SPARQL-based framework for semantically-based event processing

Ana SASA BASTINOS[a,1], Dejan LAVBIC[b]
[a]*fluidOperations fluid Operations AG, Walldorf, Germany*
[b]*Faculty of Computer and Information Science, University of Ljubljana, Slovenia*

Abstract. Event-driven architecture and complex event processing have become important topics in achieving business reactivity and proactivity. Even though many technologies to support this have been developed, there is still a need for providing support for complex events based on different levels of required expressivity. On the other hand, semantic technologies and semantic-based systems have emerged and are becoming more and more used, whereas there is no recognized solution for event processing in information systems based on semantic technologies. In this paper, we address these issues. We present a framework for ontology-based support for complex events, which allows for semantically enriched event definitions and automatic recognition. In order to automatically recognize complex events, an appropriate event definition basis has to be defined. The paper represents a continuation of our previous work by enhancing this basis and developing a SPARQL-based framework to provide a richer mechanism for event definitions and more efficient event detection. As a proof-of-concept, we provide an implementation of this framework using the semantic integration platform - Information Workbench. The implementation is available as an app that runs on top of the Information Workbench platform.

Keywords. Event processing, complex event, ontology, SPARQL, RDF, OWL.

1. Introduction

With increasing demands for agility and reactivity of business processes, scientific circles and leading information technology companies have paid a lot of attention to EDA (event-driven architecture) and CEP (complex event processing). EDA and CEP enable event-driven systems - systems in which actions result from business events [1]. Even though many technologies to support this have been developed [2], there is still no recognized approach available that would provide semantically-enriched support for complex event processing based purely on standard semantic technologies. Existing EDA and CEP approaches do not take into consideration different expressivity requirements that are needed in definition of a large number of diverse complex events. Because of this, detection of complex events that require semantically expressive descriptions is not automated and experts are required to monitor business operation in order to determine if a complex event has occurred. Such an approach can decrease reactiveness and proactiveness of an organization [3][4].

We addressed some of these issues in our previous work presented in [3], where we discussed ontology-based framework for complex events, and later in [4] by

[1] Corresponding Author.

enhancing that basis and applying the aspectual model from the field of linguistics to our event ontology. In this paper, we take an alternative approach to defining the event ontology and defining event conditions. We do not use the Semantic Web Rule Language (SWRL) in order to construct rules that define when an event occurs, as we did in our previous work. Instead, we use SPARQL-based approach, which makes our framework more widely applicable to a wide range of available SPARQL-based tools and systems. Furthermore, this approach is not limited only to OWL-based systems, but can be used with any RDF-based system.

2. Background and related work

2.1. Events, event-driven architecture and complex event processing

The business operations of today's enterprises are heavily influenced by numerous of internal and external business events. With the Event Driven Architecture and particularly the Complex Event Processing (CEP), many research activities have been dedicated to technologies required for identifying complex correlations in these large amounts of event data right after its appearance [2].

An event is anything that happens, or is contemplated as happening [5]. An entity is considered event-driven when it acts in response to an event. EDA is a paradigm that describes an approach to information systems development with a focus on developing an architecture that has the ability to detect events and react intelligently to them [6]. EDA represents a complement to the service-oriented architecture (SOA), which has become one of the most recognized paradigms in information systems development in the recent years [7]. By enhancing the paradigm of SOA, enterprises can gain improve their ability for business transformation by implementing event-driven architectures that automatically detect and react to significant business events [6][8]. An important part of every EDA that enables and predetermines to what level a system is able to detect and respond to complex events is CEP. CEP refers to computing that performs operations on complex events, including reading, creating, transforming, abstracting, or discarding them [5].

The event-related terminology that we use in this paper, complies to the Event Processing Glossary – Version 2.0 [5]. We use the following terms from the glossary:

- Event (event object): An object that represents, encodes, or records an event, generally for the purpose of computer processing.
- Event type: A class of event objects.
- Simple event: An event that is not viewed as summarizing, representing, or denoting a set of other events.
- Complex event: An event that summarizes, represents, or denotes a set of other events.
- Derived event: An event that is generated as a result of applying a method or process to one or more other events.

2.2. Ontologies

An ontology represents an explicit specification of a conceptualization [9]. There are several languages available for ontology representation, such as DAML, CGs, OIL, DAML+OIL, OWL (OWL 1, OWL 2). To support definition and processing of events, our work is based on OWL 2 [10]. The reasons for this are a rich and useful group of ontology features for defining and describing the domain, a high level of support, and its XML foundations, which make it appropriate to be used in conjunction with other Web technologies. The main concepts of OWL are Classes, Properties, and Individuals [10]. This means that besides an ontology that provides a conceptual representation, an OWL document can also comprise instance level descriptions of an enterprise. The main types of OWL properties are object properties and data properties. An object property relates an instance to another instance, while a data property relates an instance to a literal.

Resource Description Framework (RDF) is a directed, labeled graph data format for representing information in the Web. RDF data model has a form of subject–predicate–object expressions. RDF expressions are known as triples in the RDF terminology. A common way to persist RDF data in its native representation is in a triple store.

OWL 2 ontologies can be used along with information written in RDF, and OWL 2 ontologies themselves are primarily exchanged as RDF documents [10]. In order to define the event types, we use the SPARQL 1.1 query language [11]. SPARQL (SPARQL Protocol and RDF Query Language) is a set of specifications that provide languages and protocols to query and manipulate graph content on the Web or in an RDF store.

2.3. EDA and CEP approaches based on semantic technologies

The use of ontologies in the field of EDA and CEP has been identified as a suitable approach for the semantic definition of events by many researchers. Cheng et al. [12] have proposed a framework for context-aware processing of business rules in event-driven architectures. They have developed a context ontology in order to resolve a problem of inconsistent dictionaries at knowledge sharing and merging of rules. A key difference between their approach and ours is that they focus on achieving semantic interoperability, whereas our goal is to use ontology to define and detect complex events. Sen and Ma [13] have proposed an approach for combining reactive rules with ontologies. They have used ontologies to capture the context in which certain behaviour is appropriate, together with techniques for finding similarities with the primary objective to enable detection of similar complex event patterns. Their research is related to our approach with the key difference that they do not talk about how complex event types can be defined by the ontology, and how they can be used for detection in the context of EDA. Moser et al. [14] have proposed semantic correlation of events by using ontologies. Their work is based on the observation that the event correlation is necessary for CEP to relate events obtained from various sources in order to detect patterns and situations of the business context. Paschke [15][16] has proposed a language for semantic design of CEP patterns and a Web infrastructure for distributed rule-based event processing multi-agent eco-systems. Adaikkalavan and Chakravarthy [17] propose an interval-based event specification language developed by expressing simple and composite events that are part of active rules. By contrast with these two approaches, our approach builds on standard semantic languages, does not define a specific event-processing language, and allows using available query engines and tools that support these standards. Several authors propose domain-specific definition and detection of events by using ontologies, such as [18],[19],[20], and [21]. On the other hand, our work is generic and not domain specific. Our study of related work has shown that there is no generic complex-event processing approach available to be applied to information systems based on semantic technologies based purely on standard semantic languages and protocols.

3. Overview of the SPARQL-based framework for event processing

In order to enable CE definitions, detection and triggering based on ontologies and the OWL language, we have developed an event-driven framework based on the publish-subscribe model. It is illustrated in Fig. 1.

The core part of the framework is the RDF database containing the ontologies and instance data. The ontology is composed of two parts: the base event ontology, and the domain ontology. The base event ontology is a generic part of the framework that can be reused in different domains and environments. We have developed it with the purpose of providing the common basis for event types and events. It provides classes and relationships for event instances (events) and event types. More details on the event ontology can be found in section 4.

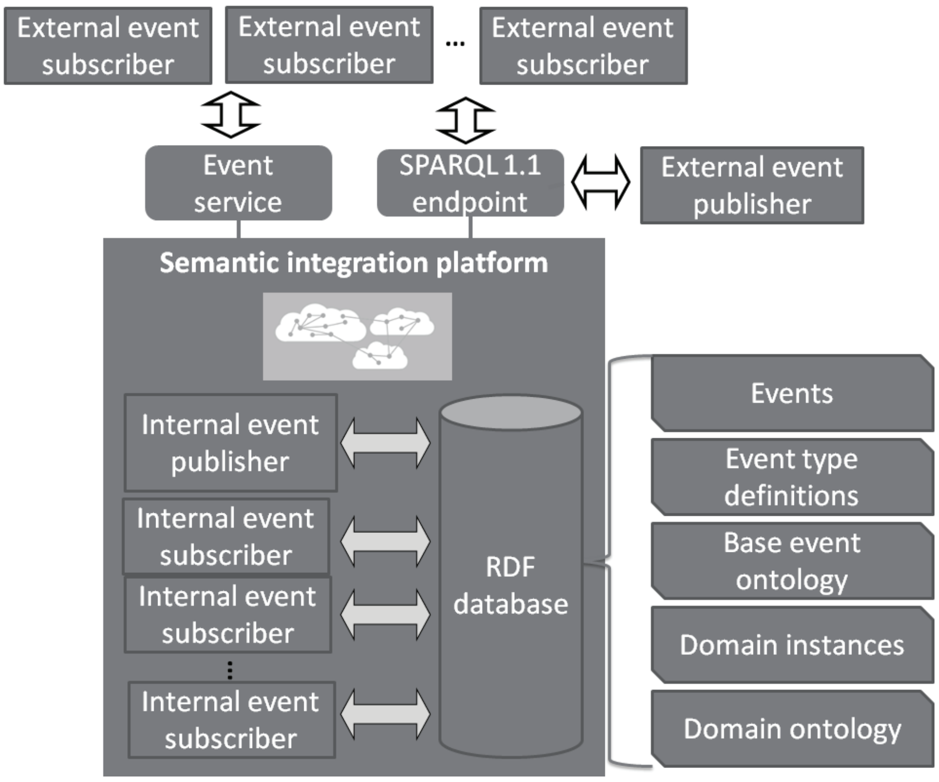
External event subscriber
External event subscriber
...
External event subscriber
Event service
SPARQL 1.1 endpoint
External event publisher
Semantic integration platform
Internal event publisher
Internal event subscriber
Internal event subscriber
Internal event subscriber
RDF database
Events
Event type definitions
Base event ontology
Domain instances
Domain ontology

Figure 1. Overview of the SPARQL-based framework for event processing.

Domain ontology is the environment specific ontology that is used in event types to specify domain-specific conditions for event triggering. Any OWL-based or RDF-based domain ontology can be used in this place.

Both base event and domain ontology are persisted in the RDF database. The RDF database also contains information about event types, event instances, and domain instances.

Producing the events and informing other components about the event that interests them is based on the publish-subscribe model. The event publisher detects when an event occurs based on the event type definition. If an event is detected, it publishes it into the RDF database as a new event together with its semantic properties. The event publisher can be a component internal to the semantic integration system (internal event publisher), or an external component (external event publisher). An external event publisher uses the SPARQL 1.1. endpoint in order to get the information from the RDF database and to publish events. In order to publish events, the external event publisher uses the CONSTRUCT SPARQL 1.1. query types.

Event subscribers can subscribe to events by specifying the event type(s) they are interested in. They can find the events invoking a parametrized predefined SPARQL query on the database, where the parameter is the event type they are interested in. Similarly, as in case of the publisher, a subscriber can be a component that is either internal or external to the system. External subscribers can use the provided SPARQL 1.1 endpoint or a customized endpoint (event service) to retrieve events based on the event type.

The publish-subscribe model used in our approach can be compared to the JMS publish-subscribe model, where the publisher publishes messages to a topic. Instead of publishing messages to a topic, in our case a new event is created of a given event type. Based on the event type, the subscriber can always find the events of the type they are interested in. In this comparison, an event type plays the role similar to the role of a topic in the JMS publish-subscribe model.

4. Base event ontology and event type definitions

Base event ontology defines classes and relationships that provide the basis for defining, detecting and publishing of events (Fig. 2). It is implemented using OWL 2 DL ontology language.

Event ontology distinguishes between the *Event* class and the *Event Type* class. Members of the *Event Type* class represent event definitions which contain information about when an event of the type occurs. Every event type carries information that uniquely defines when an event of this type occurs. On the other hand, members of the Event class are event instances with concrete information about an event that occurred. An instance of the Event class is related to its event type via *Has event type* functional object property.

An event type defines an event expression via the *Has event condition expression* property. Event expressions are the key part of our support for CEP. They represent the semantical expression that uniquely defines when an event occurs. They should be written in an agreed upon RDF-based or OWL-DL syntax. In our study, we use two types of SPARQL 1.1 query forms, i.e. the SELECT query form, and the CONSTRUCT query form. A SELECT SPARQL query returns all, or a subset of, the variables bound in a query pattern match. For the event publisher, its result indicates whether an event of the given type has occurred. If the result is an empty set, then the event did not occur. A CONSTRUCT QUERY form returns a graph, which the provider adds into the triple store, provided that the statements resulting from the query do not yet exist. The SELECT SPARQL query is used to define simple and underived complex events, while the CONSTRUCT query form is used to represent derived events.

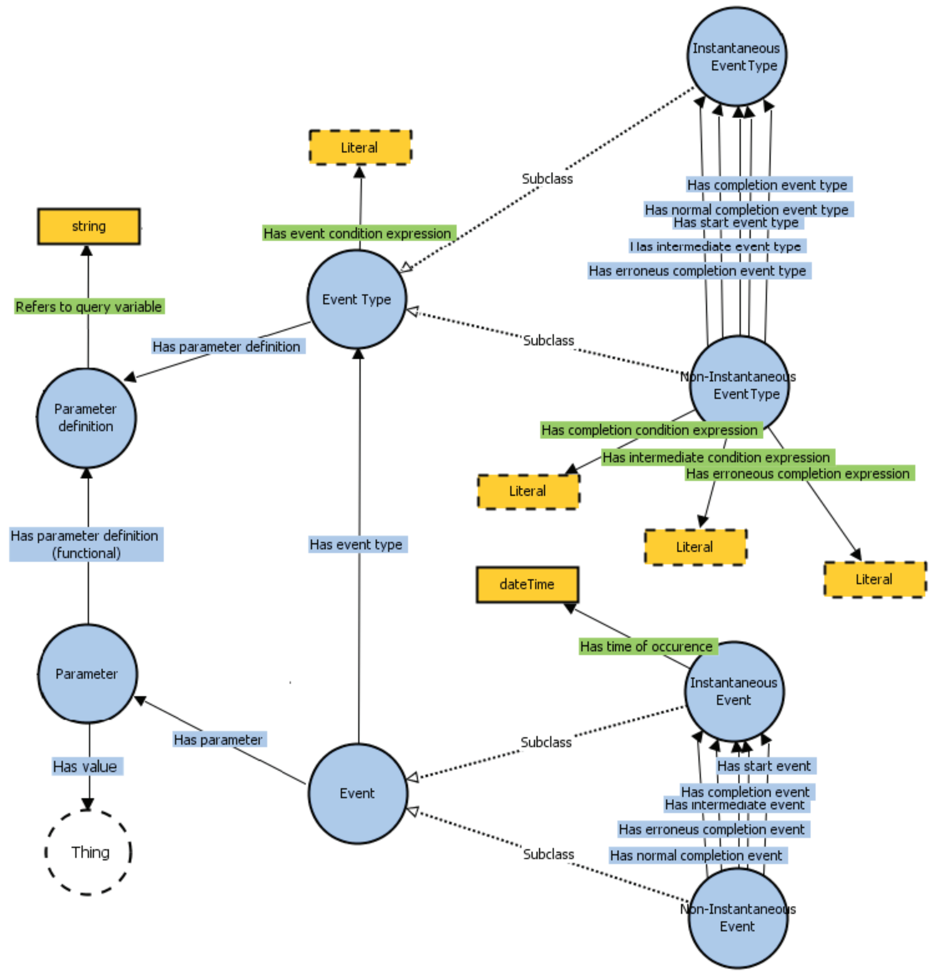
Instantaneous Event Type
Literal
Subclass
Has completion event type
Has normal completion event type
Has start event type
Has intermediate event type
Has erroneus completion event type
string
Has event condition expression
Event Type
Refers to query variable
Has parameter definition
Subclass
Non-Instantaneous Event Type
Parameter definition
Has completion condition expression
Has intermediate condition expression
Has erroneus completion expression
Literal
Has parameter definition (functional)
Has event type
Literal
Literal
dateTime
Has time of occurence
Parameter
Instantaneous Event
Has parameter
Subclass
Has value
Has start event
Has completion event
Has intermediate event
Event
Has erroneus completion event
Thing
Subclass
Has normal completion event
Non-Instantaneous Event

Figure 2. Base event ontology overview.

The condition defining when an event occurs may be true for a longer period of time, even though it actually refers to only one event. For example, the condition for an event type “Out of credit” could be that the credit balance is 0. However, this event should only be triggered once during the time when the credit balance is 0. For this reason, we may need to uniquely identify an occurrence of an event in a way that would prevent incorrect publishing of more than one event instance. For this reason, an additional check that verifies the uniqueness of the event must be included in the conditional event expression for such cases. In the given example, the unique identifier of the event would be the time when the user runs out of credit and the unique user name. Therefore, the event condition expression must also include the check if the event with such a unique identifier already exists. If it does not, a new event will not be published.

4.1. Instantaneous events vs. non-instantaneous events

In order to support different events, we distinguish between two basic types of events:

- instantaneous events, which are characterized by a point in time when they occur, and
- non-instantaneous events, which have a duration.

To reflect this in the ontology, the *Event type* class has two subclasses: *Instantaneous Event Type* and *Non-Instantaneous Event Type*. Correspondingly, Event class has two corresponding subclasses *Instantaneous Event* and *Non-Instantaneous Event.*

In comparison with an instantaneous event, a non-instantaneous event has a duration. About non-instantaneous events, one can ask a question in a continuous tense, such as “is it happening?”. This is not relevant for instantaneous events.

The conceptualization of the non-instantaneous event type, allows us to treat it in a different way that instantaneous events, which is particularly important for the event subscribers. An event subscriber may be interested when a non-instantaneous event starts, when it ends, whether it ends with one or another event type, and may also be interested in certain intermediate events that can occur between the start and the end if this event. In order to support different types of end events, the object property *Has completion event type* has two subproperties: *Has normal completion event type*, *Has erroneous completion event type*. Furthermore, one can define intermediate events using the property *Has intermediate event type*.

One can observe that the non-instantaneous event is actually a specific form of a complex event, since it is defined by related occurrence of one or more instantaneous events. An alternative approach would be to represent the non-instantaneous events by two or more separate events using the instantaneous event concept, for example by defining a separate event type designating its start, and by defining its dependent intermediate and end event types. However, with such approach the subscriber would have to be aware of all the possible subevents and subscribe to all of them. In our approach, we include the non-instantaneous event as its own concept in order to simplify its definition and implementation, especially compared to managing several separate event types that really represent one complex event. It is a very common type of an event that is widely applicable. Because of the conceptual model that supports non-instantaneous event types, implementation of the mechanism that allows sending the corresponding information to the subscribers is very straightforward.

4.2. Parameter definitions and parameters

Besides the event expression, an event type has one or more parameter definitions. A parameter definition defines which semantic properties an event has. When an event occurs, it can carry some information. An event instance is related to this information via its *Has parameter* object property. Every parameter points to exactly one parameter definition. As our approach is based on semantic technologies, the parameters are also part of the semantic world. Each parameter has a value that is either an IRI of an OWL class or a data type. An event type can have zero, one or more parameter definitions. If it has more than one, this means that the event can be related to different instances of different OWL classes or different data values with their own meaning. Event publisher retrieves the information about the event parameters from the queries using the property *Refers to query variable*, which maps the parameter definition to the query variable. In this way, the publisher has a uniquely identifiable value that it can assign to each event parameter.

5. Example

In order to illustrate the concepts presented in the previous section, this section provides example event type definitions and parameter definitions. Example domain of user account credit balance is chosen, because it is easy to understand while it provides a complex-enough basis to show the different concepts introduced in this paper.

A very common event type in the example domain is the *Credit change event type*. Its OWL definition written in the turtle format is given in Table 1. In this and the following examples, the namespace prefixes *action* and *credit* stand for namespaces from the domain ontology, whereas the namespace prefixes *event* and *eventIns* refer to the event ontology namespace and event instance namespace, respectively.

Table 1. Credit change event type

```
eventIns:CreditChangeEventType rdf:type event:EventType, event:InstantaneousEventType ;
                    rdfs:label "Credit change event type" ;
                    event:hasEventConditionExpression """

SELECT distinct ?newCreditBalance ?oldCreditBalance ?time1 ?timeOfLastChange ?user WHERE {
  ?user a <http://www.fluidops.com/User>.
?record a action:Record .
?record credit:refersToUser ?user.
?record action:startDate ?timeOfLastChange .
?record action:createdBy action:CreditBalanceChange.
?record credit:creditBalance ?newCreditBalance .

?record1 a action:Record .
?record1 credit:refersToUser ?user.
?record1 action:startDate ?time1 .
?record1 action:createdBy action:CreditBalanceChange.
?record1 credit:creditBalance ?oldCreditBalance .

FILTER NOT EXISTS {
            ?record2 a action:Record .
            ?record2 credit:refersToUser ?user.
            ?record2 action:startDate ?time2 .
            ?record2 action:createdBy action:CreditBalanceChange.
            FILTER (?timeOfLastChange > ?time2 && ?time1 < ?time2 ).
}

FILTER (?oldCreditBalance != ?newCreditBalance && ?timeOfLastChange > ?time1) .

FILTER NOT EXISTS {
            ?event a event:Event .
            ?event event:hasEventType eventIns:CreditChangeEventType.
            ?event event:hasParameter ?param .
            ?param event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
            ?param event:hasValue ?timeOfLastChange.
            ?event event:hasParameter ?userParam .
            ?userParam event:hasParameterDefinition eventIns:userParameterDefinition.
            ?userParam event:hasValue ?user.
}}
                    """ ;
                    event:hasParameterDefinition eventIns:newCreditBalanceParameterDefinition ,
                                    eventIns:oldCreditBalanceParameterDefinition ,
                                    eventIns:timeOfLastChangeParameterDefinition ,
                                    eventIns:userParameterDefinition .
```

The event expression of the *Credit change event type* defines that the event of this type occurs, when two credit records exist with a different credit balance, both of the action *CreditBalanceChange*, and there is no other record of this action that was recorded between their action times. To ensure uniqueness of the event, the second “FILTER NOT EXISTS” block of the query is used to determine that the same user does not already have a *Credit change event type* with the same time stamp.

One can observe that the Credit change event type points to four parameter definitions, which indicates that each event of this type will have exactly four parameters. An example of the *New credit balance parameter definition* is shown in Table 2. The relationship between the variable *newCreditBalance* in the SELECT SPARQL query of the event type definition and the parameter definition is given with the property *Refers to query variable* in the parameter definition.

Table 2. New credit balance parameter definition

```
eventIns:newCreditBalanceParameterDefinition rdf:type event:ParameterDefinition ;
            rdfs:label “New credit balance parameter definition”;
            event:refersToQueryVariable "newCreditBalance"^^xsd:string .
```

Table 3 shows an example of a derived event type definition. *Out of credit event type* can be derived from the *Credit change event type*. The corresponding event condition expression of this event type is a CONSTRUCT SPARQL query, which searches for events of type *Credit change event type* with the new credit balance of 0. If there is such a credit change event type, the event publisher will add a statement that this event is also of the *Out of credit event type*, unless such a statement already exists.

Table 3. Out of credit event type

```
eventIns:ZeroCreditEventType rdf:type event:InstantaneousEventType, event:EventType ;
                rdfs:label "Out of credit event" ;
                event:hasEventConditionExpression """
CONSTRUCT  {
            ?event event:hasEventType eventIns:ZeroCreditEventType .
} WHERE            {
            ?event a event:Event .
            ?event event:hasEventType eventIns:CreditChangeEventType .
            ?event event:hasParameter ?param .
            ?param event:hasParameterDefinition eventIns:newCreditBalanceParameterDefinition.
            ?param event:hasValue ?newCreditBalance .
  FILTER(?newCreditBalance = 0).
}
"""^^xsd:string .
```

Table 4 represents an example of a non-instantaneous event type, i.e. the *Subscription renewal event type*. An event of this type starts when an *Order credit request event* occurs. An *Order credit request event* is a derived event of the *Out of credit event*, which happens if the *Out of credit event* occurs when the user has automatic subscription renewal turned on (Table 5). The Subscription renewal event type completes when a *Positive credit change event* occurs after the start event and the event completion condition returns a result. The completion condition expression also relates the start and the completion events.

Table 4. Subscription renewal event type

```
eventIns:SubscriptonRenewalEventType rdf:type event:NonInstantaneousEventType;
                         rdfs:label "Subscription renewal event type" ;
                         event:hasCompletionEventType eventIns:PositiveCreditChangeEventType ;
                         event:hasStartEventType eventIns:OrderCreditRequestEventType ;
                         event:hasCompletionConditionExpression
"""
SELECT distinct ?nonInstEvent ?startEvent ?completionEvent WHERE {
  ?startEvent a event:Event .
  ?startEvent event:hasEventType eventIns:OrderCreditRequestEventType .
  ?startEvent event:hasParameter ?userParam1 .
  ?userParam1 event:hasParameterDefinition eventIns:userParameterDefinition.
  ?userParam1 event:hasValue ?user.
  ?startEvent event:hasParameter ?lastChange1 .
  ?lastChange1 event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
  ?lastChange1 event:hasValue ?time1.

  ?nonInstEvent event:hasStartEvent ?startEvent .
  ?nonInstEvent event:hasEventType eventIns:SubscriptonRenewalEventType .

  ?completionEvent a event:Event .
  ?completionEvent event:hasEventType eventIns:PositiveCreditChangeEventType .
  ?completionEvent event:hasParameter ?userParam2 .
  ?userParam2 event:hasParameterDefinition eventIns:userParameterDefinition.
  ?userParam2 event:hasValue ?user.
  ?completionEvent event:hasParameter ?lastChange2 .
  ?lastChange2 event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
  ?lastChange2 event:hasValue ?time2.

  FILTER(?time2 > ?time1) .

  FILTER NOT EXISTS {
            ?startEvent2 a event:Event .
            ?startEvent2 event:hasEventType eventIns:OrderCreditRequestEventType .
   ?startEvent2 event:hasParameter ?userParamS2 .
   ?userParamS2 event:hasParameterDefinition eventIns:userParameterDefinition.
   ?userParamS2 event:hasValue ?user.
   ?startEvent2 event:hasParameter ?lastChangeS2 .
   ?lastChangeS2 event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
   ?lastChangeS2 event:hasValue ?timeS2.
            FILTER(?timeS2 < ?time2 && ?timeS2 > ?time1) .
  }

  FILTER NOT EXISTS {
            ?event3 a event:Event .
            ?event3 event:hasEventType eventIns:PositiveCreditChangeEventType.
            ?event3 event:hasParameter ?userParam3 .
            ?userParam3 event:hasParameterDefinition event:userParameterDefinition.
            ?userParam3 event:hasValue ?user.
            ?event3 event:hasParameter ?lastChange3 .
            ?lastChange3 event:hasParameterDefinition event:timeOfLastChangeParameterDefinition.
            ?lastChange3 event:hasValue ?time3.
            FILTER(?time3 < ?time2 && ?time3 > ?time1) .
            }

   FILTER NOT EXISTS {
                         ?nonInstEvent          event:hasCompletionEvent ?completionEventX .
   }
}
"""^^xsd:string .
```

Table 5. Order credit request event type

```
eventIns:OrderCreditRequestEventType rdf:type event:InstantaneousEventType, event:EventType;
                          rdfs:label "Order credit request event type" ;
                          event:hasEventConditionExpression
"""
CONSTRUCT  {
?event event:hasEventType eventIns:OrderCreditRequestEventType .
} WHERE {
?event a event:Event .
?event event:hasEventType eventIns:ZeroCreditEventType .
?event event:hasParameter ?param .
?param event:hasParameterDefinition eventIns:userParameterDefinition.
?param event:hasValue ?user.
?user credit:hasAutomaticSubscriptionRenewal ?renewal.
FILTER(?renewal = xsd:boolean('true')).
}
"""^^xsd:string .
```

Using the proposed approach, a number of different event types of various complexities can be defined, while basing their event expressions either on the domain information, either on the event information, or both. A complex event from the example domain is an event that is published if the user runs out of credit three times within the last five days. If that happens, we may want to offer the user a subscription or an option of automatic subscription renewal. The event expression for such an event is given in Table 6.

Table 6. Repeated Zero Credit Event Type

```
eventIns:RepeatedZeroCreditEventType rdf:type event:EventType, event:InstantaneousEventType ;

                          rdfs:label "Repeated zero credit event type" ;

                          event:hasEventConditionExpression """
CONSTRUCT  {
            ?e3 event:hasEventType eventIns:RepeatedZeroCreditEventType .
} WHERE {
            ?user a <http://www.fluidops.com/User>.
            ?e1 a event:Event.
            ?e1 event:hasEventType eventIns:ZeroCreditEventType.
            ?e2 a event:Event.
            ?e2 event:hasEventType eventIns:ZeroCreditEventType.
            ?e3 a event:Event.
            ?e3 event:hasEventType eventIns:ZeroCreditEventType.
            FILTER(?e1 != ?e2 && ?e2 != ?e3).
            ?e1 event:hasParameter ?userParam1 .
            ?userParam1 event:hasParameterDefinition eventIns:userParameterDefinition.
            ?userParam1 event:hasValue ?user.

            ?e2 event:hasParameter ?userParam2 .
            ?userParam2 event:hasParameterDefinition eventIns:userParameterDefinition.
            ?userParam2 event:hasValue ?user.
            ?e3 event:hasParameter ?userParam3 .
            ?userParam3 event:hasParameterDefinition eventIns:userParameterDefinition.
            ?userParam3 event:hasValue ?user.
            ?e1 event:hasParameter ?timeParam1 .
```

```
        ?timeParam1 event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
        ?timeParam1 event:hasValue ?time1 .
        ?e2 event:hasParameter ?timeParam2 .
        ?timeParam2 event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
        ?timeParam2 event:hasValue ?time2 .
        ?e3 event:hasParameter ?timeParam3 .
        ?timeParam3 event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
        ?timeParam3 event:hasValue ?time3 .
        BIND (<http://www.fluidops.com/service/timestamp>() AS ?now) .
        BIND (<http://www.fluidops.com/service/timestamp>(?time1) AS ?time1TS) .
        FILTER(?now - ?time1TS < 432000000).
        BIND (<http://www.fluidops.com/service/timestamp>(?time2) AS ?time2TS) .
        FILTER(?now - ?time2TS < 432000000).
        BIND (<http://www.fluidops.com/service/timestamp>(?time3) AS ?time3TS) .
        FILTER(?now - ?time3TS < 432000000).
        FILTER(?time3 > ?time2 && ?time2 > ?time1) .
        FILTER NOT EXISTS {
        ?event a event:Event .
        ?event event:hasEventType eventIns:RepeatedZeroCreditEventType.
        ?event event:hasParameter ?param .
        ?param event:hasParameterDefinition eventIns:timeOfLastChangeParameterDefinition.
        ?param event:hasValue ?time3.
        ?event event:hasParameter ?userParam .
        ?userParam event:hasParameterDefinition eventIns:userParameterDefinition.
        ?userParam event:hasValue ?user.
        }
}
"""^^xsd:string .
```

6. Proof of concept

As a proof-of-concept, we provide an implementation of our SPARQL-based framework for event processing using the Information Workbench semantic integration platform.

The Information Workbench (Fig. 3) is a Web-based open platform for Linked Data and Big Data solutions in the enterprise developed by fluid Operations [22]. It exists as a community edition that is freely available, as well as a commercial edition. Our approach is based on the community edition. Information Workbench uses data providers that collect data from internal and external sources, convert it into the semantic Resource Description Framework (RDF) format, and interlink it. The integrated data set can be stored in a central repository or managed virtual layer over federated data sources. The provider architecture is extensible and supports the on-demand integration of additional data. Information Workbench is based on semantic technology standards, such as RDF, OWL and SPARQL. It offers a SDK for building apps to support your individual scenarios and requirements. Adapting and extending the functionality can be achieved using the open API interface, semantic data integration through data providers, rules, workflows and an extensible pool of predefined widgets for creating dynamic visualizations and building reports.

Information Workbench integrates the OpenRDF Sesame triple store and, among other things, it provides a SPARQL 1.1 endpoint. For these reasons, our proof-of-concept implementation was developed using the Information Workbench and is available as an app that runs on top of the Information Workbench platform. It can be found and downloaded in the fluidOps AppCenter [23].

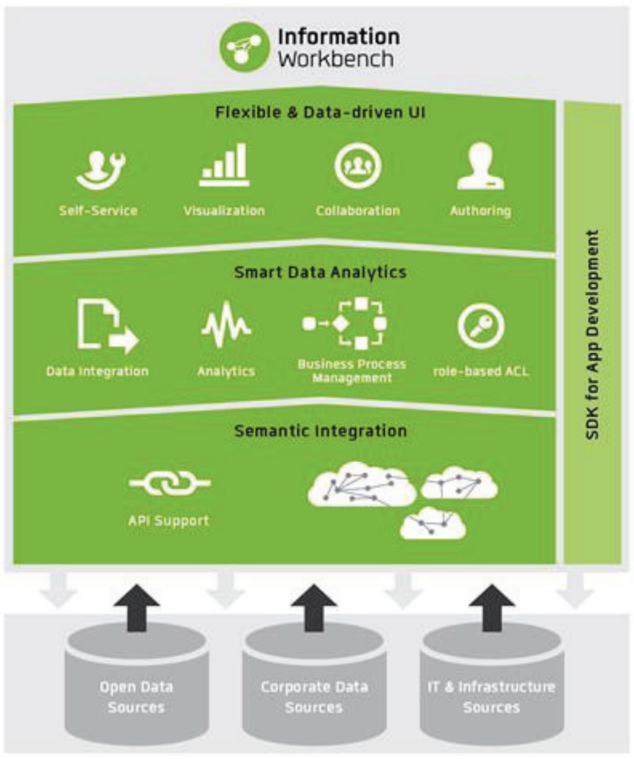

Figure 3. Information Workbench

In our app, the base event ontology, event type definitions and events reside in the OpenRDF Sesame triple store. Event publisher is internal and is implemented as a data provider which reacts on event type expressions in order to publish new events which it integrates with the event data into the triple store. In order to facilitate creation of internal event subscribers, a generic event subscriber component was developed. Specific event subscribers can extend this component and define the event type(s) they want to subscribe to. These can be chosen from the library of existing event types in the triple store. External event subscribers can use the SPARQL 1.1 endpoint service to retrieve information about the events. In order to define new event types, a special user interface is provided which allows the user to define different aspects of an event type, including the name, description, event expression and parameter definitions.

7. Discussion and conclusion

In this paper, we have presented a novel approach to defining and detecting complex events, which provides highly expressive complex event definitions using RDF and OWL ontologies and the SPARQL query language. It is a generic framework that can be applied to different application domains. A proof-of-concept implementation was developed and is available online.

Compared to other, non-semantic based event processing approaches, our approach offers several advantages. Firstly, it is based on W3C standards, especially RDF, OWL and SPARQL, which means that not special event-processing language or formalism, needs to be introduced. Consequently, no special tools that would understand and process specific formalisms need to be developed, as existing available open-source and commercial systems and tools that support these standards can be used. For example, in our proof-of-concept implementation the community edition of the fluidOps Information Workbench together with the Sesame open source Java framework for processing RDF data were used.

Secondly, it supports definition of complex event types with semantic expressivity requirements of various complexities. Besides, using the OWL- and SPARQL- based semantic approach to event processing has several intrinsic advantages. Our framework integrates event and domain information, thus enabling to perform more complete and accurate inferences about complex event occurrences. As the event expressions can use the domain ontology as well as the event ontology, an OWL-DL reasoner can be used in order to detect events based on the information inferred from the integrated asserted models.

Furthermore, with our mechanism of defining the derived events, one can implicitly define events with a taxonomical structure. An event type can have several specializations, for example each with specific additional conditions. This allows for greater flexibility and allows for working with different levels of detail. Based on the information that is available, more or less specific event can be published. In this way, different subscribers can subscribe to different levels of event abstractions.

References

[1] D. Bugaite and O. Vasilecas, “Events Propagation from the Business System Level,” in *Information Systems Development*, C. Barry, M. Lang, W. Wojtkowski, K. Conboy, and G. Wojtkowski, Eds. Springer US, 2009, pp. 1105–1116.

[2] Julian Krumeich, Benjamin Weis, Dirk Werth, and Peter Loos, “Event-Driven Business Process Management: where are we now?,” *Bus. Process Manag. J.*, vol. 20, no. 4, pp. 615–633, Jul. 2014.

[3] A. Sasa Bastinos and O. Vasilecas, “Ontology-based support for complex events,” *Elektron. Ir Elektrotechnika*, vol. 7, pp. 83–88, 2011.

[4] A. Sasa Bastinos and M. Krisper, “Context ontology for Event-Driven Information Systems,” in *The Seventh International Conference on Systems*, Saint Gilles, Reunion Island, 2012.

[5] D. Luckham, W. R. Schulte, J. Adkins, P. Bizzaro, H.-A. Jacobsen, A. Mavashev, B. M. Michelson, P. Niblett, and D. Tucker, “Event Processing Glossary – Version 2.0.” Event Processing Technical Society, Jul-2011.

[6] H. Taylor, A. Yochem, L. Phillips, and F. Martinez, *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2009.

[7] A. Sasa, M. B. Juric, and M. Krisper, “Service-Oriented Framework for Human Task Support and Automation,” *IEEE Trans. Ind. Inform.*, vol. 4, no. 4, pp. 292–302, Nov. 2008.

[8] K. Chandy and W. R. Schulte, *Event Processing: Designing IT Systems for Agile Companies*, 1 edition. New York: McGraw-Hill Osborne Media, 2009.

[9] T. R. Gruber, “Toward Principles for the Design of Ontologies Used for Knowledge Sharing,” *Int. J. Hum.-Comput. Stud.*, vol. 43, no. 5–6, pp. 907–928, Nov. 1995.

[10] “OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation.” W3C, 11-Dec-2012.

[11] “SPARQL 1.1 Overview, W3C Recommendation.” W3C, 21-Mar-2013.

[12] S. -y. Cheng, W. -r. Jih, and J. Y. -j. Hsu, “Context-aware Policy Matching in Event-Driven Architecture,” in *Proceeding of the AAAI 2005 Workshop: Contexts and Ontologies: Theory, Practice and Applications*, Pittsburgh, Pennsylvania, USA, 2005, pp. 140–141.

[13] S. Sen and J. Ma, “Contextualised Event-driven Prediction with Ontology-based Similarity,” in *Proceedings of the 2009 AAAI Spring Symposium, Intelligent Event Processing*, 2009, pp. 73–79.

[14] T. Moser, H. Roth, S. Rozsnyai, R. Mordinyi, and S. Biffl, “Semantic Event Correlation Using Ontologies,” in *On the Move to Meaningful Internet Systems: OTM 2009*, R. Meersman, T. Dillon, and P. Herrero, Eds. Springer Berlin Heidelberg, 2009, pp. 1087–1094.

[15] A. Paschke, “Reaction RuleML 1.0 for Rules, Events and Actions in Semantic Complex Event Processing,” in *Rules on the Web. From Theory to Applications*, A. Bikakis, P. Fodor, and D. Roman, Eds. Springer International Publishing, 2014, pp. 1–21.

[16] A. Paschke and H. Boley, “Rule responder: rule-based agents for the semantic-pragmatic web,” *Int. J. Artif. Intell. Tools*, vol. 20, no. 06, pp. 1043–1081, Dec. 2011.

[17] R. Adaikkalavan and S. Chakravarthy, “Event Specification and Processing for Advanced Applications: Generalization and Formalization,” in *Database and Expert Systems Applications*, R. Wagner, N. Revell, and G. Pernul, Eds. Springer Berlin Heidelberg, 2007, pp. 369–379.

[18] A. Briassouli, S. Dasiopoulou, and I. Kompatsiaris, “Ontology-Based Trajectory Analysis for Semantic Event Detection,” in *International Conference on Semantic Computing, 2007. ICSC 2007*, 2007, pp. 735–742.

[19] A. R. J. François, R. Nevatia, J. Hobbs, R. C. Bolles, and J. R. Smith, “VERL: an ontology framework for representing and annotating video events,” *IEEE Multimed.*, vol. 12, no. 4, pp. 76–86, Oct. 2005.

[20] T. Metzke, A. Rogge-Solti, A. Baumgrass, J. Mendling, and M. Weske, “Enabling Semantic Complex Event Processing in the Domain of Logistics,” in *Service-Oriented Computing – ICSOC 2013 Workshops*, A. R. Lomuscio, S. Nepal, F. Patrizi, B. Benatallah, and I. Brandić, Eds. Springer International Publishing, 2013, pp. 419–431.

[21] H. Paulheim, “Efficient Semantic Event Processing: Lessons Learned in User Interface Integration,” in *The Semantic Web: Research and Applications*, L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds. Springer Berlin Heidelberg, 2010, pp. 60–74.

[22] fluid Operations, “Information Workbench,” *Information Workbench*. [Online]. Available: http://www.fluidops.com/en/portfolio/information_workbench/.

[23] fluid Operations, “fluidOps AppCenter.” [Online]. Available: http://appcenter.fluidops.net/.