# Defining Services and Service Orchestrators Acting on Shared Sensors and Actuators

Rayhana Bouali Baghli[1,2], Elie Najm[1] and Bruno Traverson[2]
[1]*LTCI, Télécom ParisTech, Université Paris Saclay, 46 rue Barrault, 75013, Paris, France*
[2]*EDF R&D, Département ICAME, 7 boulevard Gaspard Monge, 91120 Palaiseau, France*

Keywords: Services, Service Orchestration, Internet of Things, Condition/Action Logic, Consistency Rules, Smart Home.

Abstract: In the context of the Internet of Things (IoT), it is necessary to design services that are loosely coupled to the objects on which they act. We call these loosely coupled services generic services. Based on a previous work (Baghli et al., 2016) that defines a three-levelled architecture for the IoT, we first propose a declarative approach to the design generic services for the IoT. Then, based on this declarative description, we define service orchestrators which are high level services that are able to manage access conflicts of services to connected objects. Next, we describe consistency rules to check validity of a generic service or an orchestrator. Finally, we illustrate our approach with use cases around services in a smart home.

## 1 INTRODUCTION

In the current context of the Internet of Things (IoT), services and the devices they manage come often bundled. This strong coupling between objects and services creates technological silos which do not foster an open market for IoT. On the other hand, several pragmatic initiatives, such as IFTTT (If This Then That) (Ovadia, 2014), have recently been launched aiming at breaking these silos. These initiatives ease the design and development of services (called recipes) which are not tied with a specific brand of connected devices. While providing practical and running solutions, these endeavors have limitations since they do not rely on an explicit architecture or on solid foundation principles. For instance, the issue of service composition is not tackled, and thus recipes cannot act on shared devices in a coordinated manner. The figure below is an illustrative example of the difficulty to reason, in an asynchronous operational environment, about conflicts that can emerge when concurrently running services act on shared devices. In this example, the *temperature regulation service* reacts to events sent by the temperature sensor and may request the window to be closed. The *air quality service*, on the other hand, may react to events emanating from the air quality sensor by requesting the window to be open. This simple example shows the need for a higher level of abstraction whereby services are described and reasoned about on the basis of their service logic.

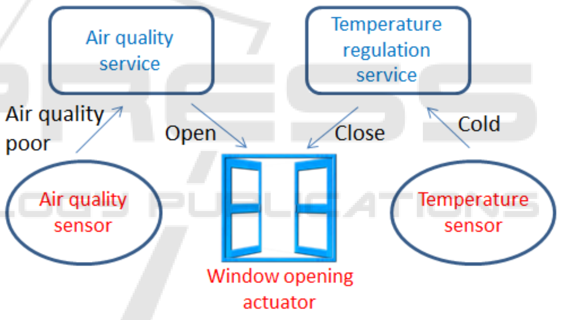


Figure 1: Example of conflicting services acting on shared devices.

In (Baghli et al., 2016), an architecture called SAR is encompassing 3 levels of abstraction: Semantics, Artifacts and Resources (loosely inspired by the 3 mid viewpoints of RM-ODP (Linington, 1995)). SAR is aimed to serve as a basis for the design and development of services for the IoT. It prescribes the existence of type repositories for connected objects. Hence, it allows the design of generic services that can be bound to specific objects at deployment phase, provided the respective types match. SAR provides also a basis to reason about service orchestration. It allows to define orchestrators that detect, manage and resolve conflicts that may arise when two or more services are bound to shared actuators. The focus of this paper is the Semantics level. In this level, services and orchetrators are modeled using a logical declarative approach. Model transformations (not discussed in this paper) complement the service models

Baghli, R., Najm, E. and Traverson, B.
Defining Services and Service Orchestrators Acting on Shared Sensors and Actuators.
DOI: 10.5220/0006609402370246
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 237-246
ISBN: 978-989-758-283-7

defined in the Semantics level. A first model transformation generates artifact representations of these services (featuring events and messages) while a second transformation provides for its restful (resource oriented) execution environment. The reminder of the paper is composed as follows. In section two, we provide an informal introduction to the Semantics level. Section three presents a simplified meta-model of this level. In section four we present a formalization of the concepts defining the Semantics level. In section five we discuss a validation approach using Alloy. Section six is devoted to related work. Finally, in section seven we conclude.

## 2 OVERVIEW OF THE SEMANTICS LEVEL

Simply stated, in this level, the system is viewed as a collection of typed read and write variables that are read and written on by services. Sensors are abstracted as read only variables. Actuators are abstracted as write only variables. We also introduce a third category of devices that we call sensactuators and that correspond to the kind of actuators that have also the capacity to sense and reveal their current state. In the semantics level, sensactuators are abstracted as read/write variables. The current state of the system is reflected by the current values of the variables composing the system. The physical environment has the capacity to change the state of the sensors, while services behave as agents that react to these changes by providing the values that should be The state of the system is given by the current values of the variables. the which considers that the system is managed by invariants. In this level, we deal only with notions of states. All operational aspects like messages exchanges between services are considered only in the artifacts and resources levels. Semantic level abstracts all these notions and provide a high level way to model, orchestrate and resolve conflicts between services.

In our approach we consider that the user choose in a catalog some generic services and some connected objects according to his personal project. After that each of selected services is instantiated and linked to an execution context. Also, each connected object is installed and linked to his physical localization. When, a least one service is instantiated, it can be binded with objects he will rely on. The binding process is first a selection of connected objects that interest that instance of service, and secondly link is created between service instance and connected objects he is binded to. A least, if an object is shared by several services, an orchestration functionality is set up to manage conflicts between these services.

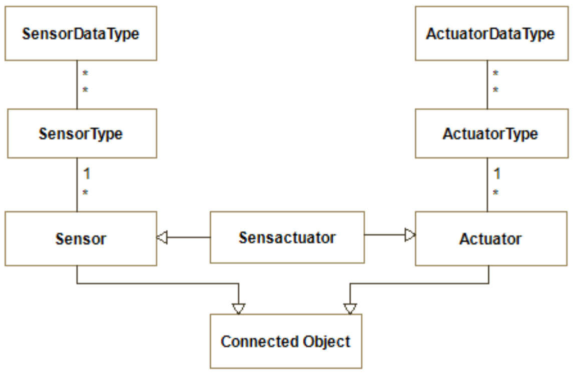


Figure 2: Meta-model of connected objects.

## 3 META-MODEL OF THE SEMANTICS LEVEL

In this section, we describe in some detail the concepts that populate the semantic level of the SAR architecture.

We start by describing the meta model of connected objects and then we present the services meta-model.

### 3.1 Connected Objects

We model connected objects as they are viewed by services which use them. The purpose of this section is not to provide a complete meta-model to represent connected objects, but to represent concepts that are relevant from the semantic service model point of view.

Objects that form the IoT are able to observe their environment and to provide measures recorded in corresponding variables or to act on the envoronment by enforcing actuators into certain states. We classify objects of the IoT in three classes : Sensors, Actuators and Sensactuators. Sensors can sense a physical measure and send it on a network; actuators can actuate real world objects; and sensactuators can sense state and actuate same real world objects. So, sensors only present data (they provide to services); actuators only receive the target states that they need to be reached by them; and sensactuators do both.

Figure 2 shows connected object meta-model. A connected object could be, sensor, actuator or sensactuator. Each ”Sensor” is of one ”SensorType” and each ”SensorType” can be related to several sensors. Semantically, this relation represents a classification of sensors.”Temperature sensor” and ”Light sensor” are two examples of sensor types.

In the same way, actuators are classified into actuator types.

On the other hand, each ”SensorType” concept is related to ”SensorDataType” concept. Sensor data types are types of data sent by sensors on the network.

”ActuatorType” concept is related to a ”ActuatorDataType” concept. Actuator data types define sets of elements which could be handled by actuators.

”Sensactuator” concept is subtype of ”Sensor” and ”Actuator” concepts. When a connected object is considered as a sensactuator, its sensor type and actuator type are linked to each other. Its sensor data types and actuator data types are also linked to each other. This relation is an equality or an inclusion relation.

## 3.2 Services for the IoT

This section describes services meta-model which allows to describe generic services independently from the connected objects they would rely on.

Orchestrators are services that are able to manage access conflicts of services to connected objects.

Service meta-model is split into two compatible meta-models : Structural meta-model and Behavioural meta-model. These meta-models are presented in detail in the two following sections.

### 3.2.1 Structural Modelling

Figure 3 depicts the service structural meta-model. In order to model services independently from connected objects, these services are described in terms of roles. Roles can be sensor roles, actuator roles or sensactuators roles. Each role can be played by several ”service object types” which are ”service sensor type”, ”service actuator type” or ”service sensactuator type”. An example of a sensactuator role for a ”temperature regulation service” could be a ”heat source”. Some examples of service sensactuator types which can play this role could be ”electric heater”, ”gas boiler” or ”heating floor”.

Service structural meta-model describes ”SensorDataType” and ”ActuatorDataType ”. Sensor data types and actuator data types concepts define data types that the service can handle.

Service rules describe behaviour of the service, this concept is depicted in detail in the section 3.2.2.

A ”Cluster” is an entity that groups a set of services that share actuators or sensactuators. Two services of the same cluster could directly or via a transitive closure share actuators or sensactuators. Services that are part of the same cluster are orchestrated by one Orchestrator.

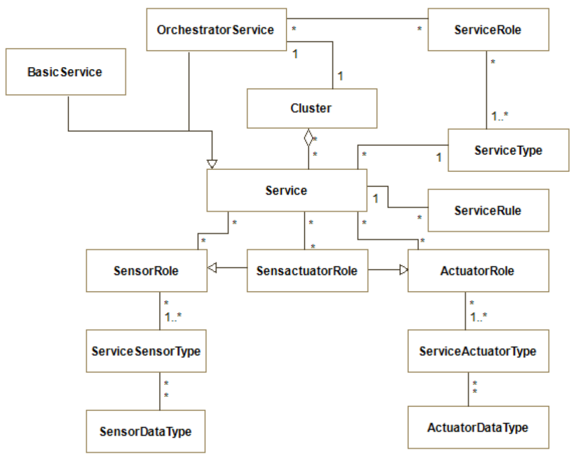


Figure 3: Meta-model of IoT services structure.

The structural meta-model allows description of basic services and orchestors. In addition to structural concepts which are described in basic services, orchestrators include service roles. These roles can be played by several service types.

We have described structural modelling of services and we present, in the next section, behavioural modelling of these services based on concepts of the structural part of the meta-model.

### 3.2.2 Behavioural Modelling

The aim of the semantic level of our architecture is to propose a modelling of services that is close to user needs. So, we choose to model the services in terms of goals rather than actions.

Declarative approaches are used to describe objectives to be achieved rather than procedures to be followed. Our proposition has naturally been based on a declarative approach for describing service behaviour as a set of service rules. These rules describe service objectives disregarding on how these goals can be achieved.

Figure 4 depicts the service behaviour meta-model. Each rule is composed by one premise and one conclusion.

The premise describes the condition that triggers the rule, and the conclusion represents the consequence of triggering the rule. The premise concept is related ”CurrentState” and ”PreCondition” concepts. The combination of instances of these two concepts triggers a service rule that describes them in its premise.

The precondition concept is an expression that can be evaluated, for a particular current state, as ”true” or ”false”. The current state concept is a specialization of the ”State” concept. It represents the current state of some roles that are part of the structural model of the service.

In addition to objects states, each actuator role is related to an ”Agent” concept. Agent concept describes the responsible of the current actuator state. This responsible could be a human, a service or the environment.

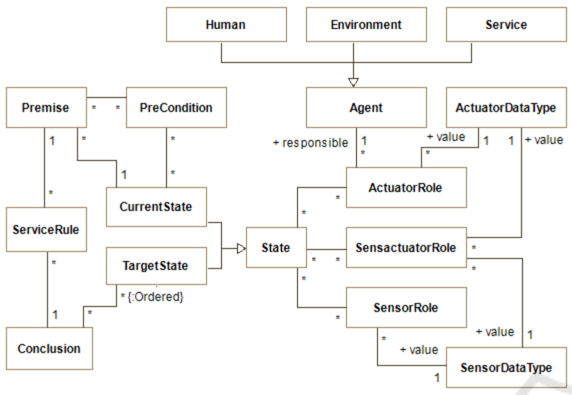


Figure 4: Meta-model of IoT services behaviour.

The concept of conclusion represents the consequence of a service rule. The conclusion is related to a set of ”TargetState” and each target state is related to a ”StateCategory”. ”Desired state” and ”Tolerated state” are two examples of state category instances. The ordered set of target states represents choices, from the most desired to the least desired target state of the service rule. This ordered set of target states represents several preferences of the service rule.

In the same way, each service could describe a set of ordered target states as potential reactions to one current state. The advantage of describing a set of target states in a conclusion instead of a single target state can be seen in a configuration that contains several services acting on same objects. In this case, one or more orchestrators will orchestrate these services. If the services express several choices of target states, then the orchestration process will be more flexible. The orchestrator would be able to satisfy several services by choosing a target state which satisfies the greatest number of services but which is not necessarily the first choice for each service.

We have described meta-models which are part of the semantic level, we present in the next section, formal representations of services and service compositions and a set of checking rules for each of them.

## 4 FORMALIZATION OF THE SEMANTICS LEVEL

In this section, we present formal descriptions of services and service compositions;

## 5 SMARTHOME

A smart home is defined as a triple made of a set of devices, a collection of services and a set of bindings that allow to deploy services on the devices of the smart home.

SmartHome = (DEV, Serv, Bindings)

## 6 DEVICES

The set of devices is a disjoint union of sensors ($SE$), actuators ($AC$) and sensactuators ($SA$).

$DEV = SE \uplus AC \uplus SA$ We will use $d$ to denote any device and we will assume the existence of a mapping $kind : DEV \Rightarrow \{S, A, SA\}$

with $d.kind = S \Leftrightarrow d \in SE(respectively\, A,\ SA)$.

Actually, as seen by services, devices represent read (sensors), write (actuators) and read/write (sensactuators) variables.

The set $RD = SE \cup SA$ is the subset of variables that can be read by services and $WD = SA \cup AC$ those that can be written.

We denote by $Val_{DEV}$, $Val_{SE}$, $Val_{AC}$, $Val_{SA}$, $Val_{RD}$, $Val_{WD}$ the set of values taken respectively by all devices, sensors, actuators, sensactuators, readable and writable variables.

We let $se$ denote any sensor, $ac$, any actuator, $sa$, any sensactuator, $rd$, any readable device (sensor or sensactuator) and $wd$ any writable device (actuator or sensactuator).

## 7 STATE OF A SMART HOME

The running state of smart home is any function that maps devices to values. States can be global on all devices or partial on a subset of devices:

global state : $GS : DEV \rightarrow Val_{DEV}$
read state : $RS : RD \rightarrow Val_{RD}$
write state : $WS : WD \rightarrow Val_{WD}$

## 8 SERVICES

Informally, a service is an active entity that acts on a set of associated roles , reading from the read roles and reacting by writing on the write roles.

More formally,

A service $S$ is a couple $S = (Ro, Ru)$ with

$Ro$ a set of roles

$Ru$ a set of rules : $Ru = R_1, ..., R_n$

$Ro$ is equiped with the *kind* function similar to $DEV$.

Hence, for a role $r$, $r.kind \in \{S, A, SA\}$ is its *kind*: sensor, actuator or sensactuator.

$Ro.RR = \{r | r.kind \in \{S, SA\}\}$ is the subset of readable roles and
$Ro.WR = \{r | r.kind \in \{A, SA\}\}$ is the subset of writable roles.

Each role has an associated domain of values (which depends on its type).

We let $Val_{RR}(Ro)$ the set of values of all readable roles in Ro and $Val_{WR}(Ro)$ the set of all writable roles in Ro.

A rule $R \in Ru$ is of the format $R = (Pre, NS)$ with

*Pre* is the precondition of the rule. It is a predicate that evaluates on readable roles. The concrete syntax depends on the types of readable roles.

If $rs$ is a readable state on roles i.e. $rs : Ro.RR \rightarrow Val_{RR}(Ro)$ then $Pre(rs) \in \{true, false\}$

$NS$ is a priorized list of next states of write roles with

$NS = (ns_1, ..., ns_k)$

A next state $ns$ is a partial function $ns : Ro.WR \rightarrow Val_{WR}(Ro)$

We denote $NS.WR$ the set of write roles appearing in $NS$.

A service is well formed iff the following condition holds

$\forall\ R_i,\ R_j\ with\ i \neq j,\ R_i = (Pre_i,\ NS_i),\ R_j = (Pre_j,\ NS_j),\ \forall\ rs,\ Pre_i(rs) = True\ \Rightarrow\ Pre_j(rs) = False$

## 9 BINDINGS

Given a smart home equiped with a set of devices and a collection of independently defined services ready to be deployed, we need to define the notion of deployable and deployed services. To do so, we use, for each service, a bind function that maps roles to devices.

### 9.1 Valid Binding

Given a well formed service, $S = (Ro, Ru)$ and *Bind*, a map, $Bind : Ro \rightarrow DEV$

Bind is said to be a valid deployment map with regard to S iff it satisfies the following two conditions:

(i) Kind compatibility: $\forall r \in Ro, Bind(r).kind = r.kind$

(ii) Injectivity on writable devices: $\forall\ r,\ r' : r \neq r'$ with $r\ \in_R o.WR,\ r'\ \in Ro.WR : Bind(r) \neq Bind(r')$

Condition (i) stipulets that roles are mapped on devices of the same kind. We will assume also that there is a compatibility between the types of roles and the types od devices they are bound to and that the domains of values are identical between roles and the bound devices.

Condition (ii) enforces that wr writable roles cannot be bound to the same device.

## 10 DEPLOYED SERVICES

A deployed service is a couple $(DD, DRu)$ where $DD$ is a set of devices and $DRu$ is a set of rules acting on the set of devices. A deployed service is similar to a service, except that roles are replaced with devices.

Given a couple $(S, Bind)$, with $S$ a well formed service and *Bind* a valid deployment map, with $S = Ro, Ru$ and $Ru = \{(Pre_1, NS_1), ..., (Pre_n, NS_n)\}$

We can define a deployed service DS where roles are substituted wth devices as follows :

$Bind\,(S) = (Bind\,(Ro)\,, Bind\,(Ru))$

Where $Bind\,(Ro)$ is the codomain of Bind and $Bind\,(Ru)$ is the set of rules obtained by replacing in each rule of $Ru$ roles by their corresponding bound devices.

Claim: if $S$ is well formed then $Bind\,(S)$ is well formed.

## 11 DEPLOYED SMART HOME

Considering a smart home with devices $DEV$, services *Serv*, and *BIND*, a set of valid deployable maps, one can define a deployed smart home where services are replaced by deployed services. To that end, since different services can act on shared actuators, we need to introduce mechanism to resolve conflicts between services accessing the same actuators.

A deployed smart home is a couple $(DEV, DServ)$ where $DEV$ is a set of devices and *DServ* a list of prioritized deployed services, i.e. a totally ordered set of deployed services.

### 11.1 Cluster

A cluster is the unit for orchestration. Services of the same cluster are orchestrated based on their priorities given a read state.

A cluster of services $C$ is an ordered list of deployed services compatible with the order in $DServ$, $(DS_1,...,DS_n)$ and such that $\forall\ i,\ j:$
$DS_i\ sh^*\ DS_j$

where $sh$ is a binary relation on deployed services defined by

$DS\ sh\ DS' \leftrightarrow \exists\ wd\ such\ that\ wd \in DS\ and\ wd \in DS'$

and $sh^*$ is the transitive closure of $sh$ i.e.

$DS\quad sh*\quad DS'\quad iff\ \ \exists\ \{DDS_1,...,DDS_n\} \subset DSERV\ such\ that$

$DDS_1 = DS \wedge DDS_k = DS' \wedge \forall\ i < k-1 : DDS_i\ sh\ DDSi+1$

### 11.2 Conflict Resolution within a Cluster of Orchestrated Services

Given a cluster $C = (DS_1,...,DS_n)$ with $DS_i = (DD_i, DRui)$ where
$DRui = (Pre_i^1, NS_i^1),...,\ (Pre_i^k,\ NS_i^k)$

For given read state RS, let $DS_{i_1},...,DS_{i_m}$ the subset of services for which one premisse is true on RS.

Let us call $Pre_{i_p}$ the premiss true for service $DS_{i_p}$ and $NS_{i_p}$ the associated next state.

$NS_{i_1} = ns_{i_1}^1,...,ns_{i_1}^{q_{i_1}}$
.
.
.
$NS_{i_m} = ns_{i_m}^1,...,ns_{i_m}^{q_{i_m}}$

The next state that maximizes satisfaction of high priority service is given as follows :
1) We define the set $Max_1$, the set of next state that maximize the function $f$
$f = 2^m(ns_{i_m}^1 \vee ... \vee ns_{i_m}^{q_{i_m}}) + ... + 2(ns_{i_1}^1 \vee ... \vee ns_{i_1}^{q_{i_1}})$
2) For $ns \in Max_1$ we define function $f'$
$f'(ns) = 2^m(2^{q_{i_m}}(ns \rightarrow ns_{i_m}^{q_{i_m}}) + ... + 2(ns \rightarrow ns_{i_m}^1)) + ... + 2(2^{q_{i_1}}(ns \rightarrow ns_{i_1}^{q_{i_1}}) + ... + 2(ns \rightarrow ns_{i_1}^1))$

Let $Max_2$ the set of states $ns'$ that maximizes function $f'$

$Max_2$ contains exactly one element that we call $NS_{sat}$ which is selected as the next state corresponding to read state $RS$, that maximizes satisfaction.

In this section, we presented formal service and service composition descriptions. These descriptions and properties allows to represent formally services and service compositions to check validity of them and then to bind them with connected objects. The next section presents how we validate our approach using smart home use cases.

## 12 VALIDATION

In order to validate our approach, we define three generic services in the field of smart home use-cases, including two basic services and a service composition of the two basic services we define.

As a first step, each generic service is described in natural language and part of its structural and behavioural descriptions are exposed.

Next, we implement in the Alloy Analyzer (Jackson, ), our meta-models, services and compositions models, and checking rules. In order to make validation process less complex, we suppose that generic services have already been instantiated and bound to connected objects. So we implement instantiated services instead of generic ones.

### 12.1 Smart Home Use Cases

We define in this section three generic services in the field of smart homes. Two basic services ares described : a temperature control service and an energy management service. The third one is a composition of the first two basic services and concerns an economical temperature control service.

In this section each generic service example is first described in natural language. Then, part of structural and behavioral descriptions of each generic service are shown.

#### 12.1.1 Temperature Control Service

The temperature control service we define aims to regulate the temperature in its execution context. The execution context could be a room, a set of rooms, a floor or the whole house. This service is intended to regulate temperature in winter mode. It is interested in all objects that produce heat, such as heaters or boilers. It also considers objects which measure temperature inside and outside its execution context.

**Structural Definition.** The structural definition describes sensors, actuators and sensactuators roles. We give here an example of a sensactuator role which is defined by our temperature control service :

SensactuatorRole: Source of heat

- ServiceSensactuatorType1: Electric heater.
  - ActuatorDataType1: ”On, Off”
  - SensorDataType1: ”On, Off”

- ServiceSensactuatorType2: Gas boiler.
  - ActuatorDataType2: ”On, Off”
  - SensorDataType2: ”On, Off”

This sensactuator role describes a source of heat role. This role could be played by two types of sensactuator objects : electric heater and gas boiler. The role description defines also sensactuators data types which the service can handle. In this description, sensor and actuator data types are ”On, Off” for both sensactuator types.

**Behavioural Definition.** The behavioural definition describes a set of service rules. An example of the temperature control service rule is given as follows

- If internal temperature is lower than target temperature and source of heat is an electric heater then the electric heater is on.

This rule links between the ”source of heat” role and the ”electric heater” object type. Another rule describes the case of that role is played by the ”gas boiler” object type.

#### 12.1.2 Energy Management Service

The energy management service aims to control energy consumption of its execution context by avoiding a set of energy loss scenarii. This service is interested in all objects which consume energy and all objects through which there may be a loss of energy. An example of an energy loss scenario is an opened window when a heater is turned on. Another example of energy loss may be a light which is switched on in an empty room.

**Structural Definition.** The energy management service structural definition describes roles of sensors, actuators and sensactuators. Each of these roles could be played by one or several objects types and each object type is linked to the object data type which the service can handle.

SensactuatorRole: Energy loss source

- ServiceSensactuatorType1: Window.
  - ActuatorDataType1: ”Open, Close”
  - SensorDataType1: ”Open, Close”
- ServiceSensactuatorType2: Light.
  - ActuatorDataType2: ”On, Off”
  - SensorDataType2: ”On, Off”

This sensactuator role describes a source energy loss role. This role could be played by two types of sensactuator objects : Window and Light. In this example, window sensor and actuator data types are ”Open, Close” and Light sensor and actuator data types are ”On, Off”.

**Behavioural Definition.** The behavioural definition describes the rules of the service. As for the other services, each energy management rule links between roles, objects types which could play these roles and object data types.

- If heater is on and and energy loss source is a window then the window is close.

This rule aims to avoid the energy loss scenario where a window is open when a heater is turned on.

#### 12.1.3 Economical Temperature Control Service

The economical temperature control service we define is a composition of two services : a temperature control service and an energy management service. This composite service aims to control temperature in its execution context while limiting the loss of energy.

**Structural Definition.** Our example of economical temperature control service defines two service roles: a temperature control service roles and a service role for managing energy consumption. It does not define object roles but relies on roles which are defined by temperature control and energy management services.

**Behavioural Definition.** The economical temperature control service does not define behavioural rules, but it relies on rules which are defined by temperature control and energy management services.

## 12.2 Validation on Alloy

Alloy is a notation inspired by the Z language. It adopts a declarative modelling approach and relies on logic of the first order. Alloy allows to describe models and constraints which apply on these models (Jackson, 2012).

In addition to describing models and constraints, Alloy allows to check validity of these models. Alloy relies on a SAT solver which transforms constraints into booleans and performs validations on models according to these constraints. For more details please refer to (Jackson, 2012).

For validations of our approach, we use the Alloy Analyzer (Jackson, ) which supports description of Alloy models and their automated analysis. The Alloy Analyzer allow two types of checking : Simulation and assertion checking. Simulation looks for an instance of the model which conforms the specification. And assertions checking verify, in a scope that all instances verify validity constraints. The scope could be formed by billions of instances.

For our simulations and validations in alloy, we translate smart home use cases models in Alloy and implement them in the Alloy Analyzer. We implement service and service composition properties to simulate and validate the services we define.

When translating our models and meta-models, we followed the procedure described in (Anastasakis et al., 2007) where classes are translated to signatures and association ends are translated to signature fileds. It is possible to add facts to the Alloy model to translate the multiplicity constraints of the metamodel.

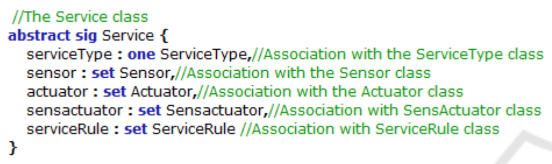
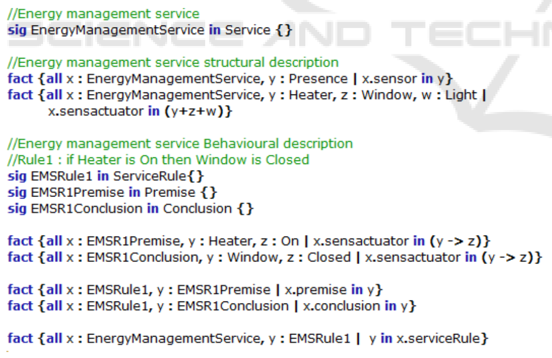

```
 //The Service class
abstract sig Service {
   serviceType : one ServiceType,//Association with the ServiceType class
   sensor : set Sensor,//Association with the Sensor class
   actuator : set Actuator,//Association with the Actuator class
   sensactuator : set Sensactuator,//Association with SensActuator class
   serviceRule : set ServiceRule //Association with ServiceRule class
}
```

Figure 5: Portion of the metamodel implemented in Alloy.

Figure 5 shows a portion of our Alloy code. This portion of code implements the Service class which is defined as a signature (sig) and each association is defined as a fields of that signature. The multiplicity constraints on the association ends are here translated by the ”one” and ”set” keywords.


```
//Energy management service
sig EnergyManagementService in Service {}

//Energy management service structural description
fact {all x : EnergyManagementService, y : Presence | x.sensor in y}
fact {all x : EnergyManagementService, y : Heater, z : Window, w : Light |
      x.sensactuator in (y+z+w)}

//Energy management service Behavioural description
//Rule1 : if Heater is On then Window is Closed
sig EMSRule1 in ServiceRule{}
sig EMSR1Premise in Premise {}
sig EMSR1Conclusion in Conclusion {}

fact {all x : EMSR1Premise, y : Heater, z : On | x.sensactuator in (y -> z)}
fact {all x : EMSR1Conclusion, y : Window, z : Closed | x.sensactuator in (y -> z)}

fact {all x : EMSRule1, y : EMSR1Premise | x.premise in y}
fact {all x : EMSRule1, y : EMSR1Conclusion | x.conclusion in y}

fact {all x : EnergyManagementService, y : EMSRule1 |  y in x.serviceRule}
```

Figure 6: Portion of the energy management service implemented in Alloy.

Figure 6 depicts a portion of our Energy management service code in Alloy. The first part of our code (Line 1) defines the energy management service as a signature which is in the ”Service” signature. The second part of our code describes structural description of the energy management service. In the structural description, we define connected objects on which our service acts. Connected objects which are rely to our service are presence sensor, heater, window and light. The third part of the Alloy code depicts a portion of the behavioural description of our energy management service. It describes one rule which states that if heater is on then window is close.

After translating our meta-models and models in Alloy, we translate service properties and service composition properties. Once translated in Alloy, these properties represent checking rules to validate services and service compositions.

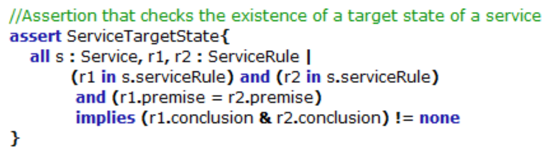

```
 //Assertion that checks the existence of a target state of a service
 assert ServiceTargetState{
   all s : Service, r1, r2 : ServiceRule |
          (r1 in s.serviceRule) and (r2 in s.serviceRule)
           and (r1.premise = r2.premise)
           implies (r1.conclusion & r2.conclusion) != none
 }
```

Figure 7: Service target state checking implemented in Alloy.

Figure 7 shows an example of service checking rule. This rule is expressed as an assertion and it allows to verify existence of a target state for a service. This assertion rule example verifies that for two rules that are triggerable simultaneously, it is possible to find a common target state. Concretly, it verifies that intersection of these triggerable rules conclusions is not empty.

After defining checking rules as assertions, we can verify validity of services. We implement an example of non valid energy management service. This service has two triggerable rule whose conclusions intersection is empty. Figure 8 shows that the Alloy Analyzer founds a counter example which invalidates the ”service target state” assertion. It means that the target state assertion is invalide for our model. The Alloy analyzer can also show graphically the counter example he finds and which invalidates the assertion.

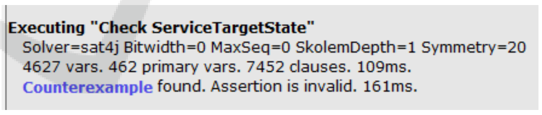

```
Executing "Check ServiceTargetState"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   4627 vars. 462 primary vars. 7452 clauses. 109ms.
   Counterexample found. Assertion is invalid. 161ms.
```

Figure 8: Service target state checking result.

The sample code and results we present show the approach we take to verify the validity of services and service compositions using the verification rules we describe in section V and using the Alloy Analyzer tool.

## 13 RELATED WORK

In the field of service composition and services conception, many efforts have been done in order to provide platforms and languages for building heterogeneous systems. Standards have be defined to discover, describe and invoke Web services like WSDL (Web Services Description Language) (Christensen et al., 2001), UDDI (Universal Description, Discovery, and Integration) (Bellwood et al., 2002), SOAP (Simple Object Access Protocol) (Box et al., 2000) or also DAML-S ontology (Ankolekar et al., 2002). BPEL4WS (Business Process Execution Language for Web Services) (Andrews et al., 2003) allows to compose dynamically services by describing messages exchanges between services when they are known a priori. Authors of (Ponnekanti and Fox, 2002) proposed another approach called SWORD which consists in a set of tools to compose web services.

In the filed of conflicts resolution for the Internet of Things, authors of (Bertran et al., 2014) proposed Diasuite, a tool to develop SCC (Sense Compute Control) applications. The SCC paradigm allows to collect data from sensors, to compute them and to command actuator for acting on their environment. To manage conflicts, authors also proposed in (Jakob et al., 2011) an approach based on a DSL extension. This approach generates at runtime a code to manage conflicts. The major limitation of this approach is that conflicts management is not fully automatic and developers have to act in conflicts resolutions.

Other recent efforts are done in conflicts resolution between shared devices. In (Hadj et al., 2016), authors describe an approach which aims to manage conflicts when devices are shared by several applications. They proposed to add an autonomic access layer in pervasive platform that is called iCASA (Escoffier et al., 2014). This platform provides a service oriented component model to develop pervasive applications. These efforts seem to be interesting but are already in a preliminary phase. There is not yet a formal model which could resolve conflicts management.

The difference between these approaches and the one we develop in this paper resides in the fact that we have described in (Baghli et al., 2016) a multilevelled architecture. This architecture allows to abstract at the semantic level all exchanges of messages and temporal aspects which are related to conception and composition of services and conflicts management resolution between services on shared objects. Our approach considers that the semantic level is managed by invariants and deals only with states of the system. All aspects of message exchanges, actions that move the system from one state to another one are dealt by the artefacts level of our architecture.

Many work have also been done in the field of connected objects modelling, ontologies like (Compton et al., 2012) and (Seydoux et al., ) propose to model connected sensors and actuators. In fact our work is not focused on objects modelling but represent, for each level of our architecture, objects like they are viewed from that level services perspective. So, in each level, we describe a meta-model of connected objects which is adapted to the needs and principles of this level.

## 14 CONCLUSION AND FUTURE WORK

We present in this paper an approach to model generic services independently of connected objects. This approach deals with service composition and objects sharing between several services. We have described the meta-models on which our proposal is based. We also described formal representations of services and service compositions. Checking rules have been defined to verify the validity of services and service compositions. In order to validate our proposal, we have implemented smarthome use cases in the Alloy Analyzer tool.

In our future work, we would complete our approach by implementing a platform for the design-time modeling and runtime management of IoT services. We would also formally define an operational meta-model and model tranformations.These model tranformations should allow to translate the high level semantic models to operational models.

## REFERENCES

Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2007). Uml2alloy: A challenging model transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 436–450. Springer.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., et al. (2003). Business process execution language for web services.

Ankolekar, A., Burstein, M., Hobbs, J. R., Lassila, O., Martin, D., McDermott, D., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T., et al. (2002). Daml-s: Web service description for the semantic web. In *International Semantic Web Conference*, pages 348–363. Springer.

Baghli, R. B., Najm, E., and Traverson, B. (2016). Towards a multi-leveled architecture for the internet of things. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2016 IEEE 20th International*, pages 1–6. IEEE.

Bellwood, T., Clément, L., Ehnebuske, D., Hately, A., Hondo, M., Husband, Y., Januszewski, K., Lee, S., McKee, B., Munter, J., et al. (2002). The universal description, discovery and integration (uddi) specification. *Rapport technique, Comit OASIS*.

Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Balland, E., and Consel, C. (2014). Diasuite: A tool suite to develop sense/compute/control applications. *Science of Computer Programming*, 79:39–51.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple object access protocol (soap) 1.1.

Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., et al. (2001). Web services description language (wsdl) 1.1.

Compton, M., Barnaghi, P., Bermudez, L., GarcíA-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., et al. (2012). The ssn ontology of the w3c semantic sensor network incubator group. *Web semantics: science, services and agents on the World Wide Web*, 17:25–32.

Escoffier, C., Chollet, S., and Lalanda, P. (2014). Lessons learned in building pervasive platforms. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, pages 7–12. IEEE.

Hadj, R. B., Chollet, S., Lalanda, P., and Hamon, C. (2016). Sharing devices between applications with autonomic conflict management. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 219–220. IEEE.

Jackson, D. Alloy website. http://alloy.mit.edu/alloy/index.html. Accessed on Sep. 30, 2017.

Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.

Jakob, H., Consel, C., and Loriant, N. (2011). Architecturing conflict handling of pervasive computing resources. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 92–105. Springer.

Linington, P. F. (1995). Rm-odp: the architecture. In *Open Distributed Processing*, pages 15–33. Springer.

Ovadia, S. (2014). Automate the internet with if this then that (ifttt). *Behavioral & Social Sciences Librarian*, 33(4):208–211.

Ponnekanti, S. R. and Fox, A. (2002). Sword: A developer toolkit for web service composition. In *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, volume 45.

Seydoux, N., Alaya, M. B., Drira, K., Hernandez, N., Monteil, T., and Haemmerlé, O. San (semantic actuator network). https://www.irit.fr/recherches/MELODI/ontologies/SAN.html. Accessed on Sep. 30, 2017.