come with different data, clocks, moments and duration of execution.

Since we leave the Cloud part abstract, we can avoid dealing with the complexity of managing synchronization of asynchronous agents. We can suppose the adaptivity of our application to be synchronous w.r.t the client-side. Therefore, we model its operation in a distributed setting in terms of synchronous multi-agent models.

For a reader interested in the theory on asynchronous multi-agent ASMs, we refer to [BS03].

2.1.2. *Control state ASMs*

For modeling the behavior of a single agent, we here use a class of ASMs called *control state ASMs* [BS03]. They are particularly useful to model system *modes* (or *control states*). Control state ASMs have an intuitive graphical representation by means of FSM-like state diagrams, and are defined as follows [BS03]:

Definition 2.2 A control state ASM is an ASM whose rules are all of the form as in Fig. 2: in a given control state i, only one of the conditions $cond_k$ can be true, $1 \leq k \leq n$, if any; the machine executes $rule_k$ if $cond_k$ is true and changes control state from i to j_k; the machine does nothing when no condition $cond_k$ is satisfied.

Note that $rule_k$ can be any ASM rule, so also the parallel composition of more transition rules that is graphically depicted as sequence of rule blocks (or rectangles).

2.2. Ground model and model refinement

A *ground model* is an ASM that can be considered a rigorous high-level system blueprint ("system contract"), specified using domain-specific terms, which can be understood by all stakeholders. The ground model is *abstract*, i.e., it avoids irrelevant details necessary later for the implementation, *correct*, i.e., it reflects the intended initial requirements, and *consistent*, i.e., it removes ambiguities of the initial textual requirements. However, it does not need to be *complete*, i.e., it may leave some given functional requirements unspecified.

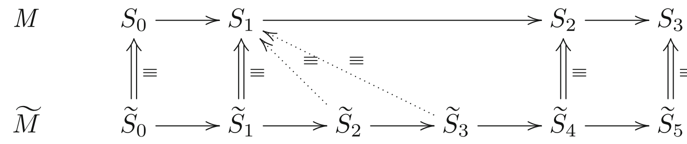

Fig. 3. Stuttering refinement

Model refinement allows to obtain from an abstract model a more detailed one. ASM refinement allows one to refine either the signature (*data refinement*) or the control (*operation refinement*). Also a combination of both changes is possible, while many notions of refinement in the literature keep these two features separated [FL09, Abr96].

At each refinement step, a refined model must be proved correct w.r.t. the abstract one. The definition of *correct refinement* between ASM models has been originally given in [Bör03, Bör07] and simplified for proof automation purposes in our refinement prover. To prove that an ASM $\widetilde{M}$ is a correct refinement of an ASM M, the following items must be defined:

- a notion of *locations of interest* and of *corresponding* locations, i.e., pairs of (possibly sets of) locations one wants to relate;
- a notion of *equivalence* $\equiv$ of the data in the locations of interest which induces a notion of *conformance* between abstract and refined states.

Once the notion of equivalence (in terms of locations of interest) has been determined, one can define when an ASM $\widetilde{M}$ is a correct refinement of M.

Definition 2.3 (*Stuttering refinement*) An ASM $\widetilde{M}$ is a correct *stuttering refinement* of an ASM M if and only if for each $\widetilde{M}$-run $\widetilde{S}_0, \widetilde{S}_1, \ldots$, there is an M-run $S_0, S_1, \ldots$ such that (i) initial states are conformant, i.e., $\widetilde{S}_0 \equiv S_0$, and (ii) if $\widetilde{S}_i$ is conformant to S_j (i.e., $\widetilde{S}_i \equiv S_j$), then $\widetilde{S}_{i+1}$ is conformant with either S_{j+1} or S_j (i.e., $\widetilde{S}_{i+1} \equiv S_{j+1}$ or $\widetilde{S}_{i+1} \equiv S_j$). Moreover, both runs terminate and their final states are the last pair of conformant states, or both runs are infinite.

This notion of refinement requires that every refined state $\widetilde{S}_i$ is conformant with an abstract state S_j, and its successor state $\widetilde{S}_{i+1}$ is conformant with either S_j or with S_{j+1} (being S_{j+1} a next state of S_j). More precisely, given two related runs, ρ of M and $\widetilde{\rho}$ of $\widetilde{M}$, each state of $\widetilde{\rho}$ is in relation with a state of ρ, and each state of ρ is refined by at least a state of the refined run $\widetilde{\rho}$. We therefore perform (0, 1)- or (1, 1)-refinements with the constraint of total conformance relation on the states of the refined machine.

Figure 3 shows the equivalence relation between states of a run of the refined machine and a run of the abstract machine. Refined states $\widetilde{S}_0$, $\widetilde{S}_1$, $\widetilde{S}_4$, and $\widetilde{S}_5$ are linked with abstract states S_0, S_1, S_2, and S_3. Refined states $\widetilde{S}_2$ and $\widetilde{S}_3$, that are not linked with the next abstract state, are conformant with the previous linked state S_1.

Such definition of stuttering refinement is more restrictive than the original refinement definition given by Börger [Bör03, Bör07], but it has the advantage of preserving invariants (as also the approach in [Sch08]): if a property (specified over the locations of interest) is true in every abstract state, it will be true also in every refined state. The classical refinement definition preserves the invariants only weakly, since intermediate refined states are not required to conform to some abstract state. In previous works (e.g., the modeling of a Landing Gear System [AGR15] and of a medical device software [ABG+15]), we found that it is often useful to guarantee that invariants holding in the abstract level, still hold in the refined one.

Stuttering refinement has the further advantage to be checked automatically and a technique has been already developed for this purpose. In order to prove that $\widetilde{M}$ refines M, it suffices to prove that the following property holds:[2]

$$\forall\, \widetilde{S}\,\forall\, \widetilde{S}'\,\forall\, S: \left[\begin{pmatrix} step(\widetilde{S}, \widetilde{S}') \\ \wedge \\ conf(\widetilde{S}, S) \end{pmatrix} \rightarrow \begin{pmatrix} \exists\, S' : (step(S, S') \wedge conf(\widetilde{S}', S')) \\ \vee \\ conf(\widetilde{S}', S) \end{pmatrix}\right] \tag{1}$$

where $conf$ is the conformance relation between states, and $step$ the transition relation.

In case of a multi-agents ASM, model refinement can proceed either in *horizontal* or in *vertical* way, depending if the models of all the agents are taken in consideration at each refinement step, or just the specification of only one agent is considered. Of course, combination of the two approaches is possible. However, to keep the complexity of the system under control, it is preferable to proceed vertically by refining each component at a time. When proving correctness of the refinement, the ASMs of the other agents, kept at a higher level of abstraction, are considered as *mock* models to mimic the behavior of the rest of the system.

The following theorem establishes the relationship between the stuttering refinement of a single ASM and the refinement of a multi-agent ASM. It guarantees that, under some assumptions, local refinement always induces a global refinement.

Theorem 2.4 (Vertical Refinement) *Let M be a synchronous multi-agent ASM consisting of the pairs (a, M_a) and (b, M_b) such that (i) $shared(M_a, M_b) = \emptyset$, or (ii) $\forall\, x \in shared(M_a, M_b)$, x is a read function for a and write function for b, or vice versa, but not both.*

If $\widetilde{M_a}$ is a stuttering refinement of M_a and $shared(M_a, M_b) = shared(\widetilde{M_a}, M_b)$ then $\widetilde{M} = \{(a, \widetilde{M_a}), (b, \widetilde{M_b})\}$ is a stuttering refinement of $M = \{(a, M_a), (b, M_b)\}$.

Proof By hypothesis, Eq. 1 holds among states of M_a and $\widetilde{M_a}$ and for all possible monitored (in M_a and $\widetilde{M_a}$) values of functions x (if any) in $shared(M_a, M_b)$. The same equation trivially holds for M and $\widetilde{M}$ by keeping unchanged the portion of the state that is relative to the computation of M_b, and restricting the values of functions x to those values assigned (written) by b. □

As a consequence of Theorem 2.4, it is possible to prove the following:

Corollary 2.5 *Let $M = \{(a, M_a), (b, M_b)\}$ be a synchronous multi-agent ASM such that*

(i) *$shared(M_a, M_b) = \emptyset$, or*
(ii) *$\forall\, x \in shared(M_a, M_b)$, x is a read function for a and write function for b, or vice versa, but not both.*

If $\widetilde{M_a}$ is a stuttering refinement of M_a, $\widetilde{M_b}$ is a stuttering refinement of M_b, and $shared(M_a, M_b) = shared(\widetilde{M_a}, \widetilde{M_b})$, then $\widetilde{M} = \{(a, \widetilde{M_a}), (b, \widetilde{M_b})\}$ is a stuttering refinement of $M = \{(a, M_a), (b, M_b)\}$.

Banach et al. have shown similar results in [BZSW14], studying the problem of decomposing a monolithic ASM system design (embodying a dual controller/controlled nature) into separate controller and controlled subsystems. They give the conditions that must be guaranteed to be able to split the monolithic ASM (similar to the conditions we are giving in Theorem 2.4 on the shared variables) and show that if the refinement holds for the individual subsystems, it also holds for the global system.

3. ASM-based development process

In this section, we introduce the essential concepts of the design methodology that we exploit in our ASM-based rigorous modeling of the client-Cloud adaptivity component.

The concepts of ground model and model refinement bring to the definition of a rigorous process for ASM-based development. The process is depicted in Fig. 4: the modeling activity is complemented with other validation and verification activities on models, already applicable at ground level and along the chain of refined models. Such activities help to guarantee correctness of the developed system.

[2] A property over initial states must also be proved. It is not reported here.

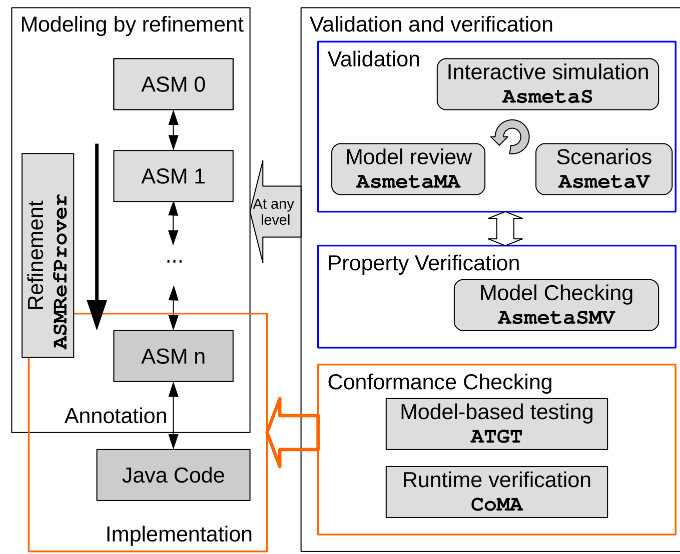

Fig. 4. ASM-based development process

```
asm model
// header
import StandardLibrary
signature:
    dynamic monitored value: Integer
    ...

// body
definitions:
    function ...

    rule r_open = ...

    main rule r_main = ...

// initial state
default init s0:
    ...
```

Code 1. A template for an ASM

A set of tools exists to support the developer in the modeling and analysis activities and to make the ASM method useful in practice. Tools are part of the ASMETA (ASM mETAmodeling) framework[3] [AGRS11, GRS08b], and are strongly integrated in order to permit reusing information about models during different development phases. The IDE `AsmEE` is available to assist the user when editing an ASM model by using the concrete syntax `AsmetaL` [GRS08a].

An `AsmetaL` model is structured into three sections, as shown in Code 1: a *header* in which external models can be imported and the signature is declared, a *body* in which functions, domains, invariants, and rules (including the *main rule*) are defined, and an *init* which initializes the machine. An ASM without the main rule is called *module*.

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

3.1. Modeling by refinement

Using ASMs, the process of *requirements capture* results in constructing rigorous *ground model* (ASM0 in Fig. 4) specified using terms of the application domain, possibly with the involvement of all stakeholders. ASM ground model helps us to solve three major problems of requirements capture: language and communication problem, validation problem, and verification-method problem [BS03].

We start from the description of the informal requirements, and the ASM *ground model* is developed trying to capture the behavior of the system by means of transition rules, at a very high level of abstraction. The process of developing the ground model is iterative and can be error-prone. It goes through a sequence of sketchy intermediate models that could be neither "correct" nor "complete". Rather, they can expose errors, ambiguities, or incompletenesses that are typical of the original requirements. Validation and verification techniques (see Sect. 3.2) are useful to discover such weaknesses. We can try to achieve correctness through continuously reasoning on requirements, executing the ongoing models, checking them again the expected behavior (by validation) and required properties (by verification), till the level of the ground model when we have enough confidence that every feature in the requirements that is relevant for the intended system behavior is present in the reached model. This ground model is not necessarily complete (and usually it is not) since, although it captures a significant subset of initial requirements, it can skip irrelevant details necessary later for the implementation.

Starting from the ground model, by *step-wise refined models*, further details can be added to capture all the functional requirements and the design decisions. In this way, the complexity of the system can be always taken under control, and it is possible to bridge, in a seamless manner, the gap between specification and code.

Each time a model is specified as a refinement of an abstract one, refinement correctness should be checked. This can be done by hand, but we provide an automatic way to achieve this assurance in case of *stuttering refinement*. The tool `ASMRefProver` automatically checks stuttering refinement between two ASM models (see Sect. 6.2).

In Sect. 6, we show the application of the ASM modeling-by-refinement approach for the specification of the client-Cloud adaptivity component.

3.2. Model validation and verification

Modeling activity is supported, at each level of refinement, by model *validation* and *verification* (V&V).

Model validation should be applied, already at ground model level, in order to ensure that the specification really reflects the user needs and statements about the system, and to detect faults in the specification as early as possible with limited effort. ASM model validation is possible by means of the model simulator `AsmetaS` [GRS08a] (see Sect. 7.1) and by the validator `AsmetaV` [CGRS08] (see Sect. 7.2) that allows to build and execute *scenarios* of expected system behaviors. A further validation technique is *model review* (a form of static analysis) that determines if a model has sufficient *quality* attributes (as minimality, completeness, consistency). Automatic ASM model review is possible by means of the `AsmetaMA` tool [AGR10b] (see Sect. 7.3). Typical vulnerabilities and defects that can be introduced during the modeling activity using ASMs are checked as violations of suitable *meta-properties*, expressed as CTL formulae. The violation of a meta-property means that a quality attribute is not guaranteed, and it may indicate the presence of a real fault or only of a stylistic defect.

There is no mandatory order in which these validation activities have to be performed. A modeler can prefer to first check the model statically, and then execute it interactively or by constructing scenarios, or get feedback proceeding in the other way around.

Validation usually precedes the application of more expensive and accurate methods, like formal requirements analysis and verification of properties, that should be applied only when a designer has enough confidence that the specification captures all informal requirements. However, this is only a usual attitude: the order between validation and verification is not mandatory.

Formal verification of ASMs is possible by means of the model checker `AsmetaSMV` [AGR10a] (see Sect. 8). *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulae can be proved on models. In case of multi-agent ASMs, in order to keep the time and memory consumption of model checking under control, one can consider a single ASM model at a time and abstract the other models by using monitored functions (to be understood as *mock objects*).

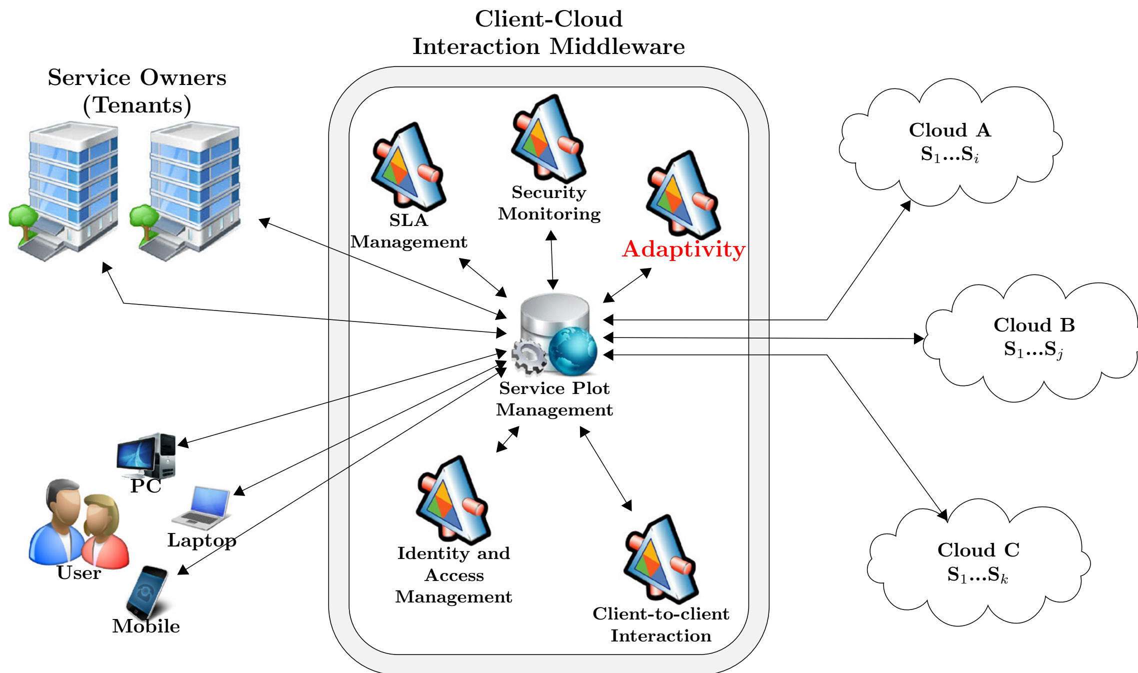

Fig. 5. CCIM architecture

In case a system implementation is available, either derived from the model as last low-level refinement step, or externally provided, also *conformance checking* is possible. For ASMs, techniques and tools for conformance checking w.r.t. Java code have been developed. The tool `ATGT` [GRR03] can be used to automatically generate tests from ASM models[4] and, therefore, to check the conformance *offline*; `CoMA` [AGR12], instead, can be used to perform runtime verification, i.e., to check the conformance *online*.

Note that conformance checking is out of the scope of this paper since no Java implementation is available yet for the client-Cloud adaptivity component, but it is argument for future development. As already stated, indeed, the long term goal of this work is to develop the client-Cloud adaptivity component by following a rigorous development process as that supported by the ASM method. The intention is to develop a final implementation of the component in a correct-by-construction way proceeding by correct-proved refinement of models, from the ground model to the Java code.

4. Client-cloud interaction middleware

Cloud Computing appeared already since years ago and, even though many providers are suggesting that it is a mature technology, we could still identify existing problems (e.g., the lack of client-orientation and the lack of formal specifications of the Cloud software solutions) and, therefore, we consider Cloud computing being an evolving paradigm.

The research done at the *Christian Doppler Laboratory for Client-Centric Cloud Computing* (CDCC)[5] is addressing the difficulties concerning the client side of Cloud Computing by providing a formalized model for a *client-Cloud interaction middleware* (CCIM). Figure 5 shows the system architecture of the CCIM, in which three entities can be identified: the Cloud providers, which offer the infrastructure, the service owners (tenants), which offer the software as *services* (denoted in Fig. 5 by S_n), and the Cloud users, which want to use the previously mentioned services [BHV15]. The aim of the CCIM model is to integrate various components that are designed in a loosely coupled way, such that any of them can be removed from the system, and the rest can still work properly. Each component deals with a specific problem (briefly described in the following) addressed by

[4] Note that sequences generated by `ATGT` could be used to test programs written in any programming language.
[5] http://www.cdcc.faw.jku.at/.

the research at CDCC. The *Service Plot Management* component [Bós12] solves two problems identified in the Cloud architecture model: it allows the users to combine functions of Cloud services belonging to different service owners and also gives the tenants the possibility to fully control the usage of their services. This component also realizes the integration between the other components. The *Identity and Access Management* component [Vle12] accelerates the adoption or migration to Cloud services, while providing a secure privacy-enhanced integration between a client and any given Cloud provider with respect to all the aspects related to identity management. The *Client-to-Client Interaction* [Bós13] is a mechanism which provides a special kind of services, called *channels*, through which the Cloud users can interact with each other in an almost direct way and they can also share the available Cloud resources among them. The *SLA Management* component [LR15] deals with monitoring the service execution adherence to the agreed terms. The *Security Monitoring* component [LR15] provides an intrusion detection system that tries to detect anomalies in the client-Cloud interaction by monitoring the communication protocol language effectively. The *Adaptivity component*—the one designed in this paper—provides on-the-fly adaptivity of Cloud applications to different devices (e.g., smartphones, tablets, laptops) and environment.

The CCIM results into a compound software component, which, due to the applied ambient concept [Bós12], could be easily deployed on the Cloud-side. However, we keep the components in this middleware-based configuration, in order to easily manage their interaction and avoid problems in case any of them is eliminated while the other components still remain active [BHV15]. Another advantage is that, whenever a component must be updated (for adding functionalities or fixing bugs), only one instance of the component must be redeployed. The components are independently designed and developed. Some components are already completed and others are still under development. Not all development processes started by defining a formal model, which could bring to the implementation, but some were first implemented and system verification was used at the end of the development process. In the development process of the adaptivity component, we decided to go from requirements to implementation (planned as future work) through a sequence of refined models (as presented in this paper).

5. Adaptivity component

This work considers the adaptivity component, whose main contribution in the context of CCIM is to adapt on-the-fly the cloud service content and layout based on device properties and network characteristics. In this paper we provide the specification of the adaptivity component (described in Fig. 6), which is analyzed using different validation and verification activities. Our long term goal [SBL+11] is to obtain, through a chain of refined models, a correct-by-construction implementation.

5.1. Informal requirements of the adaptivity component

The first step to be done in a development process is the requirements elicitation, which represents a natural-language problem description. The resulting requirements describe the desired functionality of the system. There are many techniques (e.g., brainstorming, interviews, domain analysis, prototyping, task analysis, user stories, etc) and approaches (informal, semi-formal, formal) one could use to manage requirements elicitation. Approaches use various representations varying from natural language, through graphical forms to formal mathematics.

We here use an informal requirements description, which is very similar to the textual description of UML use cases [BPP99]. However, one could also use *stories* (three or four sentences that describe the features of the system) like the ones exploited by the agile methodologies in the exploration phase [ASRW02, Mey14], or viewpoint methods (whose representation includes hierarchies, tabular collection forms, and diagrams) [HJD10].

Note that the process of ASM modeling works with any approach used for requirements elicitation. In Sect. 6.1 we present how informal requirements are further on captured by the ASM ground models in order to realize a sufficiently precise, unambiguous, consistent, complete and minimal description of the system. We design the logic to collect the device information necessary for the content and layout adaptation satisfying the following requirements:

1. The client initiates the communication with the Cloud by sending a request from his/her device.
2. The middleware intercepts the request and gathers information about the device.
3. The middleware forwards the client's request to the Cloud.
4. The answer coming from the Cloud is intercepted by the middleware and processed.
5. If the answer sent by the Cloud uses other languages than HTML5 and JavaScript and this type of language is not accepted by the device, then the format is converted to HTML5 (and JavaScript, if necessary).

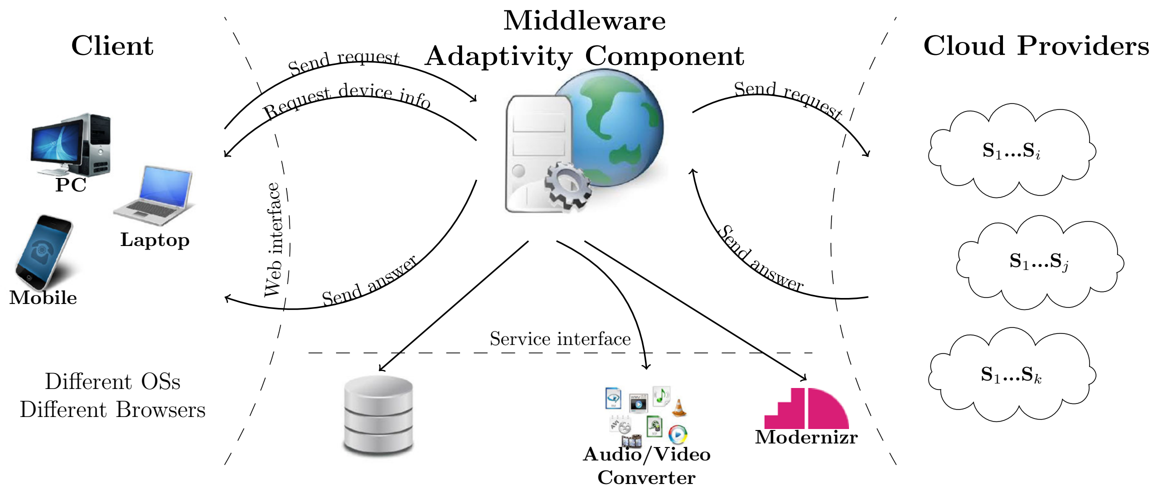

Fig. 6. Architecture of the adaptivity component

6. The Cloud's answer content and layout are adapted by the middleware in accordance with the *device profile* (a device profile contains information about the properties of the device, e.g., screen's width, audio/video format support, touch, etc).
7. The middleware sends the Cloud's answer (after processing it) to the client's device.
8. The client's device displays the message.

5.2. Architecture of the adaptivity component

Figure 6 represents the architecture of the adaptivity component. The architecture contains three main parts: the client-side containing various devices that a user could use for accessing the Cloud services, the Cloud-side with the Cloud services supplied by different Cloud providers, and in the middle there is the adaptivity component, which is integrated in the middleware framework presented in Fig. 5. The middleware software realizes the communication between the client and the Cloud, without the need of the client being aware of it. The client sends its requests through a Web interface (Web application). The communication process is initiated by the client. It selects a Cloud service from the list filtered based on its credentials. The adaptivity component checks, both in the client's cookie and in a local database, for the availability of the *device profile*. If the profile is not available, then it requires the necessary information from the device by creating JavaScript tests with the help of the modernizr framework[6] (tests which are afterwards executed on the device). The result of the JavaScript tests is inserted in a cookie and sent back to the middleware. If and when the corresponding device profile is available on the server-side, then the middleware forwards the client's request to the Cloud and waits for the answer. The device profile is used to adapt the content of the answer coming from the Cloud, which is afterwards sent to the client. The problem of missing browser functionalities can be solved by using replacement code done in JavaScript, the so-called "polyfills". If the format of images and/or videos is not accepted by the device, then third-party tools can be used to generate other formats. The device profile is also saved locally on the server, to be able to reuse the information when the user logs in again from the same device.

[6] http://modernizr.com/.

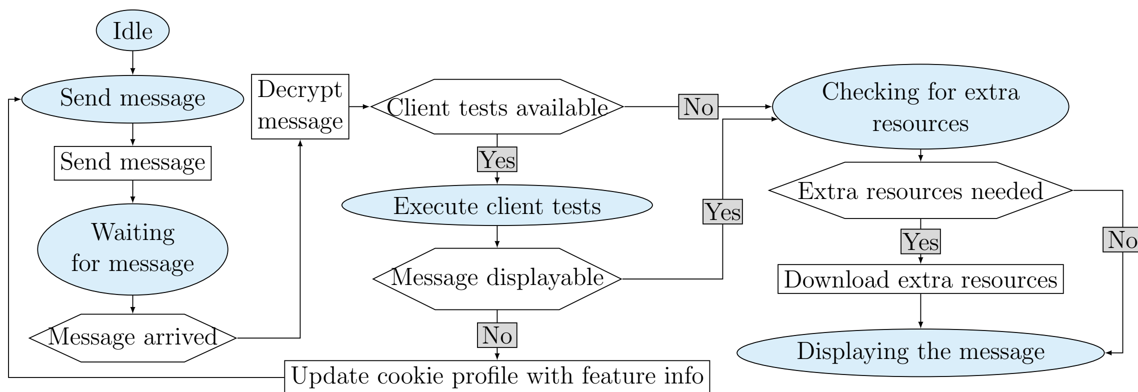

Fig. 7. Client—control state ASM ground model

6. Formal specification

6.1. Ground model

In this section we show how the requirements of the adaptivity component presented in Sect. 5.1 can be formally captured by (control state) ASM ground models.

A preliminary simplified model was already presented in [HRWW16]. We focused only on the client-side and on the interaction between the client and the middleware server, but not on the Cloud-side (left abstract) and neither on the communication with the Cloud. We only had two agent types: the client and the middleware.

In the current work, we present an extended version of the client and of the middleware. The latter includes also the design of a new agent type, *request handler*, that is responsible for handling the requests coming from the clients and processing the answers returned by the Cloud. Moreover, the complete communication cycle between the client and the Cloud, mediated by the middleware, is modeled. In particular, we added the communication between the middleware and the Cloud, and the final part of the communication flow when the middleware sends the message to be displayed to the client.

The whole multi-agent ASM results composed of an agent of type *middleware*, a family of agents of type *device*, owned by the clients, and a family of agents of type *request handler* created at runtime, one for each client request. Their behaviors, modeled in terms of control-state ASMs, are explained in details below.

Client model Figure 7 shows the (control state) ASM ground model of the client (the agents executing the flow presented in this model are the devices of the users). There are six states through which a client's device goes. The initial state is *Idle*, and the final state is *Displaying the message*. The client initiates the communication by sending a request to the Cloud (which is intercepted by the middleware) and then waits for the answer. If a message arrives, then it is automatically decrypted by the browser (we keep this functionality abstract) and the guard *Client tests available* checks if the content of the received message contains JavaScript (modernizr) tests. If so, they are executed and, if the message is not displayable, their result is used to update the cookie (in this way the middleware will be aware of the new values of the device properties). The messages coming from the middleware are labeled with a flag saying whether a message should be displayed or not. A message is not displayable if the middleware needs information about the device to process the answer on the server-side. We can still have JavaScript tests also if the message is displayable, but these are the so-called "tests per request" (the corresponding device information could change by every request, e.g., GPS location, screen orientation view, etc). The device agent reaches the state *Checking for extra resources* directly if no modernizr test is available or after the execution of the tests, if the message is displayable. In this state, if extra resources are needed, they are downloaded by the browser, and then the device agent reaches the final state *Displaying the message* and the Cloud's answer (processed by the middleware) is displayed on the device.

Fig. 8. Middleware—control state ASM ground model

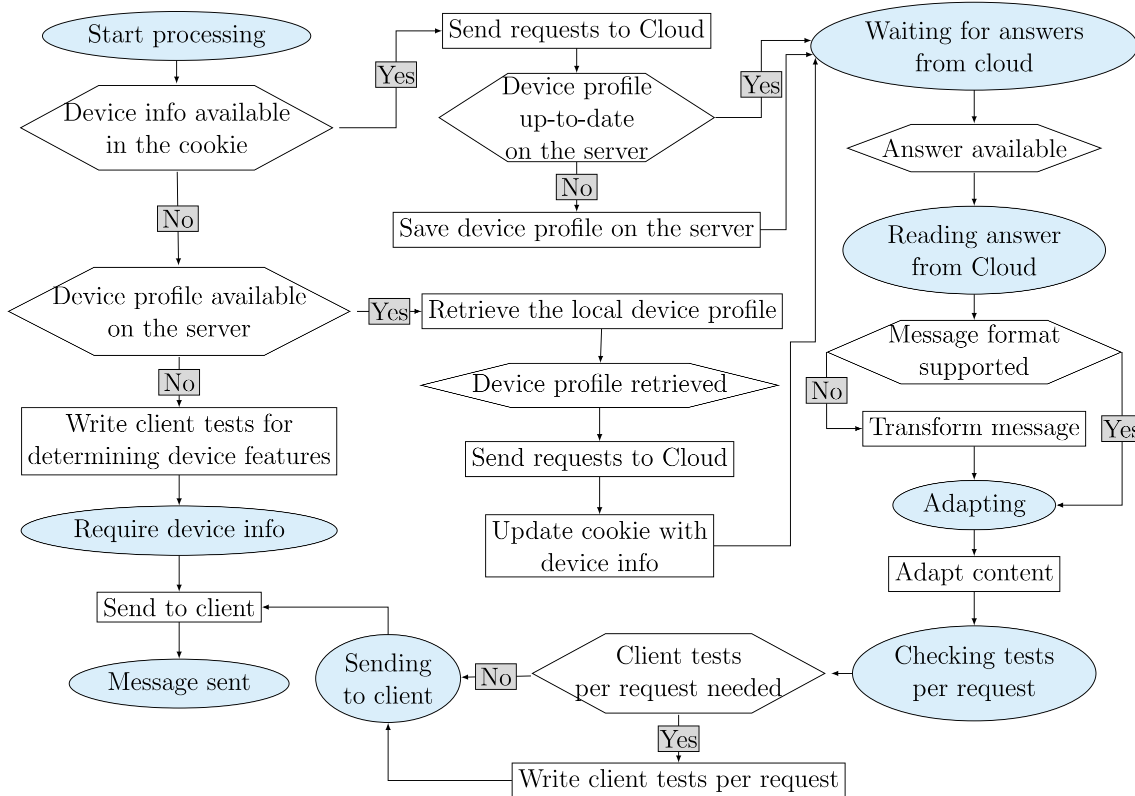

Fig. 9. Request handler—control state ASM ground model

Middleware server model On the server-side we have the specification of the *middleware* and of the *request handler*. Fig. 8 displays the (control state) ASM ground model of the middleware agent. Only one state is available for this agent: *Waiting for requests*—the middleware keeps on waiting for requests from the clients. Each device sends only one request at any given time, thus the middleware agent can deal with requests from different devices in parallel without difficulties, because requests are independent of each other. For each received request a new agent of type *request handler* is generated.

Note that we provide a basic infrastructure for a message-passing communication. When modeling the client-middleware communication, only reactive behaviors are considered, i.e., the middleware reacts upon receiving requests from the client. We abstract from considering possible "conversational" aspects (possibly implying several exchanges of information between the two parts) of the communication in distributed systems. However, such aspects can be included in further refinements of the model by using the high-level models for fundamental bilateral service interaction patterns specified by Barros and Börger [BB05] in terms of ASMs. They define ASM rules to capture the semantics of both asynchronous and synchronous message passing (the non-blocking and blocking mode) and the semantics of service interactions beyond simple request-response sequences by involving acknowledgment, resending, etc.

Figure 9 displays the model of a *request handler* agent. The initial state of a request handler is *Start processing*, followed by the guard that checks if the corresponding device profile is available in the cookie. If not available, the agent searches for the information in the local database. In case device information is not available neither in the cookie nor on the server, modernizr tests are created in JavaScript, the state of the request handler is set to *Require device info*, and the request containing the tests is returned to the client for executing the JavaScript code and updating the device information in the cookie.

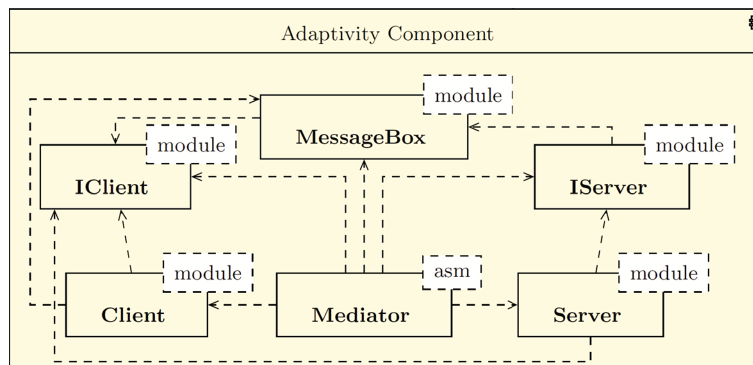

Fig. 10. The architecture of the client-middleware communication in AsmetaL

```
module IClient
signature:
    domain Device subsetof Agent
    controlled cookie: Prod(Device, String) −> String
    controlled modernizr: Device −>
    Seq(Prod(String, Boolean, Seq(Prod(String, Boolean))))
    static dev1: Device
    static dev2: Device
    static dev3: Device
```

Code 2. IClient module

After this step, the agent's state is updated to *Message sent*, which is also the final state of a request handler. Otherwise, if the information has been found, the request is forwarded to the Cloud, and its handler enters the state *Waiting for answers from cloud*. If the device profile has been retrieved from the cookie, it is updated on the server (if necessary), otherwise, if it has been retrieved from the server, the cookie is updated. For each answer that returns from the Cloud, the handler agent verifies whether the device accepts the format: If not, the content is transformed. Then, the agent goes to the next state *Adapting*. After the content is adapted using the device profile, the agent checks if "tests per request" are necessary. If so, the corresponding JavaScript tests are generated and the agent goes to the *Sending to client* state. The answer is sent to the client's device to be displayed and the handler agent reaches the final state *Message sent*.

AsmetaL models We here present (fragments of) the AsmetaL encoding of the control state ASMs presented before.[7]

The AsmetaL encoding of a multi-agent ASM requires the definition of a *main ASM* that imports all the ASM modules and, in its main rule, schedules all the other agents.

Figure 10 displays the architecture of the client-middleware communication in AsmetaL.

`IClient` and `Client` correspond to the *client* control state ASM depicted in Fig. 7. `IServer` and `Server` together represent the control state ASMs of the *middleware* and of the *request handler* shown in Figs. 8 and 9. `MessageBox` is a module containing signature to model message exchange between clients and middleware. `Mediator` is the main ASM of the multi-agent AsmetaL model. It is responsible for delivering messages among the other agents. In the figure, dashed arrows represent the *importing* relation among models: `A −−> B` means that `A` reads (part of) the signature of `B`. Such feature is particularly common in multi-agent ASMs, since one agent may need information from other agents.

`IClient` module (see Code 2) contains the signature of the client that must be accessed by other modules.

`Device` is the set of the client running agents, and static functions `dev1`, `dev2`, and `dev3` represent three concrete client agents; `cookie` and `modernizr` functions are declared in this module, because they are also accessed by the *middleware*. The module `Client` (see Code 3) formalizes, by means of the rule `r_ClientAction` (partially reported), the behavior of each client operating through a device. The translation from the graphical notation of the control state ASM to the textual notation is done according to the mapping shown in Fig. 2. Each action, reported with a rectangle in Fig. 7, becomes a rule in the AsmetaL code.

[7] All the specifications are available at http://fmse.di.unimi.it/sw/AdaptivityFAOC2016.zip.

```
module Client

signature:
    enum domain State = {IDLE | SEND_MSG |
            WAITING_FOR_MSG | EXEC_CLIENT_TESTS |
            CHECKING_FOR_EXTRA_RESOURCES |
            DISPLAYING_THE_MSG}
    controlled state: Device -> State
    controlled htmlTags: Device -> Prod(MessageType, Msg, Msg)
    monitored extraResources: Device -> Boolean
    monitored clientTestsAvl: Device -> Boolean
    derived messageArrived: Device -> Boolean

definitions:
    function messageArrived($d in Device) = ...

    rule r_DownloadExtraResources = skip //this rule remains abstract

    rule r_DecryptMessage = skip //this rule remains abstract

    rule r_updateCookieProfile =
            r_updateCookieWithModernizr[cookie(self,"deviceProfile"),
                                        modernizr(self)]
```

```
rule r_Send =
    par
        if isDef(htmlTags(self)) then
            outboxClient(self) := including(outboxClient(self),
                                  second(htmlTags(self)))
        else
            extend Msg with $m do
             outboxClient(self) := including(outboxClient(self), $m)
        endif
        htmlTags(self) := undef
    endpar
rule r_ClientAction = //Behavior of the devices of a client
    par
        ...
        if state(self) = SEND_MSG then
            par
                r_Send[]
                state(self) := WAITING_FOR_MSG
            endpar
        endif
        if state(self) = WAITING_FOR_MSG then
            ...
    endpar
```

Code 3. Client module

```
module IServer
signature:
    domain Middleware subsetof Agent
    dynamic domain ReqHandler subsetof Agent
    static middleware: Middleware
```

Code 4. IServer module

For example, rule `r_Send` describes the behavior of a client that uses the device to send a request to the Cloud (intercepted by the middleware). As one can observe, the request is not directly sent to the middleware: it is placed in a queue and processed afterwards by the `Mediator`. The client is, therefore, not aware of the processing done by the middleware.

A similarly modeling approach has been followed for the server-side. The functions which are needed in the other modules are in the signature of module `IServer` shown in Code 4. We have two declarations of agent domains on the server-side, because there are two different types of agents: `Middleware` and `ReqHandler`. The agents of type `ReqHandler` are generated at runtime (inside the `Server` module as shown in Code 5) by the unique `middleware` agent.

The module `Server` is partially shown in Code 5. Inside the signature, the enumerative domains `MiddlewareState` and `ReqHandlerState` represent the states of, respectively, the *middleware* and a *request handler*. Among the functions, `devRequests` represents the received requests, and `devProfileDB` the server database. Rule `r_ReceiveRequest` describes the behavior of the *middleware* agent as described in the control state ASM shown in Fig. 8: it generates a *request handler* agent for each new request, and awakes request handlers that did not reach their final state yet, so that they can further process their corresponding requests.

All the other rules of the module describe the actions executed by the *request handler* agents as described by the control state ASM in Fig. 9. Rule `r_ProcessRequest` is the principal rule of the request handler, that invokes all the other rules. For example, rule `r_Send` places the answer for the corresponding device in a queue, so that the `Mediator` can forward it to the client-side.

Code 6 displays the `Mediator`, the main ASM of our project. In the main rule, the programs of all the device (client) agents and of the middleware agent are executed in parallel. Moreover, rules `r_SendToServer` and `r_SendToClient` establish the connection between the client and the server by delivering messages: the messages sent by the devices (resp. by the request handlers) are taken from the `outboxClient` (resp. `outboxServer`), queued, and forwarded to the middleware (resp. devices). In the initial state section, the `Mediator` initializes the signature of all the modules and specifies the *programs* (i.e., the starting rules) of the agents.

We skip the presentation of the `MessageBox` module, that only contains signature including the `Msg` and `MessageType` domains, and the communication queues used by the agents.

```
module Server

signature:
    enum domain MiddlewareState = {WAITING_REQUESTS}
    enum domain ReqHandlerState ={START_PROCESSING|
        REQUIRE_DEVICE_INFO | WAITING_FROM_CLOUD |
        READING_ANSW_FROM_CLOUD | ADAPTING |
        CHECKING_CLIENT_TESTS_REQ |
        SENDING_TO_CLIENT | MSG_SENT}
    controlled mState: Middleware −> MiddlewareState
    controlled reqHandlerState: ReqHandler −>
                            ReqHandlerState
    controlled devProfileDB:
    Device −> Seq(Prod(String, Boolean,
                Seq(Prod(String, Boolean))))
    controlled devProfileDBRetrieved: Device −> Boolean
    controlled devRequests: Device −>
                            Powerset(Prod(Msg, ReqHandler, Msg))
    controlled reqHandlerDev: ReqHandler −>
                            Prod(Device, Msg, Msg)
    controlled clientTests: RequestHandler −> Msg
    derived requestsAvl: Boolean
    derived cookieInfoAvl: Device −> Boolean
    derived devProfileDBUpdated: Device −> Boolean
    derived existsDevProfileDB: Device −> Boolean
    derived cookieKeys: Prod(Device, String) −>
                            Seq(String)
    derived devProfileDBKeys: Device −> Seq(String)
    derived cookieSubprops: Prod(String, String, String) −>
                            Seq(Prod(String, Boolean))
    monitored answerAvl:
        Prod(Device, Msg, RequestHandler) −> Boolean
    monitored answerFormatSupp: Msg −> Boolean
    monitored clientTestsPerRequest: Msg −> Boolean

definitions:
    function requestsAvl = ...
    function cookieInfoAvl($d in Device) = ...
    function cookieKeys($d in Device, $s in String) = ...
    function devProfileDBKeys($d in Device) = ...
    function devProfileDBUpdated($d in Device) = ...
    function existsDevProfileDB($d in Device) = ...
    function cookieSubprops($s in String,
                    $d1 in String, $d2 in String) = ...

rule r_WriteClientTests ($rh in ReqHandler) =
rule r_SendRequestsToCloud($rh in ReqHandler, $r in Msg,
                                    $d in Device) = ...
rule r_UpdateDevProfileDB($d in Device) = ...
rule r_ReadDevProfileDB($d in Device) = ...
rule r_UpdateCookieProfile($d in Device) = ...
rule r_Send($d in Device, $mt in MessageType, $initialReq in Msg,
                                    $answer in Msg) =
    outboxServer($d) := ($mt, $initialReq, $answer)
rule r_ReplaceAnswer($rh in ReqHandler, $d in Device, $r in Msg,
                                    $nA in Msg)= ...
rule r_GetAnswerFromCloud($rh in ReqHandler, $r in Msg,
                                    $d in Device) = ...
rule r_TransformMessageFormat($rh in ReqHandler, $d in Device,
                                    $r in Msg) = ...
rule r_AdaptContent = skip //this rule remains abstract
rule r_UpdateModernizr($d in Device) = ...
rule r_WriteClientTestsPerRequest($rh in ReqHandler, $d in Device,
                                    $r in Msg) = ...
rule r_ProcessRequest = //Behavior of the request handler
    par
        if reqHandlerState(self) = START_PROCESSING then
            if cookieInfoAvl(first(reqHandlerDev(self))) then
                par
                if not devProfileDBUpdated(first(reqHandlerDev(self)) then
                        r_UpdateDevProfileDB[first(reqHandlerDev(self))]
                endif
                r_SendRequestsToCloud[self, second(reqHandlerDev(self)),
                            first(reqHandlerDev(self))]
                endpar
            else
                ...
    endpar

rule r_ReceiveRequest = //Behavior of the middleware
    ...
    forall $d in Device do
        choose $dr in deviceRequests($d) with true do
            if isUndef(second($dr)) then
                extend RequestHandler with $rh do
                    ...
            endif
    forall $reqH in ReqHandler
            with reqHandlerState($reqH) != MSG_SENT do
        program($reqH)
    ...
```

Code 5. Server module

```
asm Mediator

definitions:

    ...
    rule r_SendToServer =
        forall $d in Device with size(outboxClient($d)) > 0 do
            ...
            forall $m in outboxClient($d) do
                ...

    rule r_SendToClient =
        forall $d in Device with isDef(outboxServer($d)) do
        ...

    main rule r_Main =
        par
            forall $d in Device do
                program($d)
            program(middleware)
            r_SendToServer[]
            r_SendToClient[]
        endpar

default init s0:
    ...
    agent Device: r_ClientAction[]
    agent Middleware: r_ReceiveRequest[]
    agent RequestHandler: r_ProcessRequest[]
```

Code 6. Mediator module

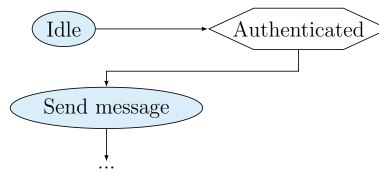

Fig. 11. Client—control state first refinement

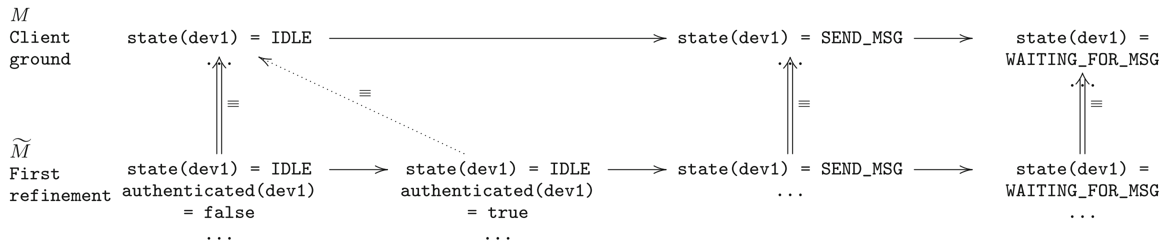

Fig. 12. First refinement—example of refined run

6.2. Refinement of the client

Starting from the ground models of the client and the middleware server, we proceeded through a chain of refinements to obtain more detailed models of the client and the middleware server. We followed the *vertical* refinement approach where each model is singularly refined. Each refinement step has been proved correct through the SMT-based tool `ASMRefProver`. In the following, we show, by means of the first two steps of refinement of the client, the two possible types of refinement: operation refinement and data refinement.

Our system guarantees the conditions of Theorem 2.4, i.e., that variables shared between agents are only written by an agent and read by the other one: indeed the client writes variables that are read by the server (that does not have write access on them) and the other way around. Therefore, according to Theorem 2.4, we can consider the agents individually and prove refinement for them; if refinement holds for the individual agents, it will also hold for the global system.

First refinement In the ground model of the client, in the initial state the device is in state `IDLE` and always moves to the state `SEND_MSG`. In the first refinement step, we have added an authentication step in which the device must login to the middleware service (see the modified control state ASM in Fig. 11). We use a monitored predicate `authenticated` for specifying whether the device has been authenticated. This refinement is an *operation refinement* since a new behavior (i.e., the authentication phase) has been added in the refined model.

The equivalence between abstract and refined states is given by the equality of the functions `state` (i.e, the locations of interests). The refinement is a correct stuttering refinement. In all the states in which the state of the client is different from `IDLE`, the proof is straightforward because the refined model behaves as the abstract model. We only need to prove that the correct refinement holds when the state of the client is `IDLE`: if in a refined state $\widetilde{S}$ a device is in `IDLE` and the state is equivalent with an abstract state S, in the next state $\widetilde{S}'$ the refined model can only be equivalent with either S (if the device failed to authenticate in the previous state) or with S', the next state of S (if the authentication was successful). Figure 12 shows an example of refined run $\widetilde{S}_0, \widetilde{S}_1, \widetilde{S}_2, \ldots$ and a corresponding abstract run $S_0, S_1, \ldots$. In the first state $\widetilde{S}_0$ of the refined run, `dev1` is in `IDLE` state and fails to authenticate (i.e., `authenticated(dev1)` is false): therefore, in the second state $\widetilde{S}_1$ it remains `IDLE`. In the second state, the device authenticates correctly and, therefore, in the third state $\widetilde{S}_2$ its state becomes `SEND_MSG`. Then the run continues with the message exchange. We can find a corresponding abstract run that respects the stuttering refinement relation: the first two states of the refined run are conformant with the first state S_0 of the abstract run in which `state(dev1) = IDLE`, and then each refined state $\widetilde{S}_i$ ($i \geq 2$) is conformant with the abstract state S_{i-1}.

Second refinement In this refinement we implement the updating of the cookie by the modernizr. This refinement is a *data refinement*. Indeed, it extends the signature with functions `cookie` and `modernizr`, and adds a rule for updating the cookie, but the transition relation of the previous model is not affected.

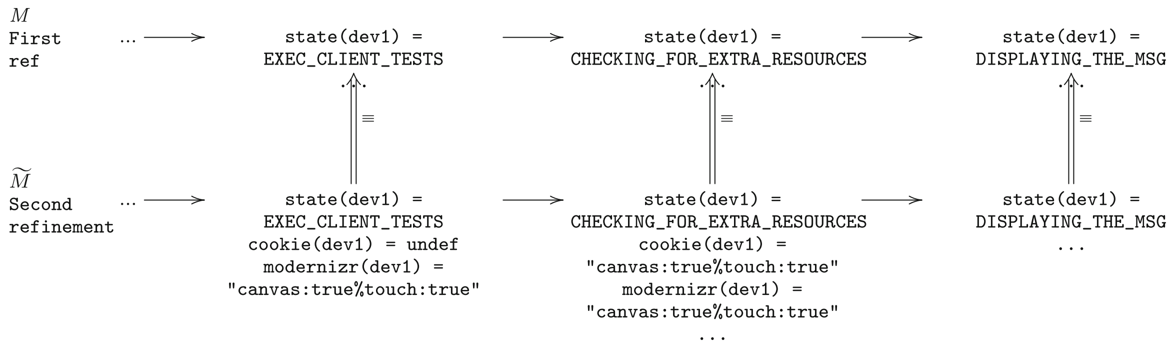

Fig. 13. Second refinement—example of refined run

Therefore, the proof of correct refinement is straightforward, since each run is associated with a run of equal length where each refined state $\widetilde{S}$ is equivalent with the abstract state S. Figure 13 shows an example of refined run.

7. Validation

Model validation is a first model analysis activity that is performed to ensure that the specification captures all system requirements, and owns some specific qualities. We exploited the simulator `AsmetaS` [GRS08a] and the validator `AsmetaV` [CGRS08] for model simulation in both an interactive and automatic way. Moreover, we used the tool `AsmetaMA` to verify *system-independent* properties (or *meta-properties*), i.e., properties that any model should guarantee.

7.1. Simulation

`AsmetaS` permits to perform either *interactive simulation*, where required inputs are provided interactively by the user during simulation, and *random simulation*, where input values are chosen randomly by the simulator itself. The simulator, at each step, performs *consistent updates checking* to check that all the updates are consistent: in an ASM, two updates are inconsistent if they update the same location to two different values at the same time [BS03]. Moreover, at each step the simulator also checks that all the invariants hold.

During the development process, we have repeatedly simulated our specification, both interactively and randomly. Figure 14 shows an extract of a simulation trace. In the second state, three devices have sent three messages to the middleware, putting them in `outboxClient`. In the following, we only describe the workflow for the request `Msg!3` of device `dev1`, but all the requests have been handled in parallel during the simulation. In the fourth state, the `middleware` has created the request handler `ReqHandler!3` for the request of `dev1` and has associated it to the corresponding message `Msg!3` by means of function `reqHandlerDev`; the state of the handler (i.e., function `reqHandlerState`) is set to `START_PROCESSING`. In state 6, the middleware has updated the `cookie` of `dev1`, because the information is available in the database (`devProfileDB`); the state of the corresponding request handler has been updated to `WAITING_FROM_CLOUD`, meaning that the request has been forwarded to the Cloud. In the following state, through the setting of monitored location `answerAvl(dev1,Msg!3,ReqHandler!3)` to true, we model the fact that an answer came back from the Cloud for the message `Msg!3`. In state 8, the middleware reads the answer coming from the Cloud (`Msg!6`) and inserts it in `devRequests` for `dev1` and in `reqHandlerDev` (a function recording the request of each request handler) for `ReqHandler!3`. In state 9, by means of the monitored function `answerFormatSupp`, the middleware checks whether the message `Msg!6` returned from the Cloud for `dev1` has the correct format. In this case the format is correct; if the format had been not correct, a new message would have been generated. In the next state, `ReqHandler!3` enters state `ADAPTING`, i.e., it adapts the message coming from the Cloud with the information contained in the cookie. In state 13, `ReqHandler!3` checks whether client tests are necessary for handling the current request (e.g., the GPS device must be tested); this is done by means of function `clientTestsPerRequest`. In this case, tests are necessary. Therefore, in the next state a new message is created (`Msg!7`) for `dev1` and the `modernizr(dev1)` location is updated with the new test.

```
...
<State 2 (controlled)>
devRequests(dev1)={}
devRequests(dev2)={}
devRequests(dev3)={}
outboxClient(dev1)={Msg!3}
outboxClient(dev2)={Msg!2}
outboxClient(dev3)={Msg!1}
state(dev1)=WAITING_FOR_MSG
state(dev2)=WAITING_FOR_MSG
state(dev3)=WAITING_FOR_MSG
mState(middleware)=WAITING_REQUESTS
</State 2 (controlled)>
...
<State 4 (controlled)>
outboxClient(dev1)={}
state(dev1)=WAITING_FOR_MSG
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,undef)
reqHandlerState(ReqHandler!3)=START_PROCESSING
devRequests(dev1)={(Msg!3,ReqHandler!3,undef)}
outboxServer(dev1)=undef
...
</State 4 (controlled)>
...
<State 6 (controlled)>
state(dev1)=WAITING_FOR_MSG
cookie(dev1,"deviceProfile")="canvas:true%touch:true"
devProfileDB(dev1)=[("canvas",true,undef), ...]
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,undef)
reqHandlerState(ReqHandler!3)=WAITING_FROM_CLOUD
..
</State 6 (controlled)>
...
Insert a boolean constant for
                 answerAvl(dev1,Msg!3,ReqHandler!3):
true
<State 7 (monitored)>
answerAvl(dev1,Msg!3,ReqHandler!3)=true
</State 7 (monitored)>
<State 8 (controlled)>
outboxServer(dev1)=undef
devRequests(dev1)={(Msg!3,ReqHandler!3,Msg!6)}
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!6)
reqHandlerState(ReqHandler!3)=READING_ANSW_FROM_CLOUD
...
</State 8 (controlled)>
...
Insert a boolean constant for answerFormatSupp(Msg!6):
true
<State 9 (monitored)>
answerFormatSupp(Msg!6)=true
</State 9 (monitored)>
<State 10 (controlled)>
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!6)
reqHandlerState(ReqHandler!3)=ADAPTING
state(dev1)=WAITING_FOR_MSG
...
</State 10 (controlled)>
...
```

```
<State 12 (controlled)>
state(dev1)=WAITING_FOR_MSG
cookie(dev1,"deviceProfile")="canvas:true%touch:true"
devProfileDB(dev1)=[("canvas",true,undef), ...]
devRequests(dev1)={(Msg!3,ReqHandler!3,Msg!6)}
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!6)
reqHandlerState(ReqHandler!3)=CHECKING_CLIENT_TESTS_REQ
...
</State 12 (controlled)>
...
Insert a boolean constant for clientTestsPerRequest(Msg!6):
true
<State 13 (monitored)>
clientTestsPerRequest(Msg!6) = true
</State 13 (monitored)>
<State 14 (controlled)>
state(dev1)=WAITING_FOR_MSG
modernizr(dev1)=[("geolocation",true,undef)]
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!7)
reqHandlerState(ReqHandler!3)=SENDING_TO_CLIENT
...
</State 14 (controlled)>
...
<State 16 (controlled)>
outboxServer(dev1)=(DISPLAY,Msg!3,Msg!7)
devProfileDB(dev1)=[("canvas",true,undef), ...]
reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!7)
reqHandlerState(ReqHandler!3)=MSG_SENT
...
</State 16 (controlled)>
<State 17 (controlled)>
htmlTags(dev1)=(DISPLAY,Msg!3,Msg!7)
...
</State 17 (controlled)>
...
Insert a boolean constant for clientTestsAvl(dev1):
true
<State 17 (monitored)>
...
<State 18 (controlled)>
state(dev1)=EXEC_CLIENT_TESTS
...
</State 18 (controlled)>
<State 20 (controlled)>
state(dev1)=CHECKING_FOR_EXTRA_RESOURCES
...
</State 20 (controlled)>
</State 21 (controlled)>
...
Insert a boolean constant for extraResources(dev1):
false
<State 21 (monitored)>
...
<State 22 (controlled)>
state(dev1)=DISPLAYING_THE_MSG
...
</State 22 (controlled)>
...
```

Fig. 14. Example of simulation trace

In state 16, message `Msg!7` for `dev1` has been sent by adding it to `outboxServer` and the state of `ReqHandler!3` is updated to `MSG_SENT`. In the next state, message `Msg!7` is stored on `dev1` (in the function `htmlTags`). The function `clientTestsAvl` checks now if Javascript tests are available in the answer. `dev1` executes the test in state 18, checks for extra resources in state 21, and can finally display the message in state 22.

7.2. Scenario-based validation

Although simulation is useful in the first stages of the model development, when the model becomes particularly big, following a long simulation can be a tedious task for the developer. Scenario-based validation by the tool `AsmetaV` [CGRS08] permits to specify *scenarios* describing the interaction between a user (i.e., the environment)

and the machine. The `Avalla` language provides constructs to **set** the values of the monitored functions, to execute a **step** of simulation of the ASM, and to **check** that a given closed first order formula (*assertion*) holds in a given state. The validator `AsmetaV` simulates (using the simulator `AsmetaS`) the ASM model according to the commands of the scenario, and checks if all the assertions are satisfied. As soon as an assertion is not satisfied, the simulation is interrupted reporting the violation.

We have produced some scenarios of interaction sequences with suitable checks describing our expectations about the model states (similarly to what is done with unit testing in code development). Moreover, such scenarios have been executed every time we modified and/or enhanced our models to check that no faults were introduced (in a kind of regression testing).

Code 7 shows the scenario reproducing the simulation reported in Fig. 14.

7.3. Model review

This approach aims at determining if a model is of sufficient *quality* to be easy to develop, maintain, and enhance. This technique permits to identify defects early in the system development, reducing the cost of fixing them. For this reason, it should be applied also on preliminary models. The `AsmetaMA` tool [AGR10b] (based on the model checker `AsmetaSMV`) allows *automatic* review of ASMs. Typical vulnerabilities and defects that can be introduced during the modeling activity using ASMs are checked as violations of suitable *meta-properties* (MP, defined in [AGR10b] as CTL formulae). The violation of a meta-property means that a quality attribute is not guaranteed, and it may indicate the presence of a real fault (i.e., the ASM is indeed faulty), or only of a *stylistic defect* (i.e., the ASM could be written in a better way).

For this work, we have identified some meta-properties tailored for distributed systems:

- MP_{nc}: *no concurrent* writing errors occur in the specification. This particular concurrency error occurs when the same element is *simultaneously* modified by different agents. Such error corresponds to the ASM notion of *inconsistent update* when the same location is updated to two different values at the same time [BS03]. In our case, we may have inconsistencies in the communication, for example, if the outbox of the middleware is simultaneously updated with multiple replies. Another inconsistency may occur in the database handling, for example, if multiple request handlers can update the information of a given device at the same time. This meta-property guarantees consistency since it checks that no location is simultaneously updated by different agents. Note that a violation of this meta-property indicates a real fault in the model.
- MP_e: all specified behaviors are *executed*. Such meta-property guarantees that all the rules specified in the model are executed, i.e., all the agents are active and all their configurations (as shown in the control state ASMs in Figs. 7, 8, and 9) are actually reachable. Note that this meta-property only guarantees the rechability of a rule, not its correctness, that can only be checked with application-dependent properties specified by the user, as described in Sect. 8.
- MP_m: the signature is *minimal*, i.e., it does not contain locations that are *unnecessary* (they are never read nor updated) or that do not assume all the values of their codomains. A violation of this meta-property may indicate over-specification, i.e., the model contains signature elements that are not needed, or that the model is not complete, i.e., that the designer forgot to add some behaviors (i.e., transition rules) to modify such locations. Note that removing unnecessary elements at model level is mandatory because if they are kept in the final implementation, they can negatively influence the performances of the whole system and increase the costs without any motivation.

Meta-property verification works as follows. A meta-property MP is instantiated on the model under review producing a set of CTL properties which are proved over the model. If no violation is found, the meta-property holds. For example, meta-property MP_e requiring that each rule R is executed, is defined as $\mathbf{EF}(firingCondition(R))$, where $firingCondition(R)$ specifies the condition that guards the execution of R. Verification of MP_e is done by instantiating the meta-property on all the rules of the model.

During the development process, we have executed `AsmetaMA` on all the models. We have found several minimality violations (meta-property MP_m) in all the models, since some functions were declared but never used: we discovered that some of these functions were indeed unuseful (and so they could be removed), while some others were useful, but we forgot to use (read or update) them.

```
scenario communication

load Mediator.asm

step
step
check outboxClient(dev1)={Msg!3} and state(dev1)=WAITING_FOR_MSG;
...
step
check devRequests(dev1)={(Msg!3,ReqHandler!3,undef)} and reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,undef) and
reqHandlerState(ReqHandler!3)=START_PROCESSING;

...
step
check cookie(dev1,"deviceProfile")="canvas:true

set answerAvl(dev1,Msg!3,ReqHandler!3): = true;
step
check devRequests(dev1)={(Msg!3,ReqHandler!3,Msg!6)} and reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!6) and
reqHandlerState(ReqHandler!3)=READING_ANSW_FROM_CLOUD;

set answerFormatSupp(Msg!6) = true;
step
check reqHandlerState(ReqHandler!3)=ADAPTING;

...
step
check reqHandlerState(ReqHandler!3)=CHECKING_CLIENT_TESTS_REQ;

set clientTestsPerRequest(Msg!6) := true;
step
check modernizr(dev1)=[("geolocation",true,undef)] and reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!7) and
reqHandlerState(ReqHandler!3)=SENDING_TO_CLIENT;

...
step
check outboxServer(dev1)=(DISPLAY,Msg!3,Msg!7) and reqHandlerDev(ReqHandler!3)=(dev1,Msg!3,Msg!7) and
reqHandlerState(ReqHandler!3)=MSG_SENT;

step
check htmlTags(dev1)=(DISPLAY,Msg!3,Msg!7);

set clientTestsAvl(dev1) := true;
step
check state(dev1)=EXEC_CLIENT_TESTS;

...
step
check state(dev1)=CHECKING_FOR_EXTRA_RESOURCES;

set extraResources(dev1) := false;
step
check state(dev1)=DISPLAYING_THE_MSG;
```

Code 7. Example of scenario

A more serious error that we discovered in our first specification (originally published in [Che13]) was the presence of a consistency violation (meta-property MP_{nc}). We found that the client could, under some conditions, simultaneously update a location of the function `state` (`ctl_state` in [Che13]) to two different values. Although a normal simulation or the scenario-based validation can sometimes unveil the presence of inconsistent updates, when the model becomes particularly complex, inconsistencies may be more difficult to find, and an automatic approach as that provided by the model reviewer is helpful. Moreover, simulation can show only *some* inconsistencies (i.e., those detected in the executed runs), whereas model review detects *all* the inconsistencies.

```
forall $k in Key do
    par
        cookie(self, $k) := modernizr(self, $k)
        forall $c in SubKey with keyParentalRel($k, $c) do
            cookieSub(self, $k, $c) := modernizrSub(self, $k, $c)
    endpar
```

Code 8. Rule `r_updateCookieProfile`

We also checked more general meta-properties defined in [AGR10b]. In the second refinement of the client model, we have found several violations of meta-property *MP4*, requiring that *no assignment is always trivial* [Gur00]: an update rule $l := t$ is always trivial if, when the rule is applied, l is already equal to t. In our specification we always update all the keys of a cookie through the modernizer, even if they are already up to date, as shown in Code 8.

In this way, the locations of function *cookie* that refer to keys that never change, will be always updated to the same value. Although this is not a real error, it gave us a more deep understanding of the behavior of our specification. In a further refinement of our model, we could avoid to update keys that are already up to date; surely such a control should be done in the final implementation, in order to improve the performances.

In a preliminary version of the middleware server, we also discover a violation of meta-property *MP7*, requiring that *every controlled location is updated* at least once (otherwise it should be defined static). We discovered that controlled function `devProfileDB`, representing the database hosted on the middleware, was never updated, i.e., the middleware never copied in the database the information retrieved from the cookie. In this case, the meta-property violation was the signal of a requirement that was not modeled.

8. Verification

Once the designer has enough confidence that the specification captures all the intended requirements, (s)he can apply heavier techniques, as model checking, to guarantee the specification correctness.

We used `AsmetaSMV` [AGR10a], a tool that translates ASM specifications into models of the NuSMV model checker, and allows the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulae. As underlined also in [Leu08, ADK08, Hei98], declaring a property for a high level-model of the system is definitely easier than writing the same property for a low-level model, as the one we would obtain directly using the syntax provided by model checkers (e.g., Promela, the input language of SPIN, or the input syntax of NuSMV).

Since model checking requires a finite number of states to verify, we have slightly modified our models in order to make them suitable for model checking. We have abstracted all the infinite domains with finite ones and we have considered a finite number of interactions between the client and the middleware. For example, we have modified the signature of functions `cookie` and `modernizr` of the client model as follows:

```
controlled cookie: Prod(Device, Key) -> Boolean
controlled modernizr: Prod(Device, Key) -> Boolean
```

being `Key` an enumerative domain representing the possible keys of a cookie. `cookie(d,k)` is true if the device `d ∈ Device` has the key `k ∈ Key` in its cookie; `modernizr(d,k)` is true if the modernizr associates the key `k` to the device `d`. Two other functions record the sub-keys of the cookie keys.

```
controlled cookieSub: Prod(Device, Key, SubKey) -> Boolean
controlled modernizrSub: Prod(Device, Key, SubKey) -> Boolean
```

Such abstractions have been devised in a way to guarantee that the abstract model bisimulates the original model [BK08], and therefore properties proved over the abstract model also hold in the original model. In ASM terms, each run of the original model can be mapped to a run of equal length of the abstract model and vice versa.

The same abstractions have also been applied when we executed model review (see Sect. 7.3), since it is based on model checking.

We have verified classical temporal properties to guarantee correctness and reliability of the client-Cloud adaptivity application. Differently from [HRWW16], we have verified properties not only for the client and for the middleware, but also for their communication. In order to keep the verification times reasonable, we have first verified the communication, abstracting from the client and the server (mimicking their behaviors by suitable monitored functions). Then, we have verified the client and the middleware singularly, mimicking the communication with the other actors.

Each property has been specified on the model containing the elements involved in the property; for example, properties related to cookies were specified starting from the client model in which cookies were added. Moreover, all the properties specified at a given refinement level have also been reproved on all the successive refinement levels. Indeed, as explained in Sect. 2.2, stuttering refinement only guarantees that invariant properties are preserved by refined machines. As future work, we plan to investigate which temporal properties are preserved by stuttering refinement, so to avoid reproving all temporal properties on the refined models. This is usually guaranteed by stuttering bisimulation [MPMO10]; we will therefore study whether—or under which assumptions—stuttering bisimulation holds in our context.

All the properties have been specified in CTL. We verified all the properties on a Linux machine, Intel(R) Core(TM) i7 CPU @ 2.67 GHz, 4 GB RAM.

8.1. Communication

In order to verify the communication, we consider one device (`dev1`) sending one request to the middleware which has only one request handler (`RQ1`). In this way, we are able to identify the request handler which is used to handle the request.

We verify that the communication flow (as specified in Sect. 5.1) is correct by means of the following set of properties. We link each property with the requirement that it is supposed to (partially) verify.

Second requirement implies that whenever a device sends a request, the request handler will eventually start handling it.

ag(state(dev1) = WAITING_FOR_MSG implies ef(reqHandlerState(RQ1) = START_PROCESSING))

As specified by the third requirement, when the request handler starts handling the request, it can eventually send the request to the cloud.

ag(reqHandlerState(RQ1) = START_PROCESSING implies
ef(reqHandlerState(RQ1) = WAITING_FROM_CLOUD))

Note that the request is not sent to the Cloud when there is no information regarding the device, neither in the cookie nor in the database (this is why we use **ef** and not **af** in the consequent of the implication).

If the request handler is waiting for an answer from the cloud, it can eventually receive it and start adapting the content (as specified by fourth requirement).

ag(reqHandlerState(RQ1) = WAITING_FROM_CLOUD implies ef(reqHandlerState(RQ1) = ADAPTING))

Note that the cloud could not reply for different reasons (e.g., network problems).

As specified by the seventh requirement, the request handler, after having adapted the message, sends it back to the client.

ag(reqHandlerState(RQ1) = ADAPTING implies af(reqHandlerState(RQ1) = SENDING_TO_CLIENT))

Finally, the message sent by the server is eventually received by the client that can display it (eighth requirement).

ag(reqHandlerState(RQ1) = SENDING_TO_CLIENT implies
af(reqHandlerState(RQ1) = MSG_SENT and state(dev1) = DISPLAYING_THE_MSG))

8.2. Client

With the following properties we check that the clients behave correctly. In this case, we consider multiple clients running in parallel and we abstract from the communication framework and the middleware. Note that some properties partially overlap with properties already verified for the communication framework.

We first verify that each device can actually receive a message to display[8] (see eighth requirement).

(**forall** $d **in** Device **with** ef(state($d) = DISPLAYING_THE_MSG))

Then, through a set of properties, we check that the client correctly handles the test, as specified by the sixth requirement. First, we check that, if a message containing a test arrives and the device is waiting for the message, then the device surely (in the next state) executes the test.

(**forall** $d **in** Device **with** ag((state($d) = WAITING_FOR_MSG and msgArrived($d) and clientTestsAvl($d))
implies ax(state($d) = EXEC_CLIENT_TESTS)))

Second, we verify that a test is executed correctly: if a device executes a test, then afterwards it has some information in its cookie (i.e., at least a key of its cookie is defined).

(**forall** $d **in** Device **with** ag((state($d) = EXEC_CLIENT_TESTS) implies
(**exists** $k **in** Key **with** ax(cookie($d, $k) != undef))))

The previous property guarantees that the execution of a test provides some information to the middleware. However, the property does not check that the information copied in the cookie is indeed correct. Therefore, we check that during a test the cookie is updated with the information contained in the modernizr.

(**forall** $d **in** Device **with** ag(state($d) = EXEC_CLIENT_TESTS implies
(**forall** $k **in** Key **with** ax(cookie($d, $k) = modernizr($d, $k)))))

8.3. Middleware

The signature of the middleware contains functions `cookie` and `cookieSub` as declared for the client; moreover, it contains functions `devProfileDB` and `devProfileDBsub`,[9] representing the server database that stores the device profiles.

With the following properties we verify that the middleware behaves correctly.

First, we check that every request received from the client is eventually considered (it is either sent to the cloud or returned to the client to have more info), as specified by the second requirement.

(**forall** $r **in** ReqHandler **with** ag(reqHandlerState($r) = START_PROCESSING implies
af(reqHandlerState($r) = WAITING_FROM_CLOUD or reqHandlerState($r) = REQUIRE_DEVICE_INFO)))

The middleware has a database for memorizing the devices profiles extracted from the cookies. The following three properties check that the memorization mechanism behaves correctly (according to the sixth requirement). First of all, we check that, if a key is present in a cookie, it will be eventually present in the database as well.

(**forall** $d **in** Device, $k **in** Key **with** ag(cookie($d, $k) != undef implies ef(devProfileDB($d, $k) != undef)))

The previous property does not check that the value stored in the database is correct. So, we prove that the key value of a cookie (top-level key or sub-key) is eventually copied in the database.

(**forall** $d **in** Device, $k **in** Key **with**
ag(cookie($d, $k) != undef implies ef(devProfileDB($d, $k) = cookie($d, $k))))
(**forall** $d **in** Device, $k **in** Key, $c **in** SubKey **with**
ag(cookieSub($d, $k, $c) != undef implies ef(devProfileDBsub($d, $k, $c) = cookieSub($d, $k, $c))))

The device configurations stored in the database can also be used to update a cookie if this does not report any information about the device (i.e., the corresponding location is *undef*). So, the following properties check that a device information stored in the database (as top-level key or sub-key) is eventually copied in the undefined cookie location.

[8] Note that we have actually checked a slightly different property, because in NuSMV (the model checker used by `AsmetaSMV`) a CTL formula holds if it holds in all initial states. More information can be found in the NuSMV FAQ http://nusmv.fbk.eu/faq.html#007.

[9] **controlled** devProfileDB: Prod(Device, Key) −> Boolean
controlled devProfileDBsub: Prod(Device, Key, SubKey) −> Boolean

```
(forall $d in Device, $k in Key with ag((devProfileDB($d, $k) != undef and cookie($d, $k) = undef) implies
                                              ef(cookie($d, $k) = devProfileDB($d, $k))))
(forall $d in Device, $k in Key, $c in SubKey with ag((devProfileDBsub($d, $k, $c) != undef and
        cookieSub($d, $k, $c) = undef) implies ef(cookieSub($d, $k, $c) = devProfileDBsub($d, $k, $c))))
```

In the middleware model, we assume that the information about a device does not change. So, the following properties check that, once a cookie gets a value for one of its keys (top-level or sub-key), it cannot change it.

```
(forall $d in Device, $k in Key, $b in Boolean with ag(cookie($d, $k) = $b implies ag(cookie($d, $k) = $b)) )
(forall $d in Device, $k in Key, $c in SubKey, $b in Boolean with
                                  ag(cookieSub($d, $k, $c) = $b implies ag(cookieSub($d, $k, $c) = $b)) )
```

9. Related work

Our work presents a Web solution for the *Content Adaptation* of Cloud services to different end-devices topic and its formalization. Therefore, we position our research in the area of content adaptation of Web application and their formal validation and verification.

Content Adaptation is of major interest in the context of mobile devices. [Cre11] shortly explains the different mobile Web content adaptation techniques. Each technique has its own advantages and disadvantages. We think that the *hybrid approach*—retrieving the information regarding the device on the client-side, but processing the content on server-side—is the technique that better suits our project. A similar approach is presented in [RR11] that uses *device detection databases*. However, differently from [RR11], we prefer to only use the *modernizr* framework.

The analysis and verification of Web applications is not a novel idea, because in the past years several papers regarding this topic appeared, but not many of them are based on formal approaches. The research literature splits into two groups: on one side there are papers proposing forward engineering methods (starting by specifying the requirements, then going to the design phase and from this building the Web application), and on the other side there are papers using reverse engineering methods to extract, from an existing Web application, the corresponding models, and afterwards verify the models. There are also several analysis methods [ACD09] that are used, like modeling the navigational aspects of Web applications, or modeling the behavior and the features of Web applications, or modeling, validation and verification of the completeness and correctness of Web pages. The survey in [ACD09] presents the desirable properties for Web application modeling, and compares and categorizes some existing modeling methods based on the level of Web application modeling.

Several research papers are using reverse engineering methods. In [HPS04], a communicating finite automata model is generated from a recorded browsing session. The Spin model checker is used to verify the user-defined properties based on the obtained model. The authors used the black-box approach by executing the Web application under test (WAUT) and observed the application's behavior using a proxy server, that was intercepting the HTTP requests and responses. In the approach presented in [HPBS13], a Web application is monitored while it is explored by a user or a program and traces are collected: in this way the Web application behavior is modeled. The difference with the work in [HPS04] is that in this work the authors are extending the Linear Temporal Logic (LTL), by defining specialized operators using scopes. They do this in order to cover the problem of property specification in LTL over a subset of the states. The system presented in [BGV06] verifies if correctness and completeness properties are fulfilled by a Web site. The system uses a rule-based language for the specification and the verification of syntactic and semantic properties of collections of XML/XHTML documents. Using this method, one can only detect problems in the content of a Web page, but no model, that could be used for the improvement of the Web site, is generated. The use of validation and verification in the design phase helps to ensure the Web application reliability.

Other works apply forward engineering methods in order to check if the Web application satisfies the requirements. Unified Modeling Language (UML) is used in [SDM+05, MsYhSbJ10] to build the navigation model which would then be verified using SMV, respectively NuSMV. Since UML cannot be directly used for automated verification, the navigation model is defined as a Kripke structure. In [SDM+05], a parser of the XML Metadata Interchange (XMI) output of the UML is used to generate the SMV model. In [MsYhSbJ10], the description of the Web application is completed by incorporating the session mechanism into the navigation model. Another proposal to Web application navigation model is presented in [HIA+10], where the model is represented by using two finite-state automata, a page automaton and an internal state automaton, and then expressed using Promela, the input language for Spin model checker. A drawback would be that they do not use a tool to automatically transform the

models into Promela. A non-formal model of the presentation layer of a Web application is presented in [OW10] with the aim of testing the application; however, the model does not support asynchronous server-client interactions and concurrency, and it is based on a static technique. A correct-by-construction design process for secure and reliable distributed systems (in particular, Cloud systems) is proposed in [EMMW15]; it is based on process algebra and rewriting logic. The paper presents the notion of stuttering bisimilarity among different specifications and how this notion preserves all $CTL^*\backslash X$ formulas: we plan to investigate whether (or under which assumptions) this notion also applies to our notion of refinement.

Many successful applications exist in literature regarding the use of the ASMs for complex system modeling and analysis. Due to their multiplicity, we prefer to refer to [BS03] for a complete introduction on the ASM method and the presentation of the great variety of its successful applications.

10. Discussions and conclusions

The paper presents the formal design of a client-Cloud adaptivity application devised for making Cloud services available to different devices having different profiles [SBL+11]. The framework consists of a middleware server that intercepts requests from the clients, forwards them to the Cloud, and adapts the answers coming from the Cloud on the base of the profiles of the clients' devices.

We have modeled the client, the middleware server, and the complete communication cycle (mediated by the middleware) between the client and the Cloud. The resulting model is a multi-agent ASM. Each ASM has been obtained through a chain of refinements, starting from a high level model to more detailed ones. The use of the refinement approach helped us to manage the complexity of the distributed system. The particular kind of refinement we consider (i.e., stuttering refinement) allowed us to automatically prove each refinement step.

The ASMs method provided us a mathematical founded, yet easy to use, notation for reasoning on the requirements of the system and developing a correct specification. Thanks to the modular nature of ASMs, we have been able to reason both on the single components and on their communication.

Different validation activities have been performed along all the model development, using an iterative process between model specification and model validation. We found particularly useful the application of model review for checking application-independent properties tailored for distributed systems. Indeed, thanks to this technique, we were able to discover real errors in our models (inconsistencies); moreover, other meta-properties violations, although were not real faults, allowed us to find some weaknesses of the models (e.g., controlled locations that were never updated).

Once we gained enough confidence that the specification captured the intended requirements, we verified correctness properties through model checking. The ASM-based design approach facilitated this activity, since property specification can be done at the same level of system specification. Indeed, using the ASMETA framework, the user does not need to worry about translating the ASM specification into the language of the model checker. The mapping from an ASM model into a NuSMV model is automatic and the temporal properties can be directly expressed as part of the ASM model itself. Although some limitations exist on the class of ASMs that can be model checked (e.g., only finite domains), the alternative would be to encode the system under development directly in the model checker syntax, arising two problems: i) model checkers syntaxes usually have a low expressive power and it may be difficult to model complex systems with them, ii) we could produce a model not equivalent with the ASM specification.

We do not have an implementation of the designed client-Cloud adaptivity component yet. The long term goal of the project presented in [SBL+11]—which the current work is part of—is to develop, in a controlled and verified way, through a chain of refinements, the implementation of a prototype. When the implementation will be available, we could check its conformance w.r.t. the specification using model-based testing and runtime verification approaches provided by the ASMETA framework.

Acknowledgements

The research reported in this paper has been partly supported by the Charles University research funds PRVOUK and by the Christian Doppler Society.

References

[ABG+15] Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Formal validation and verification of a medical software critical component. In: 13th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2015). IEEE (2015)

[Abr96] Abrial, J.-R.: The B book: Deriving Programs from Meaning. Cambridge University Press, Cambridge (1996)

[ACD09] Alalfi, M.H., Cordy, J.R., Dean, T.R.: Modelling methods for web application verification and testing: State of the art. Softw. Test. Verif. Reliab. **19**(4), 265–296 (2009)

[ADK08] Arvind, N.D., Michael, K.: Getting formal verification into design flow. In: Jorge, C., Tom, M., Kaisa, S. (eds.) FM 2008: Formal Methods, vol. 5014 of Lecture Notes in Computer Science, pp. 12–32. Springer, Berlin Heidelberg (2008)

[AGR10a] 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), vol. 5977 of Lecture Notes in Computer Science, pp. 61–74. Springer, Berlin (2010)

[AGR10b] Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of abstract state machines by meta property verification. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), pp. 4–13. NASA (2010)

[AGR12] Arcaini, P., Gargantini, A., Riccobene, E.: CoMA: conformance monitoring of Java programs by Abstract State Machines. In: Sarfraz, K., Koushik, S. (eds.) Runtime Verification, vol. 7186 of Lecture Notes in Computer Science, pp. 223–238. Springer, Berlin (2012)

[AGR15] Arcaini, P., Gargantini, A., Riccobene, E.: Rigorous development process of a safety-critical system: from ASM models to Java code. Int. J. Softw. Tools Technol. Transf. 1–23 (2015)

[AGRS11] Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw. Pract. Exp. **41**, 155–166 (2011)

[ASRW02] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J.: Agile Software Development Methods: Review and Analysis. Technical Report 478. VTT PUBLICATIONS (2002)

[BB05] Barros, A., Börger, E.: A compositional framework for service interaction patterns and interaction flows. In: Lau, K.-K., Banach, R. (eds.) Formal Methods and Software Engineering, vol. 3785, Lecture Notes in Computer Science, pp. 5–35. Springer, Berlin Heidelberg (2005)

[BGV06] Ballis, D., García-Vivó, J.: A rule-based system for web site verification. Electron. Notes Theor. Comput. Sci. **157**(2), 11–17 (2006)

[BHV15] Bósa, K., Holom, R.-M., Vleju, M.B.: A formal model of client-cloud interaction. In: Thalheim, B., Schewe, K-D., Prinz, A., Buchberger, B. (eds.) Correct Software in Web Applications and Web Services, Texts and Monographs in Symbolic Computation, pp. 83–144. Springer International Publishing, New York (2015)

[BK08] Baier, C., Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press, Cambridge (2008)

[Bör03] Börger, E.: The ASM refinement method. Formal Aspect. Comput. **15**, 237–257 (2003)

[Bör05] Börger, E.: The ASM method for system design and analysis. A tutorial introduction. In: Gramlich, B. (ed.) Proceedings of Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, Sep 19–21, 2005, vol. 3717 of Lecture Notes in Computer Science, pp. 264–283. Springer, New York (2005)

[Bör07] Börger, E.: Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. Formal Aspect. Comput. **19**, 225–241 (2007)

[Bós12] 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, Austria (2012)

[Bós13] Bósa, K.: An ambient ASM model for client-to-client interaction via cloud computing. In: José, C., Marca, D.A., van Sinderen, M. (eds.) ICSOFT 2013 Proceedings of the 8th International Joint Conference on Software Technologies, Reykjavík, Iceland, 29–31 July, pp. 459–470. SciTePress, Portugal (2013)

[BPP99] Back, R.-J., Petre, L., Paltor, I.P.: Analysing UML use cases as contracts. In: Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard, UML'99, pp. 518–533. Springer-Verlag, Berlin, Heidelberg (1999)

[BS03] Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag, Berlin (2003)

[BZSW14] Banach, R., Zhu, H., Su, W., Wu, X.: ASM, controller synthesis, and complete refinement. Sci. Comput. Program. **94**(2), 109–129 (2014)

[CGRS08] Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Proceedings of the 1st International Conference on Abstract State Machines, B and Z (ABZ 2008), vol. 5238 of Lecture Notes in Computer Science, pp. 71–84. Springer-Verlag, Berlin (2008)

[Che13] Chelemen, R.-M.: Modeling a web application for cloud content adaptation with ASMs. In: Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on, pp. 44–51 (2013)

[Cre11] Cremin, R.: Mobile web content adaptation techniques. http://mobiforge.com/starting/story/mobile-web-content-adaptation-techniques (2011)

[EMMW15] Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. Sci. Comput. Program. **99**, 24–74 (2015)

[FL09] Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press, Cambridge (2009)

[GBC14] Gervasi, V., Börger, E., Cisternino, A.: Modeling web applications infrastructure with ASMs. Sci. Comput. Program. **94**(P2), 69–92 (2014)

[GRR03] Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) Abstract State Machines 2003, vol 2589., Lecture Notes in Computer Science, pp. 263–277. Springer, Berlin Heidelberg (2003)

[GRS08a] Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univ. Comput. Sci. **14**(12), 1949–1983 (2008)

[GRS08b] Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: the ASMETA case study. In: Int. Conf. on Software Engineering Advances, ICSEA, pp. 373–378 (2008)

[Gur00] Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic **1**(1), 77–111 (2000)

[Hei98] Heitmeyer, C.L.: On the need for practical formal methods. In: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT ’98, pp. 18–26. Springer-Verlag, London (1998)

[HRWW16] Hildebrandt, T., Ravara, A., van der Werf, J.M., Weidlich, M. (eds.) Web Services, Formal Methods, and Behavioral Types. 11th International Workshop, WS-FM 2014, Eindhoven, The Netherlands, September 11-12, 2014, and 12th International Workshop, WS-FM/BEAT 2015, Madrid, Spain, September 4-5, 2015, Revised Selected Papers, vol. 9421. Springer (2016)

[HIA+10] Homma, K., Izumi, S., Abe, Y., Takahashi, K., Togashi, A.: Using the model checker spin for web application design. In: Proceedings of the 2010 10th IEEE/IPSJ International Symposium on Applications and the Internet, SAINT ’10, pp. 137–140. IEEE Computer Society, Washington, DC (2010)

[HJD10] Hull, E., Jackson, K., Dick, J.: Requirements Engineering, 3rd edn. Springer-Verlag New York Inc, New York (2010)

[HPBS13] Haydar, M., Petrenko, A., Boroday, S., Sahraoui, H.: A formal approach for run-time verification of web applications using scope-extended LTL. Inform. Softw. Technol. **55**(12), 2191–2208 (2013)

[HPS04] Haydar, M., Petrenko, A., Sahraoui, H.: Formal verification of web applications modeled by communicating automata. In: Formal Techniques for Networked and Distributed Systems-FORTE 2004, vol. 3235 of Lecture Notes in Computer Science, pp. 115–132. Springer, Berlin Heidelberg (2004)

[Leu08] Leuschel, M.: The high road to formal validation. In: Proceedings of the 1st international conference on Abstract State Machines, B and Z, ABZ ’08, pp. 4–23. Springer-Verlag, Berlin, Heidelberg (2008)

[LR15] Lampesberger, H., Rady, M.: Monitoring of client-cloud interaction. In: Thalheim, B., Schewe, K.-D., Prinz, A., Buchberger, B. (eds.) Correct Software in Web Applications and Web Services, Texts & Monographs in Symbolic Computation, pp. 177–228. Springer International Publishing, New York (2015)

[Mey14] Meyer, B.: Agile! The Good, the Hype and the Ugly. Springer, New York (2014)

[MPMO10] Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. J. Logic Algebr. Program. **79**(2), 103–143 (2010)

[MsYhSbJ10] Mao-shan, S., Yi-hai, C., Sheng-bo, C., Jia, M.: A model checking approach to Web application navigation model with session mechanism. In: Computer Application and System Modeling (ICCASM), 2010 International Conference on, vol. 5, pp. V5-398–V5-403 (2010)

[OW10] Offutt, J., Ye, W.: Modeling presentation layers of web applications for testing. Softw. Syst. Model. **9**(2), 257–280 (2010)

[RR11] Rieger, B., Rieger, S.: Adaptation: why responsive design actually begins on the server. In: Breaking Development Conference, Nashville, Sep. 12–14 (2011)

[SBL+11] Schewe, K.-D., Bósa, K., Lampesberger, H., Ma, H., Vleju, M.B.: The christian Doppler laboratory for client-centric cloud computing. In: 2nd Workshop on Software Services (WoSS 2011), Timisoara, Romania (2011)

[Sch08] Schellhorn, G.: ASM refinement preserving invariants. J. Univ. Comput. Sci. **14**(12), 1929–1948 (2008)

[SDM+05] Sciascio, E., Donini, F.M., Mongiello, M., Totaro, R., Castelluccia, D.: Design verification of web applications using symbolic model checking. In: Lowe, D., Gaedke, M. (eds.) Web Engineering, vol. 3579, Lecture Notes in Computer Science, pp. 69–74. Springer, Berlin Heidelberg (2005)

[SSB01] Stärk, R.F, Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, New York (2001)

[Vle12] Vleju, M.B.: A client-centric ASM-based approach to identity management in cloud computing. In: Advances in Conceptual Modeling., vol. 7518 of Lecture Notes in Computer Science, pp. 34–43. Springer, Berlin Heidelberg (2012)

Received 1 March 2015

Accepted in revised form 12 March 2016 by Thomas Hildebrandt, Joachim Parrow, Matthias Weidlich, and Marco Carbone

Published online 13 April 2016