Stream Reasoning and Complex Event Processing in ETALIS

Darko Anicic a, Paul Fodor b and Sebastian Rudolph c and Nenad Stojanovic a

a FZI Forschungszentrum Informatik, Karlsruhe, Germany
E-mail: darko.anicic@fzi.de, nenad.stojanovic@fzi.de
b State University of New York at Stony Brook, USA
E-mail: pfodor@cs.sunysb.edu
c Karlsruhe Institute of Technology, Germany
E-mail: rudolph@kit.edu

Abstract. Addressing the dynamics and notification in the Semantic Web realm has recently become an important area of research. Run time data is generated by multiple social networks, sensor networks, various on-line services, and so on. The challenge is how to get advantage of a huge amount of real time data, i.e., how to integrate heterogeneous data streams, combine data streams with the background knowledge (ontologies), and perform stream reasoning. In this paper we describe ETALIS system which enables specification and monitoring of changes in real time. Changes can be specified as complex event patterns, and ETALIS system can detect complex events in real time. Moreover the system can perform the task of reasoning over streaming events (e.g., RDF triples) with respect to the background knowledge (e.g., ontologies). ETALIS system implements two languages for event patterns: ETALIS Language for Events, and Event Processing SPARQL. The described system has various applicabilities in capturing changes in semantic networks, broadcasting the notifications to interested parties, and creating further changes (based on processing of the temporal, static, or slowly evolving knowledge).

Keywords: Complex Event Processing, Stream Reasoning, EP-SPARQL

1. Introduction

The amount of semantically annotated data and related ontologies in the Web is rapidly growing. More and more of this data is real time information, where fast delivery is important with respect to an underlying Web service. For instance, consider a social network where capturing changes (updates) and broadcasting them in real time is of significant importance to users of the network, as well as, to an on-line advertisement deployed in the network.

Semantic Web tools utilize various techniques to understand the meaning of information, and further to reason about that information. With rapidly changing information on the Web this capability is not sufficient. A conclusion that is derived a minute ago, may not be valid right now. Hence, instead of a reasoning service, there is a requirement to enable reasoning on rapidly changing (or streaming) data.

Stream Reasoning. While reasoning systems year-over-year improve in reasoning over a static (or slowly evolving) knowledge, reasoning over streaming knowledge remains a challenge. For instance, ontological knowledge can efficiently represent the domain of interest (of an application) and help in harnessing the semantics of information. However, the task of reasoning over streaming data (e.g., RDF triples) with respect to the background ontology constitutes a new challenge known as Stream Reasoning [15].

On the other hand, to process data in real time Complex Event Processing (CEP) has arisen significant attention in the research community. CEP is concerned with detection of near real time situations (complex events) that are of a particular business interest. A complex event can be perceived as a composition of
various more simple events (e.g., elementary changes, updates etc.) that happened satisfying different temporal relations between themselves.

However CEP also suffers two limitations, it can provide on the fly analysis of data streams, but cannot combine data streams with background knowledge (ontologies), and they cannot perform reasoning tasks. Therefore, CEP could be extended toward Semantic Complex Event Processing (SCEP).

**Semantic Complex Event Processing.** In state-of-the-art event processing systems [2,10,7,14,13] complex patterns are detected only by examining temporal relations between events. Yet the question is whether examining temporal relationships is (necessarily) sufficient to detect real time situations of interest. For instance, complex events can be used to trigger time critical actions or decisions and hence ensure appropriate responses to certain situations. The question is whether event patterns from current event processing systems are expressive enough to capture complex events in all their aspects. For example, how likely is that business critical decisions are taken merely on event patterns of the type: event $a$ is followed by event $b$ in last 10 seconds? For some applications, such patterns are expressive enough; however for semantic-rich applications, they are certainly not. In such applications time critical actions and decisions are taken not only based upon events, but also upon additional knowledge (e.g., ontologies). This knowledge captures the domain of interest, or context related to business critical actions and decisions.

The state-of-the-art event processing systems provide integration of event streams with databases. However explicitly represented data from databases are not always sufficient. Very often to match two events in a pattern we need to prove semantic relations between them; and to do that, we need to process the background knowledge describing the relations between events. From this knowledge we can discover, not only explicit semantics, but also to derive implicit relationships. Being able to evaluate the knowledge on-the-fly, we can for example enrich recorded events with the background information; detect more complex situations; propose certain intelligent recommendations in real time; or to accomplish complex event classification, clustering, filtering and so forth. We refer to these advanced features as SCEP.

This paper presents an open source tool called ETALIS system [6]. The tool is capable, not only to detect complex events over streaming data, but also to evaluate domain knowledge on-the-fly (e.g., in order to prove semantic relations among them). In nutshell, ETALIS system combines complex event processing functionality with stream reasoning.

**2. ETALIS System**

**2.1. Conceptual Architecture**

An event represents something that occurs, happens or changes the current state of affairs. For example, an event may signify a problem or an impending problem, a threshold, an opportunity, an information becoming available, a deviation and so forth. Simple events are combined into complex events depending on their temporal, causal and semantic relations.

The task of complex event processing and stream reasoning in ETALIS is depicted in Figure 1, and it can be described as follows. Within some dynamic setting, events from multiple event sources take place (see “Events” in Figure 1). Those atomic events are instantaneous, i.e., they happen at one specific point in time. Notifications about these occurred events together with their timestamps and possibly further associated data (such as involved entities, numerical parameters of the event, or provenance data) enter ETALIS system in the order of their occurrence.

![Fig. 1. Conceptual Architecture of ETALIS System](image-url)

ETALIS system further features a set of complex event descriptions (denoted as “Event Patterns”) by means of which “Complex Events” can be specified as temporal constellations of atomic events (see Figure...
The complex events thus defined can in turn be used to compose even more complex events and so forth. As opposed to atomic events, those complex events are not considered instantaneous but are endowed with a time interval denoting when the event started and when it ended.

To detect complex events, ETALIS system may consult the background (contextual) knowledge too. For instance, two events match an event pattern only if they (apart from temporal relationships) also satisfy certain semantic relationships, and the semantic relations are specified in the background knowledge or they are inferred from that knowledge. ETALIS system needs to evaluate the knowledge and reason about it (illustrated with the upper part of Figure 1). The reasoning is performed on-the-fly as streaming events occur.

### 2.2. ETALIS Language for Events

ETALIS system implements a rule-based language defined in [5]. It is called ETALIS Language for Events (ELE). In this subsection we briefly review its capabilities, at the same time reflecting the capabilities of ETALIS system.

Figure 2 demonstrates the various ways of constructing complex event patterns (descriptions) in ETALIS Language for Events. Moreover, the figure informally presents the semantics of the language (a formal semantics can be found in [5]).

![Fig. 2. Language for Event Processing - Composition Operators](image)

Let us assume that instances of three complex events, $P_1$, $P_2$, $P_3$, are occurring in time intervals as shown in Figure 2. Vertical dashed lines depict different time units, while the horizontal bars represent detected complex events for the given patterns. In the following, we give the intuitive meaning for all patterns from the figure:

- $(P_1)_3$ detects an occurrence of $P_1$ if it happens within an interval of length 3, i.e., 3 represents the (maximum) time window.
- $P_1$ SEQ $P_3$ represents a sequence of two events, i.e., an occurrence of $P_1$ is followed by an occurrence of $P_3$; here $P_1$ must end before $P_3$ starts.
- $P_2$ AND $P_3$ is a pattern that is detected when instances of both $P_2$ and $P_3$ occur no matter in which order.
- $P_1$ PAR $P_2$ occurs when instances of both $P_2$ and $P_3$ happen, provided that their intervals have a non-zero overlap.
- $P_2$ OR $P_3$ is triggered for every instance of $P_2$ or $P_3$.
- $P_1$ DURING $(0$ SEQ $6)$ happens when an instance of $P_1$ occurs during an interval; in this case, the interval is built using a sequence of two atomic time-point events (one with $q = 0$ and another with $q = 6$, see the syntax above).
- $P_3$ STARTS $P_1$ is detected when an instance of $P_3$ starts at the same time as an instance of $P_1$ but ends earlier.
- $P_1$ EQUALS $P_3$ is triggered when the two events occur exactly at the same time interval.
- NOT$(P_3).[P_1, P_1]$ represents a negated pattern. It is defined by a sequence of events (delimiting events) in the square brackets where there is no occurrence of $P_3$ in the interval. In order to invalidate an occurrence of the pattern, an instance of $P_3$ must happen in the interval formed by the end time of the first delimiting event and the start time of the second delimiting event. In this example delimiting events are just two instances of the same event, i.e., $P_1$. Different treatments of negation are also possible, however we adopt one from [1].
- $P_3$ FINISHES $P_2$ is detected when an instance of $P_3$ ends at the same time as an instance of $P_1$ but starts later.
- $P_2$ MEETS $P_3$ happens when the interval of an occurrence of $P_2$ ends exactly when the interval of an occurrence of $P_3$ starts.

It is worth noting that the defined pattern language captures the set of all possible 13 relations on two temporal intervals as defined in [3]. The set can also be used for rich temporal reasoning.
2.3. An Example of ETALIS Application

It is worthwhile to demonstrate how ETALIS system can be used in practice. To do this, let us consider a monitoring service based on processing stock quotas (events) from Google Finance web service. The service combines CEP capabilities with background knowledge in order to provide the real time intelligence.

Let us assume we want to detect the stock price increase in a supply chain system of certain companies. The following pattern monitors two stock price increases in two companies (occurred within certain time window), and checks whether the companies are parts of the supply chain system.

\[
trendIncrease() \leftarrow \\
(stockIcr(\text{CompanyA}) \text{ seq } stockIcr(\text{CompanyB})).\text{10} \\
\text{AND inSupChain(CompanyA, CompanyB)}.
\]

The supply chain system is represented as a set of explicit links between companies, e.g., with \( \text{linked(CompanyA, CompanyB)} \) we represent two interconnected businesses involved in the ultimate provision of a product. We assume that such explicit relationships are continuously being updated via according information events as, for instance, our data mining tool processes different information sources, delivering events of the form

\[
\text{linked(CompanyA, CompanyB)} \\
\text{linked(CompanyY, CompanyZ)}
\]

The following transitive closure pattern can then be used to span over semantic relationships between companies scenario where direct supply relationships are represented explicitly, and hence discover implicit relationships, i.e., whether two stock price increases also covered the whole supply chain system.

\[
in\text{SupChain}(X, Y) \leftarrow \text{linked}(X, Y). \\
in\text{SupChain}(X, Z) \leftarrow \text{linked}(X, Y) \text{ AND in\text{SupChain}(Y, Z)}.
\]

Alternatively, we can build more complex knowledge-bases that may serve more complex reasoning tasks such as market predictions. For example, a few semantically related events may trigger predictions about the related stock markets.

To generalize, for a given set of events that satisfy certain temporal relationships, our approach may be used to additionally check whether these events satisfy certain semantic relations with respect to domain knowledge that itself may be dynamically collected. Semantic relationships between occurring events is an important dimension, neglected in today’s CEP systems. It helps discovering the context in which events occurred by combining deductive reasoning with complex event processing.

2.4. Internals of ETALIS System

In this subsection we give more details about internal processing in ETALIS system, i.e., how events specified in ETALIS Language for Events can be detected at run time. Our approach for SCEP and stream reasoning is based on deductive (logic) rules. Such an approach enable us, not only to do event processing but to process contextual knowledge too. One difficulty is however to realize data or event-driven computation that is required in CEP. Deductive systems are rather suited for a request-response computation. That is, for given a request, an inference engine will evaluate available knowledge and respond with an answer. This means that the event inference engine needs to check if this pattern can be deduced or not. The check is performed at the time when such a request is posed. If satisfied by the time when the request is processed, a complex event will be reported. If not, the pattern is not detected until the next time the same request is processed (though it can become satisfied in-between the two checks).

ETALIS system is a rule-based reasoning system that acts as an event-driven engine. Figure 3 shows basic operational steps that are undertaken in ETALIS system. Rectangles in the diagram are used to depict certain processes in ETALIS system while ovals represent either (external/internal) inputs to these processes, or (external/internal) outputs from them.

The system diagram starts by a user written ETALIS CEP rules as input. These rules specify event patterns according to ETALIS Language for Events (see Section 2.2). ETALIS system validates the rules with respect to the language grammar, and parse them\(^1\). As a result ETALIS system produces rules in an internal format, ready for a preprocessing step called binarization.

The binarization is a process of splitting an event pattern into a set of two-input intermediate event patterns (goals). For instance, an event pattern (1) is split into \( \text{i}_{e_1} \leftarrow a \text{ seq } b \) (where \( \text{i}_{e_1} \) is an intermediate event), and \( e \leftarrow \text{i}_{e_1} \text{ seq } c \).

\(^1\)“parser.P” and “etalis.P” are source files that implement corresponding functionality (see Figure 3) in our open source implementation [6].
Using the binarization, it is more convenient to construct execution rules for three reasons. First, it is easier to implement an event operator when events are considered on “two by two” basis. Second, the binarization increases the possibility for sharing among events and intermediate events, when the granularity of intermediate patterns is reduced. Third, the binarization eases the management of rules. Each new use of an event (in a pattern) amounts to appending one or more rules to the existing rule set. However, what is important for the management of rules, we do not need to modify existing rules when adding new ones\(^2\). \textit{ETALIS Compiler} compiles binary rules into event-driven backward chaining (EDBC) rules, i.e., executable rules.

\(^2\)This holds even if patterns with negated events are added.
EDBC rules are basic mechanism in ETALIS that “converts” request-response computation into event-driven processing. It is a mechanism which enables an inference system to derive a complex event at the moment it really occurs (not at the moment when a request is posed to the inference system). The notable property of these rules is that they are *event-driven*, i.e., a rule will be evaluated when an event that matches the rule’s head occurs. In such a situation a firing rule will insert a goal into memory. The purpose of the goal is to denote that a certain event happened, and that the system “waits” for another appropriate event in order to produce a more complex goal. For example, let us consider again the pattern (1). When event \( a \) occurs, there will be an EDBC rule which will insert a goal stating that the system waits for event \( b \) to happen in order to produce goal \( xe_1 \). Later, when event \( b \) occurs the system will insert a goal stating that (intermediate) event \( xe_1 \) occurred and the system waits for event \( c \) to happen in order to produce event \( c \).

Goals are automatically asserted by EDBC rules as relevant events occur. They can persist over a period of time “waiting” to support detection of a more complex goal. Important characteristics of these goals are that they are asserted only if they are used later on (to support a more complex goal or an event pattern); goals are all unique, and persist as long as they remain relevant (after that they are deleted). Goals are asserted by rules which are executed in the *backward* chaining (Prolog) mode. Although the rules are executed backwards, overall they exhibit a *forward* chaining behavior. Further details about EDBC rules can be found in [5].

ETALIS Compiler produces EDBC rules which are executable rules (written in Prolog). These rules may be accompanied with the background knowledge to represent domain of interest (as discussed in Section 2.1). Domain knowledge is also expected to be in Prolog (but in Section 3 ETALIS system is extended to accept RDFS ontologies as background knowledge). Compiled rules, together with the domain knowledge, are then executed by a standard Prolog system (e.g., SWI, YAP, XSB etc.). The rules are triggered by events from Event streams (see Figure 3). As a result EDBC rules continuously derive complex events as soon as they happen.

Detection of complex events also depends on other important issues such as *consumption policies*, *garbage collection*, additional algebra for reasoning about time intervals, and so forth (see Figure 3). In the following, we review them briefly.

In event processing, consumption policies (or event contexts, see [12]) deal with an issue of *selecting* particular events occurrences when there are more than one event instance applicable and *consuming* events after they have been used in patterns. We have implemented three widely used consumption policies: recent, chronological, and unrestricted policy.

ETALIS system features two memory management techniques to *prune* outdated events, and hence free up its working memory. The first technique modifies the binarization step by pushing the time constraints (set by pattern’s time window information; users are always encouraged to write patterns with certain time constraints). The technique ensures that time window constraints are checked during the incremental event detection. Therefore unnecessary intermediary subcomplex events will not be generated if the time constraints are violated (i.e., time expired). Our second solution for garbage collection is to prune expired events (goals) by using periodic events, generated by the system. This technique does not check the constraints at each step during the incremental event detection. Instead, events (goals) are pruned periodically as system events are triggered.

As an algebra for reasoning about time intervals we have implemented Allen’s temporal relationships [3]. Using this algebra, the system can also reason about intervals of detected complex events (e.g., to discover whether one complex event occurred during another complex event, whether one complex event starts/finishes another one, and so forth).

Finally, it is worth noting that detected complex events are fed back into the system (to possibly produce more complex events), or they can be used externally in an application or a system. Typical situation when events are fed back into the system happens with recursive event patterns. This is denoted by the backward (dashed) edge in Figure 3.


To enable ETALIS system to handle real time Semantic Web applications we have developed Event Processing SPARQL (EP-SPARQL) language [4]. This extension enables a user to specify event patterns in a SPARQL-like language. Event streams are expected to be in an RDF format (i.e., RDF streaming triples additionally accompanied with timestamps). The background (contextual) knowledge can be specified as an RDFS ontology.
Syntactically, we defined EP-SPARQL to be SPARQL extended by the binary operators SEQ, EQUALS, OPTIONALSEQ, and EQUALSOPTIONAL used to combine graph patterns in the same way as UNION and OPTIONAL in pure SPARQL. Intuitively, all those operators act like a (left, right or full) join, but they do so in a selective way depending on how the constituents are temporally interrelated, as indicated by their naming: $P_1$ SEQ $P_2$ joins $P_1$ and $P_2$ only if $P_2$ occurs strictly after $P_1$, whereas $P_1$ EQUALS $P_2$ performs the join if $P_1$ and $P_2$ are exactly simultaneous. OPTIONALSEQ and EQUALSOPTIONAL are temporal-sensitive variants of OPTIONAL.

Moreover, we added the function getDURATION() to be used inside filter expressions. This function yields a literal of type xsd:duration to be used inside filter expressions. This function is always available in EP-SPARQL queries, conceiving them as a kind of production rule. Thereby, the result graph of such a query is assumed to be added to the RDF stream. For instance, the following statement gathers “temporally distributed” rating information to create a triple indicating an event of being downrated, which in turn can be used in other CONSTRUCT or SELECT queries.

Moreover, we allow for recursion by employing CONSTRUCT queries, conceiving them as a kind of production rule. Thereby, the result graph of such a query is assumed to be added to the RDF stream. For instance, the following statement gathers “temporally distributed” rating information to create a triple indicating an event of being downrated, which in turn can be used in other CONSTRUCT or SELECT queries.

3.1. Examples with EP-SPARQL

We provide a few examples to give some intuition on EP-SPARQL operators supported by ETALIS system. The following EP-SPARQL query is supposed to search for companies whose stock price has decreased by over 30% and subsequently risen by more than 5% within a time frame of 30 days.

$$\text{SELECT } ?\text{company} \text{ WHERE}$$

$$\{ ?\text{company} \text{ hasStockPrice } ?\text{price1} \}$$

$$\text{SEQ} \{ ?\text{company} \text{ hasStockPrice } ?\text{price2} \}$$

$$\text{SEQ} \{ ?\text{company} \text{ hasStockPrice } ?\text{price3} \}$$

$$\text{FILTER} \{ ?\text{price2} < ?\text{price1} \times 0.7 \land (1) \text{getDURATION()} < \"P10D\"^^xsd:duration \}$$

The next EP-SPARQL query will identify companies with a more than 50% stock price drop and – in case some rating agency previously downrated this company, this rating agency will be indicated as well.

$$\text{SELECT } ?\text{company} \text{ ?ratingagency} \text{ WHERE}$$

$$\{ ?\text{company downratedby ?ratingagency} \}$$

$$\text{OPTIONALSEQ} \{ ?\text{company hasStockPrice } ?\text{price1} \}$$

$$\text{SEQ} \{ ?\text{company hasStockPrice } ?\text{price2} \}$$

$$\text{FILTER} \{ ?\text{price2} < ?\text{price1} \times 0.5 \}$$

It is worth mentioning that – just like for pure SPARQL – negation (i.e., requiring the absence of some triple pattern instead of its presence) is not an explicit part of the formalism, but can be expressed via OPTIONAL and FILTER. For instance, the following query asks for companies having a larger than 50% stock price increase in less than 15 days without having acquired another company during that period.

$$\text{SELECT } ?\text{company} \text{ WHERE}$$

$$\{ ?\text{company hasStockPrice } ?\text{price1} \}$$

$$\text{SEQ} \{ \{ ?\text{company hasAcquired } ?\text{othercompany} \} \}$$

$$\text{OPTIONALSEQ} \{ ?\text{company hasStockPrice } ?\text{price2} \}$$

$$\text{FILTER} \{ ?\text{price2} < ?\text{price1} \times 1.5 \land \text{getDURATION()} < \"P15D\"^^xsd:duration \}$$

Finally, the forthcoming extended SPARQL standard\(^3\) featuring subqueries and expressions allows for as complex mechanisms as aggregation over sliding windows. As an example we present a query monitoring the average stock price of a company ACME Inc. over the last 10 days. First, we use a construct rule that aggregates counts and sums of stock prices within the given time frame and feeds this information back into the stream. Thereby, the EQUALSOPTIONAL and filter part make sure that no price signal is left out.

$$\text{CONSTRUCT } ?\text{company} \text{ downratedby ?ratingagency} \text{ WHERE}$$

$$\{ ?\text{rating1 rater ?ratingagency ; rated ?company ; score ?score1 } \}$$

$$\text{SEQ} \{ ?\text{rating2 rater ?ratingagency ; rated ?company ; score ?score2 } \}$$

$$\text{FILTER} \{ ?\text{score2} < ?\text{score1} \}$$

$$\text{CONSTRUCT } _\text{aaa} \text{ :hasCount } ?\text{count} .$$

$$\text{CONSTRUCT } _\text{aaa} \text{ :hasSum } ?\text{sum}.$$  

$$\text{SELECT } ?\text{count AS} ?\text{prevcount} + 1 \text{ WHERE} \{ \{ ?\text{aaa} :hasSum } ?\text{prevsum} + ?\text{price} \text{ AS} ?\text{prevsum} + ?\text{price} \}$$

$$\text{OPTIONALSEQ} \{ ?\text{point :hasCount } ?\text{prevcount} . ?\text{point :hasSum } ?\text{prevsum} . \}$$

$$\text{SEQ} \{ ?\text{aaa} \text{ :hasCount } ?\text{prevcount} . ?\text{point :hasSum } ?\text{prevsum} . \}$$

$$\text{EQUALSOPTIONAL} \text{ WHERE} \{ \{ ?\text{point :hasCount } ?\text{prevcount} . ?\text{point :hasSum } ?\text{prevsum} . \}$$

$$\text{FILTER} \{ \text{getDURATION()} < \"P10D\"^^xsd:duration \}$$

---

\(^3\)http://www.w3.org/TR/2009/WD-sparql-features-20090702/
3.2. Internals of EP-SPARQL Implementation

EP-SPARQL is implemented as an extension to ETALIS system, described in Section 2.4. A system diagram of our EP-SPARQL engine is shown in Figure 4.

A user is expected to write EP-SPARQL queries and to deploy them in the engine. These queries act similarly as continuous queries in Database Stream Management Systems (DSMS), i.e., once registered the queries are continuously evaluated with respect to streaming data. In our implementation, the engine incrementally matches incoming data (events) and produce complex events that satisfy queries as soon as they occur (see Section 2.4).

Since event streams and the background knowledge are represented in the RDF format, we use an RDF/XML parser to convert inputs into internal ETALIS format (see Figure 4). For event streams the conversion is applied on-the-fly. It is a straight forward mapping, and typically does not case a significant overhead at run time. The background knowledge (RDFS ontologies) can be converted in Prolog representation at design time. Similarly, we have also implemented a parser for EP-SPARQL syntax and a compiler which produces EDBC rules out of EP-SPARQL expressions. All three inputs (EP-SPARQL queries, event streams and the domain ontology) are then fed into ETALIS system where the processing, as described in Section 2.4, takes place.

4. Interacting with ETALIS System

ETALIS system can accessed in the following ways:

- interaction through the command line interface;
- access through a foreign language interface.

Command line interface is suitable for development, testing, and deployment of event-driven applications in ETALIS system. On the other hand, ETALIS system can interfaced from other applications or systems.
Typically, CEP systems belong to middleware where they serve as a part of some more complex service. ETALIS system can be interfaced from other widely used languages such as Java, C and C#. This also enables ETALIS system to be combined with existing programs and libraries. For more details on this topic, an interested reader is referred to [6].

As a future work we plan to build a graphic user interface to provide another convenient way to interact with ETALIS system.

5. Experimental Results

ETALIS system is implemented in Prolog and freely available from [6]. As an underlying Prolog engine (see Section 2.4) YAP version 5.1.3 is used.4 Presented tests were carried out on a workstation with Intel Core Quad CPU Q9400 2.66GHz, 8GB of RAM, running Windows Vista x64.

Test 1: SCEP. As a concrete example, we show the evaluation of the trendIncrease complex pattern from Section 2.3. We varied the pool of companies in the transitive closure, ranging from 100 to 100,000 linked companies. The Figure 5 shows the throughput in events/second obtained after detection of stockIcr events for two companies and checking the supply-chain connectivity (semantic relationships) for those two companies. As result we got the throughput of more than 20000 events/second, where for each complex event the system additionally needed to traverse 100000 supply chain links. It can be seen that the computation of the recursive relation inSupChain (as the background knowledge) has a relatively small effect, ∼10%, on the overall complex processing execution time (even when the system needed to traverse 100,000 links in between two stockIcr events).

Test 2: Stream reasoning. To provide a performance evaluation of stream reasoning, we decided to reconstruct an experiment from [16]. The goal of the test is to listen to streaming triples, and to infer whether the subject of a triple is an instance of the class of concern (or any of its subclasses). The class of concern is \( \text{http://spire.umbc.edu/ontologies/EthanPlants.owl#Tracheobionta} \) that has 40,080 subclasses with a maximum depth of eight. As in [16], we measured delay caused by reasoning that proves whether a streaming triple is an instance of the class of concern. The work in [16] provides three implementations: the first based on Jena, the second based on precomputed inference results stored in a hashtable, and the third based on a streaming database engine, TelegraphCQ [13] (none of these 3 implementations is available for download and testing). The fastest implementation is the third one (which also precomputes all inferences and stores them in a PostgreSQL database). Figure 6(a) shows results of the same test with ETALIS. Our system is more than 20 times faster. We did the test on a faster machine, however ETALIS system was doing stream reasoning on-the-fly (with no persisted inferences), and still performed significantly better.

As an additional stream reasoning test we evaluate the following RDFS closure.

\[
(\text{?a rdfs:type ?y}) \leftarrow \\
(\text{?x rdfs:subClassOf ?y}) \text{ SEQ (?a rdfs:type ?x)}
\]

The wine ontology5 is used as a background knowledge base. In this test, we combine event processing with stream reasoning. Two streaming triples need to satisfy the temporal relations (i.e., to appear one after the other one). Additionally, ?x must be a subclass of a class of concern. We choose the Wine class as a class of concern in the background ontology. Figure 6(b) presents results that we obtained for this test.

Note that existing approaches for stream reasoning [9,8,16,11] cannot do the event processing part of this test. Further on, the derived triples can be used in building even more complex events (creating temporal and semantic relations at higher levels) which is also a

Fig. 5. Throughput change as relation linked varies from 100 to 100,000

\[\text{Throughput change by changing recursion depth}\]

\[\text{Throughput Change} \]

\[\text{Recursion depth}\]

\[\text{Throughput (1000 x Events/Sec)}\]

\[\text{Throughput Change} \]

\[\text{Recursion depth}\]

\[\text{Throughput (1000 x Events/Sec)}\]

\[\text{Throughput Change} \]

Fig. 5. Throughput change as relation \text{linked} varies from 100 to 100,000

4YAP Prolog: http://www.dcc.fc.up.pt/~vsc/Yap/

5Wine ontology: www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf
capability beyond the state of the art in stream reasoning.

More extensive experiments on event processing as well as on stream reasoning in ETALIS system can be found in [5,4].

6. Conclusion

Addressing the dynamics in the Semantic Web realm has recently become an important area of research. Real time processing of the dynamic changes has useful applications in many areas including Web applications such as blogs and feeds, financial services, sensor networks, geospatial services, click stream analysis etc. In this paper we have presented ETALIS system which is a tool for complex event processing and stream reasoning. The tool can efficiently detect complex events in (near) real time while evaluating the background knowledge (e.g., an ontology). The knowledge is evaluated on-the-fly either to capture the domain of interest in an application or to prove certain relations between matching events. ETALIS system is an open source tool.

References


