A modeling method and declarative language for temporal reasoning based on fluid qualities

Matei Popovici, Mihnea Muraru, Alexandru Agache, Cristian Giumale, Lorina Negreanu, and Ciprian Dobre

POLITEHNICA University of Bucharest,
Splaiul Independentei nr. 313, sector 6, Bucuresti,
Postal Code: 060042
{pdmatei,mmihnea,alexandruag,cristian.giumale,
lorina.negreanu,cipsmm}@gmail.com

Abstract. Current knowledge representation mechanisms focus more on providing a static description of a modeled universe and less on capturing evolution. Ontology modeling languages, such as OWL, have no inherent means for describing time or time-dependent properties. In such settings, time is usually represented along with other application-dependent concepts, yielding complex models that are difficult to maintain, extend, and reason about. On the other hand, in imperative languages that allow the definition of time-dependent behavior and interactions such as WS-BPEL, the emphasis is on specifying the control flow in a service-oriented environment. In contrast, we argue that a declarative approach is more suitable. We propose a modeling method and a declarative language, designed for representing and reasoning about time-dependent properties. The method is applicable in areas such as ubiquitous computing, allowing the specification of intelligent device behaviour.

Keywords: temporal representation, temporal reasoning, hypergraph, declarative language

1 Introduction

Traditionally, ontologies provide static descriptions of a given domain of interest. That is the case of medical approaches such as [8] or semantic lexicons such as WordNet [10]. Existing ontology modeling methods have no inherent means for representing evolution. Also, languages such as the Ontology Web Language (OWL) [5] lack the same feature: they have no dedicated constructs that accommodate time or temporal properties. The reasoning process is focused on taking a snapshot of the modeled universe that includes concepts, relations and individuals, and derive additional properties such as concept subsumption and satisfiability [5].

In many cases temporal concepts can be defined on top of existing modeling primitives, such as in OWL-Time [12], and thus provide a high-level modeling

layer. Time-related concepts reside on the same representational level with other application-specific concepts. Creating such models proves difficult, lacks scalability, and makes the reasoning process computationally hard. When modeling real-life behavior, ontologies are challenged to accommodate temporal evolution. For example, concepts such as *married two times* or *widower* are inherently dependent on properties that held in the past, but are no longer valid presently. Dedicated mechanisms for representing and reasoning about time-dependent knowledge are required in this case.

We propose an ontology-based modeling method for time-dependent applications that accomplishes these goals. Our approach takes a static hierarchy of concepts and relations between concepts (defined in a manner similar to the one used in OWL), and extends it with a structure able to represent time and change. Temporal elements are no longer represented on the application layer; they become modeling primitives. Consequently, the method provides *native* means for representing temporality. Unlike conventional ontologies, where the instantiation relationship between an individual and a concept is unique, our approach allows for multiple instantiations with respect to the same concept-individual pair, at different moments of time. This allows individuals to be enrolled in concepts such as *single, but married two times in the past*, by simply exploring past *marriage* instantiations with respect to the same individual. All time-dependent instances are stored in a dedicated structure designed to preserve temporal order. We call such a structure a *hypergraph*. Temporal reasoning is accomplished by exploring the hypergraph.

The paper is structured as follows: Section 2 describes our modeling approach based on individuals, qualities, actions, as well as the structure responsible for storing the ordering of concept instances: the hypergraph. Section 3 introduces a language that allows both static and time-dependent modeling. Some similarities and differences with respect to other declarative languages, such as CLIPS [11] and Prolog [7], are discussed. In Section 4, a case study for intelligent device behavior in an ubiquitous environment [3] is described. Section 5 compares our method with other approaches. Section 6 presents conclusions, as well as future work.

2 A modeling method based on fluid qualities

2.1 Modeling primitives

Individuals. Concepts, together with concept instances, are useful for defining a static universe of discourse. As an alternative to such traditional representations, our framework relies on a modeling approach based on individuals, fluid qualities and actions, introduced in [4]. Individuals are atomic entities, identifiable by themselves. They are *perennial*: during the evolution of a model, individuals do not disappear or suffer structural changes. Individuals can however acquire or lose qualities.

Qualities. A (fluid) quality $Q(i_1, \ldots, i_n)$ or $Q(\bar{i})$ represents a time-dependent n-ary relationship between individuals $\bar{i} = (i_1, \ldots, i_n)$. A unary quality $Q(i)$

stands for a time-dependent property associated with individual i. Qualities hold on specific time-intervals Δt. Δt denotes the time-slice associated with $Q(\overline{i})$. Qualities are created by instantiating a quality prototype $Q(\overline{x})$. Here, $\overline{x}$ represents variables. We say that a quality was destroyed if its time-slice ended at a certain moment in time. Once introduced in a model, a quality is never completely erased. Qualities have similarities to property instances from OWL [5]. One major difference is that qualities are time-dependent and thus need not be unique. As a result, if an individual i was enrolled in Q in the past, then $Q(i)$ has a time-slice Δt associated with an interval from the past. If i is enrolled in Q once more, then a new quality $Q(i)$, with a different time-slice, will be introduced. The time-slices of the two quality instances cannot overlap. For example, *Married(John)* can be present several times in a model that describes *John*'s marriages. In a similar way, qualities such as *On(AirConditioner)* can be present at different moments, in a model for home devices. Quality prototypes are equivalent to concepts and properties from conventional ontologies. They allow the instantiation of qualities, with respect to certain individuals. Quality prototypes can form hierarchies. We use the meta-relationship *is-a* to build the hierarchy of quality prototypes. If $Q_1(\overline{x})$ *is-a* $Q_2(\overline{y})$, then all individuals enrolled in a quality Q_1 over a specific time slice Δt must also be enrolled in Q_2, as well as in all other qualities enforced by Q_2, through the chain of its existing *is-a* meta-relations. For example, assume the following quality prototype definition: *AirConditioner(x) is-a Device(x)*. In this case, if an individual possesses the quality *AirConditioner*, then it is automatically enrolled in the quality *Device*.

Actions. An action $a(\overline{i})$ represents an external stimulus that changes the state of the modeled universe. Actions can enrol one or several individuals, and can require the presence or absence of particular qualities. When actions occur, they produce side-effects: the creation and/or destruction of qualities. From this point of view, actions can be considered constructors and destructors for qualities. With respect to time, actions are instantaneous: they occur at a particular moment and have no duration. For instance, in a model describing marital evolution, where the qualities $q_j = Single(John)$ and $q_a = Single(Anne)$ hold, executing the action $marries(John, Anne)$ will terminate the qualities q_j and q_a and create a new quality $married(John, Anne)$.

2.2 Representing time

The hypergraph. The evolution of a model is encoded in a structure $H = (A, T, E_q, E_a)$, where H is an oriented, acyclic graph. A is the set of action instances, T is the set of temporal nodes, E_q is the set of qualities and E_a is the set of preconditions.

Time is represented using actions and qualities. A moment in time is defined as a set of actions $t_a = \{a_1, \ldots, a_k\}$ that occur simultaneously. $t_a \in T$ is a temporal node in a hypergraph. Action nodes are depicted in white in Fig. 1. Temporal nodes are shown in grey, and contain action nodes. There is a distinguished temporal hypernode, $Init$, that refers to the starting moment of the modeled application. It contains an implicit action denoted by a_{init}, which is

the constructor for all initial qualities in the model. Similarly, the *Current* hypernode denotes the current moment in the unfolding of the model evolution. It also has an implicit $a_{current}$ action, that is considered to be a pseudo-destructor of all existent qualities.

Time intervals are defined by pairs of actions (not necessarily consecutive). For example assume an action $a_1(\overline{x}')$ is executed. As a result, a certain quality $Q(\overline{x})$ is introduced. Assume also that another action $a_2(\overline{x}'')$ destroys $Q(\overline{x})$. Then, if t_1 is a temporal node such that $a_1 \in t_1$ and similarly $a_2 \in t_2$, then the time-slice of $Q(\overline{x})$ is $\Delta t = [t_1, t_2]$. This duration is represented as an edge between action nodes a_1 and a_2. If a_2 happens to be $a_{current}$, it means that the quality holds at the present moment. Notice that different qualities might have identical durations, as a result of being created (and destroyed) by actions that occur simultaneously. Also, it might be that multiple qualities are created/destroyed by the same action.

Temporal nodes need not have specific values. In some applications, time is relative and the focus is on event ordering only. In these situations, temporal nodes are symbolic. In cases where measurements of time are important, temporal nodes can be assigned actual timestamps. Depending on the desired precision, timestamps can encode minutes, seconds, miliseconds etc.

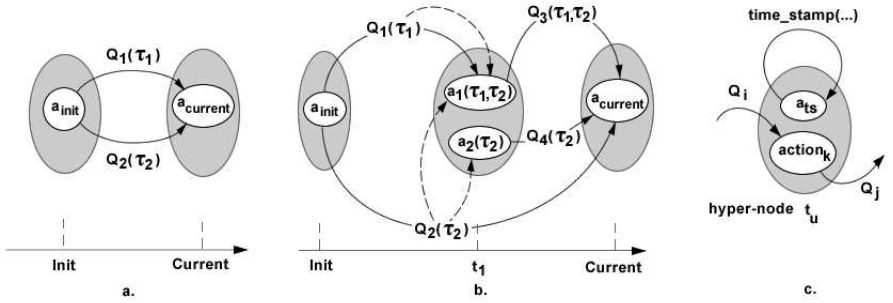

Fig. 1. The hypergraph

The E_q set contains edges that stand for n-ary qualities $Q(\overline{x})$. These edges span action nodes a_{start}, $a_{end} \in A$ that construct and destroy $Q(\overline{x})$, respectively. Edges from E_a are shown as solid arrows, in Fig. 1. The E_a set contains directed edges that designate preconditions of action nodes from A. An edge $e = (q, a)$ from quality q to action a designates q as a satisfied precondition of a. e actually connects a quality edge to an action node. Precondition edges are shown as dotted arrows in Fig. 1.

Initially, H contains only *Init* and *Current*, and all predefined qualities span action nodes a_{init} and $a_{current}$. H changes, as new stimuli are recorded by the model. If an action (or a set of actions) is executed, and the required precondi-

tions, according to the action's prototype, hold, then: (1) a new temporal node is created, and inserted in H, just before $Current$; (2) the temporal node is populated with all valid actions that are executed; (3) for each satisfied precondition, a dedicated edge between a required quality and the current action is added to E_a; (4a) if an action a destroys a quality, then its ending action node is modified from $a_{current}$ to a; (4b) if an action creates a new quality q, then a new edge $(a, a_{current})$ corresponding to q, is added to E_q.

As an example, consider the hypergraph from Fig. 1(a). It contains two qualities $q_1 = Q_1(\tau_1)$ and $q_2 = Q_2(\tau_2)$ that hold at the current moment of time. In this scenario, actions $a_1(\tau_1, \tau_2)$ and $a_2(\tau_2)$ are signalled. $a_1(\tau_1, \tau_2)$ requires the presence of q_1 and q_2, and $a_2(\tau_2)$ requires q_2. These dependencies are shown with dotted edges. Since all preconditions are satisfied, the action is executed. The effects can be seen in Fig. 1(b). a_1 will terminate q_1, and create a new quality $q_3 = Q_3(\tau_1, \tau_2)$. a_2 does not terminate any quality, but creates $q_4(\tau_2)$.

Temporal primitives. In the above examples, actions are solely conditioned by the existence of qualities at the current moment of time. In this context, the hypergraph performs as a structured log for recording events. In the following, we introduce temporal primitives as a mechanism for creating complex, temporal-based constraints on action execution. A temporal primitive enrolls two qualities, and enforces a certain temporal relation between them. In our model we use the Region Connection Calculus (RCC) relations to express temporal primitives. They are fully described in [9]. We shall not review the entire set of temporal primitives and their associated RCC relations. Instead, we will focus on some relations such as the ones shown in Table 1. These relations, as well as their inverses will be used in the following sections. For example, a possible precondition associated to an action $a(\overline{x})$ is the existence, somewhere in the past, of qualities $q_1 = Q_1(\overline{x}')$ and $q_2 = Q_2(\overline{x}'')$ such that the constraint: q_1 $just\_after$ q_2 is satisfied. A hypergraph exploration will attempt to find instances q_1 and q_2 that satisfy the $just\_after$ constraint. If such instances are found, action a is executed, along with the entire process of edge and node insertions described previously.

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)

Table 1: Examples of temporal primitives

3 A declarative language based on fluid qualities

3.1 Motivation

Assume an application that models devices in a intelligent house scenario. We consider a simple model with two devices A and B. The device A can be turned

on if both devices A and B are off. The device B can be turned on if: (1) it is off and (2) the device A has been off for a time period of at least T seconds.

In conventional programming or ontology-based languages, the program or ontology encoding the above scenario must contain specific data structures and handling mechanisms, for keeping track of time intervals associated to events. Even if these data-structures and functions are implemented as libraries (in the case of programming languages), or as time-related concepts (in the case of ontology languages) the resulting program or ontology will be larger, more cumbersome to write, modify and understand. In most existing approaches, a model for the above scenario should encode a potentially large finite-state automaton that contains states for all possible ways of turning on the two devices. Alternatively, a declarative language enhanced with temporal primitives would explicitly convey the semantic content of the model. In the following, we use the modeling method described in Section 2 in order to introduce a declarative language for time-dependent applications.

3.2 Defining quality prototypes

Assume the definitions from Program 1. Variables are designated using the symbol "?", in a manner similar to that in CLIPS [11].

```
individual ac, p
quality Device(?d), HasPower(?d,?val), AirConditioner(?d) is-a Device(?d)
```

Program 1: Individuals and qualities

The simple model from Program 1 introduces two simple 1-ary quality prototypes: `Device`, and `AirConditioner`, and a binary quality prototype (or relation): `HasPower`.

3.3 Rules

Rules are the basis for defining actions, their preconditions as well as effects. A simple rule, such as the one from Program 2, can be read in the following way: *In order to execute action* turnOn*, the individual* ?x *must be a device, and also off. If these preconditions are satisfied and the action is signalled from the external environment, then the quality* Off *will be destroyed, and* ?x *will acquire the quality* On. Program 2(a) has similarities with rule definitions from declarative languages such as CLIPS. A possible translation of Program 2(a) in CLIPS is shown in Program 2(b).

Notice that CLIPS facts have been used to model qualities. Facts are also a means for representing actions. This appears to correspond to our modeling perspective: actions are external stimuli that, in temporal contexts where their preconditions hold, produce certain effects. Nevertheless, this assumption may

cause problems, due to the instantaneous nature of actions. The following question is raised: *When is an action fact retracted in CLIPS, during the rule firing cycle?*

```
rule start_device                    (defrule start_device
    preconditions: Device(?x),           (device ?x)
    Off(?x) as ?off                      ?off <- (off ?x)
    action: turnOn(?x)                   (turnOn ?x)
    effects: destroy ?off, On(?x)        => (retract ?off)
                                         (assert (on ?x)))
           (a)                                  (b)
```

Program 2: A simple turnOn rule

Take for instance the CLIPS rule `start_device` from Program 3, where the more general fact `(turnOnAll)` replaces the particular action fact `(turnOn ?x)`. We assume that `(turnOnAll)` is externally introduced, in order to trigger the turning on of all devices. Notice that the retraction of `(turnOnAll)` is essential. In its absence, a rule such as `invalid` would be incorrectly executed, although its preconditions did not hold at the particular moment when `(turnOnAll)` was signalled. This happens because (1) the effects of the rule `multiple_start_device`, more precisely the assertion of `(on ?x)`, validate the preconditions of `invalid` and (2) the action fact `(turnOnAll)` is not removed, as it should.

```
(defrule multiple_start_device       (defrule invalid
        (device ?x)                          (on ?x)
        ?off <- (off ?x)                     (turnOnAll)
        ?all <- (turnOnAll)                  =>
        =>                                   ...)
        (retract ?off)
        (retract ?all)
        (assert (on ?x)))
```

Program 3: Rule activation

However, retracting `(turnOnAll)` causes other problems. In CLIPS, an activation record for the rule `multiple_start_device` is created for a particular device in state off and selected in a nondeterministic fashion, from the set of stopped devices. When the rule is fired for that activation record, the fact `(turnOnAll)` is removed, thus inhibiting the turning on of other devices in state off. Since activation records are created in a sequential manner, and since, for each record, preconditions are checked, the solution is to mark the devices for which the rule is applicable, but defer carrying on the effects until after `(turnOnAll)` has been

retracted. This way, newly created qualities will not be able to erroneously trigger the execution of other rules, such as *invalid*, having unsatisfied preconditions at the expected moment. The instantaneous nature of actions can only be modeled by translating single action facts that affect multiple entities, to multiple action facts associated with single entities.

In Program 4, `(turnOnAll)` is translated to multiple, particular action facts `(turnOn ?x)`. The salience declaration gives this rule a higher priority. When all particular action facts have been generated, `(turnOnAll)` is removed by the rule `remove_turnOnAll`. By appending to Program 4 the rule `start_device` from Program 2(b), the effects of turning on all devices are added to the working memory and the desired behavior is finally obtained.

```
(defrule translate
        (declare (salience 10))
        (device ?x)
        ?off <- (off ?x)
        ?all <- (turnOnAll)
        =>
        (assert (turnOn ?x)))
```

```
(defrule remove_turnOnAll
        (declare (salience 1))
        ?all <- (turnOnAll)
         =>
         (retract ?all))
```

Program 4: Modeling simultaneous actions

It is easy to see that, in CLIPS, actions are difficult to define. In contrast, our approach makes a clear distinction between facts (or qualities) and actions. The latter are equivalent to signals that remain active throughout the rule execution cycle. As a result, the execution of one rule instance can affect other instances, by means of quality changes only. If translated in our language, Program 3 would execute for each device, as `(turnOnAll)` is an action and its removal would be handled by the interpreter, at the end of the rule execution cycle.

Returning to Program 2(b) notice that, in the absence of `(turnOn ?x)`, the CLIPS rule would be executed for each entity that is a device, and that is in state *Off*. As a result, all devices in state *Off* would be turned on. Our approach is essentially different: qualities (and actions) are generated by actions only. It implicitly means that there must be at least an initial action, i.e. an entry-point in the program, that starts the model interpretation. As a consequence, the rule *start_device* from Program 2(a), would not be executed each time preconditions hold, but only when a *turnOn* action is signalled.

3.4 Representing and computing values

Numeric values. Numeric values are represented using predefined individuals. Predefined qualities can be assigned to such individuals, in order to disambiguate their numeric types. `Double(1), Integer(1), Float(1)` are such qualities. Since the modeling method does not explicitly define a mechanism

for evaluating expressions, a special behavior is defined for numeric individuals. More precisely, their symbolic/textual representation encodes their actual value. For example, in `Double(1)`, `1` refers to a numeric individual that possesses the double value 1.0. Operators are defined using predefined actions. For instance, the arithmetic addition is an action that can be applied on, and compute, numeric individuals. The expression `?x = 2 + 3.5` is a shorthand for the execution of the special action `+(2,3.5)` that produces, as a side-effect, the binding of variable `?x` to individual `5.5`. The action `+` is responsible for type casts. Since both `Float(2)` and `Integer(2)` hold for `2`, `+` will correctly evaluate the above expression. The hierarchy of types is internally defined using the `is-a` relationship as in `Integer (?x) is-a Number(?x)`.

Time and durations. The modeling approach described in Section 2.2 assumes that moments of time, modeled as hypernodes, are symbolic. In many applications, moments of time require actual values, thus allowing the application to compute durations, and make decisions based on them. As a result, the predefined qualities `Day, Month, Year, Time, Date` are introduced. The individuals enrolled in these predefined qualities have specific identifiers. In some cases, these individuals can be *polymorphic*. For instance, `Year(2010)` and `Number(2010)` enroll the same individual, `2010`, that acts as both a year and a number. In these cases, qualities act as type casts that disambiguate the usage of an individual.

We introduce another predefined quality, `Timestamp`. It is associated automatically, by the model interpreter, with every hypernode from the hypergraph. To preserve uniformity, we consider that `Timestamp` is an edge having the same action as constructor and destructor. Its time span is zero, as seen in Fig. 1(c). As a result, timestamps can be used in preconditions, like any other quality. For instance, in `Temperature(?t) just_after Timestamp(08:00/1.1.2011)`, `?t` would hold the temperature value recorded in the hypergraph just after `08:00`, on the 1st of January. Also, primitives for selecting the timestamp of nodes are defined. These can be applied on end-points of quality edges. `current_moment()` returns the timestamp associated with the *Current* node in the hypergraph. Also, `quality_start_moment(q)` and `quality_end_moment(q)` returns the timestamps associated with a quality edge end-point.

In order to compute timestamps, an alternative inspired from Haskell and from the unification process in Prolog [7] is used. Assume that, in $Q(\overline{x})$, $\overline{x}$ refers to an enumeration of individuals and variables. An example is $Q(i_1, ?x, ?y, i_2, i_3)$. Also, if S is a substitution, i.e. a set of bindings of variables to individuals, we denote by $Q(\overline{x})/S$ the replacement of variables from $Q(\overline{x})$ with their respective individuals, according to S. The expression $Q(\overline{x})$ is $Q(\overline{y})$ produces the *unification* of the two qualities and, as a side-effect, the necessary variable bindings. For example, the unification from Program 5(a) solves the problem *what timestamp ?t/?d.?m.?y incremented by 2:00/2.0.0 yields 1:00/1.1.2011?*. Therefore, $S = \{?t = 23:00, ?d = 29, ?m = 12, ?y = 2010\}$ and `Timestamp(?t + 2:00, ?d + 2.?m.?y)` is `Timestamp(23:00/29.12.2010)`. There are cases when unifications can fail, as in Program 5(b). This happens because no such values can be assigned

to `?t, ?d, ?m`. Also, the are cases such as the one shown in Program 5(c), where the unification leaves unbound variables.

```
(a)     Timestamp(1:00/1.1.2011) is Timestamp(?t + 2:00, ?d + 2.?m.?y)
(b)     Timestamp(1:00/1.1.2011) is Timestamp(?t + 2:00, ?d + 2.?m.2011)
(c)     Timestamp(?x/?y.?z.2011) is Timestamp(?t + 2:00, ?d + 2.?m.2011)
```

Program 5: Timestamp unification

4 Case study: a model for intelligent buildings

In the following, we illustrate a model for intelligent device behavior, described using the declarative language introduced in Section 3. In order to be operational, the model relies on the following assumption: devices can be controlled individually using an uniform interface: web services. In this setting, actions are mapped on service invocations. For instance, in order to turn on device `A`, the invocation `AService.turnOn()` must be performed. In Program 6 we assume the existence of an individual `ac` and three qualities: `AirConditioner(ac)`, `CurrentTemp(20deg)` and `DesiredTemp(10deg)`.

```
rule get_current_temp
   preconditions: CurrentTemp(?x) as ?crt
   actions: newTemp(?y)
   effects: destroy ?crt, CurrentTemp(?y)

rule modify_temp
   preconditions:
        DesiredTemp(?x), ?x != ?y,
        AirConditioner(ac), Off(ac)
   actions: newTemp(?y)
   effects: turnOn(ac), setTemp(?x,ac)

rule stop_cooler
   preconditions:
        DesiredTemp(?x), ?x == ?y,
        AirConditioner(ac), On(ac)
   actions: newTemp(?y)
   effects: turnOff(ac)
```

Program 6: A model for air conditioners

We have omitted from Program 6 basic quality and action definitions such as `On`, `Off` and `turnOn`, `turnOff`, respectively. They were introduced previously in Section 3.3. The model from Program 6 connects a temperature sensor with an air conditioner. Whenever the sensor detects a temperature change, it generates an action `newTemp(?y)`. If the environment's temperature is different from the desired one, the air conditioner starts, and cools or heats with the specified temperature. Notice the presence of `Off(ac)` as a precondition in the rule

`modify_temp`. This prevents starting the air conditioner, if it is already on. Rule `stop_cooler` stops the device, once the desired temperature was reached.

As seen in Program 6, `newTemp` is defined in several, possibly overlapping contexts. While `stop_cooler` and `modify_temp` have mutually exclusive contexts, the rule `get_current_temp` and either of `stop_cooler` or `modify_temp` do not. As a result, the rule `get_current_temp` can be fired at the same time with `modify_temp`. In this particular case, the overlapping can be avoided by replacing rules `modify_temp` and `get_current_temp` with a more complex one. This would also eliminate the necessity of a `CurrentTemp` quality, but would make the model harder to read and to extend.

Program 6 is able to describe intelligent device behavior, but contains no temporal constraints. In order to further restrict the behavior of devices, the constraints from Program 7 may be used. Here, we assume that an external action introduces the qualities `CanicularDay(e)`, `Rains(e)`, where `e` refers to an individual representing the current environment. Program 7(a) turns on the air conditioners exactly an hour after the `CanicularDay` quality was created. Program 7(b) adds an additional constraint: the air conditioners will start in a canicular day only if current temperature reached or exceeded 35 degrees. Finally, Program 7(c) can be used to stop the air conditioners if, in a canicular day, it starts raining.

```
(a)    current_moment() is Timestamp(?t+1:00/?d.?m.?y),
       CanicularDay(e) just_after Timestamp(?t/?d.?m.?y)
(b)    current_moment() is Timestamp(?t+1:00/?d.?m.?y),
       CanicularDay(e) just_after Timestamp(?t/?d.?m.?y),
       Timestamp(?t/?d.?m.?y) during CurrentTemp(?temp), ?temp > 35deg
(c)    current_moment() is ?t+1:00/?d.?m.?y,
       CanicularDay(e) just_after Timestamp(?t/?d.?m.?y),
       CanicularDay(e) just_before Rains(e)
```

Program 7: Temporal constraints

More complex models, conceptually similar to Programs 6, 7 have been simulated and tested using COOL [6], an object oriented extension for CLIPS. A translation mechanism, from facts to web service invocations and vice-versa was developed, in order to test real device behavior. COOL was especially useful for encoding the hypergraph. Actions were simulated using the method presented in Section 3.3.

5 Related Work

Most modeling methods aimed at temporal representation and reasoning focus more on formal specification, and less on actual implementations and computational effort. They are rather suitable for reasoning about a static dis-

course containing temporal information. Modeling time-dependent evolution is not straightforward in these approaches. It is the case of Description Logics (DL) and temporal extensions of DL. For instance, OWL [5], an ontology modeling language based on DL, focuses on representing a given state-of-the-world, using primitives such as: individuals, concepts, and properties. Here, time is not directly addressed. However, attempts to incorporate temporality exist, and we distinguish between two directions: (1) building time-related concepts on top of existing primitives, thus creating meta-ontologies able to provide some temporal reasoning, such as OWL-Time [12] and (2) extending the modeling approach with new primitives related to time. The former approach suffers from the following pitfalls: complex temporal ontologies are difficult to develop, reasoning is often intractable, and most important, evolution cannot be represented explicitly. Individuals are inherently (and permanently) bound to concepts or properties, and their enrollment cannot be changed. These ontologies are useful for the disambiguation of time-dependent information, but are unable to model the evolution of real-world processes. Temporal extensions of Description Logic's [1] are an example of the latter approach. They increase the dimensionality of the representation, by adding a new temporal component. As a result, in these settings, instances of a concept are seen as pairs consisting of individuals, and the intervals on which the they are enrolled in a particular concept. The extension of a concept becomes a Cartesian product between sets of individuals and sets of intervals. This approach makes model creation cumbersome and reasoning computationally difficult.

Another well known approach is the Temporal Logic of Intervals (TLI), proposed by Allen [2]. It introduces an interval ontology used for the representation of events, properties and temporal change. Allen defines seven basic relations (*before, meets, overlaps, starts, during, finishes, equal*) which, together with their inverses allow a complete characterisation of intervals. An implementation of TLI uses a graph-based algorithm for temporal reasoning [2]. In such a graph, nodes stand for intervals, and edges represent temporal relations between intervals. After the graph is built, temporal reasoning is done by exploring it and inferring new interval relations. Less importance is given to the dynamic nature of the discourse. In contrast, our approach doesn't solely focus on a mechanism for analysing temporal information, but provides means for modeling the desired evolution of a particular application. The evolution is captured in the hypergraph, by continuously adding new instances, according to ontology definitions.

Event Calculus (EC) is a modeling method dealing with events, and the effects they produce [2]. There are some similarities with respect to our approach: events resemble actions, effects correspond to the creation or termination of qualities, time-points are similar to hypernodes. EC also introduces time intervals, that may correspond to quality edges. Temporal representation in EC is based on clauses such as *happens(event,time-point)* – marking the occurrence of an event, *initiates(event, property, time-point)* – marking the initiation of *property*, started by *event*, at the moment *time-point* and *terminates(event, property, time-point)* – which similarly terminates a property. In our approach, actions inherently be-

long to nodes from the hypergraph, whereas nodes delimit the moments in time when a quality holds. The hypergraph doesn't contain time intervals explicitly, but using timestamps, temporal durations can be computed. Reasoning in EC is based on establishing the truth value of first-order predicates. Compared to a hypergraph, where the life-span of qualities can be easily traced, the FOL-based representation from EC makes reasoning more difficult.

In [13], a declarative language for specifying web service composition (or processes) is introduced. 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. During the execution of a process, some constraints may not be satisfied, as not all services have been invoked. Nonetheless, when the process ends, all constraints must be satisfied. The approach from [13] uses only a subset of LTL and thus has limited expressive capabilities.

6 Conclusions and Future Work

There are many cases when devices internally implement intelligent behavior, but often this behavior is rigid, cannot be adjusted to more particular needs, and cannot be extended. Most importantly, intelligent behavior comes with a considerable increase of device cost. It is therefore more convenient to use devices with simple hardware, and to transfer intelligence to software components that are easier to design and update. The proposed modeling method and language are suitable for this endeavor. Moreover, with the introduction of a hypergraph, the approach has the advantage of reducing the temporal reasoning process to simple graph traversals. The chosen declarative approach has several benefits: declarative models are small in size, easy to write, and favor model checking and verification techniques.

When using COOL to represent a hypergraph, the simulation of instantaneous actions proves difficult, and complicates model specification. In addition, the unification mechanism described in Section 3.4 is not supported natively in CLIPS. For these reasons, a specific language as well as an interpreter, are being developed. The interpreter will interface with web services and will be able to: (1) translate service state changes to actions and deliver them to the model, and (2) translate actions generated by the model to web service invocations.

A current possible limitation of the proposed language with respect to CLIPS, is the inability to have qualities introduce other qualities (as described in Section 3.3). While this behavior might be simulated using a special action (`non-stop`) that is constantly calling itself, this approach is computationally inefficient. An intelligent alternative for having qualities introduce other qualities is planned as future work.

It is important to emphasize that the modeling method has potential advantages to numerous other applications in areas not necessarily restricted to device control or Service Oriented Architectures. Such areas remain to be further investigated.

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. Artale, A., Franconi, E.: A survey of temporal extensions of description logics. Annals of Mathematics and Artificial Intelligence 30, 171–210 (March 2001), http://portal.acm.org/citation.cfm?id=590341.590357
2. Augusto, J.C.: The logical approach to temporal reasoning. Artif. Intell. Rev. 16, 301–333 (December 2001), http://portal.acm.org/citation.cfm?id=565277.565279
3. Carmichael, D.J., Kay, J., Kummerfeld, B.: Consistent modelling of users, devices and sensors in a ubiquitous computing environment. User Modeling and User-Adapted Interaction 15, 197–234 (August 2005), http://portal.acm.org/citation.cfm?id=1101018.1101052
4. Giumale, C., L., N.: Reasoning with fluid qualities. CSCS-17, 17th international conference on control systems and Computer Science 2, 197–203 (December 2009)
5. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., Sattler, U.: Owl 2: The next step for owl. Web Semant. 6, 309–322 (November 2008), http://portal.acm.org/citation.cfm?id=1464505.1464604
6. J., G.: Clips reference manual (1994)
7. Jan Wielemaker, Tom Schrijvers, Markus Triska, Torbjörn Lager: Swi-prolog. CoRR abs/1011.5332 (2010)
8. Juarez, J.M., Campos, M., Palma, J., Marin, R.: Computing context-dependent temporal diagnosis in complex domains. Expert Syst. Appl. 35, 991–1010 (October 2008), http://portal.acm.org/citation.cfm?id=1383655.1383743
9. Li, S., Ying, M.: Region connection calculus: its models and composition table. Artif. Intell. 145, 121–146 (April 2003)
10. de Melo, G., Weikum, G.: Towards a universal wordnet by learning from combined evidence. In: Proceeding of the 18th ACM conference on Information and knowledge management. pp. 513–522. CIKM '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1645953.1646020
11. NASA: Clips website (dec 2010), http://clipsrules.sourceforge.net/WhatIsCLIPS.html
12. Pan, F.: An Ontology of Time: Representing Complex Temporal Phenomena for the Semantic Web and Natural Language. VDM Verlag, Saarbrucken, Germany, Germany (2009)
13. Pesic, M.: Decserflow: Towards a truly declarative service flow language. In: Springer. pp. 1–23. Springer (2006)