Event-Based Modeling of Evolution for Semantic-Driven Systems

Peter Plessers*, Olga De Troyer, and Sven Casteleyn

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
{Peter.Plessers, Olga.DeTroyer, Sven.Casteleyn}@vub.ac.be

Abstract. Ontologies play a key role in the realization of the Semantic Web. An ontology is used as an explicit specification of a shared conceptualization of a given domain. When such a domain evolves, the describing ontology needs to evolve too. In this paper, we present an approach that allows tracing evolution on the instance level. We use event types as an abstraction mechanism to define the semantics of changes. Furthermore, we introduce a new event-based approach to keep depending artifacts consistent with a changing instance base.

1 Introduction

The Semantic Web is an extension of the current Web; information is given a well-defined meaning better enabling computers and people to work in cooperation [1]. Ontologies play a major role in the Semantic Web where the meaning of web content is formalized by means of ontologies. An ontology is defined as an explicit specification of a shared conceptualization [2]. We also see that other research domains are adopting technologies developed within the Semantic Web domain. E.g. ontologies are used in content and document management, information integration and knowledge management systems to provide extensive reasoning capabilities, intelligent query possibilities, integration and cooperation between systems, etc. We will use the term "semantic-driven" systems to refer to such systems.

Systems and their environments are not static but evolve. Domains evolve and changes in user requirements occur often. To keep semantic-driven systems up to date, changes in the environment should be reflected in the underlying ontology. Furthermore, also flaws and errors in the design of the ontologies may call for a revision of the ontology. Manual handling of the evolution process of ontologies as well as managing the impact of evolution on depending ontologies, instance bases, applications and annotations, is not feasible because it would be too laborious, time intensive and complex. Therefore, an automatic mechanism should be provided.

Ontology evolution takes place on two levels: on the instance level (e.g. a 'prime minister' who resigns after an election defeat and becomes a 'senator') and on the

* This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, LNCS 3520, pp. 63 – 76, 2005.

concept level (e.g. a new rule that forbids people to be candidate for more than one parliament). Current approaches for supporting ontology evolution [3], [4] propose a single approach dealing with evolution on the instance as well as on the concept level. In this paper, we concentrate on evolution on the instance level (evolution of the concept level is outside the scope of this paper) and the impact of this evolution on depending artifacts. Our approach delivers a number of advantages not found in existing approaches.

The remainder of the article is organized as follows. In section 2 we give a short overview of related work. A general overview of our approach is given in section 3, more details are given in section 4. The approach is further elaborated in section 5 (time aspect) and section 6 (events). Section 7 explains the handling of the impact of evolution for depending artifacts. Section 8 ends the paper with conclusions.

2 Related Work

The work presented in this paper is related to the fields of temporal databases and ontology evolution.

Conventional databases capture the most recent data. As new values become available, the existing data values are removed from the database. Such databases only capture a snapshot of reality and are therefore insufficient for those in which past and/or future data are required [5]. Temporal databases [6] typically allow differentiating between temporal and non-temporal attributes. The temporal database maintains a state history of temporal attributes using time stamps to specify the time during which a temporal attribute's value is valid.

Different conceptual models that support the modeling of temporal databases exist. They can be divided into three categories: extensions to relational data models (e.g. TER [7], TERM [8] and ERT [9]), object-oriented approaches (e.g. TOODM [10]) and event-based models (e.g. TEERM [11]). The approach described in this paper resembles most the philosophy taken by event-based models. These models don't record past states of a system, but rather events that change the state. Note that in these models events are just ‘labels’ i.e. they don’t define the meaning of the event.

Ontology evolution approaches [3], [4] propose methods to cope with ontology changes and techniques to maintain consistency of depending artifacts. Both approaches present a meta-ontology to represent changes between ontology versions. A log of changes is constructed in terms of this meta-ontology. Consistency is maintained by propagating changes (listed in the log of changes) to depending artifacts.

The difference with our approach is that we formally define the semantics of changes (by means of event types). This allows us to reason about changes on a higher level of abstraction. Furthermore, event types are used to maintain consistency between an instance base and depending artifacts.

3 Approach: Overview

To keep track of changes to the instance base, we use the following approach. Whenever a change is made to the instance base, the log of changes is updated. The log is

defined in terms of the *evolution ontology*. Third-party users can use this log structure to check if a (for their artifact) relevant change occurred. To automate this, third-party users can specify a set of *event types* in which they are interested. If after the update one ore more instances satisfy the definition of one of their event types, an instance of this event type will be created. The event type is an abstraction mechanism that allows us to reason about changes on a higher level of abstraction than possible with the log of changes. The event types and the events itself are captured in a log of events defined in terms of an *event ontology*.

After a change, the instance base and depending artifacts may be in an inconsistent state. Instead of forcing third-party users to upgrade their dependent artifacts to maintain consistency (as is done in other approaches), we present a technique based on event types that allows to validate if a dependency is still valid in the current state of the instance base and if not to refer to a previous state of the instance base. This approach allows third-party users to update at their own pace (if ever) without causing inconsistencies in the meantime.

Before proceeding with a more detailed description of our approach, we first introduce an example situation that will be used throughout the paper. We have constructed a small domain ontology describing the political domain of a federal state: its governments, parliaments, ministers, parliamentarians, etc. Also assume an instance base based on this ontology to populate the web portal of the government. Furthermore, there exists a third-party website listing the current and past ministers of the governments. The content of the website is annotated using the political instance base. Note that the owner of the instance base and the owner of the website are not necessarily the same. Moreover, the instance base does not support evolution as it is of no direct benefit for the government. Also note that the government is not necessarily aware of third-party users making use of the instance base. In addition, a third-party user may only be interested in tracing evolution for a very specific part of the changes that occur. E.g. the owner of the website is only interested in tracing the evolution of ministers; i.e. he is not interested in parliamentarians.

4 Approach: Details

4.1 Assumptions

Our approach is based on the following assumptions:

1. The underlying domain ontology of the instance base is build up using classes, object properties (relations between classes) and datatype properties (relations between a class and a data type e.g. strings, integers, ...). All these are called *concepts*.
2. Concepts are identified by a unique identifier that uniquely identifies a given concept during its 'lifetime'.
3. There exists three operations that users can apply to update an instance base:
   - *Create:* a new instance is added;
   - *Retire:* an instance is removed;

- *Modify:* an instance is modified. E.g. an instance of a concept A evolves into an instance of a concept B; the value of a datatype property is changed; etc.

Note that we explicitly need the 'Modify' operator as we can't treat it as the combination of a 'Retire' and 'Create' operator. When we modify an instance by removing the instance first and afterwards adding a new instance, we cannot assure that it is the 'same' instance as identifiers can be reused.

As these assumptions are based on common features most systems will satisfy them.

4.2 Evolution Ontology

As explained in the overview, a log of changes made to an instance base is maintained in terms of the *evolution ontology*. For every instance of a class in the instance base, a new unique instance is created in the log of changes. This instance has a reference to the original instance in the instance base (at least as long as this instance exists in the instance base). In addition, the log keeps track of all the changes that are made to that instance. This is done by means of the *Change* concept.

A Change is defined as a concept with the following properties (see fig 1): a reference to the instance to which it refers (*instanceOf*); the operation used to make the change *(hasOperation);* and a time stamp identifying the date and time of the change *(hasTransactionTime)*. We have defined three types of Changes: a change to instances of a class *(ClassChange)*, to instances of an object property *(ObjectPropertyChange),* and to instances of a datatype property *(DataTypePropertyChange)*. For an ObjectPropertyChange and a DataTypePropertyChange we also keep track of respectively the target instance *(hasTargetInstance)* and the value of the changed property *(hasValue)*.

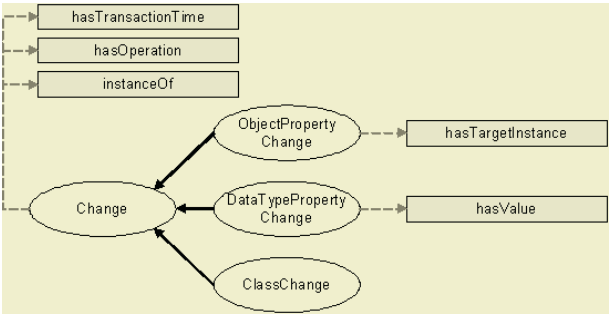

Fig. 1. Overview of Change classes

The following OWL code gives an example of a log of changes. It shows the changes applied to an instance 'john_smith'. The first change denotes that 'john_smith' is created as an instance of the concept 'Person' (see 1). Next, an instance of the datatype property 'hasName' was assigned with value 'John Smith' (see 2). Finally, he becomes politically active: 'john_smith' changes to an instance of 'Politician' (see 3) and joins a political party (see 4).

```
<EvolutionClass rdf:ID="60f28870">
  <refersTo rdf:resource="…/politics#john_smith"/>
  <changeOccurred>
    <ClassChange rdf:ID="7ece6c60">                    (1)
      <hasTransactionTime>07/05/03</hasTransactionTime>
      <instanceOf rdf:resource="#Person"/>
      <hasOperation rdf:resource="#Create"/>
    </ClassChange>
  </changeOccurred>
  <changeOccurred>
    <DataTypePropertyChange rdf:ID="7ece6c61">         (2)
      <hasValue>John Smith</hasValue>
      <hasOperation rdf:resource="#Create"/>
      <hasTransactionTime>08/05/03</hasTransactionTime>
      <instanceOf rdf:resource="#hasName"/>
    </DataTypePropertyChange>
  </changeOccurred>
  <changeOccurred>
    <ClassChange rdf:ID="7ece6c62">                    (3)
      <hasTransactionTime>18/10/04</hasTransactionTime>
      <instanceOf rdf:resource="#Politician"/>
      <hasOperation rdf:resource="#Modify"/>
    </ClassChange>
  </changeOccurred>
  <changeOccurred>
    <ObjectPropertyChange rdf:ID="7ece6c63">           (4)
      <hasTarget rdf:resource="#political_party_x"/>
      <hasTransactionTime>19/10/04</hasTransactionTime>
      <hasOperation rdf:resource="#Create"/>
      <instanceOf rdf:resource="#memberOf"/>
    </ObjectPropertyChange>
  </changeOccurred>
</EvolutionClass>
```

As the use of operations trigger changes, this log can be generated automatically. Every operation to the instance base leads to a change in the evolution ontology indicating the difference between the current and previous state of an instance. Note that the reference to the original instance in the instance base is removed as soon as the instance retires from the instance base.

4.3 Event Ontology

The log of changes can be used by third-party users to get an overview of the changes that have occurred to the instance base they are using. This log of changes forms the basic mechanism to keep depending systems consistent with the changed instance base (see section 6). However, the approach also uses an *event ontology* because the evolution ontology does not allow to deal with the following situations:

- A third-party user may only be interested in particular changes. Take for instance our example instance base and annotated web page that lists the names of all current and past ministers. For such a page, we are interested in the event where a per-

son becomes a minister or retires as minister, but we don't care when a minister changes office, or is promoted to prime minister.

- Reasoning in terms of evolution is not convenient. Changes are defined at the lowest level and few semantics are captured because no reason or meaning of a change is given. E.g. what is the reason for deleting an instance of a concept 'Minister'? Was the minister fired? Did the government fall? Or was it just the end of his term?

To associate meaning to changes and to allow indicating relevant changes, a third-party user can define a set of event types. Event types are defined in terms of changes. In this way, events in the real world can be associated with changes in the log of changes. As an example, a third-party user may define an event type 'retireMinister' as the change of an instance from being an instance of the concept Minister to a retired instance.

Letting third-party users define their own set of relevant event types, allows tracing changes from different viewpoints. Although there exists a shared agreement concerning the domain ontology and its instance base, this doesn't mean that there is also a shared agreement about events. Therefore, different users may define different event ontologies. Moreover, an event type is an abstraction mechanism that allows to reason about changes on a higher level than possible with changes in the evolution ontology .

More details and a formal representation for event types will be given in section 6.

5 Time Aspect

An important aspect when tracing evolution is the notion of time. A linear time line T is therefore used. Changes (in the log of changes) as well as events (in the log of events) are linked to this time line by means of timestamps. These timestamps represent *transaction times*. Transaction time indicates when an instance was created, modified or retired from the instance base. For each change to an instance i we use the time line T to represent transaction times. This means that we define an explicit order on the changes for a particular instance i. This can be seen as an individual, relative time line. We refer to this time line as T_i where i is a given instance from the evolution ontology. A variable ct_i refers to the current time of an instance i, i.e. the moment in time the last change took place for this instance. If c ($\in$ N) specifies the total amount of changes that occurred for an instance i, than we can use ct_i-a (where $a \in$ N, $a \leq n$) to refer to the moment in time the $(c - a)^{th}$ change occurred for that instance.

Events contain a reference to the change that triggered them. An event is indirectly linked to the time line T through the referred changes. Figure 2 gives an overview.

T_i allows us to retrieve properties of instances relative to this time line. This means that we are able to request the value of a property for an instance at a certain moment in the past. The following notation is used to retrieve past states of instances:

```
<property_name>(<inst>,  <value> | <var>, <timestamp>)
```

where <inst> is an instance from the evolution ontology, <value> is the value of a property while <var> is a substitution, and <timestamp> $\in T_i$. The <timestamp> indicates the moment in time at which we request the property.

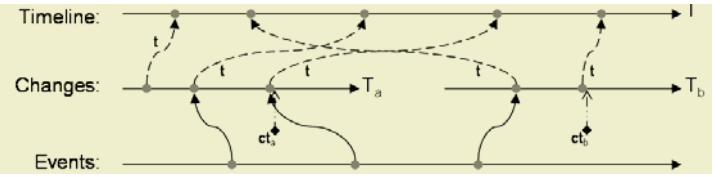

Fig. 2. Time aspect overview. 't' indicates transaction time

We give two examples. The first example checks if an instance *i* was an instance of the concept 'Minister' during the previous state of *i*. The second example retrieves the previous telephone number of the instance *i*; the result is stored in a variable 'x'.

```
Example 1: instOf(i, 'Minister', ct_i-1)
Example 2: hasTelephone(i, x, ct_i-1)
```

A first step to resolve this query is to transform the abstract timestamp (<time-stamp> $\in T_i$) into an absolute timestamp $t \in T$. Next, the past state of the instance base can be reconstructed by applying all stored changes that have occurred before the absolute time stamp. Finally, the query is resolved against the constructed state of the instance base.

6 Events

In this section we give more details about our event types. In section 6.1, we introduce basic event types. Basic event types are used to define the semantics of changes applied to one instance. We have defined a hierarchy of basic event types reflecting the meaning of basic changes. Users can subtype these basic event types to define their own set. As basic event types only define the meaning of changes to exactly one instance, we also have defined complex event types (see section 6.2) for changes involving more than one instance.

6.1 Basic Event Types

Figure 3 gives an overview of our basic event types. The root concept is the abstract class 'Event' and has three subclasses 'Creation', 'Modification' and 'Retirement'. These subclasses define the semantics of the changes resulting from the operations defined in section 4.1. The 'Modification' class is further refined into: 'Expansion', 'Contraction', 'Continuation', 'Extension' and 'Alteration'. The definitions of these event types are given in Figure 4.

To define the event types, we use the following definitions:

- The set *I* is the set of all instances of the evolution ontology (i.e. both class and property instances).

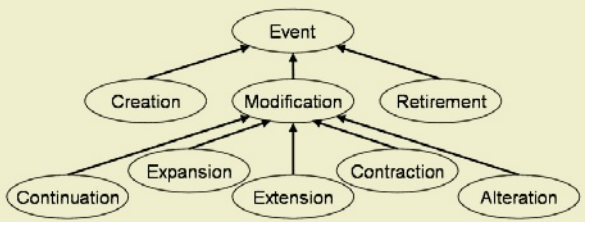

Fig. 3. Basic event type hierarchy

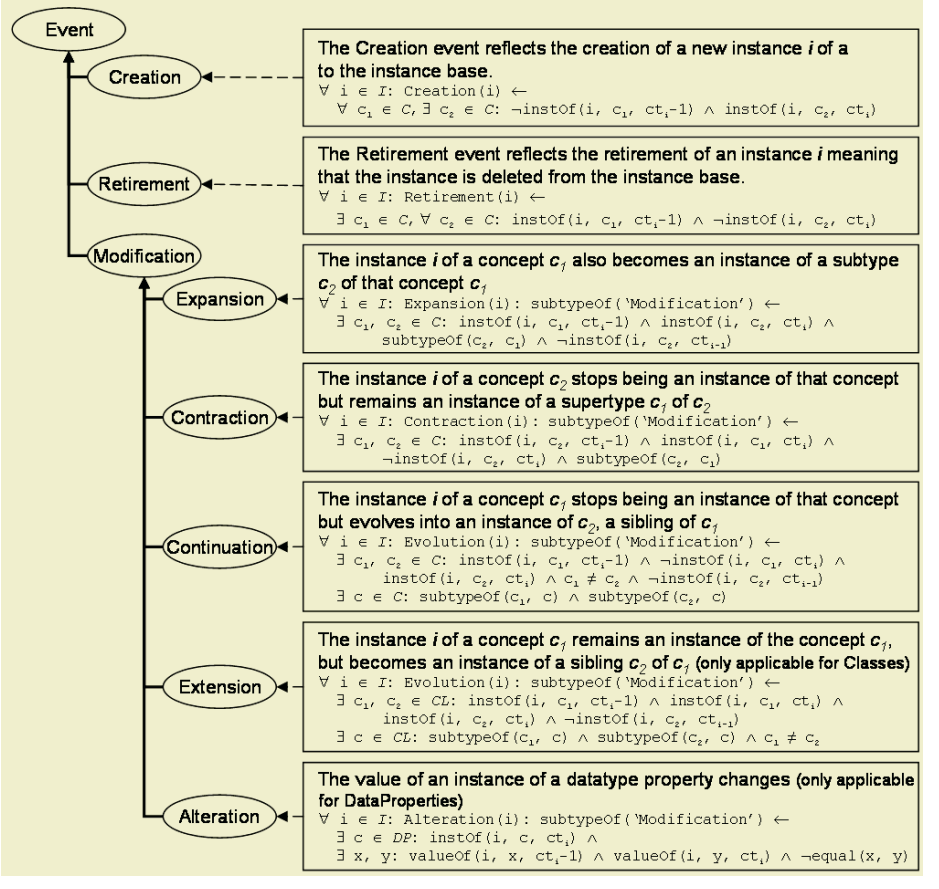

Fig. 4. Basic event type definitions

- The sets *DP*, *OP* and *CL* are the sets of respectively all datatype properties, object properties and classes defined in the domain ontology.
- The set $C = DP \cup OP \cup CL$.

As an example event type, we define 'MinisterLeavesGovernment' that describes the change when a minister leaves a government. It is defined as a 'Retirement' event of an instance of the 'memberOf' property. The remainder of the definition checks if the source and target of the property were instances of respectively 'Minister' and 'Government'.

```
∀ i ∈ I: MinisterLeavesGovernment(i):
 subtypeOf('Retirement') ←
 instOf(i, 'memberOf', ct_i-1) ∧
 ∃x, y ∈ I: source(i, x, ct_i-1) ∧ target(i, y, ct_i-1) ∧
   instOf(x, 'Minister', ct_x) ∧
   instOf(y, 'Government', ct_y)
```

6.2 Composite Event Types

Basic event types define the semantics of a change of exactly one instance. This fine-grained type of events is not always sufficient. Often evolution on a higher level, taking into account changes to more than one instance, is necessary. Therefore, we also provide *composite event types*: event types that define the semantics of changes of more than one instance. We illustrate this with an example. The example defines the change where two ministers from different governments change places (e.g. a minister from a regional government switches to the federal government and a minister from the federal government goes to the regional government). First we define the basic event type 'MinisterChangesGovernment'. The event type defines the change where a minister leaves one government to join another one.

```
∀ i ∈ I: MinisterChangesGovernment(i):
  subtypeOf('Modification') ←
  ' i remains an instance of 'memberOf'
  instOf(i, 'memberOf', ct_i-1) ∧
  instOf(i, 'memberOf', ct_i) ∧

  ∃x, y, z ∈ I:
    ' get the previous and current target and current
    ' source of i
    target(i, x, ct_i-1) ∧
    target(i, y, ct_i) ∧
    source(i, z, ct_i) ∧

    ' target and source are respectively instances of
    ' Government and Minister
    instOf(x, 'Government', ct_x-1) ∧
    instOf(y, 'Government' ct_y) ∧
    instOf(z, 'Minister', ct_z) ∧

    ' but the previous and current government are not
    ' the same instance
    ¬equal(x, y)
```

Second, we define a composite event type 'ExchangeOfMinisters' that makes use of the previously defined event type.

```
∀ i1, i2 ∈ I: ExchangeOfMinisters(i, j):
  subtypeOf('Event') ←

  ∃t1 ∈ T_i, ∃t2 ∈ T_j:
    ' the 'MinisterChangesGovernment' event occurred
    ' for both i and j
    occurredEvent[1] (i, 'MinisterChangesGovernment', t1) ∧
    occurredEvent(j, 'MinisterChangesGovernment', t2) ∧

    ' get governments
    ∃g1, g2, old_g1, old_g2 ∈ I:
      ' previous government of i
      target(i, prev_g1, ct_i-1) ∧
      ' current government of i
      target(i, cur_g1, ct_i) ∧
      ' previous government of j
      target(j, prev_g2, ct_j-1) ∧
      ' current government of j
      target(j, cur_g2, ct_j) ∧

      'check governments
      equal(prev_g1, cur_g2) ∧
      equal(prev_g2, cur_g1)
```

The event first checks if the instances *i* and *j* both changed government in the past i.e. the 'MinisterChangesGovernment' event type should have occurred before. Next, we lookup the governments they both left and joined. The last two statements check if the two ministers swapped government. Note that we didn't put any time restriction on the occurrence of the 'MinisterChangesGovernment' event. If for instance, a minister leaves government *a* and joins government *b* and three years later another minister leaves government *b* and joins government *a*, these changes will match the definition of the 'ExchangeOfMinisters' event type. However, in this situation, we can hardly speak of an exchange. We could solve this issue by adding a time constraint to the event type definition stating that the exchange must occur within a time frame of for instance 2 months.

7 Consistency Between Instances and Depending Artifacts

A change to an instance remains mostly not restricted to that single instance, but may have an impact on related instances and depending artifacts. It could bring the complete system (i.e. instance base and depending artifacts) into an inconsistent state. It is a major requirement for any evolution approach to assure that the complete system evolves from one consistent state to another.

[1] OccurredEvent(i, e, t) checks if an event type *e* has occurred for an instance *i* on a moment in time *t*.

Figure 5 shows an example system. The nodes represent an instance base (a) and two depending artifacts, the edges indicate the dependencies between them. Some of these nodes may have the same owner, others not. The circle in the figure indicates the set of nodes for which an owner has the necessary permissions to make changes. We call this a set of *controllable nodes* and refer to this set as N_c.
We distinguish three types of dependencies:

- **Intra dependency** is a dependency within one node.
- **Controllable inter dependency** is a dependency from a node a to another node b where $a \in N_c$ and $b \in N_c$.
- **Uncontrollable inter dependency** is a dependency from a node a to another node b where $a \notin N_c$ and $b \in N_c$.

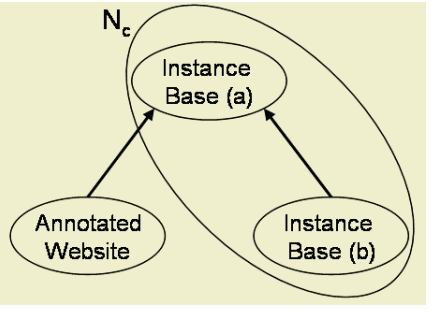

Fig. 5. Example dependency graph

Object properties, annotations, mappings between instances, etc. are forms of dependencies as they define reliance between objects. These are captured by the concept *Dependency*. A dependency exists between a source and a target instance. The concept of the source instance is called *domain* concept of the dependency, the concept of the *target* concept is called range concept. Furthermore, we have defined three subtypes of Dependency: *IntraDependency*, *ControllableInterDependency* and *UncontrollableInterDependency* referring to the distinction we introduced above. We define the set D as the set of all dependencies.

Current ontology evolution approaches provide solutions for keeping a system consistent by propagating a change to an instance to all depending artifacts [12] [13]. This means that one change may result into a chain of changes to related artifacts to avoid inconsistencies. While this may be an appropriate approach for intra and controllable inter dependencies (where one has sufficient permissions to make changes), this solution is not suitable for uncontrollable inter dependencies. In a setting like the Semantic Web, you cannot force others to update their depending artifacts to enforce consistency. Consider for example semantically annotated websites. It is not realistic to require an update of the annotations every time a change to the instance base occurs. Furthermore, it may even be not desirable to update depending artifacts. E.g. this is the case where a web page shows an image of the current prime minister that is annotated with an instance of the concept 'PrimeMinister'. When this instance evolves into an instance of for example 'Senator', it is not desirable to update the annotation or

content of the web page when the intention of the page was to show the image of the prime minister of the particular period.

Instead of propagating changes to depending artifacts, we maintain consistency without forcing third-party users to update their depending artifacts. This is done by indicating that a dependency may be state dependent, i.e. is only valid in a particular state of the instance base. To realize this, the concept 'UncontrollableInterDependency' is associated with an *invalidation event type*. The invalidation event type is used to specify that the dependency becomes invalid after the occurrence of the event. In other words, the dependency is only valid in the state of the instance base before such an event occurs for the target instance. Consider as example the following situation: *m* 'is member of' *g* where *m* is an instance of 'Minister' in an instance base (b) (of figure 5), *g* is an instance of 'Government' in instance base (a) and 'is member of' is an UncontrollableInterDependency between these two instances (the source of the dependency is *m* and the target is *g*). Attaching an invalidation event type to this dependency implies that the dependency refers to the state of the instance base (a) before the occurrence of the invalidation event for instance *g*. When no such invalidation event occurred for instance *g*, the dependency refers to the actual state of the instance base (a).

To simplify specifications, we have defined a default invalidation event type. The default invalidation event occurs when the target instance of the dependency is no longer an instance of the concept defined as range of the dependency. E.g. the retirement of the instance *g*, would trigger the default invalidation event type because *g* is no longer an instance of the range concept (i.e. 'Government').

The default invalidation event type is defined as follows:

$$
\begin{aligned}
&\forall\ i \in I\text{: DefaultInvalidationEvent(i):}\\
&\quad \text{subtypeOf('Event')} \leftarrow\\
&\quad \exists\, d \in D\text{: instOf(d, 'Unc.InterDependency', } ct_d) \wedge\\
&\quad \exists\, c \in C\text{: range(d, c) } \wedge \text{ target(d, i, } ct_d) \wedge\\
&\qquad \text{instOf(i, c, } ct_i\text{-1) } \wedge \neg \text{instOf(i, c, } ct_i)
\end{aligned}
$$

This event type specifies that there exists an uncontrollable inter dependency *d* and the range of the dependency is a concept *c*. Furthermore, an instance *i* is the target instance of the dependency *d*, but is no longer an instance of the range concept *c*.

Although, we specify a default behavior for uncontrollable inter dependencies, users can always refine this default setting by adding their own invalidation event types. Consider an annotated web page where there exist a dependency between page objects and instances of an instance base. (Note that annotations are a specific type of dependency as the source instance of the dependency is an instance of a HTML element.) Suppose the web page presents an annotated group picture of all ministers of the current government. Here, the default invalidation event type will not give the desired effect. When one or more ministers leave this government, the picture is no longer a correct representation of the state of the government. Therefore, the following event type should be added to the annotation as an additional invalidation event type and is defined as follows:

$\forall$ i $\in$ *I*: InvalidationEvent(i): subtypeOf('Event') $\leftarrow$
occurredEvent(i, 'MinisterLeavesGovernment', ct_i) $\wedge$
$\exists$ d $\in$ *D*, g $\in$ *I*:instOf(d, 'Unc.InterDependency', ct_d) $\wedge$
target(i, g, ct_i-1) $\wedge$ target(d, g, ct_d)

The definition checks if the event 'MinisterleavesGovernment' (see section 6.1) occurred for an instance *i* for which a dependency *d* exists with as target instance, the target instance of *i*.

8 Conclusion

We have presented an approach for ontology evolution on the instance level. A log of changes is maintained (by means of an evolution ontology) listing all changes applied. Third-party users can use this log to check if relevant changes occurred by specifying event types. If, after a change, instances satisfy the definitions of one of the event types, an instance of this event type is created. The event types are defined in *an event ontology* and the events itself are captured in a log of events. Instead of forcing third-party users to update their dependent artifacts to maintain consistency after a change, we have presented an event-based technique for maintaining consistency. Event types are used to invalidate dependencies and to refer to previous states of an instance base.

The advantages of our approach can be summarized as follows:

- Evolution of instances can be maintained without touching the instance base by means of the evolution ontology.
- Event types allow to filter relevant changes and to establish the semantics of changes in term of real life events. Furthermore, an event type is an abstraction mechanism that allows to reason about changes on a higher level of abstraction than possible with changes in the evolution ontology.
- Depending artifacts can be kept consistent without forcing third-party users to make updates.

References

1. Berners Lee, T., Hendler, J., Lassila, O.: The Semantic Web: A new Form of Web Content that is Meaningful to Computers will unleash a Revolution of new Possibilities. Scientific American, 5(1) (2001)
2. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2) (1993) 199-220
3. Klein, M., Noy, N. F.: A Component-based Framework for Ontology Evolution. Proceedings of the Workshop on Ontologies and Distributed Systems (IJCAI '03) Acapulco Mexico (2003)
4. Maedche, A., Motik, L., Stojanovic, L., Studer, R., Volz, R.: Ontologies for Enterprise Knowledge Mnangement. IEEE Intelligent System 18(2) (2003) 26-34
5. Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R.: Temporal databases: Theory, Design and Implementation. Redwood City, CA: Benjamin/Cummings Pub. (1993)

6. Ozsoyoglu, G., Snodgrass, R.: Temporal and Real-time Databases: A survey. IEEE Transactions on Knowledge and Data Engineering 7(4) (1995) 513-532
7. Tauzovich, B.: Toward Temporal Extensions to the Entity-Relationship Model. 10th International Conference on the Entity Relationship Approach (1991) 163-179
8. Klopprogge, M., Lockeman, P.: Modeling Information Preserving Databases: Consequences of the Concept Time. Ninth International Conference on Very Large Data Bases (1983) 399-416
9. Theodoulidis, C., Loucopoulos, P., Wangler, B.: A Conceptual Modelling Formalism for Temporal Database Applications. Information Systems 16(4) (1991) 401-416
10. Goralwalla, I., Ozsu, M.: An Object-Oriented Framework for Temporal Data Models. Spriner-Verlag, ABerlin Heidelberg (1998)
11. Dey, D., Barron, T., Storey, V.: A Conceptual Model for the Logical Design of Temporal Databases. Decision Support Systems 15 (1995) 305-321
12. Maedche, A., Motik, B., Stojanovic, L.: Managing Multiple and Distributed Ontologies on the Semantic Web. The VLDB Journal – Special Issue on Semantic Web 12 (2003) 286-302.
13. Maeche, A., Motik, B., Stojanovic, L., Studer, R., Volz, R.: An infrastructure for Searching, Reusing and Evolving Distributed Ontologies. Twelfth International World Wide Web Conference (WWW 2003), Budapest Hungary (2003) 51-62