Event-Driven Ontology Updating

Jordy Sangers, Frederik Hogenboom, and Flavius Frasincar

Erasmus University Rotterdam
PO Box 1738, NL-3000 DR
Rotterdam, the Netherlands

`jordysangers@hotmail.com`, {`fhogenboom, frasincar`}`@ese.eur.nl`

Abstract. Ontologies, as reliable resources in decision making processes, need to be accurate and up-to-date. For this purpose, ontologies have to be maintained regularly. Manual updating is tedious and time consuming, therefore we propose an event-driven automated ontology updating approach. The Ontology Update Language (OUL) and our proposed extensions are inspired by the existing SQL-triggers mechanism and make use of SPARQL and SPARQL/Update statements. We propose different execution models, providing flexibility with respect to the update process. As a proof-of-concept, we implement the language and its execution models in the Hermes News Portal (HNP), an ontology-based news personalization service.

1 Introduction

One of the most important driving factors for information in today's society is news. Every day, millions of people try to keep up-to-date with the latest developments by reading news items. Next to television and newspapers, the World Wide Web has become a good alternative for people to keep track of the state of the world. Developments in the real world – described in news items – influence a variety of activities, ranging from individual daily activities such as buying products to companies' long-term business strategies. Lately, there has been an increasing amount of effort put into automatically processing news data by extracting important information. Applications that make use of this information are plentiful, e.g., automated stock agents that keep track of financial news to exploit extracted knowledge on the stock market, news personalization services that provide users with information that matches user interests, etc.

Traditionally, news is presented as plain text and can be characterized as unstructured data, making it hard for computer systems to interpret it. With the Semantic Web, the World Wide Web Consortium (W3C) provides a framework to add structure to data through the usage of the Web Ontology Language (OWL) [1]. By means of ontologies, domain specific knowledge can be represented by creating concepts and relations between these concepts. The relations are established by defining triples that consist of a subject, a predicate, and an object.

With structured data, information can be easily extracted, and interoperability between computer systems is stimulated. This information, often described using ontologies, is used as an information source that influences the systems' actions. Due to the non-static nature of our society, the information that reflects the real world at any given time has to be updated regularly. Traditional data sources like relational databases have mechanisms for automatic updates. However, a principled way of automatic ontology updating does not yet exist. This forces domain experts to manually update ontologies, which is a tedious, repetitive, error-prone, and time-consuming job.

Numerous applications, like the Hermes News Portal [7] (an ontology-based news personalization service) take advantage of Web news items by exploiting their information through ontology matching. As a classification and querying tool, it is important that the ontology contains up-to-date information. However, such tools often lack an update language for maintaining underlying ontologies and would therefore benefit from an ontology update language. Hence, we propose the use of the event-triggered Ontology Update Language (OUL), where events are defined as phenomena requiring a knowledge base to be updated. In this paper we hypothesize that the language could be extended with techniques from active databases, i.e., with features like prefixes and negation, and also by defining various update execution mechanisms.

2 Related Work

Due to the recent explosion in (meta-)data representation technologies, information can be described in many ways. One way to do this is by making use of relational databases, which store information in tables related to each other. Additionally, the eXtensible Markup Language (XML) [3] can describe the information in a tree-structure, a common way for transportation of information between systems. Last, semantic languages for storing information exist. The Resource Description Framework (RDF) [4] adds meaning to data by using triples and can be serialized in XML. OWL extends RDF with the possibility to express additional constraints and is often used as an ontology representation language.

Common languages for retrieving information from sources are the Structured Query Language (SQL) [5] for relational databases, XPath [6] and XQuery [2] for XML documents, and SPARQL [11] for RDF and OWL. Although SQL is mainly used for querying information from tables, extra functionalities have been added to it, such as the creation, alteration, and removal of tables. These statements can be executed individually, but can also be used in combination with SQL triggers. These triggers react on predefined events based on an Event-Condition-Action model and execute SQL statements either immediately or deferred if a condition is met, hereby creating an automated way of updating relational databases.

Updating XML documents can be realized with XUpdate [9], and updating ontologies is usually done with SPARQL/Update statements [13]. These statements are similar to SPARQL queries, though specifically designed for updating ontologies. Due to the complexity of ontology updating caused by dependencies and physical distributions, we need a principled approach for automatic ontology updating. The Ontology Update Language (OUL) [10] is a blend of active (database) triggers and SPARQL/Update statements, which updates ontologies in an event-driven manner. By defining so-called *changehandlers*, specific ontology change events can be caught and handled individually.

Despite the convenient representation aspects of OUL inspired from active database triggers, the usage of SPARQL and SPARQL/Update, and the implementation of preconditions, the language lacks several key features. First, OUL does not support negation and namespaces. Second, chaining of triggers (changehandlers) is not possible. The changehandlers do not react on actions of other changehandlers. In order to trigger a changehandler, the user has to manually execute an update. Third, there is no differentiation between the order of execution of changehandlers' actions, i.e., there is no distinction between immediate (i.e., once a changehandler is matched, it is executed) and deferred (i.e., the actions are executed all at once after matching the changehandlers and collecting the actions) executions. Fourth, only the first matching changehandler is executed. Hence, when an event occurs, each changehandler is matched against the event; the first changehandler that matches, is handled. Additionally, when updates are triggered and executed, new updates could be triggered, requiring another update cycle. This kind of execution looping is currently not supported.

3 OUL Syntax

Atomic ontology update actions can be executed using SPARQL/Update statements. However, multiple ontology change actions are often required. These actions are hard to express in one single SPARQL/Update statement and can not be edited easily. Therefore, complex ontology updates should be performed as a sequence of atomic SPARQL/Update statements executed in a specific order. The Ontology Update Language (OUL) [10] is based on the automatic update mechanism in active databases: SQL-triggers. Using an Event-Condition-Action model, a list of ontology update actions are performed on event occurrence. This method, using triggers (called changehandlers here), however, does not support a fully automated ontology update process. OUL does feature a dynamic update process using an existing RDF update language, and hence we extend this language in such a way that no human intervention is needed for multiple updates.

OUL makes use of changehandlers that perform SPARQL/Update actions whenever a certain change event (represented as an RDF-graph that is either added to or deleted from the ontology) occurs. If we want to perform particular actions, whenever such triple is added to or deleted from the ontology, we can specify them in a changehandler. Each changehandler has a general form as:

```
CREATE CHANGEHANDLER <name>
FOR <changerequest>
AS
  [ IF <precondition>
    THEN ] <actions>
```

which is analogous to active database triggers. When the *changerequest* matches the change event, a *precondition* on the ontologies is checked. If this precondition is met or if no precondition has been defined, a list of *actions* will be executed. In contrast to active databases, ontology updates do not require SQL statements, but events, conditions, and actions have to be defined using SPARQL and SPARQL/Update statements.

3.1 Requesting Changes

OUL defines two different types of changerequests, i.e., insertion and deletion of information. The `add` and `delete` keywords distinguish between the two different types and every changerequest is further defined by a `WHERE`-clause of a SPARQL `SELECT` query. The syntax is defined as:

```
<changerequest> ::== add [unique] (<SPARQL>)
                   | delete [unique] (<SPARQL>)
<SPARQL>        ::== WHERE clause of a SPARQL SELECT query
```

When all the triples in the query can be deduced from a change event and the event-type matches the changerequests' type, the changerequest is matched. The set of bindings that are returned from the query can be reused later in the `AS`-clause of the changehandler definition. A `unique` property can be used to state whether only one single binding is required. Whenever this property is set, changerequests will not match when their query returns multiple bindings.

3.2 Preconditions

Whenever a changerequest matches, also optional preconditions defined in the changehandler have to be met so that the actions are executed. In contrast to the changerequest, which is used to match the occurring event, the precondition is used to check the current state of the ontology. Three different types of preconditions can be used. First, `contains` checks whether the ontology contains a set of triples. Second, `entails` checks whether the ontology entails a set of triples, i.e., using inferencing it can be concluded that the statement is logically entailed by the ontology. Third, `entailsChanged` checks whether the direct application of the requested change leads to an ontology which entails a set of triples.

Conditions can be combined by `and`- or `or`-operators and can be nested as well. Each precondition results in a set of bindings and the `and`- and `or`-operators perform join and union operations on the resulting bindings. The syntax for the precondition is defined as follows:

```
<precondition>  ::== contains(<SPARQL>)
                   | entails(<SPARQL>)
                   | entailsChanged(<SPARQL>)
                   | (<precondition>)
                   | <precondition> and <precondition>
                   | <precondition> or <precondition>
<SPARQL>        ::== WHERE clause of a SPARQL SELECT query
```

3.3 Actions

When the changerequest is matched and the precondition is met, a list of actions is executed. Actions make use of the binding information that resulted from matching the changerequest and the precondition. There are four types of actions, i.e., SPARQL/Update queries, `feedback` actions that give feedback to the user using text containing bounded variables, `applyRequest` actions that execute the events caught by the changehandler, and last, the `for` actions that iteratively execute a set of actions with binding information from a for-condition:

```
<actions>        ::== [<action>]|<action><actions>
<action>         ::== <SPARQL update>
                    | for( <precondition> ) <actions> end;
                    | feedback(<text>)
                    | applyRequest
<SPARQL update> ::== a MODIFY action (in SPARQL/Update)
<text>           ::== string (may contain SPARQL variables)
```

3.4 Extensions

In SPARQL it is possible to define prefixes, i.e., labels referring to a namespace. Since in OUL multiple SPARQL `WHERE` clauses and SPARQL/Update `MODIFY` clauses may be used, it is necessary to define in each query the used prefixes or to use the full namespaces. The latter provides too much overhead and hence, we propose to define the prefixes for the entire changehandler instead of for every separate SPARQL query.

In OUL, preconditions can be combined by using `or`- or `and`-operators. It is, however, not possible to use negation, something that could be desirable whenever ontologies should not contain certain information. Therefore, we implement negation by allowing the usage of an exclamation mark ('!') to denote negation in OUL. The syntax is altered as follows:

```
CREATE CHANGEHANDLER <name>
[<prefixes>]
FOR <changerequest>
AS
  [ IF <precondition>
    THEN ] <actions>

<prefixes>       ::== <prefix>[<prefixes>]
<prefix>         ::== <SPARQL prefix>
<changerequest> ::== add [unique] (<SPARQL>)
                    | delete [unique] (<SPARQL>)
<precondition>  ::== contains(<SPARQL>)
                    | entails(<SPARQL>)
                    | entailsChanged(<SPARQL>)
                    | (<precondition>)
                    | <precondition> and <precondition>
```

```
                 | <precondition> or <precondition>
                 | !<precondition>
<actions>       ::== [<action>]|<action><actions>
<action>        ::== <SPARQL update>
                 | for( <precondition> ) <actions> end;
                 | feedback(<text>)
                 | applyRequest
<SPARQL prefix> ::== PREFIX statement of a SPARQL query
<SPARQL>        ::== WHERE clause of a SPARQL SELECT query
<SPARQL update> ::== a MODIFY action (in SPARQL/Update)
<text>          ::== string (may contain SPARQL variables)
```

4 OUL Execution Models

Updating ontologies in an event-driven manner requires an execution model that controls aspects like selecting the proper changehandlers, executing SPARQL queries, and performing changehandler actions. In [10], the authors provide an execution environment for OUL that allows for ontology updating upon detection of change events in texts. The Ontology Update Manager plays a central role here, as it matches changehandlers based on a changerequest and executes the actions defined in the respective changehandlers. The ontology update specification describes how the ontology can be updated by providing a set of changehandlers.

By default, whenever a change event occurs, all changehandlers defined in the ontology update specification are checked upon their changerequest and precondition to determine if the change event can be handled by a specific changehandler. When a changehandler matches a changerequest with a change event and the precondition is met, the original change event is replaced by the actions defined in the matching changehandler. These actions are then stored and executed all at once later on, i.e., in a deferred manner. In situations where multiple changehandlers match a change event and meet their precondition, only the actions of the first matching changehandler are executed. As OUL does not feature chaining of changehandlers, the execution of the actions cannot trigger other changehandlers, implying that immediate execution would have the same results as deferred execution, when executing only the first matched changehandler.

With respect to the original OUL execution model, we propose several extensions. First, inspired by applications in active databases, we extend OUL by adding support for immediate updating, as opposed to deferred updating. Next, in analogy with active databases where triggers can activate other triggers, we add changehandler chaining. Although this does not ensure termination, it enhances the expressivity of the update language and it enables separation of atomic update operations, thereby enabling modularity. Similarly to active databases triggers, methods for automatic termination evaluation can be developed [12]. Additionally, execution looping is added, which is needed in situations where new updates are required after triggering and executing other updates. Last, we update the OUL execution model in a way that it does not only execute the first matching changehandler, but optionally each matched changehandler.

4.1 Deferred and Immediate Updates

The original (deferred) execution model of OUL comprises three main steps, which are illustrated in Algorithm 1. First, changerequests of all defined changehandlers with respect to the change event are matched and preconditions are verified. Second, actions are collected from the matched changehandlers and SPARQL/Update statements are created. Third and last, the latter statements are applied to the ontology. Note that the method $matchHandlers(\ldots)$ is further specified in Algorithms 3 (first match) and 4 (all matches), and $collectUpdates(\ldots)$ is described in Algorithm 5.

This execution model can be altered in such a way that immediate updating is performed. This implies that during the collection process, update statements are applied immediately to the ontology. Hence, in contrast to deferred updating, we distinguish between two steps, i.e., changehandler matching and update application. Algorithm 2 provides the immediate updating model. Note that the method $matchHandlers(\ldots)$ is further specified in Algorithms 3 (first match) and 4 (all matches), and $applyUpdates(\ldots)$, which applies updates, is described in Algorithm 6.

4.2 First and All Matching Changehandlers

There are two distinct ways of matching changehandlers. The OUL execution model proposed in [10] returns the first changehandler that matches a changerequest and meets its precondition (Algorithm 3). An iterator moves forward through the ontology update specification document until either the end of the document has been reached or a changehandler has been matched to the change event. The matching process returns non-empty binding information which should contain a single binding when a unique keyword is used in the changerequest. For matching preconditions, in case a valid binding is returned, the changehandler is added to the list of matched changehandlers.

However, one could also require multiple changehandlers to be matched. When altering Algorithm 3 by changing the loop conditions, we obtain an execution model that returns all matching changehandlers associated with a change event as given in Algorithm 4. While in Algorithm 3 in line 2 a condition for limiting the list of matched changehandlers is defined, in Algorithm 4, this is removed, making it possible to check all changehandlers defined in the ontology update specification and to add every matching changehandler to the resulting list.

4.3 Chaining Updates

After matching the changehandlers (either the first changehandler encountered, or all changehandlers), their associated update statements have to be collected and applied. This stage depends on the type of execution mechanism. In case deferred execution is applied, all update statements from the matched changehandlers have to be collected before executing them. When immediate execution is used, the statements have to be executed while inspecting them.

Algorithm 1 Deferred ontology updating (updateOntology)

Description: Update ontology with deferred execution of updates
Input: ontology O consisting of axioms,
change event *op(Ax)* where $op \in \{add, del\}$ and Ax is a set of axioms
Data: *matchedHandlers* changehandlers that match their changerequest and meet their precondition according to the provided change event,
updateList list of update actions to be applied to the ontology
Output: updated ontology O

```
1: // Find matched changehandlers
2: $matchedHandlers \leftarrow matchHandlers(O, op(Ax))$
3: // Collect updates from changehandlers
4: $updateList \leftarrow collectUpdates(O, op(Ax), matchedHandlers)$
5: // Apply updates to ontology in deferred way
6: for all $update \in updateList$ do
7:     apply $update$ to $O$
8: end for
9: return $O$
```

Algorithm 2 Immediate ontology updating (updateOntology)

Description: Update ontology with immediate execution of updates
Input: ontology O consisting of axioms,
change event *op(Ax)* where $op \in \{add, del\}$ and Ax is a set of axioms
Data: *matchedHandlers* changehandlers that match their changerequest and meet their precondition according to the provided change event
Output: updated ontology O

```
1: // Find matched changehandlers
2: $matchedHandlers \leftarrow matchHandlers(O, op(Ax))$
3: // Apply updates to ontology in an immediate way
4: $O \leftarrow applyUpdates(O, op(Ax), matchedHandlers)$
5: return $O$
```

Algorithm 3 Returning the first matching changehandler (matchHandlers)

Description: Collect matching changehandlers
Input: ontology O consisting of axioms,
ontology update specification US treated as a list of changehandlers,
change event *op(Ax)* where $op \in \{add, del\}$ and Ax is a set of axioms.
Data: *handler* changehandler that is checked for applicability
Output: list of matched changehandlers *matchingHandler*

```
1: // While not at document's end and no changehandler has been matched
2: while not $US.endOfDocument$ and $matchingHandlers.count < 1$ do
3:     // Take the next changehandler
4:     $handler \leftarrow US.nextChangeHandler$
5:     // Match the changerequest with the change event
6:     $matches \leftarrow SPARQLmatch(handler.changerequest, op(Ax))$
7:     // The bindings form the changerequest should not be empty
8:     if not $matches.isEmpty$ then
9:         // The number of bindings should be 1 when the unique keyword is used
10:        if ($handler.changerequest.unique$ and $matches.count$ == 1) or not
           $handler.changerequest.unique$ then
11:            // Substitute variables in the precondition with changerequest bindings
12:            $instPrecondition \leftarrow substitute(handler.precondition, matches.first)$
13:            // Evaluate the precondition. When this returns any binding, it is met
14:            if not $evaluate(instPrecondition, O).isEmpty$ then
15:                // Add the changehandler to the list
16:                $matchingHandlers.add(handler)$
17:            end if
18:        end if
19:    end if
20: end while
21: // Return the list of matched changehandlers
22: return $matchingHandlers$
```

Algorithm 4 Returning all matching changehandlers (matchHandlers)

Description: Collect matching changehandlers
Input: ontology O consisting of axioms,
ontology update specification US treated as a list of changehandlers,
change event $op(Ax)$ where $op \in \{add, del\}$ and Ax is a set of axioms.
Data: *handler* changehandler that is checked for applicability
Output: list of matched changehandlers *matchingHandler*

```
// While not at document's end
while not $US.endOfDocument$ do
    // Take the next changehandler
    $handler \leftarrow US.nextChangeHandler$
    // Match the changerequest with the change event
    $matches \leftarrow SPARQLmatch(handler.changerequest, op(Ax))$
    // The bindings form the changerequest should not be empty
    if not $matches.isEmpty$ then
        // The number of bindings should be 1 when the unique keyword is used
        if ($handler.changerequest.unique$ and $matches.count$ == 1) or not
        $handler.changerequest.unique$ then
            // Substitute variables in the precondition with changerequest bindings
            $instPrecondition \leftarrow substitute(handler.precondition, matches.first)$
            // Evaluate the precondition. When this returns any binding, it is met
            if not $evaluate(instPrecondition, O).isEmpty$ then
                // Add the changehandler to the list
                $matchingHandlers.add(handler)$
            end if
        end if
    end if
end while
// Return the list of matched changehandlers
return $matchingHandlers$
```

Algorithm 5 Update collection from matched changehandlers (collectUpdates)

Description: Collect updates from a list of matched changehandlers using deferred execution
Input: ontology O consisting of axioms,
change event $op(Ax)$ where $op \in \{add, del\}$ and Ax is a set of axioms,
list of changehandlers *matchedHandlers* that match the change event
Output: list of the update statements *updateList*

```
// Check whether any changehandler matches the change event
if not $matchedHandlers.isEmpty()$ then
    // Loop through all matched changehandlers
    for all $matchedHandler \in matchedHandlers$ do
        // Loop through all update statements in the changehandler
        for all $update \in matchedHandler.updates$ do
            // Chaining: add the update or the replaced update actions from other
            //              changehandlers to the list of update statements
            // Find changehandlers that match the update event
            $newMatchedHandlers \leftarrow matchHandlers(O, update)$
            // Collect updates from changehandlers
            $newUpdateList \leftarrow collectUpdates(O, update, newMatchedHandlers)$
            // Add the update statements to the list
            $updateList.add(newUpdateList)$
        end for
    end for
else
    // There is no changehandler matching the change event; therefore, the change
    // event itself is added to the list of update statements
    $updateList.add(op(Ax))$
end if
// Return the list of update statements
return $updateList$
```

The earlier introduced Algorithm 1 defines the execution steps of the deferred execution model. In the first step, update collection, statements are collected from the matched changehandlers. Algorithm 5 explains how this task is performed. First, a check is done to investigate whether any changehandlers match the change event. If this is the case, the update statements from every changehandler in the set of matched changehandlers are collected. If no changehandler matches the change event, the change event itself is applied to the ontology. In lines 7-14, each update statement is treated as a change event, representing the implementation of chaining. This part is similar to Algorithm 1, except for the fact that updates are not applied to the ontology, because this has to happen at the end of the process when using deferred execution. In the end, the algorithm returns a list of update statements that need to be applied to the ontology.

As depicted in Algorithm 6, for immediate ontology updating, no update lists are returned. In contrast to deferred updating, the updates are immediately applied to the ontology. For immediate updating, the algorithm first checks whether any changehandler exists in the list of matched changehandlers. If this is the case, for each update statement in each of the matched changehandlers, a change event is fired as shown in Algorithm 2 using the update as the change event. In this way, we provide a mechanism for chaining. If no changehandler matches the change event, the change event itself is applied to the ontology.

4.4 Looping Updates

Applying updates to the ontology and thereby changing the ontology can trigger new changehandlers to become matched, which can be used for applying additional updates in case the event is handled more than once. Hence, we introduce the possibility to iterate over the changehandlers with the same event and apply updates until there are no matching changehandlers left. In this way, the effect of the update actions can be checked and additional updates can be applied.

Ontology update looping can be implemented by adding a call to the *updateOntology*(...) methods of Algorithms 1 (deferred) and 2 (immediate) at the end of both algorithms, using the same change event and ontology as input. Algorithms 7 and 8 implement looping for deferred and immediate executions, respectively. Before the updated ontology is returned, in both algorithms the *updateOntology*(...) method is called recursively to ensure that additional updates are handled until no updates are available.

5 Implementation

OUL, including the proposed extensions, has been implemented as a stand-alone software package providing event-driven ontology updates (available at `http://people.few.eur.nl/fhogenboom/oulx.html`). We used this package in the Hermes News Portal [7], a Java-based news personalization tool implementing the Hermes framework [7]. Hermes uses an ontology for classifying and querying news items. The Hermes domain ontology has to be up-to-date with the

Algorithm 6 Update application from matched changehandlers (applyUpdates)

Description: Apply updates from a list of matched changehandlers using immediate execution
Input: ontology O consisting of axioms,
change event *op(Ax)* where $op \in \{add, del\}$ and Ax is a set of axioms,
list of changehandlers *matchedHandlers* that match the change event
Output: updated ontology O

```
 1: // Check whether any changehandler matches the change event
 2: if not matchedHandlers.isEmpty() then
 3:     // Loop through all matched changehandlers
 4:     for all $matchedHandler \in matchedHandlers$ do
 5:         // Loop through all update statements in the changehandler
 6:         for all $update \in matchedHandler.updates$ do
 7:             // Chaining: fire the update as an update event; this way, the update can
 8:             //              be handled by appropriate changehandlers
 9:             $updateOntology(update)$
10:         end for
11:     end for
12: else
13:     // There is no changehandler matching the change event; therefore, the change
14:     // event itself is applied
15:     apply op(Ax) to $O$
16: end if
17: return $O$
```

Algorithm 7 Looped deferred ontology updating (updateOntology)

Description: Update ontology with deferred execution of updates and looping
Input: ontology O consisting of axioms,
change event *op(Ax)* where $op \in \{add, del\}$ and Ax is a set of axioms
Data: *matchedHandlers* changehandlers that match their changerequest and meet their precondition according to the provided change event,
updateList list of update actions to be applied to the ontology
Output: updated ontology O

```
 1: // Find matched changehandlers
 2: $matchedHandlers \leftarrow matchHandlers(O, op(Ax))$
 3: // Collect updates from changehandlers
 4: $updateList \leftarrow collectUpdates(O, op(Ax), matchedHandlers)$
 5: // Apply updates to ontology in deferred way
 6: for all $update \in updateList$ do
 7:     apply $update$ to $O$
 8: end for
 9: // Execute this algorithm again to check for additional updates
10: if not matchedHandlers.isEmpty() then
11:     $updateOntology(O, op(Ax))$
12: end if
13: return $O$
```

Algorithm 8 Looped immediate ontology updating (updateOntology)

Description: Update ontology with immediate execution of updates and looping
Input: ontology O consisting of axioms,
change event *op(Ax)* where $op \in \{add, del\}$ and Ax is a set of axioms
Data: *matchedHandlers* changehandlers that match their changerequest and meet their precondition according to the provided change event
Output: updated ontology O

```
1: // Find matched changehandlers
2: $matchedHandlers \leftarrow matchHandlers(O, op(Ax))$
3: // Apply updates to ontology in an immediate way
4: $O \leftarrow applyUpdates(O, op(Ax), matchedHandlers)$
5: // Execute this algorithm again to check for additional updates
6: if not matchedHandlers.isEmpty() then
7:     $updateOntology(O, op(Ax))$
8: end if
9: return $O$
```

latest news and hence needs automatic ontology updates. Based on the information extraction plugin for the Hermes News Portal, i.e., Aethalides, information extracted from news items can be used for updating the ontology.

A key aspect of the Hermes News Portal is its financial ontology. For its updates, the ontology is dependent on information extracted from financial news messages, e.g., product releases, CEO appointments, bankruptcies, etc. We implement the OUL update mechanisms and connect them to the information extraction processes of Aethalides. The Aethalides plugin makes use of the Hermes Information Extraction Engine, which is used for matching user-created information extraction rules with text in news items.

To integrate automatic ontology updating, each new news item is processed and information (in the form of events) is extracted using the user-created rules. After validating the extracted information, the ontology is updated using OUL update rules. In our implementation, the execution of the SPARQL `WHERE` clauses and the SPARQL/Update statements, as well as ontology updating is performed using Jena [8]. In order to work with the latest developments in the Semantic Web, we updated ARQ, the query engine in Jena, to version 2.8.8, which features SPARQL 1.1. The changehandlers can be loaded via a plain text file that contains changehandlers specified in the proposed syntax. Parsing and compiling of changehandlers is performed via a compiler created with JavaCC [14].

6 Evaluation

In order to evaluate the extensions made to OUL, we analyze the characteristics of each proposed execution model. As it is difficult to perform a quantitative analysis and as there are no benchmarks available for OUL, we discuss at a qualitative level the advantages and disadvantages of each execution model. We assume all queries are chained (non-chained queries as originally proposed by OUL are also supported), which provides us with eight execution models:

- Immediate, looped execution of first matching changehandler;
- Immediate, non-looped execution of first matching changehandler;
- Immediate, looped execution of all matching changehandlers;
- Immediate, non-looped execution of all matching changehandlers;
- Deferred, looped execution of first matching changehandler;
- Deferred, non-looped execution of first matching changehandler;
- Deferred, looped execution of all matching changehandlers;
- Deferred, non-looped execution of all matching changehandlers.

Deferred execution of matching changehandlers could lead to erroneous updates, and hence it usually does not make sense to make use of the last four execution models. For example, it could be the case that several changehandlers can originally match, but after executing their corresponding updates in a deferred mode, the updates of the previous matches could be made invalid. Due to the nature of deferred execution, these updates would still be executed. On the other hand, deferred updating could possibly lead to more efficient updates in case multiple changes are to be made to the same entity, as these actions could be merged and transformed into simplified update statements. Additionally, duplicate actions can be merged, eliminating duplicate action executions. So, if deferred updating is used, some caution is required, making sure there are no conflicting dependencies between update actions and change requests.

When comparing models that execute the first matching changehandler with those that execute all changehandlers, one could make the following observations. The latter method is computationally more intensive due to the increased complexity on the execution mechanism. Conversely, updates are more efficient, as in one pass all the matched changehandlers are dealt with, hence eliminating the need for multiple user-triggered iterations.

In case of looped execution models, the advantage is that ontology updates performed during a pass that trigger new changehandlers to be matched are taken into account, hereby improving the efficiency of the ontology updating process, as no separate runs are needed. The looped execution models are on the other hand harder for users to grasp due to the repeated event generation until no changehandler matches the event.

There is a trade-off between easiness of writing update rules and their efficient execution. The all matching and/or looped variants are more efficient due to the automatic execution and possible optimization of their complex actions, while the first matching and/or non-looped counterparts are more intuitive and thus foster easier development of update rules. Also, it should be noted that for chaining there is an increased level of automation, as users do not have to manually trigger updates resulting from earlier updates (as in case of the OUL execution model), as these are automatically handled.

7 Conclusions

The Ontology Update Language (OUL) is based on SQL triggers and focuses on an event-driven ontology update specification. By creating changehandlers, containing an event description, a precondition, and a list of SPARQL/Update statements, update actions are executed when events occur and preconditions are met. OUL features the creation of Event-Condition-Action rules, hereby enabling automatic updates. We identified some drawbacks at language as well as execution model levels, and proposed extensions to address these.

Syntax-wise, in order to facilitate more complex expressions, we extended OUL so that it also supports negation and prefixes. Our main contribution however lies in the extension of OUL's execution mechanism. We incorporated immediate updating, as opposed to deferred updating. Also, we added an internal triggering mechanism for changehandlers called updates chaining, allowing for automatic event triggering based on the matched changehandlers' actions. This contributes to the usability of the language by separating atomic update actions and thus delivering modularity and an increased possibility to reuse changehandlers. Also, we added support for looping for repetitive treatment of an event. Last, it is now also possible to execute all event-related changehandlers, instead of just the first matching handler. The here proposed extensions are viable, provided that technical experts who are accustomed to the update language work together with experts of the knowledge domain.

As future work we would like to evaluate the termination of changehandlers, i.e., which conditions need to be satisfied by a set of changehandlers so that, for any incoming events, the matching changehandlers should always terminate. For this purpose we plan to reuse results from termination of rule-based updates for databases [12]. Alternatively, one could look into developing a principled information extraction language that combines information extraction and ontology updates. For this purpose, we plan to integrate the Hermes Information Extraction Language with OUL, including the here proposed extensions.

References

1. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference. W3C Recommendation 10 February 2004, from: `http://www.w3.org/TR/owl-ref/`
2. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J.: XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation 14 December 2010, from: `http://www.w3.org/TR/xquery/`
3. Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F.: Extensible Markup Language (XML). W3C Recommendation 26 November 2008, from: `http://www.w3.org/TR/2008/REC-xml-20081126/`
4. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation 10 February 2004, from: `http://www.w3.org/TR/rdf-schema/`
5. Chamberlin, D.D., Boyce, R.F.: SEQUEL: A Structured English Query Language. In: Rustin, R. (ed.) 1974 ACM SIGMOD Workshop on Data Description, Access and Control. vol. 1, pp. 249–264. ACM (1974)
6. Clark, J., DeRose, S.: XML Path Language (XPath). W3C Recommendation 16 November 1999, from: `http://www.w3.org/TR/xpath/`
7. Frasincar, F., Borsje, J., Levering, L.: A Semantic Web-Based Approach for Building Personalized News Services. International Journal of E-Business Research 5(3), 35–53 (2009)
8. HP Labs: Jena (2011), from: `http://jena.sourceforge.net/`
9. Laux, A., Martin, L.: XUpdate (2000), from: `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`
10. Lösch, U., Rudolph, S., Vrandečić, D., Studer, R.: Tempus Fugit. In: 6th European Semantic Web Conference on The Semantic Web: Research and Applications (ESWC 2009). pp. 278–292. Springer-Verlag (2009)
11. Prud'hommeaux, E., Seaborne, A.: SPARQL. W3C Recommendation 15 January 2008, from: `http://www.w3.org/TR/rdf-sparql-query/`
12. Ray, I., Ray, I.: Detecting Termination of Active Database Rules Using Symbolic Model Checking. In: Advances in Databases and Information Systems, Lecture Notes in Computer Science, vol. 2151, pp. 266–279 (2001)
13. Seaborne, A., Manjunath, G., Bizer, C., Breslin, J., Das, S., Davis, I., Harris, S., Idehen, K., Corby, O., Kjernsmo, K., Nowack, B.: SPARQL Update. W3C Member Submission 15 July 2008, from: `http://www.w3.org/Submission/SPARQL-Update/`
14. Sun Microsystems: JavaCC (2011), from: `http://javacc.java.net/`