INTEGRATION OF A DECLARATIVE LANGUAGE BASED ON FLUID QUALITIES IN A SERVICE ORIENTED ENVIRONMENT

Matei Popovici
email: *pdmatei@gmail.com*
Cristian Giumale
email: *cristian.giumale@gmail.com*
Lorina Negreanu
email: *lorina@moon.ro*
Alexandru Agache
email: *alexandruag@gmail.com*
Mihnea Muraru
email: *mmihnea@gmail.com*
Ciprian Dobre
email: *cipsmm@gmail.com*
Computer Science Department
POLITEHNICA University of Bucharest
Splaiul Independentei nr. 313, sector 6, 060042, Bucharest, Romania

ABSTRACT
Ontology formalisms such as OWL, as well as conventional languages designed for specifying Web-Service behavior and interactions such as WS-BPEL, are limited in their ability of defining complex time-dependent processes. As an alternative, declarative languages present several advantages: they are easier to use, since desired properties are specified explicitly, they are smaller in size, as they lack additional control-flow structures, specific for imperative languages. In addition, model checking techniques can be naturally applied. In this paper, we describe a modeling method and declarative language based on fluid qualities. Based on this language, we propose two mechanisms for integration in a Service Oriented environment. Using our framework, intelligent and time-dependent device behavior can be specified in an ubiquitous environment.

KEYWORDS
ontologies, declarative languages, services, SOA, smart homes, intelligent systems

1 Introduction

Traditionally, knowledge representation systems are designed to statically describe a universe of discourse, rather than modeling its evolution. Ontology modeling languages such as OWL [1] have no inherent mechanisms for modeling temporal properties and reasoning about them. Known approaches such as OWL-Time [2] define temporal concepts on the same level with other application-specific concepts, thus yielding complex models that are difficult to maintain, extend, and reason about.

Approaches based on semantic service descriptions such as OWL-S [3] suffer from similar drawbacks. OWL-S is designed to describe services in a static manner. It provides snapshots, but is unable to capture the evolution of a SOA environment. These limitations are inherent to the modeling approach based on OWL, where the instantiation relationship between an individual and a concept is unique. In this setting, representing knowledge such as *air conditioner is currently cooling but was previously off* requires the addition of a new temporal layer to the domain description.

Similarly, there are no dedicated languages for defining temporal models. In a Service Oriented Architecture (SOA) [4] environment, WS-BPEL [5] is the conventional language for defining interactions between services. It is inherently a tool for defining the evolution of processes and workflows. Imperative languages such as WS-BPEL [5] focus more on defining a control flow, and less on the behavioral and temporal properties of service interactions. For instance, assuming we have two services S_1 and S_2, we would like to define a workflow that allows successive invocations of S_1 or S_2, but does not allow invocations of both S_1 and S_2. As a result, invocation sequences such as $S_1, S_1, S_1, \ldots$ or $S_2, S_2, S_2, \ldots$ are considered valid, but S_1, S_2, S_1 is invalid. Representing workflows such as these in WS-BPEL [5] proves difficult, and requires the definition of dedicated structures that hold all past states of the model.

As an alternative, a modeling approach based on fluid qualities, as well as a declarative language [6, 7] for specifying service interactions, is in our opinion more suitable. The declarative paradigm has numerous advantages: model specification is done at a higher level, and aims at isolating technical aspects such as message communication in WS-BPEL [5]. This makes modeling more accessible to inexperienced users. Models are less complex and easier to read. The issues of understandability and maintainability in declarative process modeling are discussed in [8]. Also, model checking techniques are easier to apply on declarative specifications.

We propose an integration mechanism between a declarative language (DL) described by Giumale et al. [6, 7] and an SOA environment. The integration mechanism connects an interpreter for the DL to Web-Services. The entire framework is used in an intelligent house environment, where devices such as air conditioners, light-systems etc. are controlled using Web-Service invocations. As a result, the DL can be deployed as a tool for defining intelligent device behavior.

In Section 2 we give a brief description of our modeling approach and DL. Further details are given in [6, 7]. In Section 3 we present the integration problem, as well as two possible solutions. The first alternative relies on defining a device taxonomy as well as a binding of existing devices to this taxonomy. The second alternative relies on simpler service annotations, but forces users to introduce a considerable amount of data. Each approach has advantages and disadvantages, depending on the particular environments. Section 5 presents similar approaches, and compares them to our own. In Section 6 we present conclusions as well as future work.

2 A modeling approach and DL

The modeling approach introduces three main ontological entities: *individuals*, *qualities* and *actions*.

2.1 Individuals

Individuals are independent, atomic entities with no structure. They are perennial, with respect to the modeled universe, meaning that they do not disappear. However, they can acquire or lose qualities.

2.2 Qualities

A quality $Q(x_1, x_2, \ldots, x_n)$ is used for describing certain properties that individuals might possess, as well as relations between two or more individuals. Examples of such properties are $q_1 = On(AirConditioner_1)$, $q_2 = HasPower(AirConditioner_2, 200KW)$. The first quality indicates the current state of the individual $AirConditioner_1$, which stands for a device. The second quality defines a relation between two individuals: $AirConditioner_2$ refers to another device, while $200KW$ refers to an actual value.

We distinguish between qualities and their prototypes. In the above examples, the qualities q_1 and q_2 are instances of quality prototypes $On(x)$ and $HasPower(x, y)$. During the evolution of an application model, prototypes can be instantiated several times: $q_1 = On(AirConditioner_1)$, $q_1' = On(AirConditioner_2)$, and it might be such that the instantiation process is done with respect to the same individual. For example, qualities $q_1 = On(AirConditioner_1)$ and $q_1' = On(AirConditioner_1)$ can exist independently. This is possible because qualities are time-dependent, and are associated with time-slices Δt. In the previous example, q_1 and q_1' have different time-slices, Δt_1 and $\Delta t_1'$, which must be mutually exclusive. In this context, q_1 and q_1' give us the following information about the model: the individual $AirConditioner_1$ has property On at different time intervals, Δt_1 and $\Delta t_1'$, which cannot overlap. As a result, there is an interval (possibly empty) between Δt_1 and $\Delta t_1'$, during which $AirConditioner_1$ does not have the On property.

2.3 Actions

Qualities are time-dependent: they are created and ceased at particular moments of time. Actions are the constructors and destructors of qualities. While qualities represent temporal intervals, actions define instantaneous, external events that change the state of the model. For instance, an action $a_1 = turnOn(AirConditioner_1)$ might terminate a quality $Off(AirConditioner_1)$ and create a new quality $q = On(AirConditioner_1)$. The time-slice Δt associated to q starts from the moment a_1 is executed, and will end when another action will terminate it.

An action has associated preconditions and effects. Preconditions refer to the existence of certain qualities, and allows the enforcement of temporal constraints between qualities. The constraints are based on Region Connection Calculus (RCC) relations [9]. Examples of constraints are given in Table 1.

2.4 The hypergraph

In order to establish an ordering of qualities based on their associated time-slices, a hypergraph $H = (A, T, E_q, E_a)$ is used. The hypergraph consists of a set T of temporal nodes, that represents symbolic moments of time. A refers to the set of action nodes. A subset of A, $\{a_1, a_2, \ldots a_k\}$ is enclosed by a temporal node t, if all actions $a_1, a_2, \ldots a_k$ occur at moment t. E_q represents the set of edges corresponding to qualities. Such edges span action nodes and stand for time-slices of qualities. In the previous example, a quality $q = On(AirConditioner_1)$ will span actions $a_1 = turnOn(AirConditioner_1)$ and $a_2 = turnOff(AirConditioner_2)$. Its time-slice Δt is delimited by the temporal nodes that contain actions a_1 and a_2. The E_a set stands for edges that denote preconditions of actions. They connect qualities q_i with actions that require the presence of q_i in order to be executed. As the model unfolds, a hypergraph H changes: new temporal nodes, action nodes and quality edges are added. When an action is signalled, its preconditions are searched for in the hypergraph, as part of a pattern matching mechanism.

Table 1. Examples of temporal primitives

Temporal Primitive	Associated RCC relation
q_1 *after* q_2	X DC Y (X and Y are disconnected)
q_1 *just_after* q_2	X PO Y (X and Y are partially overlapping)
q_1 *during* q_2	X NTPP Y (X is non-tangential proper part of Y)

2.5 A language based on fluid qualities

Based on the model briefly described above, a declarative language (DL) for modeling time-dependent applications is introduced. The basis of this language is the rule definition, as shown in Listing 1.

Listing 1. Rule definition template

```
   <precondition_1> , ..., <precondition_n>
=>
   <effect_1>, ..., <effect_n>
```

Rules allow the definition of actions in terms of their preconditions and effects. In order to check that preconditions are satisfied, the hypergraph is explored, and proper bindings between variables and qualities are done. A rule can be applied onto a part or the entire set of matched qualities in the preconditions. As described above, effects consist of the termination of qualities as well as the creation of new qualities. Regarding the rule in Listing 2, when a signal `turnOn(dev)` is received, where `dev` refers to an individual that possesses the qualities `Device` and `Off`, the action is executed, and its effects are introduced in the hypergraph: the quality `Off(dev)` is destroyed and a new quality `On(dev)` is created.

Listing 2. The `turnOn` action

```
Device(?d), Off(?d) as off
=> destroy off, On(?d)
```

3 Integration in a SOA environment

3.1 The integration problem

In order to integrate an interpreter for DL in a SOA environment, the basic entities that populate DL, i.e., individuals, qualities and actions, must be mapped onto SOA entities: Web-Services, Web-Service operations and invocations. The particularity of our setting, as mentioned in Section 1, is the ability of controlling devices by means of Web-Service invocations. As a result, individuals from DL could be mapped to Web-Services. Nevertheless, the mapping is not one-to-one. In the action `setTemperature(ac,25deg)`, `ac` and `25deg` are individuals, but only `ac` could be associated with a service that controls an air conditioning device. Assuming that the action `setTemperature` is mapped on a service invocation, `25deg` would be then associated to a parameter of `setTemperature`. Disambiguating between individuals that stand for services, and individuals that stand for operation parameters is not trivial. In the following we introduce a DL model for simple devices, as well as a service-oriented environment for controlling them. These will create a proper setting for describing two mapping alternatives.

3.2 A DL model and service-oriented environment

Assume the definitions from Listing 3, that establish the basis of a DL model. The basic entities are introduced: individuals and qualities that represent device states. Also, notice a binding between individuals and services. This will be explored later.

Listing 3. Basic entities

```
individual
        ac1 bound_to ACService1,
        ac2 bound_to ACService2,
        pc bound_to PCService,
        proj bound_to ProjectorService

AC(ac1), AC(ac2), Computer(pc),
Projector(proj)

quality On(?d) requires Device(?d)
quality Off(?d) requires Device(?d)
```

A definition of action `turnOn` has already been given in Listing 2. We will consider such a definition as part of our model. Also, the action `turnOff` can be similarly defined.

In Listing 4, qualities that are specific to air conditioning devices are introduced. Notice the presence of qualities `ModifiesTemp` and `HasCrtTemp`. `ModifiesTemp` refers to an air conditioner that currently cools or heats an environment, with a predefined temperature, while `HasCrtTemp` simply refers to an air conditioner's temperature setting. As a result, if `HasCrtTemp(ac1, 20deg)` holds, then the device `ac1` is set to cool a room to 20 degrees. Nevertheless, `ac1` might be turned off, therefore it might not actually perform cooling. If turned on, `ac1` will be enrolled in `ModifiesTemp(ac1, 20deg)`, and actually cool the room. The model simulates such common behavior of air conditioners.

Listing 4. Modeling air conditioners

```
quality ModifiesTemp(?ac, ?t)
    requires AC(?ac), Temperature(?t)
quality HasCrtTemp(?ac, ?t)
    requires AC(?ac), Temperature(?t)

HasCrtTemp(ac1, 20deg),
HasCrtTemp(ac2, 20deg)

action setTemp(?ac, ?t):
    AC(?ac), Temperature(?t),
    ModifiesTemp(?ac, _) as mt,
    HasCrtTemp(?ac, _) as ht
=>
    destroy mt, destroy ht,
    ModifiesTemp(?ac, ?t),
    HasCrtTemp(?ac, ?t)

action turnOff(?ac):
    AC(?ac), ModifiesTemp(?ac, _) as mt
=>
    destroy mt

action turnOn(?ac):
    AC(?ac), HasCrtTemp(?ac, ?t) as mt
=>
    ModifiesTemp(?ac, ?t)
```

Assume also the service definitions from Listing 5, each with their operations and parameters. One can notice that each service has particular definitions for starting and stopping. Also, multiple services (in particular, air conditioners) have identical interfaces. From the entire set of actions, we are especially interested (at the interface level) in the ones that map to external signals, i.e. to web service operations: `isNotAvailableSignal`, `turnOn`, `turnOff`, `setTemp`. As mentioned before, each individual maps to a service, and each action instance maps to a service invocation. If `ac1` is mapped on `ACService1`, then `turnOn(ac1)` maps on `ACService1.command("on")`. Nonetheless, actions are defined in a generic fashion, and for generic individuals, while services individually define their specific operations. For example, in Listing 2, when defining `turnOn(?d)`, `?d` stands for a variable that can hold any device individual. On the other hand, each service has its particular operation for turning on. For instance, `turnOn(x)` maps on `ACService1.command("on")` if `x` is an air conditioner. But, if `x` is a projector, then `turnOn(x)` maps on `ProjectorService.turnOn()`. It is therefore necessary to create a service hierarchy, where such particular mappings can be specified.

Listing 5. Service environment

```
ACService1 & ACService2
   command(cmd), cmd in {"on", "off"}
   setTemperature(tval),
      tval in {16deg  - 30deg}
   isAvailable()

ProjectorService
   turnOn()
   turnOff()
   isAvailable()

PCService
   start(true)
   stop(true)
   isAvailable()
```

3.3 An alternative based on service taxonomy

Assume the service hierarchy defined in Listing 6. Each service and its operations must be mapped to the hierarchy, in a manner illustrated in Listing 7. Also, each individual from the model must be mapped to a service, and each action must be mapped on the same hierarchy. The mapping of individuals can be seen in Listing 3, and was discussed in Section 3.2. The mapping of actions is done according to Listing 8. All connections between elements from DL and services are depicted in Fig. 1.

Listing 6. Service hierarchy

```
Device
   On()
   Off()
   isAvailable()

AirConditioner extends Device
   setTemperature(val)

Projector extends Device
Computer extends Device
```

Listing 7. Service mapping

```
ACService1 implements AirConditioner where
    ACService1.command("on") =
         AirConditioner.On()
    ACService1.command("on") =
         AirConditioner.Off()
    ACService1.setTemperature(val) =
         AirConditioner.setTemperature(val)

ProjectorService implements Projector where
    ProjectorService.turnOn() = Projector.On()
    ProjectorService.turnOff() = Projector.Off()

PCService implements Computer where
   PCService.start(true) = Computer.On()
   PCService.stop(true) = Computer.Off()
```

Listing 8. Action mapping

```
turnOn(?x) bound_to On, ?x bound_to Device
setTemp (?x, ?t) bound_to setTemperature,
              ?x bound to AirConditioner,
              ?t bound to val
```

A dedicated mechanism for translating external values to predefined individuals will be used in the mapping process. Similarly, a mechanism for handling valid range values will be used.

The mapping process progresses through the following steps: (1) when a signal `turnOn(ac1)` is generated, the action-to-service-hierarchy binding is inspected and, as a result, an abstract operation `Device.On()` must be called; (2) the individual-to-service binding identifies `ac1` as being `ACService1` which, according to the service-to-service-mapping, is an `AirConditioner`. But `AirConditioner` extends `Device`, so, `ACService1` is a `Device`; then (3) the abstract operation `Device.On()` can be transformed into `AirConditioner.On()`; finally (4) `ACService1` implements `On()` as:

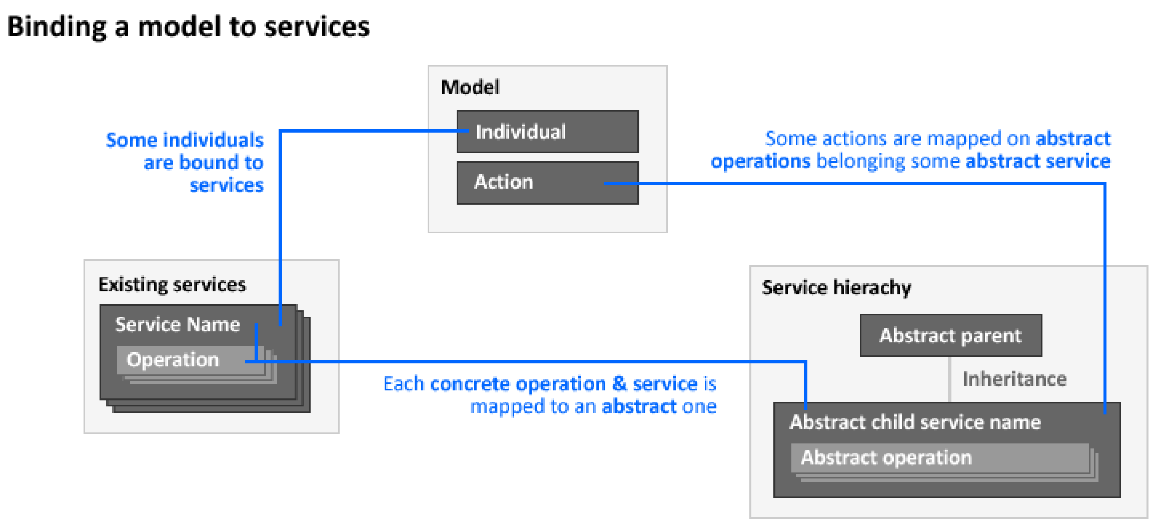
Binding a model to services
Model
Individual
Action
Some individuals are bound to services
Some actions are mapped on abstract operations belonging some abstract service
Existing services
Service Name
Operation
Each concrete operation & service is mapped to an abstract one
Service hierachy
Abstract parent
Inheritance
Abstract child service name
Abstract operation

Figure 1. Binding elements from DL to services

`ACService1.command("on")`. As a result, the operation `command("on")` must be invoked on `ACService1`. The entire process of transforming an action execution to a service call is described in Fig. 2.

3.4 An alternative based on direct bindings

The approach described in Section 3.3 has the following main disadvantage: a hybrid structure such as the service hierarchy, neither entirely part of the DL language, nor of services, must be maintained. Each device manufacturer must bind services to this hierarchy, and each modeler must bind his defined actions to the same hierarchy. The service hierarchy could be eliminated, if, instead of binding actions to services, we reverse the mapping order and bind service operations to actions. Again, assume services and operations described in Listing 5 as well as the existence of a model containing action definitions: `turnOn(?x)`, `setTemperature(?x, ?t)`. The assumptions are important. They reveal a limitation in this approach, as bindings can only be done *after* model definition, and knowing the available service implementations. In this context, individuals are bound to services, in a manner similar to the one above. Such a binding is illustrated in Listing 9.

Listing 9. Individuals are bound to services

```
individual
        ac1 bound_to ACService1,
        ac2 bound_to ACService2,
        pc bound_to PCService,
        proj bound_to ProjectorService
```

We now introduce a service-to-action binding, that can be seen in Listing 10.

Listing 10. Direct operation to action binding

```
ACService1.command("on") = turnOn(ac1)
ACService1.command("off") = turnOff(ac1)
ACService2.command("on") = turnOn(ac2)
ACService2.command("off") = turnOn(ac2)
ACService1.setTemperature(t) = setTemp(ac1,t)
ACService2.setTemperature(t) = setTemp(ac2,t)
```

Notice that, for each possible invocation, an action execution scenario is defined. In order for a binding to be correct, it must be complete i.e., cover the entire range of individuals that can be enrolled in such an action. As a result, when an action `turnOn(ac1)` is executed by the model, the interpreter checks the binding list for the proper invocation. When found, the service is called. In contrast, the execution of action `turnOn(i)` would generate a runtime error, as `turnOn(i)` is not bound to an invocation. Notice that, in this case, the binding of individuals to services becomes redundant. One pitfall of this method resides in the great number of manual bindings that must be created. In the first approach, a single action-to-operation binding was required. Also, bindings have to be done after developing the model, as mentioned above. Since the intent is to bind services to actions, actions must be developed first, and only then bound to services. The first approach suffers from this pitfall too. While the service hierarchy is fixed, and each manufacturer binds his services to the service hierarchy, the modeler still has to bind his actions to abstract operations from the hierarchy. This issue might create problems in dynamic environments, since pre-configurations would be required. The DL interpreter would be functional only after the proper bindings are created.

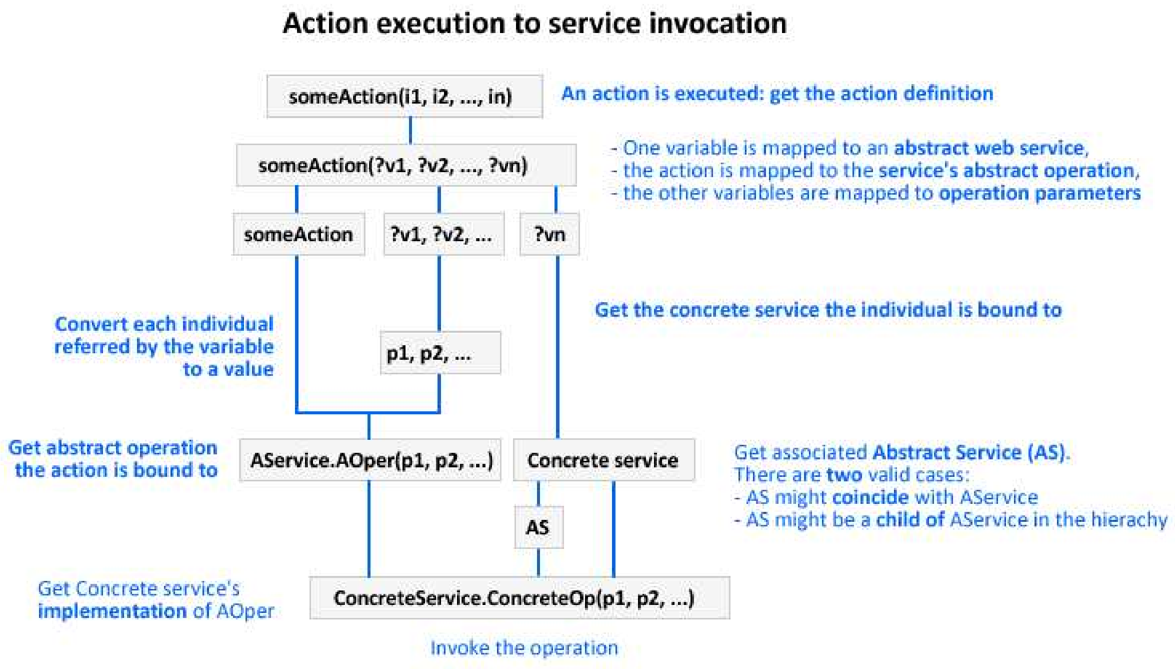
Action execution to service invocation
someAction(i1, i2, ..., in)
An action is executed: get the action definition
someAction(?v1, ?v2, ..., ?vn)
- One variable is mapped to an abstract web service,
- the action is mapped to the service's abstract operation,
- the other variables are mapped to operation parameters
someAction
?v1, ?v2, ...
?vn
Get the concrete service the individual is bound to
Convert each individual referred by the variable to a value
p1, p2, ...
Get abstract operation the action is bound to
AService.AOper(p1, p2, ...)
Concrete service
Get associated Abstract Service (AS).
There are two valid cases:
- AS might coincide with AService
- AS might be a child of AService in the hierachy
AS
Get Concrete service's implementation of AOper
ConcreteService.ConcreteOp(p1, p2, ...)
Invoke the operation

Figure 2. Mapping steps

4 A case study

In the following, we illustrate a case study for intelligent device control, based on our modeling method, web services, and the proposed integration mechanisms.

The setting consists of: (1) the two previously described web services for air conditioners, `ACService1` and `ACService2` and (2) two web services, `WinService1` and `WinService2`, connected to sensors able to detect whether a window was opened or shut. `WinService1` and `WinService2` implement Web Service Notifications (WSN) [10], that are generated by the change of window state.

In Listing 11, a simple model for air conditioner behavior is shown. It enforces the stopping of air conditioners, when windows are opened. The actions `openWindow(?w)` and `shutWindow(?w)` are mapped on the notifications of `WinService1` and `WinService2`, in a manner similar to the one described in Section 3.3. They can only be triggered externally, by any change of the window state.

Listing 11. A model for air conditioners

```
individual
    w1 bound_to WinService1
    w2 bound_to WinService2

quality WasOn(?d) requires AC(?d)
quality Shut(?w), Opened(?w)

action openWindow(?w):
        Shut(?w) as shut
=>
        Opened(?w), destroy shut

action shutWindow(?w)
        Opened(?w) as opened
=>
        Shut(?w), destroy opened

action openWindow(?w):
        On(?d) as on, AC(?d)
=>
        WasOn(?d), turnOff(?d)

action shutWindow(?w):
    WasOn(?ac) as wasOn
    ~Opened(?x), ?x != ?w
=>
    turnOn(?ac), destroy wasOn
```

A particularity for this model is the *overloading* of actions `openWindow` and `shutWindow`. In different contexts, they have different behavior.

When window `w1` is opened, its sensor detects the state change. This results in a notification generated by `WinService1`, that is further translated to the action `openWindow(w1)`. Both action rules will have their preconditions satisfied. The first action creates the quality `Opened(w1)`. Notice the preconditions of the second action `openWindow`, and the usage of the variable `?d` which isn't mapped on a particular device individual. The preconditions `On(?d) as on, AC(?d)` will match *all* individuals `?d`, modeling turned on air conditioners. Similarly, the effects `WasOn(?d), turnOff(?d)` will act upon all such individuals. They will acquire qualities `WasOn`, and actions `turnOff` will be signalled for each of them.

If another window `w2` is opened, it will also acquire a quality `Opened(w2)`, but, since all air conditioners are off, the precondition `On(?d) as on, AC(?d)` is no longer satisfied, and the second action rule definition will not execute. Similarly, when actions `shutWindow` occur, the windows change state. When the last window is closed, all air conditioners that are stopped, but were previously on, will be started.

The case study, implementation and testing were developed as part of the FCINT project (*Framework for service composition based on ontologies for the aggregation of knowledge and information for intelligent buildings*, POSCCE ID: 551).

5 Related Work

In the literature, there are numerous methods that allow temporal representation as well as reasoning. Nevertheless, such efforts focus more on formal specification and less on actual implementations.

In [11], a declarative language for specifying web service composition is introduced. Similar to our approach and to the application domain of our modeling method, [11] aims at declaratively specifying a business process. The solution is based on Linear Temporal Logic (LTL). Using LTL formulas, temporal constraints between service invocations are specified. They replace conventional control-flows specific to imperative languages. Constraint specification has several particularities: during the execution of the process, some constraints may not be satisfied, as not all services have been invoked. However, when the process ends, all constraints must be satisfied. This approach addresses temporality, but still in a static way. A declarative specification from [11] is an actual plan, a predetermined workflow. In our approach, there is no precise workflow to be executed. Depending on external stimuli (actions), qualities are asserted and new actions are triggered.

In terms of ontologies, OWL [1] is considered to be one of the most prominent ontology modeling languages. A popular variant of OWL, OWL-DL, is based on a restricted but decidable fragment of Description Logics. Description Logics (and inherently OWL) focus on representing a given state-of-the-world, using primitives such as individuals, concepts, and relations. A framework such as this does not directly address time. Nevertheless, attempts to incorporate temporality exist, and we distinguish between two directions: (1) building time-related concepts on top of existing primitives, thus creating ontologies able to provide some temporal reasoning, and (2) extending the modeling approach with new primitives related to time. An example of (1) is OWL-Time [2], where concepts such as *TemporalUnit* and *Date* are defined. These approaches suffer from the following pitfalls: complex temporal ontologies are difficult to develop, reasoning is often intractable, and most important, evolution cannot be represented. Individuals are inherently and permanently bound to concepts or relations, and their enrollment cannot be changed. Such ontologies can be useful for disambiguation of some time-dependent knowledge, but are unable to model the evolution of real-world processes.

An example of (2) are Temporal extensions of Description Logics, described in [12]. They increase the dimensionality of the representation, by adding a new temporal component. As a result, in these frameworks, instances of a concept are seen as pairs consisting of individuals and the intervals on which the they are enrolled in the concept. The extension of a concept becomes a Cartesian product between individuals and intervals. This approach makes both model creation as well as reasoning computationally difficult.

6 Conclusions and Future Work

Both alternatives described above have the following particularity: in order for models to be successfully integrated in particular environments, static mappings have to be performed. Using direct mappings, translating actions to service invocations proves computationally inexpensive. It nevertheless requires a great amount of data to be manually introduced. Using a service taxonomy greatly simplifies the task, but requires a specialized translation component, to perform the steps depicted in Fig. 2. Such component can make the entire framework too complex. Nevertheless, in distributed environments, this approach allows model migration. Therefore, models designed for particular environments can be integrated in other environments without changes. As each web service is bound to the service taxonomy, models are completely independent of actual implementations. In such a setting, mechanisms for establishing whether a model is compatible with a particular service environment, are required.

Also, it is important to notice that, no proposed alternative makes the translation process user-independent. As a result, the solution can be extended with user-oriented aiding mechanisms for action-to-service mapping. Such mechanisms can inspect models, and give suggestions about possible mappings, using certain specific heuristics.

To conclude, we emphasize the suitability of both proposed approaches in different environments, depending on available resources, model design restrictions and other environment-related features.

7 Acknowledgment

The research presented in this paper is supported by national project: "TRANSYS Models and Techniques for Traffic Optimizing in Urban Environments", Contract

No. 4/28.07.2010, Project CNCSIS-PN-II-RU-PD ID: 238. The work has been co-funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/62557.

References

[1] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *Web Semant.*, 6:309–322, November 2008.

[2] Feng Pan. *An Ontology of Time: Representing Complex Temporal Phenomena for the Semantic Web and Natural Language*. VDM Verlag, Saarbrücken, Germany, Germany, 2009.

[3] Barry Norton, Simon Foster, and Andrew Hughes. A compositional operational semantics for owl-s. In Mario Bravetti, Leila Kloul, and Gianluigi Zavattaro, editors, *Formal Techniques for Computer Systems and Business Processes*, volume 3670 of *Lecture Notes in Computer Science*, pages 303–317. Springer Berlin / Heidelberg, 2005.

[4] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, 1 edition, August 2007.

[5] Panagiotis Louridas. Orchestrating Web Services with BPEL. *IEEE Software*, 25:85–87, 2008.

[6] C. Giumale and L. Negreanu. Reasoning with fluid qualities. *CSCS-17, 17th international conference on control systems and Computer Science*, 2:197–203, December 2009.

[7] C. Giumale, L. Negreanu, M. Muraru, and M. Popovici. Modeling Ontologies for Time-Dependent Applications. In *Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 202–208, Los Alamitos, CA, 2011. IEEE Computer Society.

[8] Dirk Fahland, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus imperative process modeling languages: The issue of maintainability. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann, editors, *Business Process Management Workshops*, volume 43 of *Lecture Notes in Business Information Processing*, pages 477–488. Springer Berlin Heidelberg, 2010.

[9] Sanjiang Li and Mingsheng Ying. Region connection calculus: its models and composition table. *Artif. Intell.*, 145:121–146, April 2003.

[10] Steve Vinoski. More Web Services Notifications. *IEEE Internet Computing*, 8:90–93, 2004.

[11] M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *Towards a Truly Declarative Service Flow Language*, pages 1–23. Springer, 2006.

[12] Alessandro Artale and Enrico Franconi. A survey of temporal extensions of description logics. *Annals of Mathematics and Artificial Intelligence*, 30:171–210, March 2001.