Modeling With Fluid Qualities [1,2,3]

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

Computer Science Department, POLITEHNICA University of Bucharest Splaiul Independentei nr. 313, sector 6, 060042, Bucharest, Romania

Abstract: The current knowledge-based modeling languages have no acceptable mechanisms for working with time and the time-dependent properties of the described applications. Time-dependent aspects are usually represented using mechanisms for specifying static parameters of the described application, yielding complex models that are difficult to maintain, extend, and reason about. We argue that a modeling approach focused on the declarative and explicit specification of time-dependencies is more suitable. We propose a modeling method based on fluid qualities, and a declarative language for building time-dependent models that can be efficiently processed. The language has similarities with CLIPS and Prolog, and has direct applications in areas centered on intelligent device control.

Keywords: time-dependent behavior, modeling, ontology, declarative languages, services, SOA.

1. INTRODUCTION

Specifying and reasoning about phenomena that evolve in time requires the declarative description and storage of the past and present events occurring during the analyzed evolution, in order to efficiently derive information that aggregates events and time intervals according to complex relationships. In what follows we say that applications based on phenomena as the ones mentioned above are *fluid*. The representational infrastructure of a modeling system that is able to appropriately describe fluid applications has a direct influence on the following:

(1) The denotational gap between the model and the described application. Ideally, the entities and the time-dependent behavior of the application should be described as viewed by the application human expert.
(2) The difficulty of specifying associations between events and time intervals and of aggregating such associations into complex relationships that in turn are time dependent. Time and relationships between time intervals and the events that populate the application should be directly specified.
(3) The tractability of the inference process required by the application. Searching for entity-event relationships that are dependent on time intervals should avoid exhaustive explorations of complex data structures.

The proposed modeling approach aims at constructing domain ontologies – according to the above mentioned restrictions – able to specify fluid applications. The term *domain ontology* corresponds to that in Guarino (1998). It is a theory of the intentional meaning of a formal vocabulary associated with a particular conceptualization of a given universe of discourse. From the computational point of view, an ontology is the declarative support for decidable reasoning processes from a given Universe of Discourse (UoD). In our approach, an ontology is divided into (1) a taxonomy of entities from the given application domain, and (2) a set of models associated to the behavioral properties of specific applications populated by domain entities. While the taxonomy has a conventional structure and contents that address the intensional properties of the application domain, a model observes the extensional entity properties (i.e. properties that can depend on the modeled application and which shape the time-dependent behavior of that application). Only the proposed behavioral modeling approach is addressed in this paper. This work elaborates on the discussion presented in Giumale et al. (2010), that previously introduced the concept of modeling with *fluid qualities*.

The structure of the paper is as follows. Section 2 briefly discusses typical existing modeling methods. Section 3 presents the proposed modeling approach. Sections 4 and 5 discuss the basic representation primitives of the proposed approach. Section 6 focuses on time connectors. Section 7 shows the way of aggregating the basic modeling primitives to form rules and models. Section 8 concludes the paper.

[1] The work discussed is part of an industrial research project for controlling "intelligent buildings" (FCINT- ID 551, Code SMIS-CSNR 12038, from POS-CCE, O212)

[2] 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.

[3] The work has been 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/88/1.5/S/61178.

2. CURRENT MODELING METHODS

There are two typical methods for declaratively specifying and reasoning about fluid applications. The first method is centered on knowledge processing and uses a cognitive system as a vehicle for knowledge representation and reasoning. Examples span the full range from comprehensive taxonomies, such as WordNet, which is described in de Melo and Weikum (2009), formal notations for building ontologies, such as OWL Grau et al. (2008), and up to foundational ontology frameworks, such as DOLCE Masolo et al. (2007). The second method uses specification languages that are able to formally describe state transitions systems. Well known formal specification languages and systems, such as Z from Diller (1990), VDM from Turner and McCluskey (1994), B from Abrial (1996) etc. adhere to that view. Both methods have serious tractability drawbacks and limitations in handling complex fluid applications.

The current ontological-based approaches are successful for the static structural description of a given domain of discourse, and are difficult to use for specifying and reasoning about the evolution of the described application. For instance, besides the inability to accommodate with dynamic change, OWL has no predefined mechanisms for specifying time-dependent properties.

Consider the following example: *cool the house and, if the energy consumption was over 10 kWh in the last two days, keep the power level under 5 kW*. The constraint *keep the power level under 5 kW* is conditioned by the time-dependent property *energy consumption was over 10 kWh in the last two days*. Representing such facts requires an explicit connection between the application parameters, and the time-interval in which they hold. Building representations of time and time-dependent concepts in OWL proves extremely difficult, and produces intractable models. Although a number of OWL temporal extensions have been proposed by Hobbs and Pan (2004), Frasincar F. and U. (2007), currently there is no complete implementation of such extensions. Moreover, in some cases, specifications written in these experimental notations were proved not decidable.

The focus in most ontological frameworks is on the logical correctness of the formal description and less on the description tractability when implemented as a program. The construction and the modification of a knowledge base may rely on proofs fueled by complex constraints that are an essential part of the description of the application domain. Languages for building domain-oriented ontologies, such as OWL, have similar pitfalls, being difficult to use for fluid applications.

In the state transition approach, the focus of the description is on state transitions, where a state encodes the parameters of the modeled application at a given moment in time. Nevertheless, the states must be "closed", meaning that the set of parameters that a state encodes cannot be modified in time. As a consequence, each state must describe the full set of the application parameters, although some of them may be irrelevant in other states. In the case of small-sized applications, this requirement can be met with acceptable overhead in both modeling effort and processing. For large and even medium-sized applications, working with *full states* that encode all required properties can be a real nuisance. The set of states can become unmanageable due to the combinatorial explosion induced by the parameter values. For instance, a service-oriented application for a building may control m devices. Each device can be represented as a finite state automaton $M_i = (K_i, \Sigma_i, \delta_i, s_i), i = 1 \dots m$. Therefore, the service can also be represented as a finite state automaton $M = (K, \Sigma, \delta, s)$, where $K = K_1 \times K_2 \times ... \times K_m$. The total number of states of the service finite automaton will be $|K|$, quite impressive in complex cases. Another essential problem is related to the explicit representation of the time intervals associated with the modification of state parameters. Often, in a fluid application, the time moment of a transition is less important compared to the time interval during which a state parameter has a specified value.

3. THE PROPOSED MODELING APPROACH

Our modeling approach views the state of an application as a set of relationships between the application parameters – relationships subsequently called *qualities* – which the objects of that application must obey over finite time intervals. A state, as seen in the conventional approach, is unpacked into a set of qualities which are stable over a finite time interval during the application evolution. A state transition corresponds to the modification of the current qualities (some qualities can cease to exist and other qualities may come into effect) according to transition signals or events, which are called *actions*. Applying an action to a subset of parameters can depend on the current qualities of the application objects and can modify the qualities of these objects. For instance, the description of the behavior of a battery of elevators E must specify information regarding the previous behavior of each elevator e from E for being able to derive information such as *does e preserves its former motion direction?* In a model that describes the individual evolution of each quality of the application parameters, it is sufficient to work with qualities such as `moving(e,direction)` and to specify that such qualities span time intervals between consecutive events that may cause these qualities to change.

The relationship between a description based on conventional state transitions system and a quality-based description is illustrated in Figure 1, where there are four actions, ai, $i = 1, 4$, and three states. Each state stores two parameters p_1 and p_2. The parameter p_1 is irrelevant in $state_3$, and p_2 is irrelevant in $state_1$. The two description styles do not contradict each other and, in fact, they are equivalent. Nevertheless, it is easy to see that the quality-oriented descriptions are better suited for fluid applications where the timing of actions is important and, moreover, the processing of the time intervals associated with specified relationships of the application parameters is an essential issue. In the conventional state transitions approach finding such time intervals implies exhaustive explorations of full paths in the data structure that represents the unfolding of the automaton execution for the described application. In the actions-qualities approach, the time intervals of parameter relationships are explicitly

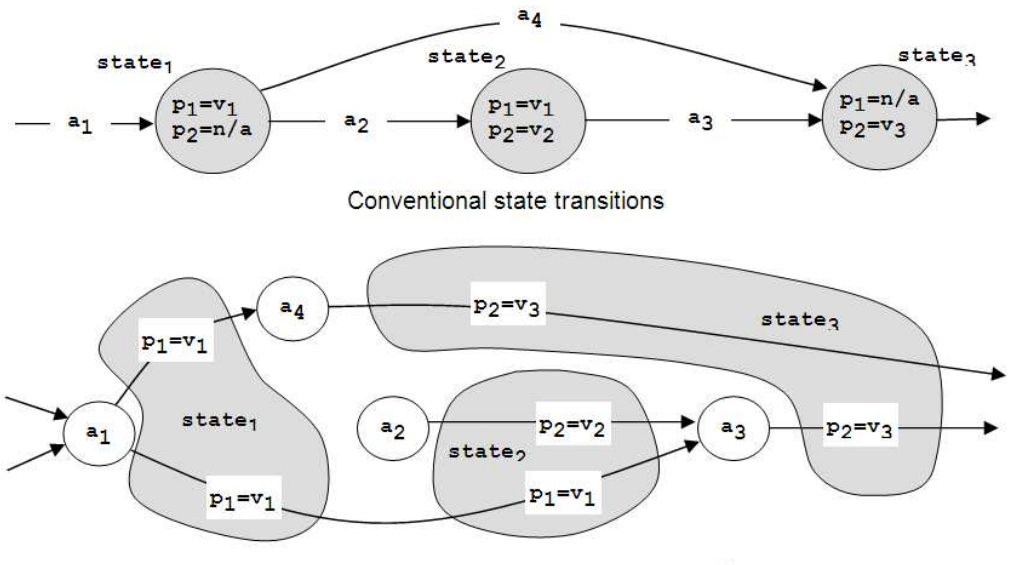

Fig. 1. Unpacking states to sets of qualities

represented as edges in a graph, as illustrated in Figure 2, speeding up the model processing.

The perdurantist modeling style, based on quality-action description, makes a notable difference between the state transitions style of description and the proposed approach. In the proposed approach the application parameters, called *individuals* in what follows, are members of qualities (relationships that characterize events and processes from the application evolution). An abstract individual, i.e. an individual with no quality, makes no sense in a description of that type; any individual must be in some relationships with other individuals or must have at least a 1-ary quality. The specification does not rely on explicit sets and set operations, as in Z, B and OWL. Time dependent sets of individuals are specified indirectly as individuals that are part of a quality according to a time constraint.

4. MODELING PRIMITIVES

The essential description primitives that compose the model of an application A from a given application domain are:

4.1 Individuals

An individual is any identifiable object from an application A. From the programming point of view, an individual is a singleton type. The value of an *individual* type is the individual itself. There are two commitments we make about the representation and the meaning of an individual that deviate from the conventional approach from Guarino and Welty (2000) and S. and J. (2004): (1) an individual is atomic; it has no structure; (2) an individual is unique and, therefore it is identifiable by itself. An individual can be member of a relation over individuals, relation called quality. The set of qualities which an individual has at given moment in time may govern the behavior of that individual. For instance, the individual bound to a variable x can have the current qualities `Elevator(x)`, `Capacity(x,8,persons)`, `Halted(x)`, `AtLevel(x,3)`, `Loaded(x,5,persons)`. In turn, `3` can be seen as an individual with the qualities `Number`, `Integer`, `Float` etc.

4.2 Qualities

Qualities stand for n-ary, $n \geq 1$, relations. The quality $Q(x_1, x_2, ..., x_n)$ designates a time-dependent relationship between the individuals x_i, $i = 1, \ldots, n$. In particular, for $n = 1$, the quality $Q(x)$ stands for a time-dependent property of x. A quality q spans a time interval Δq called the quality time-slice, and can be viewed as an edge of an oriented graph, where the graph nodes are associated with moments in time and the graph edges are qualities. A quality is built by instancing a quality prototype (a relation prototype). Quality prototypes can be structured, and can form hierarchies similar to class hierarchies in object-oriented languages. If C is a super-class of a quality prototype C' it means that all the individuals that have a quality from instances of C' must also have a quality from instances of C at a specified moment in time or during a specified time interval.

4.3 Actions

An action corresponds to a stimulus applied to one or a set of individuals, and can require existing/missing qualities of those individuals. In some cases, an action may designate an event. An action establishes or destroys relationships between individuals, and can generate actions. Therefore, an action is considered as the constructor or as the destructor of qualities. For instance if `start_moving` is an action applied onto an elevator then the preconditions of the action `start_moving(x)` can be `Elevator(x)`, `Halted(x,1)`, `Capacity(x,max,persons)`, `Loaded(x,n,persons)`, $n \leq max$. From the processing viewpoint, an action can be viewed as a node in a graph. The node may be the starting or the ending point of an edge that designates a quality. The starting node corresponds to the beginning of the quality time-slice; the

ending node corresponds to the end of that time-slice. Actions are considered instantaneous.

4.4 Time

Time is relative. A moment in time corresponds to a group of actions and a time interval to a quality spanning two actions (consecutive or not). For instance, if the action $a_1(x)$ that occurs at time t_1 builds the quality $q = Q(x)$, and the action $a_2(x)$ that occurs at time $t_2 > t_1$ destroys q, the time-slice Δq spans the time interval between t_1 and t_2. The time moments, such as t_1 and t_2, may be specified according to a particular time metric or can be considered relative to the occurrence of actions; of paramount importance is the relative ordering of actions that occur while the model of the application A unfolds according to the monitored events of A.

4.5 The evolution hypergraph

The time-dependent behavior of the individuals that populate the model of an application A is represented by a graph $G = (V, E_q, E_a)$ that is built incrementally, while the model of the application A is instanced and unfolds according to the evolution of A. G is an oriented, acyclic graph, where: (1) The nodes V designate actions from A. Distinct nodes can stand for the same action. The nodes that designate concurrent actions are grouped within the same hyper-node. A hyper-node corresponds to a time moment t during the evolution of A, and all actions within that hyper-node occur at time t. There is a distinguished node *Init* that corresponds to the starting moment of unfolding the model of the application A. There is a distinguished node *Current* that corresponds to the current moment of unfolding the model of A. (2) The edges E_q designate qualities built according to the actions V. The edge $q(a_1, a_2)$ corresponds to the quality q the lifetime of which spans the time interval $(time(a_1), time(a_2))$, where a_1 and a_2 are actions from V such that a_1 created q, while a_2 destroyed q. (3) The edges E_a designate preconditions of the actions V. An edge (q, a) is directed from a quality edge q the an action node a, meaning that q is a precondition of a.

5. PREDEFINED QUALITIES

There are predefined qualities: `day, month`, `year, time, date, number, string, boolean`, etc. that do not correspond to edges in the hyper-graph: they are *virtual*, and their time span is indefinite. There are predefined qualities that are not virtual, i.e. they correspond to edges in the hyper-graph. An essential predefined quality is `time_stamp` that is associated automatically with every node in the hyper-graph as shown in Figure 2. To preserve uniformity we consider that `time_stamp` is an unit edge in the graph with a time span value of 0.

Individuals with predefined qualities have specific identifiers, e.g. `time_stamp(8:00/8.01. 2010)`, `day(Friday)`, `time(8:00)`, `number(8)`. There are polymorphic individuals, i.e. individuals with the same identifier that have multiple predefined qualities: the individual `2010` as `year(2010)`, and `number(2010)` is both a year and a number. In such cases the qualities act as type casts that disambiguate the individuals. By convention, whenever there is no ambiguity, we specify a predefined quality `q(x)` by `x`. For instance, `time(8:00)` can be written `8:00`, `time_stamp(8:00/8.01.2010)` can be written `8:00/8.01.2010`, etc. There are specific operators that can be applied on individuals with predefined qualities. A particular class of such operators contains explicit coercions from a timestamp to a temporal quality (selectors of components from a timestamp). For example `day(8:00/8.01.2010)` returns the quality `day(Friday)`.

A variant of computing with timestamps is inspired from Prolog and is based on unification. Assume that: (1) the expression `quality1 is quality2` defines the unification of the two qualities. (2) for enumerable qualities, such a time, day, month, year we can apply the operators `+` and . The unification `quality1 is quality2` is *a special unification* such that the substitution S preserves the correctness of $quality_1/S$ and $quality_2/S$ according to the semantics and ordering of quality. For example, the unification shown in Program 1 solves the problem *which time stamp ?t/?d.?m.?y incremented by 2:00/2.0.0 yields 1:00/1.1.2010?*

```
time_stamp(1:00/1.1.2010)
    is time_stamp(?t+2:00/?d+2.?m.?y)
```

Program 1: Unification example

The unifier is `S=?t=23:00, ?d=29, ?m=12, ?y=2009` and `?t/?d.?m.?y` is `23:00/29.12.2009`.

6. TIME INTERVALS AND TEMPORAL CONNECTORS

A model that describes the behavior of an application `A` is, basically, a collection of rules that specify the actions and the events that can change the qualities of the individuals that populate `A`. A rule has the format illustrated in Program 2.

```
rule R
    preconditions Pre
    action Act
    effects Post
```

Program 2: Rule format

Part of the preconditions can be automatically inserted in the rule from the restrictions specified in the taxonomy of the application domain entities (taxonomy that is not presented in this paper). The rule `R` is applicable at the current moment t if: (1) the action `Act` is present at t, (2) the qualities mentioned in `Pre` exist at time t for some individuals, and (3) the temporal restrictions from `Pre` are satisfied at time t. Deciding rule applicability is a computationally intensive process based on pattern match. As a side effect, an applicable rule builds an application record which stores bindings of the local rule variables to appropriate values.

All rules that are applicable at the current moment t in a model are simultaneously applied. The effects of the set of applicable rules are enforced only after all these rules are applied. However, there are cases when this rule firing strategy is a source of problems. Consider the modeling

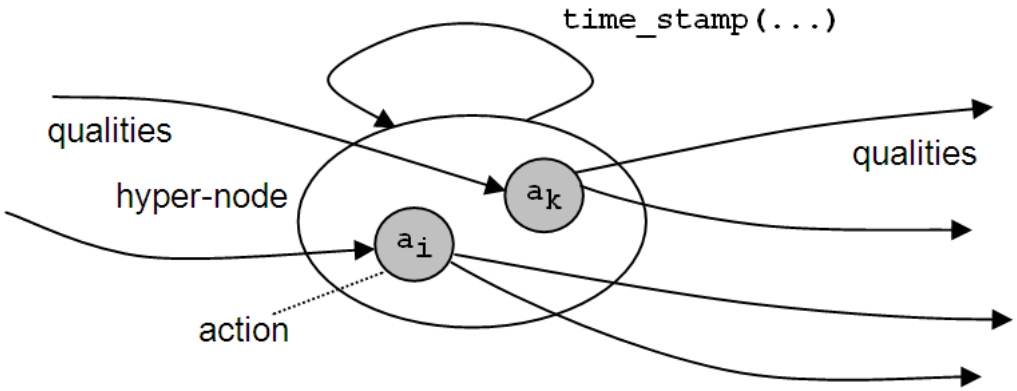

Fig. 2. Unpacking states to sets of qualities

of *turn on, in the non-decreasing order of the consumed power, the air coolers that were off the last two days such that the total consumed power does not go over a specific limit.* The wrong solution is the rule from Program 3.

```
rule turn_on
   preconditions
     PowerLimit(?limit)
     (current_moment() as ?now) is ?/?d+2.?m.?y
     AirCooler(?ac,?power_ac)
     time_interval(start ?/?d.?m.?y,end ?now)
             part_of Off(?ac)
     TotalPower(?total) as ?total_power
     ?total + ?power_ac <= ?limit
     not (AirCooler(?d,?power_d)
          and Off(?d)
          and ?power_d < ?power_ac)
   action turn_on_cooling()
   effects
       destroy ?total_power
       assert turnOn(?ac),
              TotalPower(?total+?power_ac)
```

Program 3: Wrong solution

Assume qualities `PowerLimit(1000)`, `TotalPower(0)`, as well as two air coolers (individuals with the current quality `AirCooler`) `ac1` and `ac2` with the same minimal power, say `600`. Then, the rule `turn_on` has two activation records, one for `?ac=ac1`, the other for `?ac=ac2`. If the simultaneous rule application of the two activation records is followed, then the total power is overrun. A possible correct solution is to instruct the model interpreter to process immediately some of the effects of a rule as soon as the rule becomes applicable. In the example from Program 4, the effects annotated `immediate` are enforced immediately when their containing rule becomes applicable.

Assume that, in Program 4 the first checked applicability of the `turn_on` rule is for `ac2`. Then (1) an activation record is created with `?ac=ac2, ?power_ac=600, ?total=0`; (2) the effects `destroy ?total_power` and `assert TotalPower(?total+?power_ac)` are effectively enforced (following the textual sequencing order). Subsequently, the model interpreter checks the applicability of rule `turn_on` for `?ac=ac1, ?power_ac=600, ?total=600`. The rule is not applicable since the precondition `(?total + ?power_ac)<=1000)` is not fulfilled. Therefore, a single activation record is preserved and the effects annotated as immediate are now enforced: `turnOn(ac2)`.

```
rule turn_on
   preconditions
     PowerLimit(?limit)
     (current_moment() as ?now) is ?/?d+2.?m.?y
     AirCooler(?ac,?power_ac)
     time_interval(start ?/?d.?m.?y,end ?now)
          part_of Off(?ac)
     TotalPower(?total) as ?total_power
     ?total + ?power_ac <= ?limit
     not (AirCooler(?d,?power_d)
          and Off(?d)
          and ?power_d < ?power_ac)
   action turn_on_cooling()
   effects
    immediate
      {destroy ?total_power
       assert TotalPower(?total+?power_ac)}
    assert turnOn(?ac)
```

Program 4: Correct turn_on implementation

The effective turning on of the air coolers is controlled by the simple rule from Program 5, assuming that an `AirCooler` is a particular kind of `Device`: the quality `AirCooler` is a descendant of the `Device` quality in the taxonomy of qualities (not presented in this paper) associated with the application domain of the model. The effect of trigger `Device(turnOn,?ac)` is to delegate control to a Web Service that implements the operation `turnOn` for the individual bound to the variable `?ac` and that has the quality `Device` (or a `Device` sub-quality). Such services are classified in the taxonomy of qualities used in the model. In that way, actions are processed using web services. For more information about Web Services, see Josuttis (2007).

```
rule turn_on_device
   preconditions Device(?ac),Off(?ac) as ?is_off
   action turnOn(?ac)
   effects destroy ?is_off, assert On(?ac)
           trigger Device(turnOn,?ac)
```

Program 5: Calling services

Notice the essential role of the proposed modeling approach: qualities (and actions) are generated by actions only. It implicitly means that there must be at least an initial action that starts the model interpretation. The modeler must provide such initial actions, in the same way he has to specify the initial qualities of the model individuals. If we permit general rules as in conventional rule-based languages, such as CLIPS J. (1994), i.e. a quality can be generated by other qualities and actions, then the complexity of the model processing increases dramatically. Since the qualities span time intervals, the pattern match algorithm (used for selecting the applicable rules) would be similar to that used in CLIPS, i.e. a sort of RETE algorithm described in Giarratano and Riley (2005), complicated by the data base storage of the hyper-graph, and by the (non-deterministic, device-dependent) time dependency of facts. If the pattern match is action-controlled, and if as expected there are few rules per action and few actions per time unit; the pattern match overload is limited. Moreover, assume that there is a predefined action, say initial-action (as the nitial-fact in CLIPS) which is implicitly present at any time moment. In that way the rules that depend on initial-action can be quality-driven without sacrificing the action centered approach.

Rules can be triggered by internal events (actions and qualities generated by the model rules) and by external events (actions and qualities generated from the environment of the modeled application). In addition, special service-oriented actions, such as `Device(turnOn,?ac)` from Program 5, can control directly the application environment. In that way, a model can be coupled with and can control a real-life application, using collections of web services, where individuals can correspond to devices, environmental parameters, etc.

7. CONCLUSIONS AND FUTURE WORK

The discussed work is part of an industrial research project having as a goal — among others — the development of a suitable framework for building ontologies able to describe complex time-dependent behavior of an extended range of household appliances and devices. The existing notations do not offer an acceptable support for the specified goal, although they provide strong verification mechanisms applicable for static models. The approach presented in this paper can cope with complex fluid applications. Nevertheless, the approach supports a limited range of static model checking, delegating the bulk of verification to the model interpreter at the run time. Moreover, there is a major scalability problem: the evolution hyper-graph, an essential part of a model, can theoretically grow indefinitely. We intend to limit the size of the hyper-graph that will be kept in a data base by zipping and logging the *distant past* of the modeled application to a mass storage. When necessary, that bulk of information can be searched for at the expense of a significantly slower model interpretation speed. Since in most of the real life applications such a situation occurs infrequently, we believe that – subject to the evolution hyper-graph search optimization – the model interpretation will work acceptably in most cases. A major benefit of the proposed modeling approach is the declarative description style. Services already modeled could be retrieved easier starting from semantic-oriented descriptions. An interesting application is to predict the future evolution of a model by expanding the evolution hyper-graph according to an operational semantics, the development of which is currently underway.

REFERENCES

Abrial, J.R. (1996). *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA. URL `http://portal.acm.org/citation.cfm?id=236705`.

de Melo, G. and Weikum, G. (2009). Towards a universal wordnet by learning from combined evidence. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, 513–522. ACM, New York, NY, USA.

Diller, A. (1990). *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., New York, NY, USA.

Frasincar F., M.V. and U., K. (2007). tOWL: Integrating Time in OWL.

Giarratano, J.C. and Riley, G. (2005). *Expert systems : principles and programming*. Thomson Course Technology, c2005.

Giumale, C., Negreanu, L., Muraru, M., and Popovici, M. (2010). Modeling ontologies for time-dependent applications. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, 0, 202–208.

Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., and Sattler, U. (2008). Owl 2: The next step for owl. *Web Semant.*, 6, 309–322.

Guarino, N. (1998). Formal ontology and information systems. In *Formal Ontology and Information Systems*, 3–15. IOS Press.

Guarino, N. and Welty, C. (2000). Identity, unity, and individuality: Towards a formal toolkit for ontological analysis.

Hobbs, J.R. and Pan, F. (2004). An ontology of time for the semantic web. *ACM Transactions on Asian Language Information Processing*, 3, 66–85.

J., G. (1994). Clips reference manual.

Josuttis, N.M. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, 1 edition.

Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A., and Schneider, L. (2007). The WonderWeb Library of Foundational Ontologies and the DOLCE ontology. Technical report, WonderWeb.

S., F. and J., B. (2004). General ontology baseline ontospace project report, deliverable d1, version 1.2.

Turner, J.G. and McCluskey, T.L. (1994). The construction of formal specifications: an introduction to the model-based and algebraic approaches. In *McGraw-Hill Software Engineering Series, London. ISBN*, 0–07. McGraw Hill.