Linköping Studies in Science and Technology
Licentiate Thesis No. 1782

Towards Semantically Enabled Complex Event Processing

Robin Keskisärkkä

Linköping University
Department of Computer and Information Science
Division of Human-Centered Systems
SE-581 83 Linköping, Sweden

Linköping 2017

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.
A doctor's degree comprises 240 ECTS credits (4 years of full-time studies).
A licentiate's degree comprises 120 ECTS credits.

Edition 1:1

ISBN 978-91-7685-479-2
ISSN 0280-7971
Typeset using XeLaTeX

Printed by LiU-Tryck, Linköping 2017

Abstract

The Semantic Web provides a framework for semantically annotating data on the web, and the Resource Description Framework (RDF) supports the integration of structured data represented in heterogeneous formats. Traditionally, the Semantic Web has focused primarily on more or less static data, but information on the web today is becoming increasingly dynamic. RDF Stream Processing (RSP) systems address this issue by adding support for streaming data and continuous query processing. To some extent, RSP systems can be used to perform complex event processing (CEP), where meaningful high-level events are generated based on low-level events from multiple sources; however, there are several challenges with respect to using RSP in this context. Event models designed to represent static event information lack several features required for CEP, and are typically not well suited for stream reasoning. The dynamic nature of streaming data also greatly complicates the development and validation of RSP queries. Therefore, reusing queries that have been prepared ahead of time is important to be able to support real-time decision-making. Additionally, there are limitations in existing RSP implementations in terms of both scalability and expressiveness, where some features required in CEP are not supported by any of the current systems. The goal of this thesis work has been to address some of these challenges and the main contributions of the thesis are: (1) an event model ontology targeted at supporting CEP; (2) a model for representing parameterized RSP queries as reusable templates; and (3) an architecture that allows RSP systems to be integrated for use in CEP. The proposed event model tackles issues specifically related to event modeling in CEP that have not been sufficiently covered by other event models, includes support for event encapsulation and event payloads, and can easily be extended to fit specific use-cases. The model for representing RSP query templates was designed as an extension to SPIN, a vocabulary that supports modeling of SPARQL queries as RDF. The extended model supports the current version of the RSP Query Language (RSP-QL) developed by the RDF Stream Processing Community Group, along with some of the most popular RSP query languages. Finally, the proposed architecture views RSP queries as individual event processing agents in a more general CEP framework. Additional event processing components can be integrated to provide support for operations that are not supported in RSP, or to provide more efficient processing for specific tasks. We demonstrate the architecture in implementations for scenarios related to traffic-incident monitoring, criminal-activity monitoring, and electronic healthcare monitoring.

Acknowledgments

I would like to start by thanking my supervisors Eva Blomqvist and Henrik Eriksson. You have been very supportive throughout this thesis work and I deeply appreciate the time and energy you have invested in helping me reach this far.

The research leading to the results reported in this thesis has been carried out at the Department of Computer and Information Science at Linköping University. The research was funded in part by: (1) VALCRI, financed by the European Union Seventh Framework Programme (FP7/2007–2013) under the EC Grant Agreement No FP7-IP608142; (2) E-care@home, financed by the Swedish Knowledge Foundation; and (3) STeDS, partly financed by the research organization CENIIT (project id 12.10). In these projects, I have had the honor of working with many talented individuals and I hope to have the pleasure of working with some of you again in the future.

I would also like to thank my dear colleagues, with whom I have shared many interesting discussions. A special thanks goes out to Jonas Rybing, who has reviewed parts of this thesis. Our conversations about research and writing in general have been both encouraging and inspiring.

Finally, I want to thank my family. When I have felt overwhelmed by work you have always been there to put things in perspective. The love I feel for you cannot be expressed in words.

Robin Keskisärkkä
Linköping, October 2017

Contents

List of Figures

List of Tables

List of Abbreviations

ASP	Answer Set Programming
CEP	Complex Event Processing
EPA	Event Processing Agent
EPN	Event Processing Network
IoT	Internet of Things
ODP	Ontology Design Pattern
OWL	Web Ontology Language
RDF	Resource Description Framework
RSP	RDF Stream Processing
RSP Group	RDF Stream Processing Community Group
RSP-QL	RDF Stream Processing Query Language
SCEP	Semantic Complex Event Processing
W3C	World Wide Web Consortium

1 Introduction

The Semantic Web was designed as an extension of the web to enable sharing and reuse of information across applications as Linked Data. The basic principles are similar to those of the traditional web, but the Semantic Web promotes standards and formats that allow structured information to be shared in a machine-readable format. One of the cornerstones of this model is the Resource Description Framework (RDF). RDF was originally intended as a model for expressing metadata on the web, but today it is recognized as a more general model for representing data. Statements are expressed in the form of subject-predicate-object, which are referred to as *triples*. Triples can provide paths between resources and sets of triples are used to create directed labeled graphs. Applications can traverse graphs across multiple datasets by following these links.

Ontologies are used to define schemas and vocabularies for data, which provides a common language for communicating data between applications, but they are also used as a way of aligning data represented in different formats. Additionally, ontologies can be used as a way of defining rules and constraints over the data, which can be used to implement automated inferencing and reasoning. The flexibility of Semantic Web technologies, along with the standards promoted by the World Wide Web Consortium (W3C)[1], has made these technologies attractive from the perspective of managing information, in particular when the information is heterogeneous in structure.

[1] `https://www.w3.org/`

1.1 Motivation

Traditional Semantic Web technologies have been designed to efficiently process static (or slowly changing) data, and as a result they tend to scale poorly for dynamic data. In recent years, there have been several attempts to combine Semantic Web technologies with *stream processing* and *complex event processing* with the goal of leveraging Semantic Web standards to facilitate management of heterogeneous data and to support inferencing.

When querying streaming data there is typically no single correct answer, rather there is a continuous flow of responses. These answers may be generated continuously as new data become available. In most scenarios, relevance in a stream is tightly coupled with temporal proximity, since the most up-to-date data is usually a better representation of the current state. For example, if we were monitoring the location of a moving vehicle we could use previous observations to predict where the vehicle was headed; however, our predictions would rapidly lose their predictive power unless we took into account new observations as they became available.

A number of RDF Stream Processing (RSP) systems have been proposed and implemented to process streams of RDF data. These systems differ with respect to expressiveness, scalability, reasoning support, underlying semantics, and the query languages supported. For example, EP-SPARQL [6] supports temporal operators as part of the query language, whereas C-SPARQL [12] and CQELS [43] support time-based and count-based windows over streams. These differences make it very difficult to assess the practical limitations of current state-of-art RSP systems.

While RSP systems that address the issue of streaming RDF data already exist, the goal of this research is to show how these capabilities and other Semantic Web technologies can be leveraged to support semantically enabled complex event processing. The purpose of this work is ultimately to help develop, both theoretically and practically, a framework for highly expressive event processing that can assist domain experts that need to make decisions under time constraints in a streaming context.

There are several challenges associated with using current Semantic Web technologies for event processing. From an event modeling perspective, ontology languages provide powerful constructs for defining event types, and describing the semantic relationships between them. Schemas and vocabularies provide a system independent representation for events, and automated reasoning can be used to leverage information that exists implicitly in the data. For example, rather than asserting all event types explicitly, event classifications can often be inferred from class restrictions and property constraints. However, current event models on the Semantic Web have not been designed with event processing applications in mind.

With respect to querying, it is often assumed that users interact with RSP systems by issuing string-based queries that are registered in some execution

environment. This implicitly assumes that the users have sufficient knowledge about the relevant query language, the data stream representations, the background data, and the underlying ontologies. This type of detailed knowledge cannot typically be expected of all the users of these tools. Additionally, in a streaming context the data is constantly shifting, making it difficult to validate the correctness of a particular query with respect to the intention of the user. In real-time scenarios, users may not be able to spend any significant amount of time on formulating these queries, therefore supporting efficient reuse of queries that can be prepared ahead of time is a necessity.

Using available Semantic Web technologies for event processing also poses a range of technical challenges. While SPARQL is a very expressive query language, certain operations are difficult to accomplish in any scalable way. For example, the arithmetic operations supported in standard SPARQL are very limited, making it difficult to do any advanced statistical data analysis. However, one should recognize that Semantic Web technologies do not exist in isolation, and leveraging external tools and systems is a requirement for some types of semantic complex event processing to be realized using current RSP systems.

1.2 Research Questions

This thesis work has been guided mainly by three research questions. The first question relates to how current technologies can be used to support complex event processing, with particular focus on how event abstractions can be generated and represented, and how events can be queried.

RQ1 How can events be modeled to support event abstraction and querying in RSP systems to assist in semantic complex event processing?

The focus is primarily on the underlying event model and how it can be used to support representation and generation of complex events using currently available RSP technology. The question is viewed from the perspective of some of the most popular RSP models and the current work of the RDF Stream Processing Community Group (RSP Group)[2]. The question is primarily explored as part of Chapter 4, but the event model is also practically used in the implementations presented in Chapter 6.

The second research question relates to how RSP queries can be abstracted in a way that lets them be made reusable and easier to maintain in RSP systems.

RQ2 How can RSP queries be abstracted to support reuse and maintenance of queries?

[2] `https://www.w3.org/community/rsp/`

The task of writing and validating the correctness of any non-trivial query in a streaming context is both complex and time consuming. Using prepared queries would allow optimization and validation to be done beforehand, which would reduce the time required to issue the necessary queries when applied to recurring tasks that have only minor variations. While there are several ways of supporting such parameterization, an explicit goal is to also support sharing and reuse based on some (preferably) standardized Semantic Web technology, rather than on some arbitrary string-based template format. The question is primarily answered and evaluated in Chapter 5, but the query abstraction model is also used to represent the RSP queries used in Chapter 6.

Finally, the third question, RQ3, relates to how current RSP technologies can support semantic complex event processing and the tasks faced by decision makers who need to process continuous streams of information on the fly.

RQ3 What are the limitations of current RSP technologies with respect to recurring decision-making tasks in the context of semantic complex event processing?

The focus here is on how structured information can be queried and modeled, rather than on how the information is presented visually to the users. The specific tasks considered here include information filtering, data aggregation, and pattern matching through the execution of recurring continuous queries. This question is primarily answered as part of Chapter 6, which considers the problem from an implementation standpoint, based on existing RSP technologies.

1.3 Contributions

Chapter 4 presents a new event model based on a set of requirements identified for complex event processing using Semantic Technologies. Unlike previous event models, the Event Processing ODP was developed specifically to support the representation and querying of event objects in complex event processing. The ontology provides a small set of classes and properties that, among other things, allow one event object to encapsulate another. Different approaches to representing event object boundaries are also discussed, both in general terms and with respect to existing RSP models.

Chapter 5 presents an extension to the SPIN syntax. SPIN lets SPARQL queries be represented as RDF and supports meta modeling on top of these queries to create parameterized query templates. The extension, called RSP-SPIN, can be used to represent RSP queries. By leveraging RSP-SPIN and the provided RSP-SPIN API, applications can transition between the string representation of queries and RDF. The RDF representation also serves as a query language agnostic layer, which allows a query to be serialized into

different RSP dialects. The API presently provides serializers for RSP-QL, CQELS-QL, C-SPARQL, and $\text{SPARQL}_{\text{stream}}$.

Finally, Chapter 6 presents a conceptual architecture for semantic complex event processing, where RSP queries can be viewed as individual event processing agents (EPAs) in an event processing network. This very general view allows application-specific functionality to be supported by simply incorporating it as an EPA. This is quite different from other event processing approaches in the RSP domain, which have tended to regard the RSP execution environments as standalone systems. The proposed architecture was implemented in three different scenarios. The first demonstrates a minimal setup, where the building blocks communicate directly with each other. The second and third implementations are more use-case specific, with frameworks that are considerably more scalable since they use specialized message-oriented middleware to manage the communication in the event processing networks. These implementations serve to illustrate that current RSP systems, if used appropriately, can be used to support complex event processing under realistic stream velocities.

1.4 Thesis Outline

Chapter 2 presents the relevant concepts within stream processing, complex event processing, and the Semantic Web. RDF stream processing (RSP) is introduced as a way of combining stream processing with Semantic Web technologies, and an overview of the most prominent RSP models and implementations is provided. This is followed by a brief description of the standardization efforts of the RSP Group and an overview of the RSP Query Language (RSP-QL) is provided. Finally, query parameterization is discussed briefly and SPIN is presented as a way of representing query templates as RDF.

Chapter 3 presents an overview of the scientific method viewed from a technology research perspective. The chapter then describes the research process of thesis work, and describes the way in which the work has been evaluated in the two research projects VALCRI and E-care@home.

Chapter 4 presents a set of requirements for complex event processing using Semantic Web technologies, and introduces a new event model. The model are evaluated against the requirements, and the querying ability of the model is demonstrated using standard SPARQL. The problem of event object boundaries is then discussed and possible workarounds for existing RSP systems are illustrated. Finally, the chapter discusses the impact the RSP-QL model with respect to event processing.

Chapter 5 presents an extension to the SPIN vocabulary (RSP-SPIN) to represent RSP queries as RDF. The result is a query language agnostic RDF representation that supports RSP-QL and most of the constructs of other RSP language extensions, while remaining compatible with standard SPIN. The

expressiveness of the model is demonstrated using the set of RSP-QL sample queries provided by the RSP Group. Additionally, RSP-SPIN is shown to support serialization into RSP languages that can be modeled by RSP-QL using a set of RSP benchmark queries as a reference.

Chapter 6 introduces a conceptual architecture for semantic complex event processing based on existing Semantic Web technologies. It presents three different implementations of the architecture, covering three separate use-cases: traffic-incident monitoring, criminal-activity monitoring, and electronic healthcare monitoring. The scenarios leverage the event model presented in Chapter 4 and model the queries using the SPIN extension presented in Chapter 5.

Chapter 7 discusses the results of the thesis, some of the ongoing work in the RSP domain, and presents ideas for future work. Finally, Chapter 8 draws conclusions based on the results of the thesis work and how these help answer the research questions.

1.5 Publications

- Robin Keskisärkkä. "Representing RDF Stream Processing Queries in RSP-SPIN". In: *Proceedings of the ISWC 2016 Posters Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016) Kobe, Japan, October 19, 2016.* Vol. 1690. Aachen, Germany: CEUR-WS.org, Oct. 2016.

  The paper demonstrates how RSP queries can be formulated as RSP-QL, parsed into RSP-SPIN, and then serialized into multiple RSP languages. The paper writing, coding, and evaluation were performed by the author.

- Robin Keskisärkkä. "Query Templates for RDF Stream Processing". In: *Joint Proceedings of the 3rd Stream Reasoning (SR 2016) and the 1st Semantic Web Technologies for the Internet of Things (SWIT 2016) workshops co-located with 15th International Semantic Web Conference (ISWC 2016) Kobe, Japan, October 17th to 18th, 2016.* Vol. 1783. Aachen, Germany: CEUR-WS.org, Oct. 2016, pp. 25–36.

  The paper presents RSP-SPIN as a way of representing RSP query templates as RDF. The paper writing, coding, and evaluation were performed by the author.

- Robin Keskisärkkä and Eva Blomqvist. "Supporting Real-Time Monitoring in Criminal Investigations". In: *The Semantic Web: ESWC 2015 Satellite Events ESWC 2015 Satellite Events, Portorož, Slovenia, May 31 – June 4, 2015, Revised Selected Papers.* Vol. 9341. Computer Communication Networks and Telecommunications. Springer International Publishing, June 2015, pp. 82–86.

The paper demonstrates the use of RSP technology for real-time monitoring in criminal intelligence. The paper presents an approach for creating a decoupled system using the actor model that allows different technologies to be incorporated into the same workflow. The coding and evaluation were performed by the author. The paper was written by the author and reviewed together with the co-author.

- Robin Keskisärkkä and Eva Blomqvist. "Sharing and Reusing Continuous Queries – Expression of Interest". In: *RDF Stream Processing Workshop in conjunction with the 12th Extended Semantic Web Conference (ESWC 2015) May 31st, 2015 in Portorož, Slovenia.* May 2015.

  The paper presents first ideas towards an extension of the SPIN syntax to support the upcoming RSP-QL semantics. The paper was written by the author and reviewed together with the co-author.

- Robin Keskisärkkä and Eva Blomqvist. "Towards the Use of RDF Stream Processing Engines for Event Enrichment from Social Media Streams". In: *Workshop on Semantics and Analytics for Emergency Response (SAFE2015) Collocated with the The 12th International Conference on Information Systems for Crisis Response and Management (ISCRAM2015).* May 2015.

  The paper demonstrates the feasibility of using existing RSP systems for event enrichment from social media streams. The coding and evaluation were performed by the author. The paper was written by the author and reviewed together with the co-author.

- Robin Keskisärkkä. "Semantic Complex Event Processing for Decision Support". In: *The Semantic Web – ISWC 2014 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II. Vol. 8797. Lecture Notes in Computer Science.* Springer International Publishing, Oct. 2014, pp. 529–536.

  The paper discusses different RSP systems for use in complex event processing from the perspective of monitoring social media streams, and emphasizes the value of being able to utilize external tools for specific processing tasks. The paper was written and planned together with Eva Blomqvist.

- Robin Keskisärkkä and Eva Blomqvist. "Semantic Complex Event Processing for Social Media Monitoring – A Survey". In: *SMILE 2013 Social Media and Linked Data for Emergency Response Proceedings of the Workshop on Social Media and Linked Data for Emergency Response co-located with 10th Extended Semantic Web Conference (ESWC 2013) Montpellier, France, May 26, 2013.* Vol. 1191. Aachen, Germany: CEUR-WS.org, 2013.

The paper describes the ongoing PhD work towards developing and evaluating techniques for creating aggregated and layered event abstractions. The paper was planned and written by the author.

- Robin Keskisärkkä and Eva Blomqvist. "Event Object Boundaries in RDF Streams: A Position Paper". In: *Proceedings of the 2nd International Workshop on Ordering and Reasoning Co-located with the 12th International Semantic Web Conference (ISWC 2013) Sydney, Australia, October 22nd, 2013*. Vol. 1059. Aachen, Germany: CEUR-WS.org, Oct. 2013, pp. 37–42.

  The paper presents the problem of maintaining event object boundaries in streams under different assumptions of RDF streams, and proposes a few general approaches to tackle this problem. The paper was written by the author and reviewed together with the co-author.

- Mikko Rinne, Eva Blomqvist, Robin Keskisärkkä, and Esko Nuutila. "Event Processing in RDF". In: *WOP 2013 Workshop on Ontology and Semantic Web Patterns Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns co-located with 12th International Semantic Web Conference (ISWC 2013) Sydney, Australia, October 21, 2013*. Vol. 1188. Aachen, Germany: CEUR-WS.org, Oct. 2013.

  The paper presents a new event model for representing event objects in RDF. The author contributed mainly with input regarding the model, query generalization, proofreading, and planning. The paper and pattern were prepared primarily by Mikko Rinne and Eva Blomqvist.

2 Background and Related Work

The last decade has seen an increased need for models that enable processing of streaming data [18]. The existing *stream processing systems* can be broadly divided into two classes: *data stream management systems* (DSMSs) and *complex event processing* (CEP) systems. Both provide ways of dealing with high-velocity input data, while maintaining low processing latency; however, they differ with respect their approaches and have largely different origins [18].

DSMSs have been developed from traditional relational database management systems (DBMSs). The typical DBMS allows consumers to issue one-time queries to retrieve individual results, whereas a DSMS continuously evaluates rules (or queries) as new data becomes available and makes the results available to the consumer [52]. These systems often extend the relational model by supporting queries that can express limits over potentially infinite streams. A stream-to-relation operator can be used to define a subsequence (or scope) over a stream, resulting in a finite set of elements referred to as a *window* (Definition 1) [10, 52].

Definition 1. *A **window** defines a subsequence over a stream.*

A *count-based window* (or *tuple-based window*) is defined based on the number of elements that are included in the subsequence (Definition 2). *Time-based windows* are instead defined based on the timestamps associated with the elements in the stream (Definition 3). A window slides over a stream as new data becomes available, and the step size (or interval) by which it

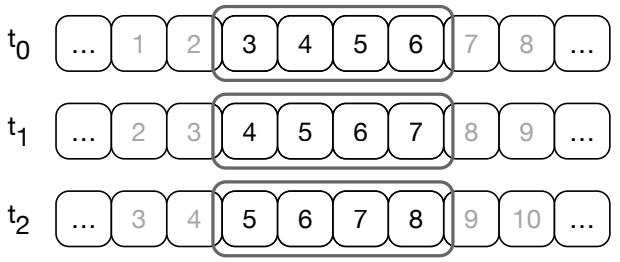


Figure 2.1: Illustration of a sliding window at three different points in time. Only the elements within the window's boundaries are considered during processing.

slides determines if two consecutive windows are disjoint or overlapping [52]. Figure 2.1 illustrates the idea of a sliding window.

Definition 2. *A **count-based window** defines a subsequence over a stream based on an integer representing (at most) the number of elements in the window.*

Definition 3. *A **time-based window** defines a subsequence over a stream based on an upper and a lower time bound.*

When a portion of a stream has been isolated, any relational operator can be used to process the data, and the results can be converted back into a stream using a relation-to-stream operator. The operator determines whether the results should contain everything from the current evaluation (i.e., snapshot semantics), or only the differences with respect to the previous evaluation (i.e., incremental semantics) [10, 18]. The three types of relation-to-stream operators that are typically supported can be defined as:

$$\begin{aligned}\text{RStream}(\omega_t) &= \{x \mid x \in \omega_t\} \\ \text{IStream}(\omega_t, \omega_{t-\delta}) &= \{x \mid x \in \omega_t, x \notin \omega_{t-\delta}\} \\ \text{DStream}(\omega_t, \omega_{t-\delta}) &= \{x \mid x \notin \omega_t, x \in \omega_{t-\delta}\}\end{aligned}$$

where ω_t represents the mappings from the current evaluation, $\omega_{t-\delta}$ represents the mappings from the previous evaluation, and x is an element in the stream. The relation stream (RStream) outputs all mappings that are in the current evaluation (ω_t). The insert stream (IStream) returns only new mappings, which includes only those that are in the current evaluation (ω_t) but were not present in the previous one ($\omega_{t-\delta}$). Finally, the delete stream (DStream) returns mappings that are not in the current evaluation (ω_t) but were present in the previous one ($\omega_{t-\delta}$). The RStream operator can lead to result duplication if the step of a window is less than its size, which causes

two consecutive windows to overlap. The other two operators avoid result duplication but require that the results from the previous evaluation are kept in memory. The Stanford Stream Data Manager (STREAM) [9] and Aurora [1] were among the first DSMSs to be implemented with support for these types of stream processing operators, but implementations that are more recent still primarily follow the same general design principles [18].

CEP systems focus on identifying meaningful events or detecting event patterns from multiple streaming sources. In this context, determining if a streamed element can be ignored is not always possible using stream windows. For example, a complex event generated from an event pattern can reference historical data in the stream that would not necessarily be captured in a fixed window. An *event* is here regarded as anything that represents something that can be thought of as "happening" (Definition 4), and a *complex event* is an event that in some way summarizes, represents, or denotes other events (Definition 5) [37].

Definition 4. *An **event** is anything that happens, or is regarded as happening.*

Definition 5. *A **complex event** is an event that summarizes, represents, or denotes a set of other events.*

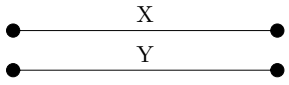
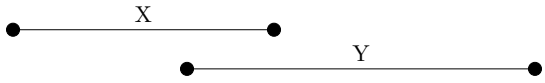
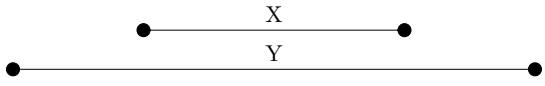
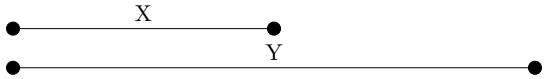
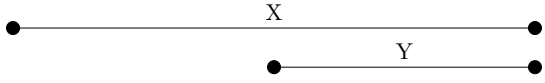
Event abstractions are formed when a number of low-level events are aggregated to form high-level events, possibly in multiple steps. Due to the importance of temporal patterns, such as event co-occurrence and event sequences, CEP systems will often contain explicit support for operators that express temporal relations. The operators that can be defined between two intervals using Allen's interval algebra [5] are illustrated in Table 2.1. The temporal operators can also be implemented as logical comparisons between the start and end times of events. If the event model uses a single timestamp per event only three of these relations are defined: before, after, and equals.

Both DSMSs and CEP systems build on models that require data streams of fixed structures [38]. This assumption is, however, generally not applicable in a web setting. Streams on the web are heterogeneous in terms of structure, format, and semantics, and implementing services that can integrate such streams in these applications is far from trivial.

The Semantic Web is an extension of the web that provides a framework in which information can be shared and reused, based primarily on the standards and recommendations promoted by the World Wide Web Consortium (W3C)[1]. At the core of the Semantic Web is the Resource Description Framework (RDF), which is a data model where the basic building block is a subject–property–object triple (or *RDF triple*) [8, pp. 67–70]. A subject is either a URI or a blank node (anonymous resource), a property is a URI, and an object is either a URI, a blank node, or a data-typed literal (Definition 6).

[1] `https://www.w3.org/`

Table 2.1: Illustration of the 13 temporal operators supported by Allen's temporal algebra [5].

Operator	Illustration
x before y y after x	
x equals y y equals x	
x meets y y metBy x	
x overlaps y y overlappedBy x	
x during y y contains x	
x starts y y startedBy x	
x finishes y y finishedBy x	

Definition 6. *An **RDF triple** consists of a subject, a property, and an object. A subject is a URI or a blank node, a property is a URI, and an object is a URI, a blank node, or a literal.*

The RDF abstract syntax defines an *RDF graph* as a set of triples, where the property represents the links between nodes (Definition 7). RDF graphs can be ordered into collections as *RDF datasets*, which are comprised of a single default graph and zero or more named graphs identified by URIs or blank nodes (Definition 8).

Definition 7. *An **RDF graph** is a directed labeled graph consisting of a set of RDF triples.*

Definition 8. *An **RDF dataset** is comprised of a default graph and zero or more named graphs.*

Although RDF was originally developed as a way of expressing metadata on the web, it has been adopted as a general-purpose format for expressing structured information. The RDF data model allows data to be semantically

enriched based on a small set of standards. Ontologies (e.g., expressed using OWL[2] or RDFS[3]) are used to provide information about concepts and relations, which in turn can be used to infer implicit information from existing data through a process that is referred to as *reasoning*. Ontologies are also used to model constraints, define vocabularies and schemas, and to provide concept definitions. Figure 2.2 shows an informal representation of a simple RDF graph, with the resources and properties prefixed for readability. The lines represent the properties that link the nodes in the graph, and the dotted line represents an inferred relation. The FOAF vocabulary[4] allows the type of `ex:Mary` to be deduced based on a domain restriction, which states that the subject of `foaf:knows` is a `foaf:Person`. The first name of `ex:John` is a literal and cannot link to other resources.

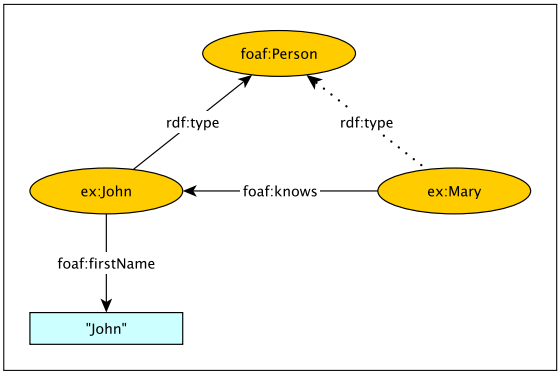


Figure 2.2: Example of data represented as an RDF graph. The dotted line denotes the inferred type for `ex:Mary` based on the domain restriction of the property `foaf:knows` in the FOAF vocabulary.

Many different text formats can be used to express RDF. Two syntaxes, Turtle[5] and TriG[6], will be used to represent RDF throughout the coming chapters. Listing 2.1 shows the data from Figure 2.2 expressed formally in Turtle. The prefixes are defined at the top of the document. Triples are terminated with a period, but the subject can be reused if the triple is terminated with a semicolon[7]. Blank nodes are represented as brackets or a label prefixed with an underscore. TriG is an extension of Turtle that supports grouping sets of triples into named graphs.

[2] `https://www.w3.org/TR/owl2-overview/`
[3] `https://www.w3.org/TR/rdf-schema/`
[4] `http://xmlns.com/foaf/spec/`
[5] `https://www.w3.org/TR/turtle/`
[6] `https://www.w3.org/TR/trig/`
[7] Turtle supports other forms of syntactic sugar that has been designed to make RDF easier to both read and express for humans.

```
@prefix ex:   <http://example.org#>
@prefix foaf: <http://xmlns.com/foaf/0.1/>

ex:John a foaf:Person ;           # John is a person
        foaf:firstName "John" .   # John has first name "John"
ex:Mary foaf:knows ex:John .      # Mary knows John
```

Listing 2.1: The data in Figure 2.2 expressed in Turtle format.

There are a number of ways in which RDF data can be queried but SPARQL[8] has been proposed as the standard query language by the W3C. SPARQL is inspired by SQL-like languages and it supports traversal of RDF data based on graph patterns. A basic graph pattern in SPARQL is similar to Turtle in structure but any part of a triple can be replaced by a variable, and bindings are produced for the variables by mapping the pattern to the target graph. Depending on the query type, the output of a query can be an RDF graph, a table, or a Boolean value. Listing 2.2 shows an example of a select query that returns all persons who know someone with the first name *John*.

```
PREFIX ex:   <http://example.org#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person1
WHERE {
   ?person1 foaf:knows ?person2 .
   ?person2 foaf:firstName "John" .
}
```

Listing 2.2: A SPARQL query returning persons who know someone with the first name *John*.

On the web, information is rarely complete and we usually need to consider the *open-world assumption*. The open-world assumption states that the absence of a statement alone cannot be used to infer that the statement is false. For example, in Figure 2.2 it would not be possible to infer that `ex:Mary` has no first name, since it is possible (and very likely) that we simply do not have any information about it.

The traditional Semantic Web technologies have been developed for scenarios that deal with large amounts of data, but have assumed that changes to data are infrequent and involve relatively small data volumes. As a result, the current reasoning approaches are often unsuitable for the dynamic nature of streaming data [38]. *Stream reasoning* is attempting to bridge this gap by unifying reasoning and stream processing [54]; however, stream reasoning is still largely unexplored, with few tools available that support any actual reasoning over streams and temporal data [38].

[8] `https://www.w3.org/TR/sparql11-query/`

2.1 RDF Stores

RDF stores (often referred to as *triple stores*) are databases designed to manage RDF data. Most modern RDF stores are in fact *quad stores* that support named RDF graphs. An *RDF quad* refers to a triple associated with a specific named graph (or context) (Definition 9).

Definition 9. *An **RDF quad** is an RDF triple associated with a specific named graph (or context).*

High-performance RDF stores are typically optimized for performance on large datasets, sacrificing memory to improve query-processing performance. The underlying storage technology of RDF stores has historically been DBMSs, such as OpenLink Virtuoso[9], but native implementations now offer competitive (or superior) performance in most scenarios. A few well-known examples of RDF stores are Apache Jena TDB[10], Stardog[11], BlazeGraph[12], and RDF4J[13]. There have also been some attempts at using other graph databases as the underlying storage, but the performance of these systems varies greatly depending on the use-case and structure of the underlying model. In terms of reasoning, most RDF stores support at least a subset of RDFS reasoning, and many support various fragments of OWL and/or user-defined rules.

The RDF store is responsible for evaluating the SPARQL queries issued by the user. This places the bulk of the work on the server, but another alternative is to have the client retrieve the data and process it locally. However, both of these approaches polarize the distribution of the workload. Linked Data Fragments [55] recognizes that there is a large gap between these two extremes, and that several approaches in this space have yet to be explored. Triple Pattern Fragments [56] is one such approach that supports only a subset of SPARQL on the server side, and relies on the client to do the remainder of the processing. The idea is to find a balance between the amount of work that has to be done by the server and the client respectively.

Despite the high performance and scalability of current state-of-the-art RDF stores, there is no solution that offers similar performance with respect to querying and reasoning over data that arrives in continuous streams.

2.2 RDF Stream Processing

A number of models and languages for RDF Stream Processing (RSP) have been proposed throughout the last decade, with the aim of combining the Se-

[9] `http://virtuoso.openlinksw.com/`
[10] `https://jena.apache.org/`
[11] `http://docs.stardog.com/`
[12] `https://www.blazegraph.com/`
[13] `http://rdf4j.org/`

mantic Web principles with processing of streaming data. The most common way of processing streaming RDF data has been to base it on *RDF triple streams* (Definition 10). An RDF triple stream is a potentially unbounded sequence of time-annotated RDF triples [12, 43]. The stream is assumed to be ordered with respect to the associated timestamps, or implicitly ordered with respect to the time of arrival. Triple streams that also provide an expiration time as part of the time annotation denote the valid intervals of the streamed triples [6].

Definition 10. *An **RDF triple stream** is a potentially infinite ordered sequence of time-annotated RDF triples.*

Triples with the same timestamp are interpreted as having occurred at the same time instant. This approach is similar to how time is often managed in DSMSs [9]. Table 2.2 shows an example of a data stream from a social event, where persons entering and exiting rooms are registered by door sensors.

Table 2.2: Example of an RDF triple stream from a social event indicating persons entering or exiting rooms.

RDF triple	Timestamp
`:John :enters :Room1`	2017-01-01 10:00:01
`:Mary :enters :Room2`	2017-01-01 10:01:10
`:Paul :enters :Room3`	2017-01-01 10:01:11
`:John :exits :Room1`	2017-01-01 10:03:01
`:John :enters :Room2`	2017-01-01 10:03:01
`:Paul :exits :Room3`	2017-01-01 10:03:02
`:Paul :enters :Room2`	2017-01-01 10:03:02

The data elements in a stream can also be RDF graphs, with time annotations potentially represented as part of the streamed data. This will be referred to as an *RDF graph stream* (Definition 11).

Definition 11. *An **RDF graph stream** is a potentially infinite ordered sequence of time-annotated RDF graphs.*

Stream joins (i.e., instances where multiple streams are queried simultaneously) typically cannot be supported reliably using implicit stream order alone due to, for example, network latencies. Similarly, any approach that relies on timestamps generated in a distributed system needs to consider strategies for coping with time synchronization, which is an issue that has been studied extensively [36, 42].

As mentioned earlier in the chapter, one of the most useful constructs in DSMSs is the window operator. In RSP a window defines either a count-based or time-based subsequence over an RDF stream (see Definitions 2 and

3). Data within the window is assumed relevant, whereas data outside the window is ignored and can potentially be discarded. The *range* (or size) of a count-based window is defined in terms of the number of streamed elements, whereas a time-based window is defined in terms of a duration or a time interval [12]. A window in the past can be specified by offsetting the upper bound of the window to an earlier point in time.

2.2.1 RSP Implementations

A number of RSP models and languages have been proposed throughout the last decade, aiming at combining the principles of the Semantic Web with stream processing and CEP.

Streaming SPARQL [15] was one of the first language extensions to SPARQL for processing RDF data streams. The contribution of the model in the RSP field can be regarded as primarily theoretical. The first work on defining an algebra for a streaming version of SPARQL [29] was also primarily theoretical, although it did provide an initial comparison with a purely static approach. Both approaches have been used as inspiration in later approaches.

C-SPARQL [12] is a query language that extends SPARQL to support processing of RDF triple streams. The language is defined as an extension to SPARQL 1.1 and supports both time-based and count-based windows over the most recent portions of a stream. The rate at which windows slide over streams is defined declaratively as part of the window definitions in the dataset clause. Queries are executed periodically and the model implicitly supports snapshot semantics (RStream). The execution interval can be specified as part of the query, or it can be based on the rate at which the windows' content changes. This separation of query execution from data arrival has been criticized as a major limitation of C-SPARQL [38], and the query repetition rate can be a limiting factor in terms of query response latency [50]. The current implementation of the C-SPARQL execution engine has some additional limitations compared with the language definition and model[14]. An important restriction is that named streams are not supported, meaning that there is no way of tracking the provenance of triples arriving in different streams, and the execution interval cannot be set explicitly. C-SPARQL also provides a special timestamp function that lets the associated timestamp of a triple be retrieved, which allows temporal comparisons between triples to be made. Listing 2.3 shows an example query formulated using C-SPARQL over the example stream in Table 2.2 that reports meet-ups between individuals if they enter the same room within two seconds of each other.

$\text{SPARQL}_{\text{stream}}$ [16] is a language that is similar to both C-SPARQL and Streaming SPARQL. The main difference is that $\text{SPARQL}_{\text{stream}}$ only supports time-based windows, which are defined using an upper and a lower

[14] At the time of writing the latest version of the engine was version 0.9.7.

```
REGISTER QUERY meetups COMPUTED EVERY 1 s AS

PREFIX : <http://example.org#>

SELECT ?person1 ?person2 ?room
FROM STREAM :social [RANGE 2 s STEP 1 s]
WHERE {
   ?person1 :enters ?room .
   ?person2 :enters ?room .
   FILTER(?person1 != ?person2)
}
```

Listing 2.3: Example of a C-SPARQL query registering meet-ups between persons that enter the same room within a two second interval.

time bound. The relation-to-stream operator can also be defined as part of the query, which makes it the only implemented RSP language that explicitly supports both snapshot and incremental semantics. The use of time bounds to specify windows over streams also provides a way of defining windows in the past (i.e., windows do not necessarily bind to the most recent portions of streams). Morph-Streams [16], the execution environment for $\text{SPARQL}_{\text{stream}}$, is implemented on top of the sensor network platform SNEE [24]. Listing 2.4 shows the query that was expressed for C-SPARQL above using $\text{SPARQL}_{\text{stream}}$.

```
PREFIX : <http://example.org#>

SELECT RSTREAM ?person1 ?person2 ?room
FROM STREAM :social [FROM NOW-1 S SLIDE 1 S]
WHERE {
  ?person1 :enters ?room .
  ?person2 :enters ?room .
  FILTER(?person1 != ?person2)
}
```

Listing 2.4: Example of a $\text{SPARQL}_{\text{stream}}$ query registering meet-ups between persons that enter the same room within a two second interval.

CQELS-QL [43] also extends SPARQL to manage streaming data but differs from the previous approaches in a number of ways. In CQELS, the query execution rates are synchronized with the rate of the streams (i.e., execution is data driven), and it only reports the additions to the current evaluation, which means that it implicitly supports incremental semantics (IStream). Multiple windows can be defined over the same stream within a single query, a feature that is not supported by any other RSP implementation to date. Defining the step size of a time-based window is optional, and leaving it empty will trigger query execution every time a new triple is processed.

CQELS-QL supports two additional keywords to define the range of windows. Using the `ALL` keyword means that the entire stream will be included in the window (i.e., the subsequence represented by the window will be a snapshot of the entire stream up to that time). The `NOW` keyword only maintains the most recent timestamp in the window, and the exact meaning depends on the temporal granularity of the stream. Listing 2.5 shows the query used in the two previous examples expressed using CQELS-QL. A new version of CQELS has also been proposed that introduces a temporal operator and path navigation to the model, with the goal of supporting more expressive CEP [19]; however, this work is still in progress and no implementation has yet been released that supports the updated syntax.

```
PREFIX : <http://example.org#>

SELECT ?person1 ?person2 ?room
WHERE {
  STREAM :social [RANGE 2s] {
     ?person1 :enters ?room .
     ?person2 :enters ?room .
  }
  FILTER(?person1 != ?person2)
}
```

Listing 2.5: Example of a CQELS-QL query registering meet-ups between persons that enter the same room within a two second interval.

Sparkwave [35] is an RSP implementation that supports continuous reasoning over RDF data streams and time-based windows. According to the authors, preliminary evaluations of the execution engine have shown that it can provide higher throughput than both C-SPARQL and CQELS under certain conditions, even with reasoning enabled [35]. However, the experiments were based on an adaptation of the Berlin SPARQL Benchmark [13], which is designed to evaluate the performance of RDF stores, and contained no queries that required more than a single data stream. Sparkwave has some known limitations with respect to the size of the background knowledge that can be supported efficiently, and it provides only limited reasoning functionality [38]. Sparkwave includes no declarative query language and the available implementation is not actively maintained.

EP-SPARQL [6] is a SPARQL 1.0 extension designed for event processing, and it explicitly supports temporal operators as part of the language. EP-SPARQL supports RDF triple streams but assumes that all triples are annotated with a time interval representing the valid time. The execution engine for EP-SPARQL translates queries into the ETALIS Language for Events (ELE), which are then executed on the ETALIS engine [6, 7]. ETALIS is based on declarative semantics and implemented in Prolog, and to support the RDF representation all event streams and background data are converted into the

internal ETALIS format. Background data is assumed to be static and is converted offline, whereas the streams are translated on the fly. Functions exist to access start time, end time, and duration of a matched graph pattern. Furthermore, construct queries allow recursive production rules to be defined. EP-SPARQL is data driven and results are returned continuously, but unlike the approaches discussed so far EP-SPARQL provides no way of defining windows over streams. There is also no way of referencing streams on a query level. Listing 2.6 shows the same query as in the previous examples expressed using EP-SPARQL.

```
PREFIX : <http://example.org#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?person1 ?person2 ?room
WHERE {
   { ?person1 :enters ?room }
   SEQ
   { ?person2 :enters ?room }
   FILTER(?person1 != ?person2 &&
          getDuration() <= "PT2S"^^xsd:duration)
}
```

Listing 2.6: Example of an EP-SPARQL query registering meet-ups between persons that enter the same room within a two second interval.

Finally, INSTANS [48] is a platform for executing continuous queries using standard SPARQL and SPARQL Update. The focus of INSTANS is to support CEP and incremental processing of queries. Continuous evaluation of incoming data is performed against the compiled set of registered queries using a data driven approach, and intermediate results are kept in cache. Streams are modeled by inserting data into graphs using SPARQL Update queries, which means that streamed elements in INSTANS can be both RDF triples and RDF graphs, and time annotations can be provided as part of the streamed elements. When query conditions are matched, the new results are immediately made available (IStream). INSTANS does not support windows over streams but indirect mechanisms can, in principle, be used to manually define similar behavior in queries. Additionally, support for reasoning has been demonstrated in INSTANS for materialization-based reasoning using collections of rules coded in SPARQL [49]. The example query used in the previous examples expressed as standard SPARQL can be seen in Listing 2.7. In the query, we assume that an RDF graph stream is used, and leverage the datetime functions included with INSTANS to calculate the time difference between the events.

```
PREFIX : <http://example.org#>
PREFIX datetime: <http://instans.org/extensions/datetime#>

SELECT ?person1 ?person2 ?room
WHERE {
   GRAPH ?g1 { ?person1 :enters ?room }
   GRAPH ?g2 { ?person2 :enters ?room }
   FILTER(?person1 != ?person2)
   ?g1 :observationTime ?ts1 .
   ?g2 :observationTime ?ts2 .
   BIND(datetime:datetime_in_seconds(?ts1) AS ?t1)
   BIND(datetime:datetime_in_seconds(?ts2) AS ?t2)
   FILTER(ABS(?t1 - ?t2) <= 2)
}
```

Listing 2.7: Example of a SPARQL query in INSTANS registering meet-ups between persons that enter the same room within a two second interval.

2.3 Limitations of Current RSP Approaches

Many of the query patterns of interest in RSP involve joining a continuous data stream with some background knowledge. The background data is often maintained externally, and RSP systems therefore usually maintain a local (or cached) version of the relevant parts for efficiency; however, current RSP models do not provide any means for refreshing this data. There has been some work on updating parts of the cached data automatically to maximize the freshness of the results [26], but current RSP implementations tend to treat background knowledge as completely static during the lifetime of a query.

With respect to streaming data, the situation is quite the opposite; streaming data is regarded as transient and persisting any part of the data is usually not possible. Most of the systems mentioned above support stateful processing in the form of aggregation (e.g., sum, average, max, and min) but only INSTANS [48] can persist, delete, and update data to maintain a local state. As an example, assume that we need to maintain a list of all people currently inside a building based on the door sensor stream used in the previous sections. Without storing any part of the data, this would not be possible using C-SPARQL, CQELS, or $\text{SPARQL}_{\text{stream}}$. EP-SPARQL supports windowless recursive production rules, which can be used to persist but not delete or change data during the query runtime.

The models discussed above primarily represent time using an explicit (or implicit) time annotation on the streamed elements; however, time can also be represented directly as RDF. Temporal labeling of the streaming data could be accomplished using, for example, the OWL Time Ontology[15], which also supports the elementary relations between intervals described in Allen's temporal algebra (see Table 2.1). Incorporating temporality into standard RDF

[15] https://www.w3.org/TR/owl-time/

is also possible, and it would not necessarily introduce any severe impact on query complexity, whether introduced as point-based or interval-based labeling [30]. However, this deviates from the RDF standard since it introduces an extra field to the RDF model.

The RSP systems discussed are generally very limited in terms of reasoning, although some can be extended using manually defined rules. The naive approach to reasoning would be to re-materialize the inferences from scratch every time some data changes, which would be computationally costly, and reasoning over streams generally requires incremental processing mechanisms. Strategies exist that allow a system to maintain a materialized graph by updating only those parts that are affected by a change. For example, the Delete/Rederive (DRed) algorithm can be used to maintain materializations by storing expiration times for new triples [11]. A maintenance program could monitor the inserted triples and keep track of the entailments, discarding the derived triples for which no justification exists. Compared to the naive approach this has been shown to scale well as long as the amount of information subject to change does not increases beyond a certain threshold, where maintaining the materialization would eventually become more expensive than simply starting from scratch [11].

Reasoning can also partially be modeled directly as a part of SPARQL queries. For example, SPARQL path expressions could be used to model the behavior of transitive and symmetric properties. ETALIS explicitly supports recursive production rules, and can be extended with more reasoning-specific rules. Furthermore, materialization-based stream reasoning covering RDFS and OWL2 RL using networks of SPARQL Update rules has been demonstrated in INSTANS [49].

Table 2.3 summarizes the features offered by the state-of-the-art RSP implementations. The table refers to the support offered by the respective implementation, as opposed to the underlying models or languages. None of the engines supports all of the listed features.

Table 2.3: List of features offered by current RSP implementations.

System	Execution strategy	Stream type	Time model	Window operator	Reasoning	Temporal operators	Recursive Processing	Stateful processing	Distributed processing
C-SPARQL	Periodic	Triple	Point	Time-/count based	RDFS[1]				
C-SPARQL on S4 [32]	Periodic	Triple	Point	Time-/count based	RDFS[1]				yes
Morph-Streams	Periodic	Triple	Point	Time based					
CQELS	Reactive	Triple	Point	Time-/count based		(*before*)[2]			
CQELS Cloud [46]	Reactive	Triple	Point	Time-/count based					yes
Sparkwave	Reactive	Triple	Point	Time based	RDFS[3]				
EP-SPARQL/ETALIS	Reactive	Triple	Interval		Rules[4]	*before*, *equals*	yes	(yes)[5]	
INSTANS	Reactive	Graph	(Point)[6]		RDFS/ OWL RL[7]		yes	yes	

[1] Supports a subset of RDFS.
[2] A CQELS-QL has been proposed to support *before* [19] but it is not supported in the released version.
[3] Implementation also supports `owl:inverseOf` and `owl:SymmetricProperty`.
[4] Supports rules expressed in ETALIS Language for Events (ELE).
[5] Partial support only since data cannot be changed or deleted.
[6] Implicitly based on time of arrival.
[7] Demonstrated for RDFS, ρdf, D*, P-entailment and OWL 2. RL [49].

2.3.1 Scalability

Evaluating RSP system performance is important when comparing different approaches, and several RSP benchmarks have been proposed, including SR-Bench [57], LSBench [44], CSRBench [20], CityBench [2], and YABench [34]. SRBench focuses mainly on the functional coverage of the query languages, LSBench targets throughput correctness by comparing and quantifying the mismatch between RSP systems, and CSRBench evaluates correctness based on the operational semantics of RSP systems. CityBench provides a configurable infrastructure and a set of queries to capture the performance metrics using smart-city datasets. Finally, YABench is a framework that attempts to unify the characteristics of existing benchmarks to measure precision, recall, performance, and scalability characteristics while varying query complexity and stream throughput.

Despite all of these benchmarks, comparing the performance of RSP systems is still difficult, due to the differences that exist both on the syntactic and semantic levels. For example, CQELS, Sparkwave, EP-SPARQL, and INSTANS use a reactive execution strategy, where queries are matched as soon as new data becomes available, while $\text{SPARQL}_{\text{stream}}$ and C-SPARQL execute queries periodically. The reactive strategy reduces latencies in real-time applications and minimizes unnecessary query execution (i.e., when no data is being processed), whereas the latter can potentially be used to adjust execution rates to help optimize performance. For example, when processing high-velocity or bursty streams the data driven query evaluation strategy may not have time to return all results before the next execution is triggered, whereas the periodic strategy would always have a fixed time period reserved for each execution.

The performance of CQELS, ETALIS (i.e., the underlying engine of EP-SPARQL), and C-SPARQL has been evaluated with respect to execution times, performance for parallel queries, and amount of background data [43]. The results indicated that C-SPARQL had low throughput compared with the other two engines, and that the average latency for a single query was at least an order of magnitude faster for CQELS than for C-SPARQL, whereas ETALIS only performed better than CQELS for the most simple query. The size of the background data had only a marginal effect on the query response time for CQELS when tested with up to 10 million triples, whereas EP-SPARQL and C-SPARQL failed to produce any results at all when the dataset exceeded 2 million triples. Both CQELS and C-SPARQL managed to scale up to 1,000 parallel queries, but at this point the C-SPARQL execution times exceeded 100 seconds. The number of parallel queries handled by ETALIS in the same scenario was typically less than 10.

Another study compared INSTANS with C-SPARQL [50]. INSTANS was superior in terms of performance, and the difference between the two increased as a function of the window repetition rate. In these tests, the dominant factor

for response latency in C-SPARQL was typically the window repetition rates. While latency is an important factor it does not necessarily measure the overall throughput of a system, but it illustrates a potential performance bottleneck of the periodical strategy when high execution rates are required.

The maximum stream rates supported by different RSP systems are difficult to estimate, since it depends greatly on, for example, query complexity, number of parallel queries, reasoning support, size of background data, and the execution strategy of the underlying engine. However, scalable stream rates have been reported in the range of 100–1000 triples/second for CQELS and C-SPARQL with up to 1,000 parallel queries [45].

Scaling beyond the centralized approach has so far received relatively little attention. CQELS Cloud [46], an adaptation of the CQELS engine, can execute parallel queries in the cloud. While it has not been made available for download, and the performance has not been compared with other implementations under similar conditions, the authors reported that it could manage throughputs of up to 100,000 triples/second while running 10,000 parallel queries [46]. C-SPARQL has also been run on top of S4[16], but although the overall performance in terms of stream processing increased considerably and it managed to process more than 100,000 triples/second, it provided no support for background data, which severely limits its potential application areas.

2.4 RDF Stream Processing Query Language

The RSP Group is currently working on defining a common model for transmitting and querying streaming RDF data. The goal of the group has been defined as:

> *The goal of the RDF Stream Processing Community Group (RSP) is to define a common model for producing, transmitting and continuously querying RDF Streams. This includes extensions to both RDF and SPARQL for representing streaming data, as well as their semantics.*
>
> *RDF Stream Processing Community Group*

The standardization efforts are still in progress and we will use the existing work [22, 21], along with the ongoing discussions in the RSP Group[17], as a reference for discussing RSP-QL. First, a key difference in the model compared with previous RSP approaches is that it assumes RDF graph streams, where the streamed graphs are explicitly annotated with metadata. The use of RDF graph streams has considerable impact on the way in which complex

[16] `http://incubator.apache.org/projects/s4.html`
[17] `https://www.w3.org/community/rsp/`

events can be communicated in streams. For example, RDF triple streams require any non-atomic event (i.e., an event described using more than a single triple) to be re-assembled on the consumer side, where event-level punctuation (see Section 2.5.2) becomes a concern. With the new model, the complex objects can be described in their entirety and transported as a single element in the stream. Second, time annotations about streamed graphs are provided as part of the associated metadata, and the temporal information can be accessed directly as RDF. Third, as in most event processing systems, events are considered immutable. This means that any derived event should generate a new unique graph, allowing other objects to link back to events without the risk of them changing over time. Finally, the draft provides guidelines to the syntax of the new query language.

Listing 2.8 shows considerably more complex query against the door sensor stream used in the previous sections, which detects a pattern where two persons appear to meet up at approximately the same time three days in a row. The language has inherited the register from C-SPARQL clause but with only slight variation. The prefixes are defined at the very top of the query and the output stream is identified using an IRI, allowing queries to generate new streams declaratively. Construct queries have been extended to support the `GRAPH` keyword, which lets the produced results match the proposed RDF stream model. From $\text{SPARQL}_{\text{stream}}$ the query language inherits the way in which the relation-to-stream operator can be set explicitly. The definition of a window has been made analogous to the `FROM NAMED` syntax that applies to named graphs in the background data. A window over a stream is associated with an IRI, allowing it to be referenced in the body of the query. The window definitions are inspired by the formats used in C-SPARQL, CQELS-QL, and $\text{SPARQL}_{\text{stream}}$ but use a standardized format to express durations (ISO 8601). Finally, windows are referenced inside queries in a way that corresponds to how named graphs are referenced in standard SPARQL.

2.5 Event Processing Using RSP

Event processing systems typically make a distinction between components that introduce events into the event processing system (*event producers*), software modules that process events (*event processing agents*), and the entities that receive the events after processing (*event consumers*) [23]. These components are interconnected to form what is referred to as *event processing networks*. The event producer has only output sources, and no part of the event processing network provides any input to the producer. The event consumer has only input sources, and generates no new data for other components to process. The event processing agents (EPAs) are the core processing component that both consume and produce data.

```
PREFIX : <http://example.org#>
REGISTER STREAM :regular-meetup AS

CONSTRUCT ISTREAM {
   GRAPH ?g { ?p1 :meetup ?p2 }
   ?g :observationTime ?time .
}
FROM NAMED WINDOW :win1 ON :social [RANGE PT1H]
FROM NAMED WINDOW :win2 ON :social [FROM NOW-PT24H TO NOW-PT23H]
FROM NAMED WINDOW :win3 ON :social [FROM NOW-PT48H TO NOW-PT47H]
WHERE {
   WINDOW :win1 { # 1 hour window
      GRAPH ?g1 { ?p1 :enters ?room1 }
      GRAPH ?g2 { ?p2 :enters ?room1 }
   }
   WINDOW :win2 { # 1 hour window yesterday
      GRAPH ?g3 { ?p1 :enters ?room2 }
      GRAPH ?g4 { ?p2 :enters ?room2 }
   }
   WINDOW :win3 { # 1 hour window two days ago
      GRAPH ?g5 { ?p1 :enters ?room3 }
      GRAPH ?g6 { ?p2 :enters ?room3 }
   }
   FILTER( ?p1 != ?p2 )        # Not the same person
   BIND(NOW() AS ?time)        # Get current time
   BIND(IRI(STRUUID()) AS ?g)  # Create unique graph URI
}
```

Listing 2.8: Example of an RSP-QL query generating a new stream consisting of people who appear to meet up during the same hour of the day three days in a row.

Using an RSP system to model an entire event processing network requires that the event streams generated by one query can be used as input to another, since it must be possible to process events in multiple steps. Only EP-SPARQL and INSTANS support the creation of such networks declaratively, but all systems mentioned in the previous section could support this, to some extent, by simply creating a wrapper for the execution environment that allows a generated stream to be fed back into the same engine instance. However, this would have to be hard coded rather than managed declaratively in queries.

2.5.1 Event Objects

In event processing, it is often useful to distinguish between the *events* that occur in the real world, and *event objects* that are representations of these events. In practice, the two terms are often used interchangeably [37] and we will make this distinction only when required for clarity. For example, a single event may have more than one event object representation, where the one used internally for processing can be very different from one that is intended for a human user. Event objects can also represent hypothetical events, such as

false-positives in fraud detection, which may not have a corresponding real-world event [23].

The building blocks typically used to describe event objects are header, payload, and open content [23]. The event header represents the attributes that are required by the event processing system, and the attributes defined here are sometimes considered to be event-type independent. The event header usually contains an event id, an event type, and at least one time-stamp. The event payload may not be required for the actual processing by the system and the payload attributes are often event-type specific. Finally, open content is considered as the part of the event object that supports free-format data, where no strict restrictions apply.

Complex events represent or denote sets of other events. What constitutes a simple or complex event is context dependent, but as a rule all derived events are considered to be complex events. Complex events may contain references to the triggering events, which forms a hierarchy in which the higher levels are often on a more abstract level. An event object can reference other event objects either via links or by encapsulating them, thus creating and independent copy of the referenced event object.

In the Semantic Web domain there are several ontologies that can be used to model events. The DOLCE+DnS Ultralite (DUL) ontology[18] can be used to model either physical or social contexts, and various extensions let it be used to represent many types of information objects. One such extension, targeted specifically at representing events in the real world, is Event-Model-F. Event-Model-F provides support for representing, for example, object participation, and causal/correlative relationships between events [51]. The ontology follows a pattern-based approach that allows it to be extended easily for a particular domain. The Semantic Sensor Network Ontology (SSN)[19], on the other hand, is an ontology designed for describing sensors, observations, procedures, features of interest, observed properties, and actuators. In many ways the modeling of sensors is related to representing events but in a narrower context. The full SSN ontology is quite large but the most recent W3C proposal includes a self-contained core ontology (SOSA) that supports only the elementary classes and properties[20]. However, none of these ontologies have been designed specifically for processing in a CEP context.

2.5.2 Stream Punctuation

In order for CEP to be possible using RSP technology the boundaries of the event objects being transferred in an event stream somehow need to be *punctuated*. Punctuations can be used to explicitly communicate the boundaries of a subsequence in the stream, allowing an infinite stream to be viewed as

[18] `http://www.ontologydesignpatterns.org/ont/dul/DUL.owl`
[19] `https://www.w3.org/TR/vocab-ssn/`
[20] `https://www.w3.org/TR/vocab-ssn/`

a mixture of finite sequences [53], and the same principles apply on the level of event objects in RDF streams [33]. The problem is most prominent in the case of RDF triple streams, which is the dominant approach in current RSP models. RDF triple streams only support atomic events, consisting of a single triple, and when a more complex event object is decomposed into triples it can cause partial events to be used in event pattern matching. In RDF graph streams, an event object can be captured and streamed in its entirety, making this issue manageable in most scenarios.

2.6 Query Parameterization

Query parameterization is a process in which fixed values in queries are defined as arguments that can be set during query instantiation, which can make queries reusable. In relational databases, stored procedures are often used to create such reusable queries, and some implementations support the enforcement of parameter constraints on individual arguments. This not only helps prevent query injections but also lets users with little to no query writing experience access the information.

Similarly, many RDF stores and frameworks (e.g., OpenLink Virtuoso[21], Apache Jena[22], RDF4J[23], and Stardog[24]) support some type of parameterization of queries, typically using some vendor-specific approach. The SPARQL Inferencing Notation (SPIN)[25], however, provides mechanisms to capture reusable SPARQL queries as RDF, enabling queries to be represented and shared using standard Semantic Web formats.

2.6.1 SPIN

SPIN consists of two main components: the SPIN SPARQL Syntax and the SPIN Modeling Vocabulary. The SPIN SPARQL Syntax provides a lightweight vocabulary that lets SPARQL queries be represented as RDF, and lets queries be stored, annotated, and shared using standard Semantic Web technologies. Queries can be represented either as text or as RDF triples. The latter is a machine-readable notation of SPARQL queries. The RDF representation, for example, allows a query to reference another as a nested subquery. The RDF representation also lets queries be accessed directly as data, for example, to retrieve the query type, list the projected variables, or query for an annotated part of a query. SPIN compliant software can transition between the RDF representation and the string-based format seamlessly. Listing 2.9

[21] `https://virtuoso.openlinksw.com/`
[22] `https://jena.apache.org/`
[23] `http://rdf4j.org/`
[24] `http://www.stardog.com/`
[25] `https://www.w3.org/Submission/spin-overview/`

shows the query from Listing 2.2 represented as RDF with the target name *John* replaced with a variable.

The SPIN Modeling Vocabulary builds on top of the SPIN SPARQL Syntax and provides additional mechanisms to model reusable templates on top of queries, where the templates support parameter constraints and default values. Listing 2.10 shows a SPIN template defined based on the query in Listing 2.9. The vocabulary can also be used to define SPARQL functions and rules, which are the core mechanisms of the inferencing engine provided with the TopBraid SPIN API[26]. Hereafter we will refer to these vocabularies collectively simply as the SPIN vocabulary.

```
@prefix sp:    <http://spinrdf.org/sp#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

:query a                    sp:Select ;
       sp:resultVariables   ( _:b0 ) ;
       sp:where             ( _:b1 _:b2 ) .

_:b0 sp:varName  "person1" .

_:b1 sp:subject    [ sp:varName  "person1" ]
     sp:predicate  foaf:knows ;
     sp:object     [ sp:varName  "person2" ] .

_:b2 sp:subject    [ sp:varName  "person2" ] ;
     sp:predicate  foaf:firstName ;
     sp:object     [ sp:varName  "name" ] .
```

Listing 2.9: The query from Listing 2.2 represented using the SPIN vocabulary, with the target name *John* replaced by the variable *name*.

```
@prefix spl:   <http://spinrdf.org/spl#> .
@prefix spin:  <http://spinrdf.org/spin#> .
@prefix arg :  <http://spinrdf.org/arg#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

_:b0 a spin:SelectTemplate;
     spin:body  :query ;
     spin:constraint [ a                spl:Argument ;
                       spl:predicate    arg:name ;
                       spl:valueType    xsd:string ;
                       spl:optional     true ;
                       spl:defaultValue "John" ] .
```

Listing 2.10: A SPIN template defined on top of the query in Listing 2.9 providing *John* as the default value for the variable *name*.

[26] http://topbraid.org/spin/api/

3 Method

The goal of science is to find out the basic laws, structures, and mechanisms of nature, society, and man [40, p. 4]. Scientific methods provide the means to test hypotheses and theories in ways that are repeatable and open to criticism. Science should therefore be viewed as a critical enterprise, where theories and claims are exposed to scrutiny, and theories can be abandoned in the light of new evidence. This is what many claim differentiates scientific research from other enterprises that claim to offer truths about the world.

In simple terms, the scientific research methods involve two primary approaches: deductive methods and inductive methods [41]. Deductive methods rely on logical reasoning to derive new facts from basic axioms and premises. Given the existence of a theory we can infer conclusions from a set of premises. Inductive methods instead rely on observations to generate principles that can be generalized into rules, with the goal of forming theories that allow us to make accurate predictions. Induction therefore requires our theories to be tested against something that is observable or measurable in the real world, and cannot be based solely on *a priori* reasoning and formal proofs.

The aim of scientific research has remained relatively unchanged throughout history, but the methods that are used have changed drastically. With the scientific revolution in the 17th century, empirical methods became the primary means of testing theories and inspiring new ones [40, p. 5]. However, any theory can be "saved" by using *ad hoc* hypotheses, and in the 20th century Popper introduced the concept of *falsifiability* [41]. Falsifiability is based on the idea that all scientific theories may eventually turn out to be wrong, and

that a theory can never be *proven* correct irrespective of how much it has been corroborated in the past. To be empirically testable a theory must be open to the possibility that the result of an experiment or observation might falsify it, which results in a balance between the descriptive power of a theory and how theoretically difficult it is to falsify. By exposing theories to falsifiability, theories can progressively be found that have greater explanatory power.

Research in computer science and software engineering primarily expect to produce new "things", such as processes, methods, algorithms, or products [27]. Technology research is an applied science where theories and models are put in real-world contexts, typically involving humans. However, while technology research strives to produce new knowledge, it follows a more pragmatic approach than traditional scientific research. For example, outside of the exploration phase, only following up on a single working method may be both sufficient and rational, which would probably not be acceptable in *pure* science. Due to the relativistic nature of technology, the falsification of a product (e.g., a procedure or technique) is rarely possible, since whether it works or not is dependent on the context within which it is applied or used. In a sense, any technology that has been shown to work at some point in time cannot be disproved. This idea of functionality as a measure of success or corroboration seems to defy the very essence of the the paradigm of falsification. However, the quality of technology can be assessed across a wide range of dimensions, and it is often possible to establish some set of performance attributes that can be used to compare products designed for similar tasks. The goal of this type of research is therefore not necessarily universal truths, but rather models and theories that are sufficiently accurate to allow us to make predictions and improve the technology beyond its current state.

The research performed in this thesis has been performed primarily in the context of two research projects. Both projects were focused on improving and developing existing technology beyond its current state to handle existing and emerging challenges.

3.1 Research Projects

VALCRI and E-care@home are two large inter-disciplinary research-focused projects, and they provide different challenges with respect to the analysis of streaming data. For the sake of brevity, the descriptions below will only provide a high-level overview of the two projects.

3.1.1 VALCRI

VALCRI (*Visual Analytics for Sense-making in Criminal Intelligence Analysis*)[1] is a project funded through the European Commission's 7th Framework

[1] `http://valcri.org/`

Programme. It focuses on developing methods and technologies for criminal intelligence analysis. Crime analysts working at law enforcement agencies engage with large volumes of data that are represented in heterogeneous formats (e.g., text documents, tabular data, images, and videos). Based on this data, the crime analysts are expected to identify important data, and to be able to detect patterns with very low frequencies. Sifting through this information manually is a monumental task and hosts of tools are employed to support this work. VALCRI targets analysis of the data by enabling police analysts to visually find and explore connections. Semantic Web technologies are applied here for organizing, integrating, and querying the data. The system and methods developed in VALCRI need to comply with the legal, ethical, security, and privacy policies of different law enforcement agencies, to protect both sensitive data and the rights of individuals.

The anonymized datasets available in VALCRI are based on data from the West Midlands Police, and are made available under a non-disclosure agreement (NDA). The NDA prevents certain details of the data from being shared publicly. An increasing amount of the information that is accessible to the police is made available in a continuous fashion. For example, bank transactions, phone records, security camera footage, and email communication all generate data continuously. The purpose of the *Event Detection* work package in VALCRI is to:

> *[…] provide methods and implemented components for accurate and timely detection of highly significant, but potentially infrequent, events within large datasets or streams of data.*
>
> — VALCRI, *Annex I: Description of Work*

The main source of streaming structured data in the VALCRI dataset is a recording of anonymized Automatic Number Plate Recognition (ANPR) camera records (i.e., a recorded stream of vehicles observed by sensors). At the time of writing, this data had not yet been aligned with the background information about crimes and suspects available in the project. For example, there is no vehicle ownership information that maps registration plate numbers to individuals, even though such data is at the disposal of the police in practice.

3.1.2 Ecare@home

E-care@home[2] is a Swedish interdisciplinary distributed research project. The project focuses on the vision of supporting the independent living of the elderly in their homes through information and communication technology (ICT). Connected electronic devices in the home, including sensors and actuators, are

[2] `https://ecareathome.se/`

envisioned to support the independent living of elderly persons in the future. The research pulls in competencies from, for example, artificial intelligence, Semantic Web, and Internet of Things (IoT). One of the main challenges that the project addresses is interoperability between devices and information at different levels, where information ranges from personal health records, to procedures and policies, to expert knowledge, and IoT resources (e.g., sensors in the home). The goal is to be accomplished by performing research on selected fundamental issues in semantic interoperability, and testing the research results on a technical platform with respect to personas and scenarios derived from the real world.

The IoT devices, such as sensors and cameras, that are envisioned in ICT-assisted living, generate large volumes of data. A lot of the data is generated from physical sensors, which means that there is a high degree of uncertainty and variance, and in order to make sense of this low-level data there is a need to provide some type of pattern detection or reasoning over the data. One of the challenges with respect to time series patterns and reasoning is the need to deal with contradictory information. For example, a person may appear to be present in two separate rooms at the same time for a brief moment due to variances in the sensors.

E-care@home differs considerably from VALCRI in three main respects. Firstly, the goal is not necessarily to build a single coherent system. Secondly, data is generated continuously from deployed sensors. Finally, data can be generated through scenarios reenacted in prepared locations with deployed sensor networks. This can be used to provide "gold standards" against which the predictions of the system can be compared.

3.2 Research Process

The research process of this thesis work is outlined in Figure 3.1. Starting from a set of broadly defined research questions (RQs), a literature overview was initiated. The papers and articles included publications stretching back approximately a decade, and were collected from the most reputable Semantic Web focused conferences, including the *International Semantic Web Conference* and the *Extended Semantic Web Conference*, and journals, including the *Journal on Data Semantics*, the *Journal of Web Semantics*, the *Semantic Web Journal*, and the *International Journal on Semantic Web and Information Systems*. The keywords used were based on the initial RQs but were expanded to include the different terminologies used in literature. The literature overview excluded certain related areas such as robotics, video and image analysis, and general rule engines. Traditional approaches that do not deal with stream processing were considered as part of the background, since they describe where and why the old paradigms fall short in the context of streaming data. Based on the increased understanding of the research domain, the

RQs were refined and condensed over several iterations, and a subset of the reviewed material was used to frame the RQs in Chapter 2 (Background and Related Work).

Each of the RQs maps roughly to a single chapter in the thesis. Chapter 4 focuses on RQ1 (*How can events be modeled to support event abstraction and querying in RSP systems to assist in semantic complex event processing?*) and defines a set of requirements for representing complex events as RDF. The proposed event ontology, or event model, was evaluated by demonstrating that it covered the list of identified requirements, and by showing that it could be used to support generalization of some recurring event patterns in standard SPARQL queries. The issue of event object boundaries in RDF streams and RSP queries were presented and discussed as a related issue, and instances where workarounds can be employed for current RSP implementations were demonstrated using basic examples.

Chapter 5 focuses on RQ2 (*How can RSP queries be abstracted to support reuse and maintenance of queries?*) and on how RSP queries can be abstracted and parameterized. Integration with existing Semantic Web technologies was considered to be a priority, and RSP-SPIN was developed as a SPIN extension for representing RSP-QL queries. The model was evaluated by using it to represent the RSP Group sample queries as RDF. Compatibility with existing RSP systems is demonstrated by showing that RSP-QL queries represented as RDF can be used to generate queries for RSP languages that can be modeled using RSP-QL.

In Chapter 6 RQ3 (*What are the limitations of current RSP technologies with respect to recurring decision-making tasks in the context of semantic complex event processing?*) was approached by proposing a general event processing architecture. In this architecture, RSP systems are not viewed as being solely responsible for CEP, but rather as generators of semantically enabled event processing agents (EPAs). The architecture was implemented in three different scenarios. In the first implementation, a single RSP engine was used and the components were connected to each other directly within the same application. The implementation was applied in traffic-incident monitoring scenario, which was inspired from work on event enrichment from social media in the VALCRI project.

In the second implementation, data and queries were based directly on a VALCRI use-case. The implementation leveraged a dedicated messaging layer, which supported the enforcement of the security constraints that apply in VALCRI (e.g., context-based access control and auditing). The data in the ANPR stream was delivered at fixed intervals, with all observations collected in the past minute streamed in a single batch. This produced a stream of high-velocity bursts followed by periods of no activity. The background knowledge in VALCRI, at the time of writing, was not yet aligned with the ANPR data. This meant that there was no way of connecting vehicle observations to individuals. A vehicle ownership dataset was constructed manually after

consulting the VALCRI partner responsible for the crime data anonymization process[3]. The ownership registry was then aligned with a synthetic ANPR stream generated based on the structure of the original data.

The third and final implementation was based on a use-case from E-care@home. A messaging layer was employed, but unlike the previous implementation, the requirements of E-care@home aim primarily at scalability, expressiveness, and reasoning support. Here, the task of RSP was primarily to limit, partition, and clean the sensor event data to support reasoning that would be performed using an Answer Set Programming (ASP) temporal reasoner. The data streams were collected from deployed sensors. This work is still ongoing and we describe only the main infrastructure and some initial conclusions, with formal evaluations planned as part of future work.

Chapter 7 discusses results of the thesis and presents some plans for future work. Finally, Chapter 8 summarizes the results of the thesis.

[3]Personal communication with Rick Adderley from A E Solutions on November 9, 2016.

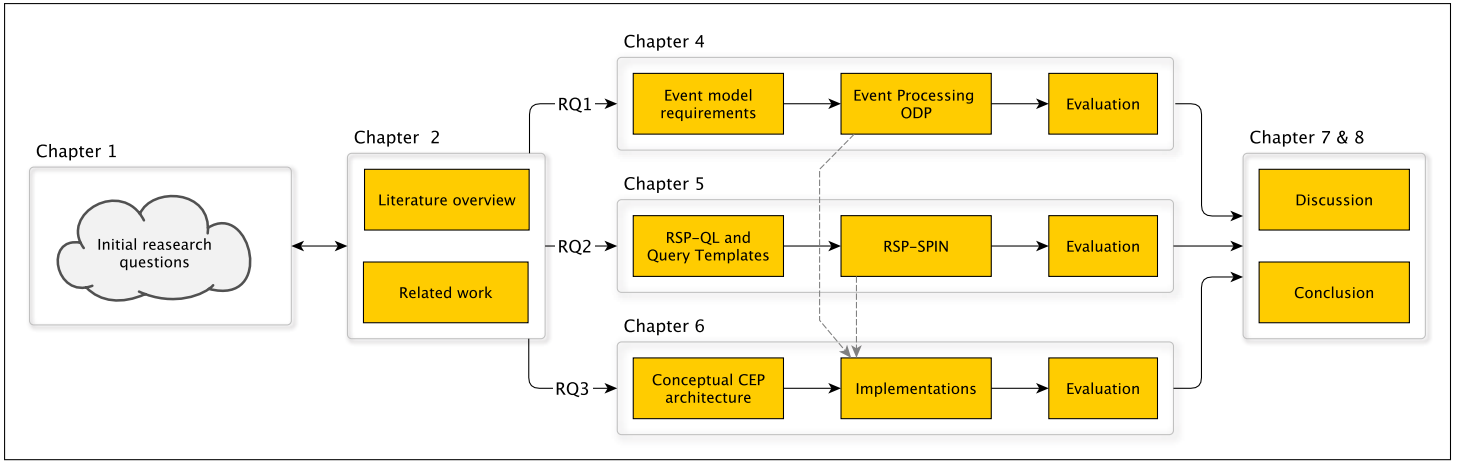


Figure 3.1: Overview of the research process. A set of initial research questions (RQs) were iteratively refined based on a literature overview. Each research question was covered in single chapter. The models created in Chapters 4 and 5 were leveraged in Chapter 6. The two final chapters discuss and summarize the results of reported in the thesis.

4 Event Modeling for Semantic Complex Event Processing

The event models currently in use on the Semantic Web have not been developed with complex event processing (CEP) as the primary objective. While some of the available models (e.g., this mentioned in Section 2.5.1) can be adapted for use in CEP scenarios, they tend to be relatively large and include many axioms that can make the models difficult to understand, learn, and extend.

In this chapter, we begin by outlining some requirements that we believe are necessary for semantic complex event processing, but which have not received sufficient attention in current event models. Based on this list, we propose a lightweight ontology that addresses the requirements and that is aligned with some of the most relevant ontologies. We then demonstrate how the new model can be used in practice.

The often-overlooked problem of event object boundaries, which is closely related to the more general issue of stream punctuation, is also discussed, and we propose a set of approaches and workarounds that could be employed to (partly) overcome this issue, using the proposed event model as a starting point. The approaches are discussed in relation to some of the current RSP implementations, as well as from the perspective of the current state of the RSP Group's standardization efforts.

4.1 Event Model Requirements

There are a number of generic functional requirements of event models, such as support for object participation, temporal duration, spatial extension, and relationships between events [51]. However, in many ways the relevant non-functional requirements are equally important and relate to such things as modularity, separation of concerns, re-usability, extensibility, formal precision, understandability, and usability. These requirements are covered to some extent by existing event models; however, none of them cover all the aspects necessary to fully represent events in the context of CEP, and in most cases the implementation of the non-functional requirements could be improved greatly. Additionally, these models are not aligned with the terminology and definitions typically used in the CEP domain, which could potentially help provide a bridge between it and the RSP domain. The list below presents an extension to a set of requirements for working with complex events identified in our previous work [47]. The list is not exhaustive in terms of event modeling requirements, but constitutes the items that we believe have not received sufficient attention in existing event models.

Req.1 *Events and event objects*: It should be possible to differentiate between *events*, which occur in the real world, and *event objects*, which are representations of such events [37]. It can be argued that all models are only representations of the events that they denote, but this separation allows us to use two parallel modeling structures for describing events. The event representations intended for a human user can be very different from the one actually present in the system.

Req.2 *Encapsulation of event objects*: The event model needs to support *composite events* that are composed of several lower-level events. Lower-level events need to be separable, and could consist of any number of additional layers of events; however, there should be a partonomy (or meronomy) relation between the composite event and its parts, where the parts are dependent on the composite event (i.e., they are encapsulated by it)[1].

Req.3 *References to triggering events*: This requirement is similar to Req.2 but refers to complex events in general. Complex events can be related to lower-level events through something that is closer to association than partonomy. This allows the triggers for each complex event to be traced back to the source, without including the referenced data in the event object structure.

[1] Lower-level events can exist in their own right, but a composite event fully encapsulates its own copy of the referenced events.

Req.4 *Multiple timestamps*: A single event can be associated with multiple timestamps, such as sensor sampling time, time of arrival in the stream, application processing time, etc. In addition, the event can be associated with time intervals denoting event validity. The event model needs to be able to support these different types of timestamps.

Req.5 *Payload support*: Event representations in CEP are often split into two parts; a *header* and a *payload* [23]. The header contains the information that is necessary for processing the event object, whereas the payload is associated with the event object without necessarily being required in the event processing. The payload structure is encapsulated and processed as part of the event object, even if no part of it is used in the actual event processing. For example, the metadata associated with a particular image file could be part of the event object header, whereas the binary file could be part of the event object payload.

Req.6 *Compatibility*: The event model should be aligned and/or compatible with the relevant ontologies that are in use today.

Req.7 *Querying ability*: The event model needs to support querying, and should not require overly long and complicated expressions to perform simple tasks, such as copying an event object.

Req.8 *Reasoning support*: The model needs to support stream reasoning. The time constraints that apply to event processing scenarios means that commitment to ontology constructs that are expensive in terms of processing should be avoided. For example, class restrictions that are computationally costly that are not strictly used for event processing should be avoided.

Req.9 *Understandability*: The model should be easily understood and learned by users.

Req.10 *Usability*: The usability aspects of ontologies and event modeling are multifaceted, but here we refer specifically to the ease with which users can apply and extend the model for a particular use-case.

4.2 Event Processing ODP

The requirements from the previous section were used to establish the desired features of a new event model. The intention was to create an autonomous ontology in the form of a *Content Ontology Design Pattern* (Content ODP). A Content ODP can be viewed as a small reusable ontology that solves a domain-oriented problem, and is directly reusable as a component or module in another ontology [25]. While this does not prevent the model from being

used in its own right, most scenarios are expected to extend it to fit their particular domain or use-case.

The model is based on the DOLCE+DnS Ultralite (DUL) ontology[2], with alignments provided for the Semantic Sensor Network (SSN) ontology [17] and the Event-Model-F ontology [51]. The SSN ontology focuses on describing the capabilities of sensors and sensor networks, and since it imports DUL it includes a total of 127 classes, 163 object properties, and approximately 2100 axioms[3]. The Event-Model-F ontology instead aims at describing events in complex socio-technical systems. It also imports DUL, making it almost as large as SSN, and includes 114 classes, 130 object properties, and approximately 1700 axioms. Both models have substantial learning curves new users.

The most important properties and classes of the proposed ontology (hereafter referred to as the Event ODP) are illustrated in Figure 4.1[4]. An event object (`EventObject`) is a subclass of `dul:InformationObject`, and the real-world event (`dul:Event`) to which it refers is referenced by the property `informationAboutEvent` (Req.1). The alignment with Event-Model-F is provided through a set of class restrictions on the `EventObject`. The `SensorOutput` class is defined as a subclass of `EventObject` and is equivalent to the `ssn:SensorOutput` class.

An event object (`EventObject`) is either a simple (`SimpleEventObject`) or a complex event object (`ComplexEventObject`). The latter refers to lower-level events using the properties `hasSubEventObject` (transitive) and `hasDirectSubEventObject` (non-transitive) (Req.3). A composite event object (`CompositeEventObject`) is a complex event capable of encapsulating its sub events through the properties `hasEventObjectPart` (transitive) and `hasEventObjectComponent` (non-transitive), which express partonomy relations (Req.2).

Multiple timestamps for event objects are represented by subproperties of `hasEventObjectTime`. Sampling time (`hasEventObjectSamplingTime`) describes the recorded time of an event in terms of the sensor's internal clock, application time (`hasEventObjectApplicationTime`) represents the timestamp assigned by the source of the data when it was transmitted, system time (`hasEventObjectSystemTime`) represents the internal clock time when the event object was received, and expiration time (`hasEventObjectExpirationTime`) expresses the time limit on the validity of the event object (Req.4).

[2] `http://www.ontologydesignpatterns.org/ont/dul/DUL.owl`

[3] The latest version of the SSN ontology (`https://www.w3.org/TR/vocab-ssn/`) modularizes the ontology further, and introduces a lightweight but self-contained core ontology called SOSA (Sensor, Observation, Sample, and Actuator) that explicitly aims at simplifying the core of the model, broadening the target audience, and increasing the potential application areas.

[4] A more detailed description of the alignments can be found in the original paper [47], the annotations of the ODP model (`http://www.ontologydesignpatterns.org/cp/owl/eventprocessing.owl`), and the pattern abstract [14]

The model supports event modeling using a header-body structure that uses separate classes for representing the *header* (`EventObjectHeader`) and the *body* (`EventObjectBody`) of an event object. This makes it possible to distinguish between the known parts of the event and the body, where the latter can include payloads for which the vocabulary may not be known (Req.5). The event header and body are referenced using `hasEventObjectHeader` and `hasEventObjectBody`. The header refers to other events using dedicated properties, and a direct sub-event is referenced using the property `refersToEventObjectConsitutent`. For event objects modeled using the header-body structure no distinction is made on the property level between references to and encapsulation of events. While this alternative way of modeling events is in line with traditional event processing systems, it may not be ideal in the context of RSP, and it was therefore designed as an optional feature. Event objects can thus be modeled directly, without explicitly distinguishing between the header and body parts.

The model was designed to support processing of event objects through standard SPARQL queries (Req.7). The model lets many query patterns be expressed in a generic manner, which facilitates query reuse and simplifies the description of event processing queries. For example, the entire RDF structure related to an event object can be matched regardless of its exact structure.

The reasoning supported by the model has been designed to be optional, and the features are not required for efficient use of the model. The alignment with DUL, SSN, and Event-Model-F was done without importing either of the ontologies to avoid the additional complexity described above. Reasoning simplifies the expression of more general patterns in queries. For example, transitive and inverse properties in the model can be used to greatly simplify the structure of queries that need to match series of connected events (Req.7 and Req.8).

The two final requirements, understandability (Req.9) and usability (Req.10), are more difficult to capture as a direct part of the design process. The use of property domain and range restrictions and high class/property ratios have been shown to make ontology patterns easier to understand and learn, while a high number of class restrictions in a model tends to have the opposite effect [31]. The external ontologies are only referenced in the model, which means that much of the quite heavy axiomatization that exists in these models is avoided. If users wish to leverage this functionality, however, they can still do so by simply importing the ontologies in question.

The ontology is around 80% smaller than the SSN or the Event-Model-F ontologies, and defines almost 90% fewer axioms. This helps make the model more understandable and easier to extend due to less semantic interaction in the model. This also has a positive effect on usability (Req.10) as well as reasoning scalability (Req.8). The Event ODP also provides examples, illustrations, and documentations as supplementary materials.

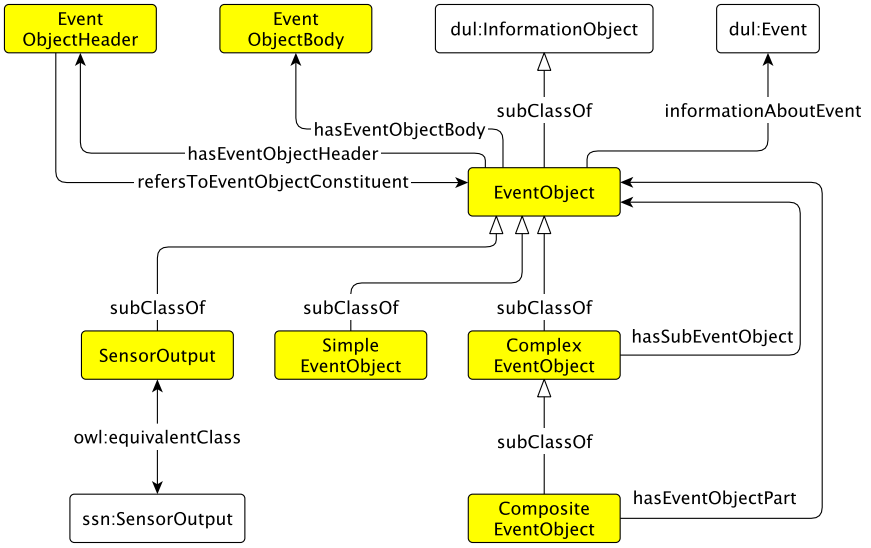


Figure 4.1: The figure shows a partial illustration of the Event ODP. It includes the topology of the classes introduced in the model and the main relationships between them. A more detailed description can be found in the original paper [47] and the pattern abstract [14].

4.2.1 Querying the Event Model

We will base the examples below on a hypothetical scenario, where a social media application is used to report traffic incidents to emergency services. Each message in the stream contains the time the event was generated (`createdAt`), a text message (`hasText`), and a location. Each message optionally contains one or more images to complement the incident report (`hasImage`). The event ontology extension for the scenario is shown in Listing 4.1, and two example messages expressed using the header-body structure are shown in Listing 4.2.

We will express the example queries using standard SPARQL 1.1. The assumption is that the streamed event objects are added to the default graph and that no reasoning is applied in any of the examples. It will be assumed that there are functions for calculating the distance between two sets of coordinates, and for calculating the duration between two timestamps[5].

In Listing 4.3, we show how all the event objects in the stream can be matched and (potentially) "copied" to a different stream. The message event is identified along with its header and body, after which the entire content of

[5]Calculations of geographic distance and datetime arithmetic is not part of standard SPARQL but are provided by several extensions.

```
@prefix : <http://example.org/ontology#>
@prefix event: <http://www.ontologydesignpatterns.org/cp/owl/
    ↪ eventprocessing.owl#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

:Message
   rdfs:subClassOf  e:SimpleEvent .
:createdAt
   rdfs:subPropertyOf  e:hasEventObjectTime ;
   rdfs:range  xsd:dateTime .
:hasText
   rdfs:subPropertyOf  e:hasEventObjectAttributeDataValue ;
   rdfs:range  xsd:string .
:hasImage
   rdfs:subPropertyOf  e:hasEventObjectAttributeDataValue ;
   rdfs:range  xsd:base64Binary .
```

Listing 4.1: Extension of the Event ODP for representing the social media messages.

```
@prefix : <http://example.org/ontology#>
@prefix ex: <http://example.org#>
@prefix e: <http://www.ontologydesignpatterns.org/cp/owl/
    ↪ eventprocessing.owl#>
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

ex:message1  a  :Message ;
   e:hasEventObjectHeader [  # event header
      :createdAt  "2017-07-01T04:30:10"^^xsd:dateTime ;
      :hasText  "A car is burning outside my house." ;
      e:hasEventLocation [
         geo:lat  "52.11"^^xsd:float ;
         geo:lon  "-1.02"^^xsd:float ]
      ] ;
   e:hasEventObjectBody [  # event body
      :hasImage  "iVBORw0KGgoAAAANS..."^^xsd:base64Binary,
                 "icBAsfw0KoikoAssS..."^^xsd:base64Binary,
                 "iTTAKfw0oMiARbssT..."^^xsd:base64Binary
   ] .

ex:message2  a  :Message ;
   e:hasEventObjectHeader [  # event header
      :createdAt  "2017-07-01T04:30:10"^^xsd:dateTime ;
      :hasText  "A vehicle is on fire next to the Montpelier Park." ;
      e:hasEventLocation [
         geo:lat  "52.10"^^xsd:float ;
         geo:lon  "-1.01"^^xsd:float ]
   ] ;
   e:hasEventObjectBody [  # event body
      :hasImage  "C67D+PAAAABGdBT..."^^xsd:base64Binary
   ] .
```

Listing 4.2: Example messages from the social media stream represented using a header-body structure.

```
PREFIX : <http://example.org/ontology#>
PREFIX e: <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>

CONSTRUCT {
   ?messageId a :Message ;
            e:hasEventObjectHeader ?header ;
            e:hasEventObjectBody ?body .

   ?headerSubject ?headerProperty ?headerObject .
   ?bodySubject ?bodyProperty ?bodyObject .
}
WHERE {
   ?messageId a :Message ;
              e:hasEventObjectHeader ?header ;
              e:hasEventObjectBody ?body .

   # match everything in the message header
   ?header (!<>)* ?headerSubject .
   ?headerSubject ?headerProperty ?headerObject .

   # match everything in the message body
   ?body (!<>)* ?bodySubject .
   ?bodySubject ?bodyProperty ?bodyObject .
}
```

Listing 4.3: Example query that matches all message events from the stream. The header and body are matched via the message id, and the content of each component is traversed using property path expressions.

the header and body is matched using property path expressions[6]. This type of query pattern is very general and can match both complex and composite events, as long as the property path only requires traversal in one direction, but reasoning could make these patterns much more expressive. For example, by inferring inverse properties in the event model a query could traverse paths that are not explicitly asserted. However, there is generally no "safe" way of traversing a property path of unknown length. If no restrictions apply, such as a depth limit or a graph context, the query could traverse and return all the available data.

In the following examples, we assume that an emergency responder wants to cluster all social media events that occur within a given radius of some location within a given time span. Listing 4.4 shows a query that generates a complex event object that groups all of the social media events that have been reported within a 1 km radius around a given location within the last hour. The generated event id and header id need to be created in a way that allows the solutions of the query to bind to the same identifiers. For example, if we generate the `eventId` in the query as a blank node in the main part of the body, each result binding could potentially return a different set of identifiers.

[6]Wild card path expressions are not supported in SPARQL, but using a negated property as shown in this query is a well known workaround.

```
PREFIX : <http://example.org/ontology#>
PREFIX e: <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX fn: <http://example.org/functions#>

CONSTRUCT {
   ?eventId a e:ComplexEventObject ;
            e:hasEventObjectHeader ?header .

   ?header e:hasEventObjectTime ?time ;
           e:refersToEventObjectConstituent ?messageId
}
WHERE {
   ?messageId a :Message ;
              e:hasEventObjectHeader ?msgHeader .
   ?msgHeader :createdAt ?datetime ;
              e:hasEventLocation ?loc .
   ?loc geo:lat ?lat ;
        geo:lon ?lon .

   # df:distance returns the distance between two coordinates in meters
   FILTER(fn:distance(?lat, ?lon, 52.10, 1.00) < 1000)
   # df:duration returns the duration between two datetimes
   FILTER(fn:duration(NOW(), ?datetime) < "PT1H"^^xsd:duration)

   { BIND(BNODE() AS ?header)
     BIND(BNODE() AS ?eventId)
     BIND(NOW() AS ?time) }
}
```

Listing 4.4: Generate a complex event with references to all message events that have been reported in the last hour within a 1 km radius of a predefined location.

This issue can be circumvented by simply grouping this part of the query, which pushes the join to the last step of the query evaluation.

Finally, in Listing 4.5 we instead construct a composite event object. The difference between this query and the former is that the new query encapsulates the matched message events, and returns them as part of the results. Like the previous query, it begins by identifying the relevant event messages, but it then also matches and returns the structure of the referenced event objects in the generic way shown in Listing 4.3.

```
PREFIX :  <http://example.org/ontology#>
PREFIX e: <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX fn: <http://example.org/functions#>

CONSTRUCT {
   ?eventId a e:CompositeEventObject ;
            e:hasEventObjectHeader ?header .

   ?header e:hasEventObjectTime ?time ;
           e:refersToEventObjectConstituent ?messageId .

   ?messageId a :Message ;
              e:hasEventObjectHeader ?msgHeader ;
              e:hasEventObjectBody ?msgBody .

   ?msgHeaderSubject ?msgHeaderProperty ?msgHeaderObject .
   ?msgBodySubject ?msgBodyProperty ?msgBodyObject .
}
WHERE {
   ?messageId a :Message ;
              e:hasEventObjectHeader ?msgHeader ;
              e:hasEventObjectBody ?msgBody .

   ?msgHeader :createdAt ?datetime ;
              e:hasEventLocation ?loc .

   ?loc geo:lat ?lat ;
        geo:lon ?lon .

   # df:distance returns the distance between two coordinates in meters
   FILTER(fn:distance(?lat, ?lon, 52.10, 1.00) < 1000)
   # df:duration returns the duration between two datetimes
   FILTER(fn:duration(NOW(), ?datetime) < "PT1H"^^xsd:duration)

   # match everything in the message header
   ?msgHeader (!<>)* ?msgHeaderSubject .
   ?msgHeaderSubject ?msgHeaderProperty ?msgHeaderObject .

   # match everything in the message body
   ?msgBody (!<>)* ?msgBodySubject .
   ?msgBodySubject ?msgBodyProperty ?msgBodyObject .

   { BIND(BNODE() AS ?header)
     BIND(BNODE() AS ?eventId)
     BIND(NOW() AS ?time) }
}
```

Listing 4.5: Generate a composite event with references to all message events that have been reported in the last hour within a 1 km radius of a predefined location. The generated event includes its own "copy" of the referenced events.

4.3 Event Object Boundaries

An aspect of RSP systems that is often overlooked is how boundaries of non-atomic events can be represented and respected in queries and streams. Querying a partial event object, for example, can result in erroneous, missed, or duplicated results [33]. The problem is not limited to how events are produced and consumed, but also occurs if event objects are split due to sliding windows. In the following section, we will discuss a number of general approaches to representing event object boundaries and discuss how these (partial) solutions can be applied in the context of semantic complex event processing. Finally, some potential workarounds that can be used when practically employing existing RSP systems for event processing will be discussed.

4.3.1 Approaches

The RSP models and implementations described in in Section 2.2.1 all assume that streamed elements are either explicitly ordered by timestamps or implicitly ordered with respect to time of arrival. Elements associated with the same timestamp should be interpreted as having *occurred* at the same time, whereas a difference in timestamp values can be used to determine the order of event occurrence. Interval-based semantics are necessary to cover the full set of operators in Allen's interval algebra (see Table 2.1); however, here we will limit our inquiries to time-point based semantics. The social media stream event from the previous section will be used to illustrate the different approaches. The solutions discussed here are by no means exhaustive, nor should they be viewed as mutually exclusive.

Punctuation

The first solution that is likely to come to mind is to directly leverage the timestamps of the streaming items to ensure that no partial event object is processed (i.e., *time-based punctuation*). In a sense, this would make this a task of the stream consumer (i.e., the event processing engine), which would have to ensure that all the event data with a given timestamp has been processed prior to executing a query over that event. However, there is no way of guaranteeing that all event data with a given timestamp has been processed until a triple with a higher timestamp value has been processed. Buffering the stream in this fashion is typically undesirable since processing is delayed and we could end up waiting indefinitely for the next item in the stream.

Therefore, an alternative is explicit stream punctuation [53], which is a technique that could be employed regardless of the exact structure of the streamed events. From an implementation standpoint there are many possible approaches, but the basic idea is to have the source of the stream explicitly denote when the event has been streamed in its entirety. Punctuation could

```
1  ex:message1 a :Message
2  ex:message1 e:hasEventObjectHeader _:b0
3  _:b0 :createdAt "2017-07-01T04:30:10"^^xsd:dateTime
4  _:b0 :hasText "My vehicle broke down on the M4... Help?!"
5  _:b0 e:hasEventLocation _:b1
6  _:b1 geo:lat "52.11"^^xsd:float
7  _:b1 geo:lon "-1.02"^^xsd:float
8  :message1 e:hasEventObjectBody _:b2
9  _:b2 :hasImage "iVBORw0KGgoAAAANS..."^^xsd:base64Binary
10 _:b2 :hasImage "icBAsfw0KoikoAssS..."^^xsd:base64Binary
11 _:b2 :hasImage "iTTAKfw0oMiARbssT..."^^xsd:base64Binary
12 [] :punctuates ex:message1
```

Listing 4.6: An event from the social media stream in Listing 4.2 represented as a stream of triples (1–12). The final triple has been injected to mark the end of the streamed event object.

be communicated as part of the event object structure itself. For example, if an event is referenced based on some unique resource identifier it could be punctuated by providing a trailing statement about that resource when all other event information has been processed. Punctuation could also be handled on a data protocol level. However, in the case of the former approach the information can be queried just like any other data and it requires no modification on the part of the query processor. Listing 4.6 illustrates one way of implementing punctuation by passing a triple in the stream that binds a blank node to the event identifier (i.e., the event object root) via the property `punctuates`.

Headers

Another possible approach is to take advantage of the fact that the structure of an event is known to the stream producer before any events are actually streamed. This means that it would be possible to associate each event with some metadata about the event (e.g., size, format, or underlying model), allowing the consumer to deduce when an event object has been processed completely. If no event objects are streamed in parallel[7], the size of the event object could be used by simply counting the number of streamed triples. Listing 4.7 illustrates the use of a header triple to indicate the number of items belonging to the next event object in the stream using the property `hasEventObjectSize`. The header information could also be communicated on a protocol level.

[7]This assumption is not dictated by the ordering requirement of RDF streams.

```
1  :message1 :hasEventObjectSize 12
2  :message1 a :Message
3  :message1 e:hasEventObjectHeader _:b0
4  _:b0 :createdAt "2017-07-01T04:30:10"^^xsd:dateTime
5  _:b0 :hasText "A car is burning outside my house."
6  _:b0 e:hasEventLocation _:b1
7  _:b1 geo:lat "52.11"^^xsd:float
8  _:b1 geo:lon "-1.02"^^xsd:float
9  :message1 e:hasEventObjectBody _:b2
10 _:b2 :hasImage "iVBORw0KGgoAAAANS..."^^xsd:base64Binary
11 _:b2 :hasImage "icBAsfw0KoikoAssS..."^^xsd:base64Binary
12 _:b2 :hasImage "iTTAKfw0oMiARbssT..."^^xsd:base64Binary
```

Listing 4.7: An event from the social media stream in Listing 4.2 represented as a list of triples (1–12). The size of the event object is injected as the first triple before the event object is streamed.

Streaming graphs

The two previous approaches have assumed that event objects are streamed as RDF triples, and the issues of event object boundaries are largely a result of the fragmentation of the event when it is transported and consumed in this format. Streaming an event object as a single item in the stream, however, allows the event to be processed directly. This approach can be extended further by supporting streaming of named graphs, which would also group the event on the consumer side and let it freely describe its own context without causing interference with other event data. Listing 4.8 illustrates the structure of an event object represented as a time-annotated named graph, as described in the current RSP-QL draft, using the TriG syntax[8].

4.3.2 Event Object Boundaries in RSP

Existing RSP systems vary in expressiveness and this greatly affects the ways in which event object boundaries can be supported, both implicitly and explicitly. We will here briefly describe the approaches from the previous section in relation to existing RSP models.

Punctuation: Time-based punctuation is only possible if the event object timestamps can be accessed from within queries. C-SPARQL, CQELS, and EP-SPARQL use timestamped triples internally, but only C-SPARQL lets them be accessed in queries. Timestamps could also be provided as an explicit part of the data on the triple level, but this would essentially require the use of reification, unless the RDF representation is extended for temporal information [30]. Reification is a strategy that allows statements about a triples to be made by binding the parts to a new resource, which can then be annotated. With reification, associating a single RDF triple with an explicit

[8] https://www.w3.org/TR/trig/

```
@prefix :     <http://example.org/ontology#>
@prefix ex:   <http://example.org#>
@prefix e:    <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
@prefix geo:  <http://www.w3.org/2003/01/geo/wgs84_pos#>
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>
@prefix prov: <https://www.w3.org/TR/prov-o/#>
ex:message1Graph {
   ex:message1
      a :Message ;
      e:hasEventObjectHeader [
         :createdAt "2017-07-01T04:30:10"^^xsd:dateTime ;
         :hasText "A car is burning outside my house." ;
         e:hasEventLocation [ geo:lat "52.11"^^xsd:float ;
                              geo:lon "-1.02"^^xsd:float ]
      ] ;
      e:hasEventObjectBody [
         :hasImage  "iVBORw0KGgoAAAANS..."^^xsd:base64Binary,
                    "icBAsfw0KoikoAssS..."^^xsd:base64Binary,
                    "iTTAKfw0oMiARbssT..."^^xsd:base64Binary
      ]
}
ex:message1Graph prov:generatedAtTime "2017-07-01T04:30:10"^^xsd:dateTime
```

Listing 4.8: An event from the social media stream in Listing 4.2 represented as a timestamped graph, where the entire event is processed as a single item in the stream. The time annotation on the graph uses a property from the PROV-O ontology (`https://www.w3.org/TR/prov-o/`).

timestamp would require (at least) four additional statements per triple in the stream. This makes the approach infeasible in a streaming context, and still does not provide an efficient way of determining whether or not the entire event object has been streamed. All the current RSP implementations could take advantage of explicit stream punctuation, provided that each event object is identified using a single identifier.

Header: The simple header approach would not be a scalable approach in any of the current RSP models. For example, while the size of an event object could be calculated using aggregates and compared with the given reference value, it would not only complicate queries greatly but would also take a heavy toll on performance. Taken as a preprocessing step, however, the approach could be used to transfer events from the producer and the RSP engine could apply another strategy for the internal ingestion of the event.

Streaming graphs: Streaming graphs would not solve the internal event object boundary issue of the RSP models that support only streams of RDF triples. If graphs are streamed the event objects still need to be decomposed into triples on the consumer side. INSTANS, which supports SPARQL 1.1 Update, is currently the only approach that could potentially leverage this type of event object punctuation.

4.3.3 Examples

Using existing RSP systems for complex event processing is not very streamlined, and many operations cannot be supported by all RSP implementations. INSTANS has been shown to support expressive complex event processing [48], and is the only system that allows data to be persisted to support stateful operations. However, INSTANS does not support windows over streams natively, and removing outdated events to reduce the volume of data to be processed is not a trivial task.

Throughout the rest of this section, we will illustrate how the queries from Section 4.2 can be expressed in C-SPARQL, which is one of the RSP languages that currently has the best support for SPARQL 1.1. Modeling the queries using CQELS would be similar, but in the current CQELS implementation it would not be generalizable to the same extent due to how the stream-to-relation operator is implemented. A query result in CQELS is returned as soon as a match is made, meaning that either additional post-processing would be required, or that the entire event object structure would have to be specified in the query (i.e., the queries would not be generalizable to unknown parts of the event). Expressing any general-purpose pattern using EP-SPARQL, on the other hand, would be even more difficult since it is limited to the constructs available in SPARQL 1.0. Explicit punctuation intuitively seems to be the most efficient solution, and in the triple-based approach we simply need to match an extra triple in the query's where clause. We will use this approach in the following examples, and we will assume that we have at our disposal a function for calculating the distance between two locations[9].

Copying the event objects from the social media stream can be done in much the same way as for standard SPARQL, and the query (see Listing 4.9) only requires a small modification to fit the syntax of C-SPARQL. With a fixed window step size, there needs to be some overlap between any two consecutive windows to ensure that no event is accidentally split when the content of a window changes.

Creating complex events referencing the messages in the social media stream is also very similar to the way it is done in standard SPARQL, as seen in Listing 4.10. It is not necessary to reference the punctuation triple in the query since we are only interested in getting the references to the event objects (i.e., there are no unspecified parts of the event to consider). The size of the stream window is indirectly specified in the original query, which limits the events of interest to the past hour.

Finally, the composite event object query uses the same general pattern as the SPARQL query in Listing 4.5. For encapsulation, we need to make sure that the punctuation triple is present for each matched message event.

[9]While the author is unaware of any similar C-SPARQL extension, the C-SPARQL engine provides several extensions that have been implemented in this way.

```
REGISTER STREAM copy AS
PREFIX :    <http://example.org/ontology#>
PREFIX e:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT {
   ?messageId a :Message ;
              e:hasEventObjectHeader ?header ;
              e:hasEventObjectBody ?body .
   ?headerSubject ?headerProperty ?headerObject .
   ?bodySubject ?bodyProperty ?bodyObject .
}
FROM STREAM <http://example.org#stream> [RANGE 2s STEP 1s]
WHERE {
   [] :punctuates ?messageId . # check for punctuation
   ?messageId a :Message ;
              e:hasEventObjectHeader ?header ;
              e:hasEventObjectBody ?body .
   ?header (!<>)* ?headerSubject .
   ?headerSubject ?headerProperty ?headerObject .
   ?body (!<>)* ?bodySubject .
   ?bodySubject ?bodyProperty ?bodyObject .
}
```

Listing 4.9: Copy all message events in the stream using C-SPARQL. The query is almost identical to the SPARQL query shown in Listing 4.3 but ensures that each message event is punctuated.

```
REGISTER STREAM complex AS
PREFIX :    <http://example.org/ontology#>
PREFIX e:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT {
   ?eventId a e:ComplexEventObject ;
            e:hasEventObjectHeader ?header .
   ?header e:hasEventObjectTime ?time ;
           e:refersToEventObjectConstituent ?messageId .
}
FROM STREAM <http://example.org#stream> [RANGE 60m STEP 10s]
WHERE {
   ?messageId a :Message ;
              e:hasEventObjectHeader ?msgHeader .
   ?msgHeader e:hasEventLocation ?loc .
   ?loc geo:lat ?lat ;
        geo:lon ?lon .

   # df:distance returns the distance between two coordinates in meters
   FILTER(fn:distance(?lat, ?lon, 52.10, 1.00) < 1000)

   { BIND(BNODE() AS ?header)
     BIND(BNODE() AS ?eventId)
     BIND(NOW() AS ?time) }
}
```

Listing 4.10: Generate a complex event with references to all message events that have been reported in the past hour within a 1 km radius of a predefined location using a C-SPARQL query. The query is very similar to the query in Listing 4.4.

```
REGISTER STREAM composite AS
PREFIX :    <http://example.org/ontology#>
PREFIX e:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
REGISTER STREAM composite AS
CONSTRUCT {
   ?eventId a e:CompositeEventObject ;
            e:hasEventObjectHeader ?header .
   ?header e:hasEventObjectTime ?time ;
           e:refersToEventObjectConstituent ?messageId .
   ?messageId a :Message ;
              e:hasEventObjectHeader ?msgHeader ;
              e:hasEventObjectBody ?msgBody .

   ?msgHeaderSubject ?msgHeaderProperty ?msgHeaderObject .
   ?msgBodySubject ?msgBodyProperty ?msgBodyObject .
}
FROM STREAM <http://example.org#stream> [RANGE 60m STEP 10s]
WHERE {
   [] :punctuates ?messageId . # check for punctuation
   ?messageId a :Message ;
              e:hasEventObjectHeader ?msgHeader ;
              e:hasEventObjectBody ?msgBody .
   ?msgHeader e:hasEventLocation ?loc .
   ?loc geo:lat ?lat ;
        geo:lon ?lon .

   # df:distance returns the distance between two coordinates in meters
   FILTER(fn:distance(?lat, ?lon, 52.10, 1.00) < 1000)

   # match everything in the message header
   ?msgHeader (!<>)* ?msgHeaderSubject .
   ?msgHeaderSubject ?msgHeaderProperty ?msgHeaderObject .

   # match everything in the message body
   ?msgBody (!<>)* ?msgBodySubject .
   ?msgBodySubject ?msgBodyProperty ?msgBodyObject .

   { BIND(BNODE() AS ?header)
     BIND(BNODE() AS ?eventId)
     BIND(NOW() AS ?time) }
}
```

Listing 4.11: Generate a composite event with references to all messages that have been reported in the past hour within a 1 km radius of a predefined location using a C-SPARQL query. The generated event object includes its own "copy" of the referenced events. The query is very similar to the query in Listing 4.5 but ensures that each message event is punctuated.

4.3.4 Event Object Boundaries in RSP-QL

The RSP Group has defined the current version of RSP-QL with the event object boundary issue taken into consideration. In RSP-QL, RDF streams are ordered sequences of time-annotated graphs, and the query language itself can be used to declaratively generate new RDF streams.

From the perspective of CEP, this new model has four major implications. Firstly, an event object can constitute a single item in the stream, which solves most of the issues related to event object boundaries, if an appropriate event model is used (e.g., one that supports event object encapsulation). Secondly, the stream model lets time annotations be defined for sets of RDF statements, rather than requiring costly reification. The relevant time properties for time-window processing can thereby be accessed without relying on knowledge about the underlying event model. Thirdly, events can define their own contexts. This means that the events can provide assumptions about the world, and be used to support certain types of abductive reasoning, where the hypotheses used to explain the observations in the event do not need to interfere with other events objects. For example, we could generate a derived event where we explain the similarity between two messages by hypothesizing that they have been written by the same person. Finally, the ability to declaratively create event streams can support event processing that takes place in multiple steps, which is one of the requirements of CEP.

4.4 Summary and Discussion

We have presented an event model that covers a range of requirements that we believe have not received sufficient attention in previous work. The Event ODP has been designed as a pattern that can be extended to support different use-cases, but was created with querying ability in mind. The event queries can often be generalized and reused in different contexts with little modification to the overall query.

The features of the model were partly illustrated by showing how it could be used to generate both complex and composite event objects using standard SPARQL. The queries demonstrate that event patterns can be generalized over the more complex event header-body pattern, and that even composite event objects that encapsulate lower-level events can be formulated in a general way that does not require the entire event object structure to be specified in the query.

Examples were provided showing how, by injecting a punctuation triple, event object boundaries can be represented in existing triple-based RSP systems, even when the queries become quite complex. The method used in the examples connected the punctuation triple to the event object root. Without a similar handle on the event object the punctuation triple approach is only reliable if there is no parallel processing of events in the stream.

Reliably processing complex event objects using existing RSP technology is not trivial. The strategies proposed above offer some suggestions on how the issue of event object boundaries can be approached; however, not all RSP engines and event models are subject to the same issues. There are some additional potential problems with systems that support only triple streams. For example, it is not clear how triple streams support anonymous resources (i.e., blank nodes). Internally, this is naturally an implementation issue, but it is not clear how (or if) the identity of blank nodes is supported in stream. The approach currently proposed as part of the RSP *Abstract Syntax and Semantics Document*[10] is that each RDF stream defines an *RDF surface*, where blank node mappings are shared between all graphs in the stream. This seems to be an implicit assumption in most RSP implementations.

Although some details about RSP-QL have yet to be fully defined, the model extends the existing RSP models in terms of expressiveness. In the RSP Group discussions and the *RSP Requirements Design Document*[11] the concept of event immutability has been partially explored, and adopting this general assumption allows a system to be made more predictable. This is an implicit assumption in the event model presented here, since complex events only provide references to the related events.

Two of the requirements that were defined could not fully be captured in the design process alone; namely, understandability (Req.9) and usability (Req.10). Several design choices of the model were made with these requirements in mind, and the examples demonstrate how the event model can be both extended and queried. However, real-world use-cases and actual end-users are required for a more detailed evaluation. In Chapter 6, we provide some work in this direction by extending the model to represent events in traffic-incident monitoring and criminal-activity monitoring.

[10] `https://github.com/streamreasoning/RSP-QL`
[11] Ibid.

5 Query Abstraction

Generalizing queries through the use of parameterization, where some fields can be specified through user input, is a common way of making queries reusable. For example, monitoring a specific vehicle based on a stream of observations from vehicle sensors could be formulated as a parameterized query, where the registration plate number could be specified each time the query is instantiated. When implemented in a service layer, prepared queries can support data access control, protect against query injections, separate business logic from application code, reduce parsing and query execution time, and significantly lower the threshold for users who have no detailed knowledge of the underlying data models or query language.

Parameterization of standard SPARQL queries is supported by many RDF stores and frameworks (e.g., OpenLink Virtuoso[1], Apache Jena[2], RDF4J[3], and Stardog[4]). Parameterization of RSP queries, however, has been limited to simple string-based substitution. General-purpose template managers, such as Mustache[5] or FreeMarker[6], could be used to define templates. However, neither of these frameworks can take advantage of background data or on-

[1] `https://virtuoso.openlinksw.com/`
[2] `https://jena.apache.org/`
[3] `http://rdf4j.org/`
[4] `http://www.stardog.com/`
[5] `https://mustache.github.io/`
[6] `http://freemarker.org/`

tologies represented on the Semantic Web, and the support for defining and enforcing parameter constraints is typically very limited.

In this chapter, an extension to SPIN is proposed to support the current version of RSP-QL (see Section 2.4). SPIN provides a vocabulary for representing SPARQL queries as RDF, and lets query templates be modeled on top of these (see Section 2.6.1). The extended vocabulary provides a query language agnostic representation of RSP queries, and supports the SPIN metamodeling features for defining templates over these. The language can be used to represent queries expressed in all the RSP query languages captured by the semantics of RSP-QL [21].

5.1 Extending SPIN for RSP-QL

The extension to the SPIN vocabulary presented in this chapter lets RSP queries be encapsulated as RDF, without losing compatibility with the standard version of SPIN. The model, hereafter referred to as RSP-SPIN, adds classes and properties to the SPIN vocabulary to capture the SPARQL extensions provided in RSP-QL. The current version of the RSP-QL abstract syntax and semantics, the discussions leading up to this draft, and the RSP-QL sample queries provided by the RSP Group have been used as a reference for the RSP-QL language. The RSP-QL sample queries were inspired by the DEBS Grand Challenge 2015[7], and were formulated against a stream of taxi trips. Listing 5.1 shows an example from the taxi stream represented as RDF.

We will use the example query in Listing 5.2 as a reference to describe the various constructs used in RSP-SPIN throughout the following sections. The query demonstrates the use of a time-based window (defined using a range), a time-based window in the past (defined using upper and lower time bounds), and a count-based window (defined in terms of number of streamed elements). It generates a stream consisting of all taxi trips that exist in all three windows simultaneously, and returns only the new mappings (IStream). This example query is quite trivial but is used purely for illustrative purposes.

[7] `http://www.debs2015.org/call-grand-challenge.html`

```
@prefix : <http://example.org#> .
@prefix debs: <http://debs2015.org/onto#> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
:t1 {
 :trip1 debs:dropoff     :dropoff1 ;
        debs:pickup      :pickup1 ;
        debs:taxi        :taxi25 ;
        debs:license     :driver2 ;
        debs:duration    "PT11M"^^xsd:duration ;
        debs:distance    "1.5"^^xsd:double ;
        debs:paymentType "CSH" ;
        debs:fare        "3.50"^^xsd:double ;
        debs:surcharge   "0.50"^^xsd:double ;
        debs:tax         "0.50"^^xsd:double ;
        debs:tip         "0.00"^^xsd:double ;
        debs:tolls       "0.00"^^xsd:double .

 :pickup1 prov:startedAtTime "2013-01-01T00:00:00"^^xsd:dateTime ;
          prov:atLocation [ geo:lat "40.716976"^^xsd:double ;
                            geo:long "-73.956528"^^xsd:double ] .
 :dropoff1 prov:startedAtTime "2013-01-01T00:11:00"^^xsd:dateTime ;
         prov:atLocation [ geo:lat "41.716976"^^xsd:double ;
                           geo:long "-73.956528"^^xsd:double ] .
}
:t1 prov:atTime "2013-01-01T00:11:01"^^xsd:dateTime
```

Listing 5.1: Example from the DEBS taxi trip stream represented as RDF. The stream outputs a single timestamped graph every time a taxi trip is completed.

```
PREFIX :     <http://example.org#>
PREFIX debs: <http://debs2015.org/onto#>
PREFIX prov: <http://www.w3.org/ns/prov#>

REGISTER STREAM :my-stream AS

CONSTRUCT ISTREAM {
  GRAPH ?g {
    ?s ?p ?o .
  }
  ?g prov:atTime ?time .
}
FROM NAMED WINDOW :w1 ON :trips [ITEM 1000 STEP 1000]
FROM NAMED WINDOW :w2 ON :trips [RANGE PT2H STEP PT1M]
FROM NAMED WINDOW :w3 ON :trips [FROM NOW-PT3H TO NOW-PT1H STEP PT1M]
WHERE {
  WINDOW :w1 {
    GRAPH ?g { ?s ?p ?o }
    ?g prov:atTime ?time
  }
  WINDOW :w2 { ?g prov:atTime ?time }
  WINDOW :w3 { ?g prov:atTime ?time }
}
```

Listing 5.2: The query generates a stream consisting of all taxi rides that are in all three windows simultaneously.

5.1.1 Register Clause

RSP-QL lets new streams be defined declaratively from continuous queries using the *register clause*. It is assumed here that no distinction is made between select and construct queries, which is the case for C-SPARQL, since neither of the RSP-QL sample queries differentiates between them. The example query defines a construct query and specifies a resource identifier on which the resulting stream will be made available. In RSP-SPIN, the value of the output-stream identifier is represented by the property `hasOutputStream`. Listing 5.3 shows the relevant RDF excerpt.

```
@prefix :   <http://w3id.org/rsp/spin#> .
@prefix sp: <http://spinrdf.org/sp#> .
@prefix ex: <http://example.org#> .

ex:my-query a                 sp:Construct ;  # Query type
            :hasOutputStream  ex:my-stream .  # Name of resulting stream
```

Listing 5.3: Representation of the register clause in RSP-SPIN.

5.1.2 Relation-to-stream Operator

The *relation-to-stream operator* indicates the semantics used in the production of query results. In RSP-SPIN, the operator is treated as a solution modifier and it is therefore attached to the query root. The value of the property `hasOutputStreamOperator` specifies that the query returns either `Istream`, `Rstream`, or `Dstream`, which are all defined as subclasses of `OutputStreamOperator`. The relevant portion of the RSP-SPIN representation is shown in Listing 5.4.

```
@prefix :   <http://w3id.org/rsp/spin#> .
@prefix ex: <http://example.org#> .

ex:my-query :hasOutputStreamOperator :Istream . # Istream/Rstream/Dstream
```

Listing 5.4: Representation of the relation-to-stream operator in RSP-SPIN.

5.1.3 Named Graphs in Construct

RSP-QL supports named graphs in the output of construct queries, which is not supported in standard SPARQL. To handle this in SPIN we added support for the `NamedGraph` block in the output portion of construct queries. The SPIN vocabulary itself does not implement any constraints against this usage; however, since SPIN is based on SPARQL 1.1 it is regarded as an adaptation. Listing 5.5 shows the relevant portion of the example query represented using standard SPIN.

```
@prefix sp:   <http://spinrdf.org/sp#> .
@prefix ex:   <http://example.org#> .
@prefix prov: <http://www.w3.org/ns/prov#> .

ex:my-query  sp:templates  ( _:b0 _:b1 ) .   # query result
_:b0  a                  sp:NamedGraph ;     # _:b0 is a named graph
      sp:elements        ( _:b2 ) ;
      sp:graphNameNode   [ sp:varName "g" ] .

_:b1  sp:subject     [ sp:varName "g" ] ;     # _:b1 is a triple
      sp:predicate   prov:atTime ;
      sp:object      [ sp:varName "time" ] .

_:b2  sp:subject     [ sp:varName "s" ] ;     # _:b2 is a triple
      sp:predicate   [ sp:varName "p" ] ;
      sp:object      [ sp:varName "o" ] .
```

Listing 5.5: Representation of named graphs in construct results using SPIN.

5.1.4 Named Windows

The size and step of count-based windows are defined in terms of the number of streamed elements, and an optional slide interval. The count-based window is referred to as a `PhysicalWindow` in RSP-SPIN, with the size and step indicated by integers linked from the properties `physicalRange` and `physicalStep`.

The size of a time-based window in RSP-QL can be specified using either a time range, or an interval with upper and a lower time bounds. The former format is subsumed by the latter in terms of expressiveness, since any window based on a time range can be expressed using upper and a lower time bounds. In accordance with the current RSP-QL version, RSP-SPIN supports both definitions. The two window types are represented by `LogicalWindow` and `LogicalPastWindow`[8]. The size of a `LogicalWindow` is denoted using the property `logicalRange`. The upper and lower time bounds of a `LogicalPastWindow` are defined using the properties `from` and `to`, where values indicate the offset from the current time. Both window types support an optional step parameter expressed as a duration and represented with the property `logicalStep`.

The window types are defined as subclasses of `Window`, and are linked to a stream and a window identifier via the properties `streamUri` and `windowUri`. The three types of windows from the example query are represented using RSP-SPIN in Listing 5.6.

[8]The rationale behind the name is that interval-based windows can be offset relative to the current time to define windows in the past.

```
@prefix :     <http://w3id.org/rsp/spin#> .
@prefix ex:   <http://example.org#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

ex:my-query  :fromNamedWindow  [ a                :PhysicalWindow ;
                                 :physicalRange   "1000"^^xsd:int ;
                                 :physicalStep    "1000"^^xsd:int ;
                                 :streamUri       ex:trips ;
                                 :windowUri       ex:w1 ] ;

             :fromNamedWindow  [ a                :LogicalWindow ;
                                 :logicalRange    "PT2H"^^xsd:duration ;
                                 :logicalStep     "PT1M"^^xsd:duration ;
                                 :streamUri       ex:trips ;
                                 :windowUri       ex:w2 ] ;

             :fromNamedWindow  [ a                :LogicalPastWindow ;
                                 :from            "PT3H"^^xsd:duration ;
                                 :to              "PT1H"^^xsd:duration ;
                                 :logicalStep     "PT1M"^^xsd:duration ;
                                 :streamUri       ex:trips ;
                                 :windowUri       ex:w3 ] .
```

Listing 5.6: The three windows from the example query represented using RSP-SPIN.

5.1.5 Named Window Block

Named window blocks in RSP-QL queries are analogous in structure to named graph blocks. Therefore, the RSP-SPIN extension for referencing a named window in a query is very similar to how named graphs are described in SPIN. The class `NamedWindow` indicates the window block, and the property `windowNameNode` links it to a window identifier. Listing 5.7 shows how the named window blocks from the example query are represented using RSP-SPIN.

```
@prefix :     <http://w3id.org/rsp/spin#> .
@prefix sp:   <http://spinrdf.org/sp#> .
@prefix ex:   <http://example.org#> .

ex:my-query sp:where  ( _:b3 _:b4 _:b5 ) ;
_:b3  a               :NamedWindow ;
      :windowNameNode ex:w1 ;
      sp:elements     ( _:b6 _:b7 ) .
_:b4  a               :NamedWindow ;
      :windowNameNode ex:w2 ;
      sp:elements     ( _:b8 ) .
_:b5  a               :NamedWindow ;
      :windowNameNode ex:w3 ;
      sp:elements     ( _:b9 ) .
```

Listing 5.7: The window blocks in the example query represented using RSP-SPIN. The graph patterns represented by `sp:elements` have been excluded for brevity.

5.2 Extending the SPIN API

We extended the SPIN API to be able to support queries expressed in RSP-QL using the RSP-SPIN vocabulary. The implementation was used to practically evaluate the vocabulary but was also intended to be a resource that would promote the use of RSP-SPIN within the RSP community. The code has been made available as open source[9] under the MIT license, and the implementation is fully compliant with standard SPIN. The full vocabulary is included as part of Appendix A.

Although the design of the extended vocabulary was made with RSP-QL as the primary reference, no commitment was made with respect to the actual underlying RSP language. The syntax can support any RSP syntax that can be modeled using RSP-QL, and the list of currently supported RSP languages includes C-SPARQL, CQELS-QL, and $\text{SPARQL}_{\text{stream}}$. Queries expressed using INSTANS are also supported, since they can be represented using standard SPIN.

5.2.1 RSP-QL Parser

The SPIN API builds extensively on the Apache Jena framework[10]. The API internally maps a SPIN structure to an ARQ query. In order to extend the API for RSP-SPIN it was necessary to extend the structure of the query classes and to provide a parser for the RSP-QL syntax. The goal of this work was to promote re-usability of queries, and the parser therefore allows the values of output streams, stream identifiers, and window parameters to be represented as variables. With respect to named windows identifiers and the relation-to-stream operator, no scenario in which parameterization would be necessary could be identified, so the implementation does not support parameterization of these.

From the RSP Group discussions, it is not clear how (or if) windows and streaming queries can be nested. The parser was implemented under the assumption that windows can be referenced in non-streaming sub-queries. The mandatory features and constructs of RSP-QL queries have not been specified, and the parser therefore assumes that all the new RSP-QL constructs are optional. For example, if no relation-to-stream operator was defined we would assume that the query falls back to the default configuration of the execution environment.

Following the implementation of the parser the ARQ classes were modified to support the new constructs. Finally, the SPIN API was extended to be able to serialize the query syntax to and from the RSP-SPIN representation.

[9] `https://w3id.org/rsp/spin`
[10] `https://jena.apache.org/`

5.2.2 Serializers for RSP-SPIN

Transitioning between the RDF representation and query string is an essential feature of SPIN. RSP-QL captures the semantics of CQELS-QL, C-SPARQL, and $\text{SPARQL}_{\text{stream}}$ [21], and since RSP-SPIN is query language agnostic it can be used to serialize a query into either of these languages.

As a proof of concept, RSP-SPIN serializers were created for the previously mentioned query languages. Features present in RSP-QL but not in the target language were handled by using *exclusion* or *simplification*. The register clause is only supported by C-SPARQL and was excluded for CQELS-QL and $\text{SPARQL}_{\text{stream}}$. C-SPARQL only supports basic strings as the name of the resulting stream, and URIs therefore had to be simplified accordingly. C-SPARQL and $\text{SPARQL}_{\text{stream}}$ do not support multiple windows over the same stream, which required the employment of a simplification strategy. The upper and lower bounds of a window that would include all the windows over the stream were established and together with the smallest step size present a new window was defined. Combinations of count-based and time-based windows were not permitted. Named graphs in windows and in the output of construct queries were not supported by either of the RSP languages in question, so the implementation instead collapses the named graphs into the default graph. $\text{SPARQL}_{\text{stream}}$ only supports time-based windows, which required windows defined using a range to be translated into this format. Windows defined in the past (i.e., with an upper bound that is offset from the current time) are only supported by $\text{SPARQL}_{\text{stream}}$, therefore C-SPARQL and CQELS-QL instead used a range-based window defined instead. For many queries, these translations should be regarded as approximations, since the simplifications do not always capture the semantic differences between the languages.

5.3 Evaluation

The modeling capabilities of SPIN with respect to templates have not been affected by the proposed extension. For this reason, the evaluation focused on query representation only, and the reader is referred to the examples provided as part of the RSP-SPIN API[11] and the SPIN API[12] for details on template usage. In this section, we first show that the vocabulary can be used to represent the sample queries from the RSP Group. We then show that each of the CSRBench [20] queries can be expressed using a single RSP-QL query and that the underlying RSP-SPIN representation can be used to recreate the equivalent benchmark queries for C-SPARQL, CQELS-QL, and $\text{SPARQL}_{\text{stream}}$.

[11] `https://w3id.org/rsp/spin/`
[12] `http://topbraid.org/spin/api/`

5.3.1 Modeling RSP-QL Queries

The 8 sample queries[13] from the RSP Group were used to validate the expressiveness of RSP-SPIN, as well as the functionality of the RSP-SPIN API. All queries contained minor syntax errors (e.g., missing prefix declarations, misspellings, and unbalanced parentheses). For other query inconsistencies, the intention of the query was typically clearly communicated by the accompanying text description and the query could be adjusted or repaired accordingly. For the purposes of the evaluation, all of these errors were corrected prior to parsing. An additional query was included to illustrate query features that have been discussed in the RSP Group but were not present in any of the sample queries; namely, count-based windows over streams, and named graphs in window blocks and construct results. The added query, shown Listing 5.8, filters the taxi trip stream to include only trips of more than 2 kilometers into a new stream.

```
PREFIX :      <http://debs2015.org/streams/>
PREFIX ex:    <http://example.org#>
PREFIX debs: <http://debs2015.org/onto#>

REGISTER STREAM ex:longTrips AS
CONSTRUCT ISTREAM {
  GRAPH ?g {
    ?s1 ?p1 ?o1 .
  }
  ?g ?p2 ?o2 .
}
FROM NAMED WINDOW :w ON :trips [ITEM 1]  # Window contains one item
WHERE {
  WINDOW :w {
    GRAPH ?g {
      ?ride debs:distance ?distance .    # Find the distance
      FILTER (?distance > 2)             # Filter by distance
      ?s1 ?p1 ?o1 .                      # Match all data in the graph
    }
    ?g ?p2 ?o2 .                         # Match metadata about the graph
  }
}
```

Listing 5.8: RSP-QL query filtering a stream of named graphs based on the distance of the trip.

All RSP-QL queries except one were successfully parsed into RSP-SPIN and serialized back into an equivalent query string. The query that failed could not be parsed due to a nested construct query. This feature was not included as part of RSP-QL, and is not supported in SPARQL 1.1. The full set of RSP-QL queries used in the evaluation are listed in Appendix A.

[13] https://github.com/streamreasoning/RSP-QL

5.3.2 Modeling CSRBench Queries

CSRBench [20] was developed to allow the correctness of different RSP implementations to be evaluated based on their operational semantics. The benchmark contains seven carefully handcrafted queries expressed using C-SPARQL, CQELS-QL, and $\text{SPARQL}_{\text{stream}}$, which represent the equivalent queries for each different RSP query language.

We formulated the queries using RSP-QL and parsed them into RDF using the RSP-SPIN API. The RDF representations were then serialized into C-SPARQL, CQELS-QL, and $\text{SPARQL}_{\text{stream}}$ queries. The generated query strings were manually compared to the reference queries from the benchmark, and for each of the queries the generated strings were equivalent, though not necessarily identical, to the handcrafted queries. Listing 5.9 shows the first benchmark query expressed as RSP-QL, and Listings 5.10– 5.11 show the queries that were generated for each of the three languages. All CSRBench queries expressed as RSP-QL can be found in Appendix A.

```
PREFIX :        <http://ex.org/streams/>
PREFIX om-owl:  <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

REGISTER STREAM :query1 AS
SELECT ?sensor ?obs
FROM NAMED WINDOW :w ON :test [RANGE PT10S STEP PT10S]
WHERE {
  WINDOW :w {
    ?obs om-owl:observedProperty weather:_AirTemperature ;
         om-owl:procedure ?sensor ;
         om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER (?value > 80)
}
```

Listing 5.9: The first CSRBench query expressed using RSP-QL.

```
REGISTER QUERY query1 AS

PREFIX :        <http://ex.org/streams/>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX om-owl:  <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>

SELECT  ?sensor ?obs
FROM STREAM <http://ex.org/streams/test> [RANGE 10s STEP 10s]
WHERE {
  ?obs  om-owl:observedProperty  weather:_AirTemperature ;
        om-owl:procedure  ?sensor ;
        om-owl:result  ?res .
  ?res  om-owl:floatValue  ?value .
  FILTER (?value > 80)
}
```

Listing 5.10: The query in Listing 5.9 translated into C-SPARQL using the RSP-SPIN API.

```
PREFIX :        <http://ex.org/streams/>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX om-owl:  <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>

SELECT  ?sensor ?obs
FROM STREAM <http://ex.org/streams/test> [NOW-10 S SLIDE 10 S]
WHERE {
  ?obs  om-owl:observedProperty  weather:_AirTemperature ;
        om-owl:procedure  ?sensor ;
        om-owl:result  ?res .
  ?res  om-owl:floatValue  ?value .
  FILTER (?value > 80)
}
```

Listing 5.11: The query in Listing 5.9 translated into $\text{SPARQL}_{\text{stream}}$ via the RSP-SPIN API.

```
PREFIX :        <http://ex.org/streams/>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX om-owl:  <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>

SELECT  ?sensor ?obs
WHERE {
  STREAM <http://ex.org/streams/test> [RANGE 10s SLIDE 10s] {
    ?obs  om-owl:observedProperty  weather:_AirTemperature ;
          om-owl:procedure  ?sensor ;
          om-owl:result  ?res .
    ?res  om-owl:floatValue  ?value .
  }
  FILTER (?value > 80)
}
```

Listing 5.12: The query in Listing 5.9 translated into CQELS-QL via the RSP-SPIN API.

5.4 Summary and Discussion

In this chapter, an extension to the SPIN vocabulary has been presented to support RSP-QL queries. SPIN is, to date, the only model that lets SPARQL queries be represented and shared as RDF. SPIN templates can support parameterization, allowing variables to be bound to user-defined values when the queries are instantiated. Formulating queries, and validating that they reflect the intention of the user, is often time-consuming, and writing the queries in first place requires knowledge not only of the particular RSP dialect, but also of the underlying data and ontologies. Query templates can be viewed as abstractions of the queries that let users leverage the expressiveness of SPARQL, even if they have little or no experience in writing queries themselves.

The work presented here represents, to the author's knowledge, the first attempt at extending SPIN for use in RSP. In combination with the RSP-SPIN API it is presently the only Semantic Web technology that supports any type of parameterization and sharing of RSP queries. While the modifications supported by query templates of this type are minor with respect to the

overall query, the effects in terms of query re-usability are significant. The RSP-SPIN API supports parameterization of variables, range and bounds for named windows, input stream identifiers, and the output stream identifier.

The vocabulary extension and the extended API were evaluated based on the RSP Group sample queries, which have been used as in the RSP-QL standardization work. The evaluation focused on query representation only, since the template functionality of SPIN remains unchanged in the extension. We showed that the model captures all of the constructs currently supported in RSP-QL and that it can capture any RSP language that that can be modeled by RSP-QL. This was demonstrated by showing that an RSP-QL query expressed as RSP-SPIN can be serialized into the languages C-SPARQL, CQELS-QL, and $\text{SPARQL}_{\text{stream}}$. One of the implications of this is that even if no RSP implementation yet supports RSP-QL, the proposed extension can still be used to represent query templates for some of the most well known RSP languages. For example, some RSP benchmarks could be implemented based on a single set of RSP-QL queries, which can be automatically translated into any of the supported languages. Using RSP-SPIN as an abstraction layer can also help users transition between different RSP languages, allowing them to run tests and experiments without investing time in developing and validating queries for each engine separately. However, RSP-SPIN does not resolve the possible semantic differences that exists between these RSP models.

There are other aspects to RSP that are not currently communicated in RSP-QL queries, such as query evaluation strategies (e.g., if queries should be executed on data input, based on a fixed interval, or every time a window slides), and specification of the timestamp properties that apply on different streams. These features could be communicated as part of the template representation, for example, to configure an execution environment.

One aspect that has not yet been explored in depth is the possibility of increasing the types of parameter constraints that can be supported in SPIN templates. For example, we could support minimum and maximum values, tests against regular expressions, or match input against data in ontologies[14]. While many constraints can be defined as part of the queries, using the SPIN meta modeling features to do some of these checks lets the queries be less complex and easier to both optimize and reuse.

Future work involves evaluating RSP-SPIN in context and aligning it with the updated proposal of the RSP Group. Potential additions to the model that have already been discussed include, for example, support for temporal operators (e.g., as supported by EP-SPARQL). We also envision the possibility of describing pipelines and defining chains of queries to process data in several steps, with particular focus on CEP applications.

[14]SPIN supports both testing for both subclass and subproperty relations but patterns that are more expressive could be described using, for example, the Shapes Constraint Language (`https://www.w3.org/TR/shacl/`).

6 Architecture and Applications

The RSP systems described in Chapter 2 support continuous processing of streaming data and in this chapter we leverage some of these systems for complex event processing. We begin by introducing a loosely coupled (sometimes inaccurately referred to as *decoupled*) architecture, where the dependencies between individual system components have been relaxed. The architecture was implemented in three different scenarios, dealing with traffic-incident monitoring, criminal-activity monitoring, and electronic healthcare monitoring.

6.1 RSP for Event Processing

Adopting RDF as the main model in CEP is likely to cause performance overhead compared to many other approaches; however, it can offer a simpler overview of the data and the processing taking place. The Semantic Web standards can support the integration of data represented in heterogeneous formats, and reasoning can enable systems to leverage information that is represented implicitly in the data.

In the future, RSP could ideally provide a single formalism for today's divergent CEP approaches. However, there are some inherent limitations in current RSP approaches that affect the degree to which they can be applied to CEP. Today, an RSP system that exists completely in isolation is severely limited in terms of processing capabilities. For example, none of the current RSP systems supports natural language processing, advanced statistical analysis, or machine learning. Expecting all such functionality to be incorporated

into a single RSP execution framework is, at the time of writing, not feasible. Instead, we must be able to efficiently interact with and leverage external systems and tools for this type of processing.

Another limitation of current RSP systems is their scalability. There are of course many factors to be considered when discussing RSP performance, such as stream velocity, query execution rate, window size and slide frequency, number of parallel streams, query complexity, and size of the background data. Distributed RSP systems have reported total throughputs of up to 200,000 triples/second [32, 46], but most engines report input stream velocities of less than a few thousand triples per second. To put this figure in perspective, lets consider the social media service Twitter. Twitter reported an average of around 5,700 tweets per second in 2013[1]. The JSON object returned by the Twitter API can contain more than a hundred fields, including user-related data, multimedia links, user mentions, re-tweet counts, timestamps and more. Each tweet could thus conservatively result in dozens of triples, generating a total average stream rate of up to 100,000 triples per second. Even the distributed approaches would struggle to handle these stream velocities. While the Twitter stream is an exceptional case in terms of velocity, similar stream rates can result from querying collections of streams, such as data generated from smart sensors.

RSP-QL can be used to model some of the most common RSP languages. In the following scenarios, we have used RSP-QL to define all the RSP queries and used the RSP-SPIN extension (see Chapter 5) to convert these representations into the required RSP language. For space reasons, we report only the RSP-QL queries here, but the translated queries are listed in Appendix B.

6.2 Architecture

In the conceptual architecture, we consider a system that consists of three main types of building blocks: *event producers* (producers), *event consumers* (consumers), and *event processing agents* (EPAs) (see Section 2.5). The components are linked up to form event processing networks (EPNs). A producer passes data to one or more message channels, consumers serve as message sinks, and EPAs both consume and output data within the EPN. Based on this definition, EPAs include any component capable of both consuming and producing data.

Furthermore, we explicitly model the channels through which messages are passed between components. The channels must support *one-to-many* communication, *at-least-once* delivery semantics, and maintain the original order of the streaming data. One-to-many communication simply states that more than one component should be able to consume data from a channel.

[1] `https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html` last retrieved on 2017-09-14

At-least-once delivery semantics requires the transportation of a message to a consumer to be guaranteed. A more strict definition, *exactly-once* delivery semantics, where data is guaranteed to be delivered to consumers exactly once, is seldom feasible in distributed systems.

Making the message channels explicit highlights the intention of maintaining loose coupling within the EPN. However, no assumptions are made with respect to how these are implemented in practice. For example, messaging could be managed using a framework that supports the publish-subscribe pattern or the actor model, but could also be implemented as a part of the application. The basic relationships between the building blocks in the architecture are illustrated in Figure 6.1. The results reported in following sections were all measured on a 2015 Macbook Pro (2,8 GHz Intel Core i7) with 16 GB 1600 MHz DDR3 with 80 GB of available SSD storage space.

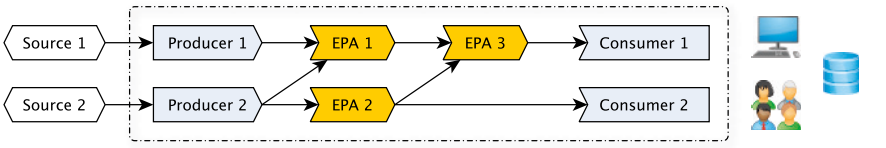


Figure 6.1: Illustration of the event processing architecture. Data passes from the sources to the producers that make it available internally. Messages may be passed through a network of EPAs before reaching the consumers, and ultimately the end-users or external applications.

6.3 Scenario 1: Traffic-Incident Monitoring

In the traffic-incident scenario, the goal was to increase emergency responders' situation awareness about traffic incidents by leveraging social media. The scenario was inspired by similar work on event enrichment in the VALCRI project.

The objective was to supplement traffic-incident reports with potentially relevant information by automatically capturing associated status updates from Twitter status updates (or tweets). The traffic incidents were manually collected from online traffic services, but in a real world setting the incidents would be generated based on data from some trusted source (e.g., the report of an emergency responder).

6.3.1 Data

The data used in the scenario included a list of traffic-incident reports, a stream of tweets, and data from the Ordnance Survey[2]. The traffic-incident reports were collected from Traffic England[3] and Traffic Scotland[4], which are two online services that provide a lot of information regarding traffic conditions. The two sites differ somewhat in the type of information they publish with respect to incidents, but both report the road and junction affected, a reason for the incident, a text description, and information about the number of lanes that are affected. Additionally, an incident is plotted on a map showing the exact geographic location of the incident. Usually an estimate of how long it will take to clear the site is also provided. Table 6.1 shows an example of an accident involving an overturned vehicle on the M1 between the junctions J46 and J47.

Table 6.1: Example of a traffic incident from Traffic England. The report describes an accident involving an overturned vehicle between the junctions J46 and J47 on the northbound M1. The nearest city, northing, and easting were extracted from the incident location plotted on a map.

Severity	Severe
Road	M1
Type	Incident
Location	The M1 northbound between junctions J46 and J47
Reason	Accident involving an overturned vehicle
Status	Currently Active
Time to clear	The event is expected to clear between 13:30 and 13:45 on 14 September 2017
Return to normal	Normal traffic conditions are expected between 14:30 and 14:45 on 14 September 2017
Lanes closed	3 of 3
Easting	438587
Northing	433905

[2] `http://data.ordnancesurvey.co.uk/`
[3] `http://www.trafficengland.com/`
[4] `https://trafficscotland.org/`

Twitter is one of the world's most popular microblogging platforms[5]. A tweet is limited to a short text snippet, containing a maximum of 140 characters[6]. Depending on the privacy settings, public tweets can provide location information connected to the user or the tweets. For geotagged tweets, the location is derived from the device on which the tweet was posted; however, the percentage of users that provide an exact location is very low. One study showed that only about 0.7 percent of all tweets provide a device location [28]. On the other hand, users often provide some location information as part of their account profile, or associate their tweets with a place description.

In the scenario, the Twitter API was used to retrieve tweets that could potentially be related to a particular traffic incident. The search API was used to collect tweets around and before the reported time of the incidents, while the streaming API was used to collect newly posted tweets continuously. The search API returns a subset of the complete set of tweets matching the search criteria, whereas the streaming API returns a rate limited percentage of all new tweets that match the search criteria. Generally, the search API results in much higher latencies than the streaming API, and the number of requests are rate limited. In theory, search queries can be issued by a single application every 2–5 seconds on average[7]. The maximum number of tweets returned by a search request is 100, which places the maximum average throughput at around 50 tweets per second. The rate limit of the streaming API, on the other hand, is about 1 percent of all posted tweets in the entire world, which means that its throughput is up to about 60 tweets per second. The theoretical throughput of the two APIs is therefore roughly the same. For data collection, both APIs were used in combination. A subset of a JSON response returned from the Twitter service is shown in Listing 6.1.

[5]In early 2017, Twitter reported 313 million active users per month, 88 percent of which were active on mobile devices (`https://about.twitter.com/company`

[6]Twitter updated their character policy in 2016, and no longer counts multimedia links and quoted tweets (re-tweets) towards character limit.

[7]The rates reported at the time of writing were 180 search requests per authenticated user and 450 requests per authenticated application for each 15-minute window (see `https://dev.twitter.com/rest/public/rate-limits`)

```
{
   "text": "Slow traffic on the M1 :(",
   "retweeted": false,
   "place": null,
   "created_at": "Thu Sep 14 13:41:36 +0000 2017",
   "user": {
      "screen_name": "JohnDoe",
      "time_zone": "GMT",
      "location": "Leeds, England",
      "lang": "en"
    },
    "geo": null,
    ...
}
```

Listing 6.1: The listing shows a few included in a JSON object returned by the Twitter API.

The background data in the scenario consisted of data from the Ordnance Survey. The Ordnance Survey is Great Britain's national mapping agency, and it provides the most accurate and up-to-date geographic data about Great Britain. The *50K Gazetteer Linked Data*[8] contains more than 250,000 place names and is the most detailed gazetteer (or geographical dictionary) available for Great Britain. The Ordnance Survey maps all locations to the British National Grid, which lets positions be referenced in using a grid-based system where each square can be subdivided into progressively smaller squares. A location is identified by a string of letters and numbers, or using two integer coordinates. An illustration of the grid divided into squares of size 100 km by 100 km is shown in Figure 6.2.

6.3.2 Ontologies

The traffic incidents and tweets were modeled using and an extension of the Event ODP (see Chapter 4). In the model, a `TrafficIncident` was defined as a `ComplexEventObject` and associated with the time of the incident (`reportedAt`), a named place (`hasLocation`), keywords (`hasKeyword`), a text description (`hasDescription`), a country (`hasCountry`), and the nearest city (`hasCity`). A tweet was represented using the class `Status`, which was was defined as a `SimpleEventObject`. Tweets were also linked with two classes representing the associated place (`Place`) and a user account (`Account`). The ontology included data properties to represent a subset of the attributes returned by the Twitter API.

[8] `http://data.ordnancesurvey.co.uk/datasets/50k-gazetteer`

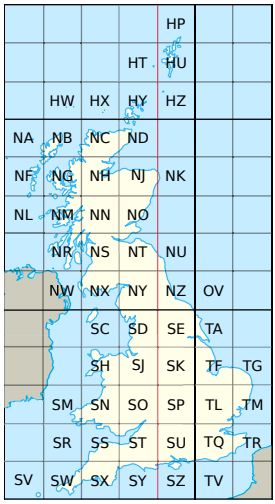


Figure 6.2: British National Grid divided into squares of size 100 km by 100 km. `https://upload.wikimedia.org/wikipedia/commons/1/18/British_National_Grid.svg`. Distributed under a CC BY-SA 3.0 license.

The relevance class (`Relevance`) was used to associate tweets with a set of relevance values described in terms of spatial distance (`hasSpatialDistance`), temporal distance (`hasTemporalDistance`), and content similarity (`hasContentSimilarity`). The named place class (`gaz:NamedPlace`) defined in the Ordnance Survey ontology provided an alignment with the gazetteer data. The classes and object properties of the ontology are presented in Figure 6.3.

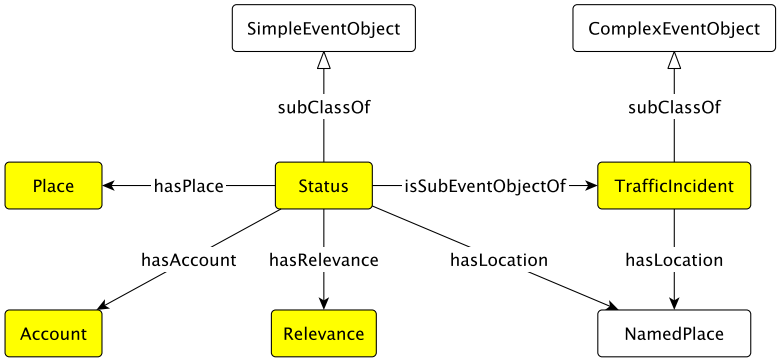


Figure 6.3: Illustration of the classes and object properties defined in the traffic-incident ontology. Events were associated with grid locations via the `NamedPlace` class defined in the Ordnance Survey ontology.

6.3.3 Relevance Metrics

Three metrics were used to describe the relevance between tweets and traffic incidents: spatial distance, temporal distance, and content similarity.

Spatial distance between two grid locations can be calculated using Pythagoras's theorem. The actual meaning of a spatial distance is, however, dependent on the geographic granularity of the locations themselves. For example, there is no obvious way of measuring the distance between the location of a traffic incident and a city, although both can be represented as grid locations. The approach adopted here adds a penalty to distances calculated based on a town or city, as opposed to an exact coordinate. When a tweet is enriched with an indirect location collected from a user profile, the penalty was higher still.

Temporal distance between a traffic incident and a tweet was defined as the duration between their respective timestamps. A small duration indicated that the two events occurred near each other in time. Since the tweet stream can contain tweets that were posted before the reported time of the traffic incident, durations can potentially be negative.

Content similarity can be assessed in many different ways but here we simply chose to count the number of matches between the text of a tweet and the keywords from the traffic incident. This allowed the content similarity metric to be calculated using a simple SPARQL aggregate query. One way of improving this could be by, for example, providing weights for the individual content words.

6.3.4 Implementation

In the implementation, direct component-to-component communication was used. The application was written in Java and run within a single JVM. Events were internally represented and streamed as RDF graphs represented by Jena models. An overview of the event processing pipeline is shown in Figure 6.4.

Event Producers

The traffic incidents were manually captured from Traffic England and Traffic Scotland, and translated into RDF based on the event ontology extension described above. For each traffic incident, the reported time, grid location, and nearest city were identified. Incident keywords were automatically created based on the traffic-incident description, with stopwords, time notations, and numbers removed. Listing 6.2 shows the complete structure of the traffic-incident example from Table 6.1.

The information about a traffic incident was used to automatically set up a twitter stream, using the traffic-incident keywords as the search query. The

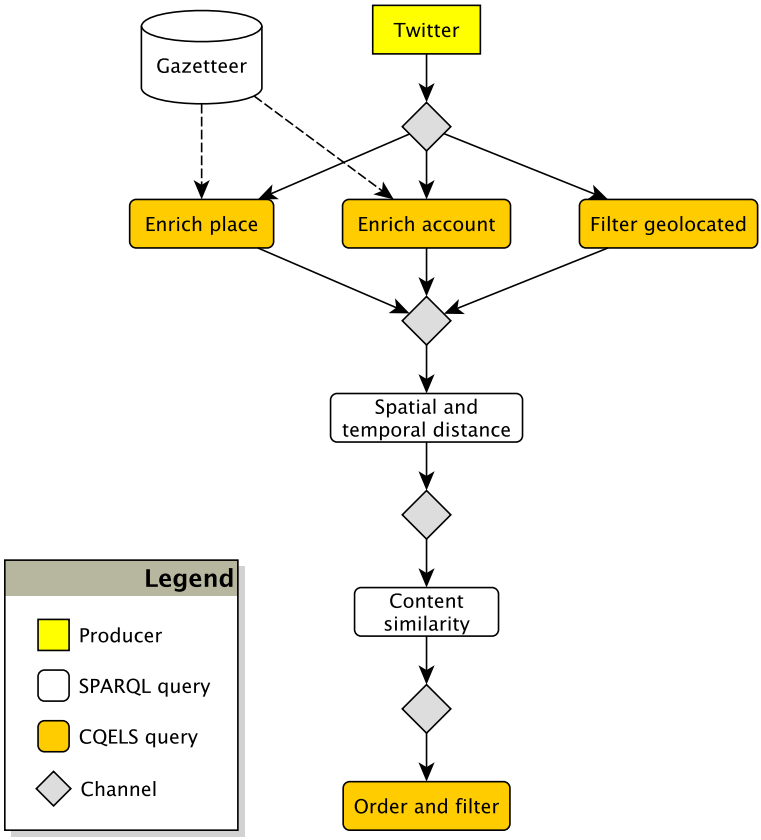


Figure 6.4: Diagram illustrating the event processing pipeline implemented in the traffic-incident monitoring scenario.

stream was produced as a two-part process. The search API was used to retrieve matching tweets from the past, stretching back to two hours before the reported time of the incident, while the streaming API was used to retrieve tweets continuously. Each tweet was implicitly ordered based on the time of arrival in the stream, rather than on the generation time of the post. The message was formatted as RDF prior to being published to the internal channel and was bound to the traffic incident for which it was retrieved. An example of a tweet, represented using the Twitter ontology, is shown in Listing 6.3.

```
@prefix : <http://trafficincident.com/ontology#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix gaz: <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix spatial: <http://data.ordnancesurvey.co.uk/ontology/
    ↪ spatialrelations/> .

<http://trafficincident.org/incident1>
   a  :TrafficIncident ;
   :hasDescription  "Accident involving an overturned vehicle on the M1
     ↪ northbound between junctions J46 and J47. The event is expected to
     ↪ clear between 13:30 and 13:45 on 14 September 2017. Normal traffic
     ↪ conditions are expected between 14:30 and 14:45 on 14 September
     ↪ 2017. All lanes are closed." ;
   :hasKeyword  "expected" , "j47" , "lanes" , "conditions" , "traffic" ,
     ↪ "vehicle" , "junctions" , "clear" , "j46" , "northbound" ,
     ↪ "overturned" , "closed" , "event" , "m1" , "september" ,
     ↪ "accident" , "normal" , "involving" ;
   :hasLocation [ a  gaz:NamedPlace ;
                  spatial:easting  "438587.2965974021"^^xsd:double ;
                  spatial:northing  "433905.46081581945"^^xsd:double ;
                  :hasCity  "Leeds" ;
                  :hasCountry  "England" ;
                  geo:lat  "53.799805"^^xsd:double ;
                  geo:long  "-1.414074"^^xsd:double
                ] ;
   :reportedAt "2017-09-14T14:00:00+00:00"^^xsd:dateTime .
```

Listing 6.2: A traffic incident modeled using the traffic-incident ontology.

```
@prefix twitter: <http://twitter.com/ontology#> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix place: <http://twitter.com/place#> .
@prefix event: <http://www.ontologydesignpatterns.org/cp/owl/
    ↪ eventprocessing.owl#> .
@prefix account: <http://twitter.com/account#> .
@prefix status: <http://twitter.com/status#> .

status:908312966403055616
   a  twitter:Status ;
   twitter:createdAt  "2017-09-14T14:54:21.000+02:00"^^xsd:dateTime ;
   twitter:hasAccount  account:298423910 ;
   twitter:hasLang  "en" ;
   twitter:hasStatusId  "908312966403055616"^^xsd:long ;
   twitter:hasText  "Driver in M1 crash that killed eight people did not
     ↪ have lorry licence, court hears https://t.co/8y5oZsI4su ^DailyT
     ↪ https://t.co/YLJM7FjfTf" ;
   twitter:isRetweet  false ;
   event:isSubEventObjectOf  <http://trafficincident.org/incident1> .

account:298423910
   a  twitter:Account ;
   traffic:hasLocation  [ a gaz:NamedPlace ;
                          rdfs:label "ENGLAND" ] ;
   twitter:createdAt  "2011-05-14T10:58:47+0200"^^xsd:dateTime ;
   twitter:hasAccountId  "298423910"^^xsd:long ;
   twitter:hasLang  "en" ;
   twitter:hasScreenName  "MrTraffic" .
}
```

Listing 6.3: A tweet modeled using the traffic-incident ontology.

Event Processing Agents

In the event processing pipeline, four of the EPAs were implemented as queries in the CQELS engine. However, CQELS-QL is not expressive enough for all the processing required in the traffic-incident scenario. For example, the current CQELS engine has no support property paths, optionals, or unions. There is also no support for datetime arithmetic and square root. In order to calculate the spatial and temporal distances we instead employed standard SPARQL queries against an in-memory dataset represented using Apache Jena [9].

Practically, CQELS needs to decompose the streamed graphs into triples prior to ingestion, which gives rise to the issue of stream punctuation and event-object boundaries (see Section 4.3). Boundaries in the scenario were managed by matching the entire event-object structures in the query patterns, which removed the risk of partial matches.

The traffic incidents in the scenario were always associated with a grid location. While tweets rarely had any such detailed location information, they were often linked to a more general location description, either directly or via the associated user profile. In both cases, the identified place descriptions were used as an estimate of the tweet origin. The location descriptions typically consisted of a comma-separated city and country, and all items in such a list were added as labels to the associated location during the RDF translation phase. To allow comparisons with the gazetteer, all device locations were also converted into grid locations.

From the gazetteer, 388,000 triples were loaded into the CQELS engine, including a total of 42,549 towns, cities, and other settlements. Three queries were used to produce a stream of tweets associated with some location derived from the device, associated place, or user account. The queries are represented using RSP-QL in Listings 6.4–6.6. The CQELS queries generated from these can be found in Appendix B.

The spatial and temporal similarity between the tweets and the associated traffic incident was calculated using a SPARQL query. The distance between two grid coordinates can be calculated by applying the Pythagoras's theorem; however, CQELS does not support square root. With respect to the temporal distance datetime arithmetic is required, which is not possible in CQELS either. Both operations are supported in the Apache Jena and a single query was used to enrich the stream with both relevance metrics. The precision of the spatial information depends on the source of the location information. In the query, any location information enriched from a user account received an added spatial distance of 10 km, while one enriched from a user account received an addition of 100 km. The query is shown in Listing 6.7.

Content similarity was calculated as the number of keyword in the traffic incident that matched the text in a tweet. This calculation could not be done

[9] `https://jena.apache.org/`

in any straightforward way using CQELS since it does not support nested aggregates. The EPA was therefore implemented as a SPARQL query and issued against an in-memory model in Apache Jena. The query is shown in Listing 6.8.

```
PREFIX  :        <http://traffic.org/data#>
PREFIX  gaz:     <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

REGISTER STREAM :enrich-place AS

CONSTRUCT ISTREAM
  {
    ?tweet  rdf:type  twitter:Status ;
            twitter:hasText  ?text ;
            event:isSubEventObjectOf  ?incident ;
            twitter:createdAt  ?createdAt ;
            traffic:hasLocation  ?location .
    ?location  rdf:type  gaz:NamedPlace ;
               rdfs:label  ?locationLabel ;
               spatial:northing  ?northing ;
               spatial:easting  ?easting ;
               traffic:source  "place" .
  }
FROM NAMED WINDOW :w ON :twitter [RANGE PT2S]
WHERE
  { WINDOW :w
      { ?tweet  rdf:type  twitter:Status ;
                twitter:hasText  ?text ;
                event:isSubEventObjectOf  ?incident ;
                twitter:createdAt  ?createdAt ;
                twitter:hasPlace  ?location .
        ?location  rdf:type   gaz:NamedPlace ;
                   rdfs:label ?locationLabel
      }
    _:b0  rdf:type  gaz:NamedPlace ;
          rdfs:label  ?locationLabel ;
          spatial:easting  ?easting ;
          spatial:northing  ?northing
  }
```

Listing 6.4: RSP-QL query generating a stream of twitter messages enriched with location information from the gazetteer based on the place associated with the tweet.

```
PREFIX  :        <http://traffic.org/data#>
PREFIX  gaz:     <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  event:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  traffic: <http://trafficincident.com/ontology#>

REGISTER STREAM :enrich-account AS

CONSTRUCT ISTREAM
  {
    ?tweet a twitter:Status ;
           twitter:hasText ?text ;
           event:isSubEventObjectOf ?incident ;
           twitter:createdAt ?createdAt ;
           traffic:hasLocation ?location .
    ?location a gaz:NamedPlace ;
              rdfs:label ?locationLabel ;
              spatial:northing ?northing ;
              spatial:easting ?easting ;
              traffic:source "account" .
  }
FROM NAMED WINDOW :w ON :twitter [RANGE PT2S]
WHERE
  { WINDOW :w
      { ?tweet twitter:hasText ?text ;
               event:isSubEventObjectOf ?incident ;
               twitter:createdAt ?createdAt ;
               twitter:hasAccount ?account .
        ?account a twitter:Account ;
                 traffic:hasLocation ?location .
        ?location a gaz:NamedPlace ;
                  rdfs:label ?locationLabel
      }
    [] a gaz:NamedPlace ;
       rdfs:label ?locationLabel ;
       spatial:easting ?easting ;
       spatial:northing ?northing .
  }
```

Listing 6.5: RSP-QL query generating a stream of twitter messages enriched with location information from on the gazetteer based on the associated user account.

```
PREFIX  :        <http://traffic.org/data#>
PREFIX  gaz:     <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

REGISTER STREAM :geolocated AS

CONSTRUCT ISTREAM
  {
    ?tweet rdf:type twitter:Status ;
           twitter:hasText ?text ;
           event:isSubEventObjectOf ?incident ;
           twitter:createdAt ?createdAt ;
           traffic:hasLocation ?location .
    ?location rdf:type gaz:NamedPlace ;
              spatial:northing ?northing ;
              spatial:easting ?easting ;
              traffic:source "status" .
  }
FROM NAMED WINDOW :w ON :twitter [RANGE PT2S]
WHERE
  { WINDOW :w
      { ?tweet rdf:type twitter:Status ;
               twitter:hasText ?text ;
               event:isSubEventObjectOf ?incident ;
               twitter:createdAt ?createdAt ;
               traffic:hasLocation ?location .
        ?location rdf:type gaz:NamedPlace ;
                  spatial:northing ?northing ;
                  spatial:easting ?easting
      }
  }
```

Listing 6.6: RSP-QL query identifying the twitter messages that are geolocated.

```
PREFIX twitter: <http://twitter.com/ontology#>
PREFIX spatial: <http://data.ordnancesurvey.co.uk/ontology/
    ↪ spatialrelations/>
PREFIX afn:     <http://jena.hpl.hp.com/ARQ/function#>
PREFIX event:   <http://www.ontologydesignpatterns.org/cp/owl/
    ↪ eventprocessing.owl#>
PREFIX traffic: <http://trafficincident.com/ontology#>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
   ?tweet a twitter:Status ;
          traffic:hasRelevance ?relevance .
   ?relevance a traffic:RelevanceObject ;
              traffic:hasSpatialDistance ?spatialDistance ;
              traffic:hasTemporalDistance ?temporalDistance .
   ?s ?p ?o .
}
WHERE {
   ?tweet a twitter:Status ;
          twitter:createdAt ?createdAt ;
          (!<>)* ?s .
   ?s ?p ?o .
   { SELECT ?tweet ?incident (MIN(?distance) AS ?spatialDistance)
     WHERE {
        ?tweet event:isSubEventObjectOf ?incident ;
               traffic:hasLocation ?location .
        ?location spatial:easting ?e1 ;
                  spatial:northing ?n1 ;
                  traffic:source ?source .
        GRAPH <http://traffic.org/incidents> {
           ?incident traffic:hasLocation _:b0 .
           _:b0 spatial:easting ?e2 ;
                spatial:northing ?n2 .
        }
        BIND(IF(?source = "status", 0,
             IF(?source = "place", 10000, 1000000)) AS ?penalty)
        BIND(xsd:double(?e1)-xsd:double(?e2) AS ?eDif)
        BIND(xsd:double(?n1)-xsd:double(?n2) AS ?nDif)
        BIND(afn:sqrt(?eDif * ?eDif + ?nDif * ?nDif) + ?penalty AS
    ↪ ?distance)
     } GROUP BY ?tweet ?incident
   }
   GRAPH <http://traffic.org/incidents> {
      ?incident traffic:reportedAt ?reportedAt .
   }
   BIND((?createdAt - ?reportedAt) AS ?temporalDistance)
   { BIND(BNODE() AS ?relevance) }
}
```

Listing 6.7: SPARQL query calculating the spatial and temporal distance between a traffic incident and a tweet.

```
PREFIX twitter: <http://twitter.com/ontology#>
PREFIX event:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX traffic: <http://trafficincident.com/ontology#>

CONSTRUCT
  {
    ?tweet traffic:hasRelevance ?relevance .
    ?relevance traffic:hasContentSimilarity ?score .
    ?s ?p ?o .
  }
WHERE
  { ?tweet traffic:hasRelevance ?relevance .
    ?tweet (!<>)* ?s .
    ?s ?p ?o
    { SELECT ?tweet (COUNT(?keyword) AS ?score)
      WHERE
        { ?tweet twitter:hasText ?text
          GRAPH <http://traffic.org/incidents>
            { ?incident traffic:hasKeyword ?keyword }
          FILTER regex(?text, ?keyword, "i")
        }
      GROUP BY ?tweet ?incident
    }
  }
```

Listing 6.8: SPARQL query calculating the number of keywords from the traffic incident that are matched in the status text of a tweet.

```
PREFIX  :        <http://traffic.org/data#>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

REGISTER STREAM :filtered AS

SELECT ISTREAM ?text ?createdAt ?easting ?northing
               ?contentSimilarity ?spatialDistance ?temporalDistance
FROM NAMED WINDOW :w ON :stream4 [RANGE PT5H]
WHERE
  { WINDOW :w
      { ?tweet    twitter:hasText       ?text ;
                  twitter:createdAt     ?createdAt ;
                  traffic:hasLocation   ?location ;
                  traffic:hasRelevance  ?relevance .
        ?location  spatial:easting      ?easting ;
                  spatial:northing      ?northing .
        ?relevance  traffic:hasContentSimilarity  ?contentSimilarity ;
                  traffic:hasSpatialDistance  ?spatialDistance ;
                  traffic:hasTemporalDistance  ?temporalDistance
      }
    FILTER (?contentSimilarity > 2)
    FILTER (?spatialDistance < 50000)
  }
```

Listing 6.9: An RSP-QL select query filtering out tweets based on user-defined threshold values.

Event Consumer

The event consumer was implemented as a CQELS query and results printed in a continuously growing table of tweets. The query filtered the stream of enriched tweets by the relevance metrics. The filters were intended to be user-defined threshold values, with lower values resulting in a higher volume of retained tweets, and higher thresholds would result in fewer tweets of a higher overall relevance. The query is represented as an RSP-QL query in Listing 6.9.

6.3.5 Summary and Discussion

The traffic-incident monitoring scenario presented several challenges from an RSP perspective, and it highlighted a number of features required in many CEP scenarios. For example, it is often necessary be able to calculate the difference between two timestamps, and to find the distance between geographic locations. Generally, neither of these are supported in current RSP implementations[10].

The content similarity EPA could not be implemented as a CQELS-QL query, since the CQELS engine does not support nested queries, which was necessary to calculate the required aggregate value. When tested individually, the EPAs based on SPARQL and Jena both processed up to 100 events per second without causing any noticeable delays. It should also be noted that all of these EPAs would have been supported in RSP-QL.

Perhaps the most notable shortcoming of the CQELS engine in the scenario was the lack of support for the optional clause and the union pattern. This made it necessary to formulate three separate queries for the location enrichment phase, where one would have otherwise sufficed. Also, despite being represented in a grid system, the limitations in terms of arithmetic operations prevented CQELS from being used to calculate the spatial distances.

Querying the Ordnance Survey gazetteer introduced a noticeable performance impact on CQELS. The performance was relatively stable up to around 40 tweets/second (700 triples/second) but beyond this rate the engine became periodically unresponsive. Interestingly, the SPARQL queries issued against the events in the in-memory Jena models were ultimately not the bottleneck in the pipeline. The explanation for this is twofold. Firstly, the direct communication in the pipeline allowed the Jena models to be communicated with no serialization overhead. Secondly, the in-memory models that were queried were very small, containing only a single traffic incident and a twitter event per query evaluation.

The tweets were associated with the traffic incidents based on a set of relevance metrics, and with the exception of temporal distance (which in

[10]Recent version of INSTANS provide support for some datetime arithmetic by loading a set of extension functions.

CQELS-QL can only be defined in the window) the thresholds for these values could be calibrated based on user preferences in the final query. For example, increasing the required content similarity and reducing the allowed temporal and spatial distance would reduce the number of tweets in the final output, while increasing the overall relevance of the remaining tweets.

6.4 Scenario 2: Criminal-Activity Monitoring

In the criminal-activity monitoring scenario, the focus was on how scalable event processing can be achieved when dealing with data requiring sophisticated access-control. In the VALCRI project (see Section 3.1.1), information is associated with security classifications ranging from publicly accessible to strictly confidential, but data access is also context dependent. For example, the information accessible to an analyst working on a burglary case can be very different from that which can be legally and ethically accessed in a murder investigation. These access-control restrictions cannot be covered by only using data encryption or group-based access control.

The VALCRI context provides constraints with respect to how components are allowed to communicate. Specifically, the software employed to manage access control has been integrated with Apache ActiveMQ[11] (hereafter referred to simply as ActiveMQ). ActiveMQ is a messaging framework that supports, among other things, the publish-subscribe pattern. The messages are ordered within *topics* and managed by a broker that allows components to communicate with each other without knowing about each other's existence (see Figure 6.5).

Figure 6.5: Relationship between producer, consumer, and broker in a basic setup of ActiveMQ. The producer and consumer each have a reference to a broker. The broker can perform load balancing by distributing messages for a topic between registered consumers.

The access-control software required all messages to be wrapped in an XML-based message structure[12]. The access-control policies that applied on top of the communicated messages lie outside the scope of this thesis and for the remainder of the chapter the process will be viewed as an implicit part of the message communication framework.

[11] `http://activemq.apache.org/`

[12] This requirement was in place during the time of the implementation but has been dropped in the more recent versions of the VALCRI infrastructure.

Another requirement in the VALCRI scenario was that each system component had to be encapsulated as a Docker[13] container. Docker containers allow applications to be packed together with all the required libraries and dependencies, which greatly simplifies deployment and collaborative development. This, however, increased the technical complexity of the pipeline and added to the memory requirements of the pipeline as a whole.

The Automatic Number Plate Recognition (ANPR) data stream was the primary source of streaming data in the VALCRI. In the scenario, we limited ourselves to three primary tasks: (1) filtering ANPR observations based on a list of vehicles of interest, (2) generating routes traveled by vehicles of interest, and (3) detecting possible rendezvous between persons that were suspects in the same crime.

6.4.1 Data

The datasets available in the VALCRI project at the time of the implementation were made up by more than 120 millions RDF quads. This is considerably more than what current RSP implementations can support efficiently, and throughout the implementation we used only a small subset of the crime and person records.

The ANPR records contained five fields: the number plate of the observed vehicle, a camera description, the location of the camera sensor, and a timestamp (see Table 6.2). The ANPR cameras were considered to be static, and they were not moved during the application runtime. This means that three of the fields from each sensor reading could be stripped and instead be provided as part of the static metadata. Additionally, placing the camera information in a static dataset allowed the distances between all pairs of ANPR cameras to be calculated and added to the static dataset offline, allowing approximate distances to be looked up quickly.

Table 6.2: Example of an ANPR observation.

Number Plate	A2RJQ
Camera description	WPT 267
Longitude	52.46333
Latitude	-1.88695
Observation time	2017-03-17 20:10:00 GMT

The ANPR dataset in VALCRI contains approximately 600,000 observations per day; however, the total ANPR volume in Great Britain returns more than 20 million observations during the same time-period[14]. At the

[13] `https://www.docker.com/`

[14] The ANPR cameras in the United Kingdom report between 25 and 35 million records daily (`http://www.npcc.police.uk/FreedomofInformation/ANPR.aspx` visited on 2017-10-09).

time of writing, the vehicle observations in VALCRI had not yet been temporally aligned with the rest of the VALCRI data, and no links between vehicle registration numbers and persons in the crime datasets were available.

In order to provide this missing alignment, as well as simulate a higher stream rate, we generated a new ANPR dataset using the original data as a reference. The ANPR cameras were manually positioned at intersections and roads around a city center, after which a series of routes were laid out between cameras. Vehicle observations were then generated along these routes and the timestamps for each observation along a route were offset relative to the estimated distance to the previous camera. The number of observations were configured to increase throughout the course of the day and decrease again by the evening. Figure 6.6 shows the placement of a subset of the cameras included in the new dataset.

Figure 6.6: The map shows the locations of a subset of the ANPR camera sensors used in the scenario.

We further included a set of blacklisted vehicles and a vehicle owner registry. The blacklisted vehicles represented vehicles of interest, which were not necessarily connected to a specific crime. In the owner registry, individuals were linked to vehicle registration numbers, providing links between the owner of a car and suspects. This data is representative of the type of alignments that would be expected in a real-world scenario (see Section 3.1.1).

6.4.2 Ontologies

Event objects were modeled based on an extension of the Event ODP (see Chapter 4). The ontology is illustrated Figure 6.7 and shows the classes and object properties defined in the model. The ontology was aligned with the

main VALCRI ontologies via the classes `Person` and `Crime`. The `Distance` class was used to link an estimated distance between two ANPR cameras.

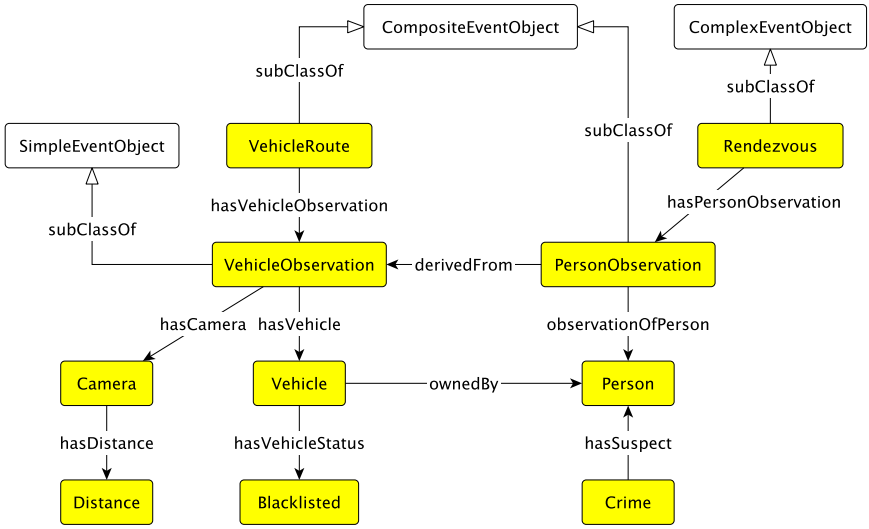


Figure 6.7: The diagram shows an overview of the classes and object properties in the ANPR ontology.

6.4.3 Implementation

The pipeline implemented for this scenario is illustrated in Figure 6.8. The ANPR producer published a comma-separated string to topic in ActiveMQ, which was converted into RDF by an adapter. All RDF events were published as serialized RDF graphs and wrapped in the XML-message structure required by the access-control software. The ANPR data stream was consumed by two separate CQELS queries, after which the results were processed by two C-SPARQL queries. Finally, the data was made available in a client service to which clients could subscribe.

ActiveMQ was deployed as Docker container with the necessary security and access-control add-on, and configured to run with a single message broker. For each stream a topic (represented by URI) was generated. Topics that did not already exist were set to be created automatically, which meant that the components in the event processing network could be started or stopped in any order without causing any issues with respect to inter-component dependencies.

CQELS and C-SPARQL were integrated as RSP components in the system, and the static datasets were loaded into the default graph of each engine.

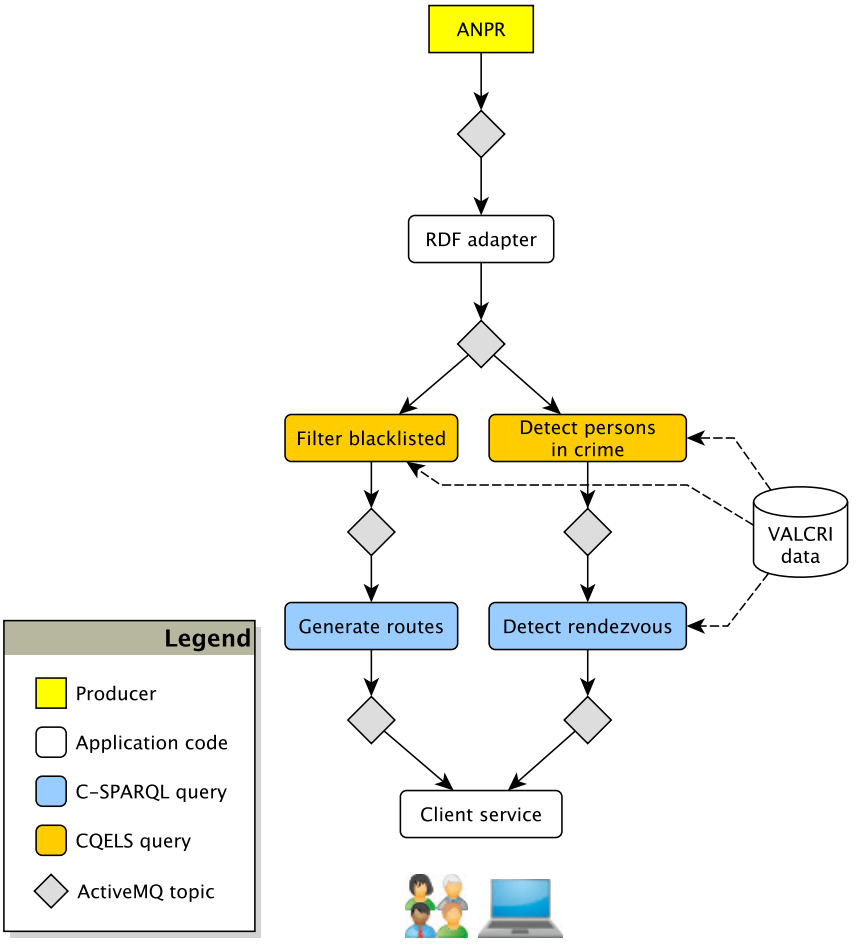


Figure 6.8: Diagram illustrating the event processing pipeline implemented in the criminal-activity monitoring scenario.

CQELS implicitly supported incremental semantics (IStream) as the default relation-to-stream operator. This made CQELS suitable for the tasks that required filtering of events streams since it can avoid duplicates. C-SPARQL, on the other hand, had more complete support for SPARQL and supported snapshot semantics (RStream). C-SPARQL was therefore better suited for tasks that required the output of an entire state, and tasks that required more expressive queries.

```
1 "01NRWS,camera_16,2017-03-01 03:27:00"
2 "Y6ZWNC,camera_21,2017-03-01 03:27:00"
3 "G4I8HF,camera_23,2017-03-01 03:27:00"
4 "RJCADE,camera_28,2017-03-01 03:28:00"
5 "CYL4MT,camera_35,2017-03-01 03:28:00"
```

Listing 6.10: Five observations from the ANPR stream represented as comma-separated strings.

The issue of event object boundaries (see Section 4.3) was handled partly by streaming the event objects as RDF graphs. This meant that any streamed event object was transferred in its entirety as a single message. However, since the graphs had to be decomposed into triples internally when consumed by the RSP engines we handled this by always specifying the full structure of the event objects in the query patterns, although this severely limited the possibility of generalizing the event patterns for different use-cases.

Event Producers

The ANPR data stream was generated from a file stored on disk. The producer generated an average of around 140 events/second, which is around 20 times the streaming rate of the data available in VALCRI. However, due to the temporal granularity of the events they were delivered in bursts once per minute, causing spikes of high stream velocities. ActiveMQ here acted as a message buffer that allowed the stream consumers to ingest the data at a slower rate. Listing 6.10 shows 5 observations from the ANPR stream.

Event Processing Agents

The RDF adapter converted the stream of ANPR observations into on-the-fly using Apache Jena[15]. Listing 6.11 shows the first ANPR observation from Listing 6.10 expressed as RDF. The vehicle identifier is a blank node to account for the fact that many crimes are committed using stolen vehicle registration plates. In many cases, it is fair to assume that the registration plate number does indeed represent a specific vehicle; however, we here required this to be a conscious decision in the processing pipeline. The RSP queries in this section are represented using RSP-QL, but the CQELS-QL and C-SPARQL translations are listed in Appendix B.

Blacklisted vehicles were filtered out using a CQELS query. The query was defined to match the entire vehicle observation to ensure that the event-object boundary was respected. To avoid the risk of missing an event due to delays the window range was set to 1 second. No window step parameter

[15] `https://jena.apache.org`

```
@prefix event:  <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#> .
@prefix anpr: <http://ns.valcri.org/ontology/anpr#> .
@prefix camera: <http://ns.valcri.org/data/camera/> .
@prefix vehicle: <http://ns.valcri.org/data/vehicle/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

_:b0
   a anpr:VehicleObservation ;
   anpr:hasCamera camera:camera_16  ;
   anpr:hasObservationTime "2017-03-01T03:27:00+00:00"^^xsd:dateTime ;
   anpr:hasVehicle _:b1 .

_:b1
   a anpr:Vehicle ;
   anpr:hasRegno "01NRWS" .
```

Listing 6.11: The first ANPR vehicle observation from Listing 6.10 represented as RDF.

was used to allow the query to match incoming events as quickly as possible. CQELS maintained consistent behavior and sub-second delays up to almost 1000 events/second. The resulting stream was reduced to only a small fraction of the original stream, averaging at less than 1 event/second. The query represented using RSP-QL query can be seen in Listing 6.12.

Observations of persons that were suspects in some crime in the dataset were generated using a CQELS query. The query matched vehicle observations where the owner of the vehicle person was a suspect in some crime. The query binds a blank node to a variable in the query body; however, this severely impacts the performance of CQELS, bringing down the average throughput to less than 10 events/second. Replacing the variable with a blank node defined in the construct result had the unintended effect of binding all events to the same blank node, although it brought up throughput to the levels reported in the previous query. This poor performance was assumed to be a bug in the current version of the engine. The query is represented as an RSP-QL query in Listing 6.13.

The vehicle routes of blacklisted vehicles were captured using a C-SPARQL query that leveraged the previously filtered stream. A route was defined as all observations of a particular vehicle within the last 30 minutes, capturing the start and stop of the majority of all trips in the stream. An alternative would have been to use a larger temporal window and put a duration threshold between the observations to mark the end of a trip; however, vehicles could travel along roads where no cameras were installed, thus creating gaps with respect to what could be observed. A route was defined to be a composite event object, and all referenced observations were included in the produced event. C-SPARQL maintained a consistent performance of up

```
PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>

REGISTER STREAM :blacklisted AS

CONSTRUCT ISTREAM
  {
    ?observation
       a anpr:VehicleObservation ;
       anpr:hasCamera ?camera ;
       anpr:hasObservationTime ?time ;
       anpr:hasVehicle ?vehicle .
    ?vehicle
       a anpr:Vehicle ;
       anpr:hasVehicleStatus anpr:Blacklisted ;
       anpr:hasRegno ?regno .
  }
FROM NAMED WINDOW :w ON :anpr [RANGE PT1S]
WHERE
  { WINDOW :w
      { ?observation
           a anpr:VehicleObservation ;
           anpr:hasCamera ?camera ;
           anpr:hasObservationTime ?time ;
           anpr:hasVehicle ?vehicle .
        ?vehicle
           a anpr:Vehicle ;
           anpr:hasRegno ?regno
      }
    _:b0
       rdf:type anpr:Vehicle ;
       anpr:hasRegno ?regno ;
       anpr:hasVehicleStatus anpr:Blacklisted
  }
```

Listing 6.12: RSP-QL query filtering out observations of blacklisted vehicles.

to around 100 events/seconds. The query is represented using RSP-QL in Listing 6.14.

Detecting possible rendezvous between persons of interest was also done using a C-SPARQL query. The query generated a rendezvous event when two persons were observed that were connected to the same crime and were spotted within 100 meters of each other in a 10 minute window. The calculated distance between cameras was used for the spatial filter, while the size of the window was used to provide the temporal boundaries. As in the previous query, the C-SPARQL query maintained a consistent performance up to around 100 events/seconds. Listing 6.15 shows the query represented using RSP-QL.

Event Consumers

The pipeline had, at the time of writing, no visual component that was part of the main VALCRI user-interface. The implementation, however, offered a web service from which the results could be retrieved by clients.

```
PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

REGISTER STREAM :personsofinterest AS

CONSTRUCT ISTREAM
  {
    ?personObservation
       a anpr:PersonObservation ;
       anpr:observationOfPerson ?person ;
       anpr:derivedFrom ?observation ;
       anpr:hasObservationTime ?time .
    ?observation
       a anpr:VehicleObservation ;
       anpr:hasCamera ?camera ;
       anpr:hasObservationTime ?time ;
       anpr:hasVehicle ?vehicle .
    ?vehicle
       a anpr:Vehicle ;
       anpr:hasRegno ?regno .
  }
FROM NAMED WINDOW :w ON :anpr [RANGE PT1S]
WHERE
  { WINDOW :w
      { ?observation
           a anpr:VehicleObservation ;
           anpr:hasCamera ?camera ;
           anpr:hasObservationTime ?time ;
           anpr:hasVehicle ?vehicle .
        ?vehicle
           a anpr:Vehicle ;
           anpr:hasRegno ?regno
      }
    _:b0
       rdf:type anpr:Vehicle ;
       anpr:hasRegno ?regno ;
       anpr:ownedBy ?person .
    ?crime
       a anpr:Crime ;
       anpr:hasSuspect ?person .
    BIND(BNODE() AS ?personObservation)
  }
```

Listing 6.13: RSP-QL query generating a stream of observations of persons that are suspects in some crime.

```
PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

REGISTER STREAM :vehicleroutes AS

CONSTRUCT RSTREAM
  {
    [] rdf:type anpr:VehicleRoute ;
       anpr:hasStartTime ?minTime ;
       anpr:hasEndTime ?maxTime ;
       anpr:hasVehicleObservation ?observation .
    ?observation
       a anpr:VehicleObservation ;
       anpr:hasCamera ?camera ;
       anpr:hasVehicle ?vehicle ;
       anpr:hasObservationTime ?time ;
       anpr:hasRegno ?regno .
  }
FROM NAMED WINDOW :w ON :blacklisted [RANGE PT30M STEP PT10S]
WHERE
  { WINDOW :w
      { { SELECT ?regno (MAX(?time) AS ?maxTime) (MIN(?time) AS ?minTime)
          WHERE
            { ?event anpr:hasVehicle/anpr:hasRegno ?regno .
              ?event anpr:hasObservationTime ?time
            }
          GROUP BY ?regno
        }
        ?observation
           a anpr:VehicleObservation ;
           anpr:hasCamera ?camera ;
           anpr:hasVehicle ?vehicle ;
           anpr:hasObservationTime ?time .
        ?vehicle anpr:hasRegno ?regno
      }
  }
```

Listing 6.14: RSP-QL query generating vehicle routes for blacklisted vehicles.

6.4.4 Summary and Discussion

The VALCRI scenario presents several challenges with respect to security and access-control, which are two topics that are rarely discussed in the context of RSP. ActiveMQ was used in the VALCRI infrastructure to manage inter-component communication for this exact reason. While several message frameworks today offer performance superior to that of ActiveMQ (e.g., Apache Kafka[16], and Akka[17]) the performance was clearly sufficient for the VALCRI requirements.

In the scenario, we leveraged two different RSP engines. CQELS offered great performance in terms of latency and throughput. In one of the queries, however, CQELS performed very poorly, and the average latency was almost two orders of magnitude higher than what was expected for the query. The

[16] https://kafka.apache.org/
[17] https://akka.io/

```
PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>

REGISTER STREAM :rendezvous AS

CONSTRUCT
  {
    [] a anpr:Rendezvous ;
       anpr:hasPersonObservation ?personObservation1 ;
       anpr:hasPersonObservation ?personObservation2 .
  }
FROM <http://ns.valcri.org/data>
FROM NAMED WINDOW :w ON :personsofinterest [RANGE PT5M STEP PT10S]
WHERE
  { ?personObservation1
       a anpr:PersonObservation ;
       anpr:observationOfPerson ?person1 ;
       anpr:derivedFrom ?observation1 .
    ?observation1
       a anpr:VehicleObservation ;
       anpr:hasCamera ?camera1 .

    ?personObservation1
       a anpr:PersonObservation ;
       anpr:observationOfPerson ?person2 ;
       anpr:derivedFrom ?observation2 .
    ?observation2
       a anpr:VehicleObservation ;
       anpr:hasCamera ?camera2 .
    ?crime
       anpr:hasSuspect ?person1 ;
       anpr:hasSuspect ?person2
  }
```

Listing 6.15: RSP-QL query generating possible rendezvous between two suspects some crime.

query in question used the `BIND` function, which therefore assume must be related to a bug in some part of the implementation.

The implemented pipeline was, with the exception of the above mentioned CQELS query, sufficient to cope with the full rate of the ANPR data stream, which in the experiment was streamed at approximately 10 times the original VALCRI velocity. However, efficiently integrating the RSP processing with large scale background information remains a challenge.

6.5 Scenario 3: Electronic Healthcare Monitoring

In the third scenario, we focused on some of the stream reasoning requirements of the E-care@home project (see Section 3.1.2). The goal was to specifically target the problem of making sense of the structured data arriving from sensors installed in the home. With respect to the total volume and velocity of the steamed data per user, the system requirements are much less demanding than in Scenario 2; however, smart-home occupants are expected to share

some of the infrastructure components in an implemented system. To put this scalability concern in perspective: A system including only 50 users, where each home is equipped with an average of 20 sensors generating data every second, would generate a total of 70 million readings daily. This is around two times the total volume generated by all ANPR cameras installed in England, Wales, Scotland, and Northern Ireland combined[18].

The sensor data in E-care@home is sometimes preprocessed in a sensor node, generating what is referred to as a *sensor feature stream*. A sensor feature stream does not represent the raw values reported by the sensors but rather represents a feature that can be inferred from the original values. For example, a set of pressure sensors attached to the legs of a chair could be used to generate a feature stream that indicated whether the chair was occupied or not. For the remainder of this section, we will use the terms sensor stream and sensor feature stream interchangeably.

The reasoning capabilities in the current RSP implementations are very limited, and in E-care@home we aim to overcome this limitation by adding an external reasoner. Answer Set Programming (ASP) is a declarative programming paradigm, where the given search problem is represented by a logic program where the *answer sets* correspond to the solutions. In E-care@home, we employed an incremental ASP solver (hereafter referred to as the ASP reasoner) to perform reasoning over time-series data, and the rules of the logic program was generated from an ontology at runtime [3]. An interesting aspect of this reasoner is the fact that it had support for representing both ontologies and data as RDF, which made it well suited to be used in combination with RSP technologies.

One of the main challenges with applying reasoning over streaming data is that new observations can provide new inferences as well as invalidate previous ones. While the ASP reasoner was reasonably efficient at this task but it had three major drawbacks: (1) the reasoner was unable to efficiently partition the incoming stream into time-series, (2) noise in the data could cause contradicting information to be provided within the same time-series, and (3) there was no way of reducing the volume of the consumed streams before applying reasoning, which could cause delays and backlogs in the reasoner. A time-series in this context is similar to a stream window but requires the temporal information to be discretized and represented using the timestamp approach.

The work in this scenario was still ongoing during the time of writing, and a lot of work still remained to be don. We here describe the data, ontologies, and the basic infrastructure of the scenario. The goal was to reduce the inconsistencies and volume of two sensor streams through filtering and aggregation, and show how RSP can partition data for the ASP reasoner.

[18]At present, the ANPR cameras in the United Kingdom submit 25–35 million records daily (`http://www.npcc.police.uk/FreedomofInformation/ANPR.aspx` visited on 2017-10-09)

6.5.1 Data

The data streams in E-care@home were generated from sensors installed in an experimental apartment. The sensors include pressure sensors in furniture, motion sensors, a sensor visually checking the luminosity of the TV, and more. Data was collected by asking persons to act out different scenarios in the apartment, creating a set of reference activities of the occupant. An illustration of the apartment can be seen in Figure 6.9.

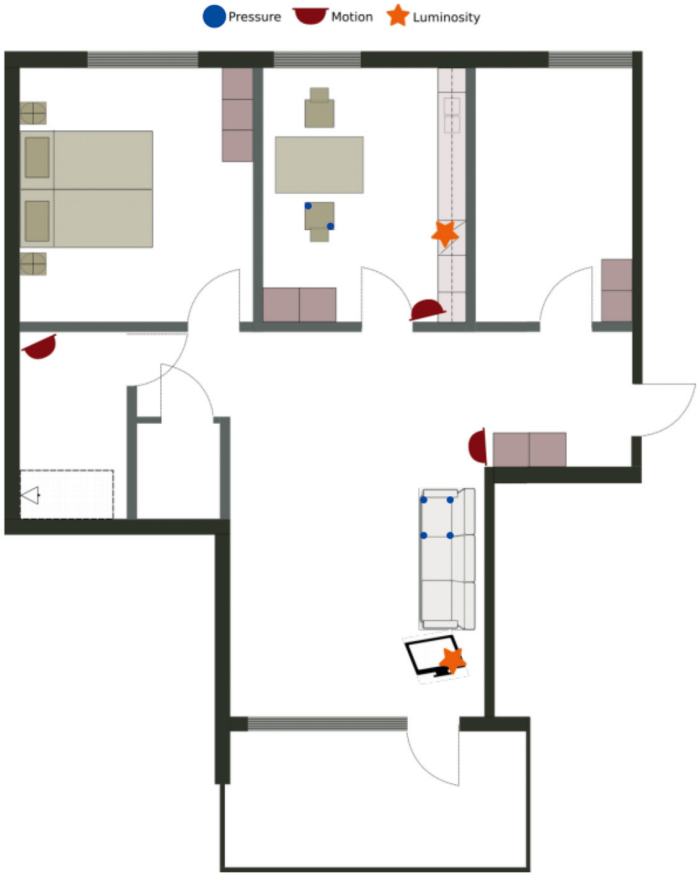


Figure 6.9: Illustration of the experimental apartment showing the location of furniture and some of the installed sensors [4].

In the scenario, we include all sensor streams but we focus specifically on the sensors related to the TV and couch, and we attempted to infer whether a person was engaged in watching TV or not. In the living room, there was a sofa equipped with pressure sensors, which are preprocessed to produce a feature stream of true/false values. The pressure sensors occasionally report

incorrect observations, especially when someone is not sitting perfectly still. This can result in situations where the occupant can appear to repeatedly sit down and stand up, and may even cause the person to appear in two locations at once.

All sensor streams in the scenario have very simple formats. Each data point contains a nanosecond timestamp, and a value or list of values. Metadata about the stream provides a sensor id and the reported value type. In Table 6.3 and 6.4, examples of the TV-sensor stream and the couch sensor stream are presented in tabular form. It is worth noting the difference between the values recorded by the two streams. The first was represented as a raw sensor values while the latter has been preprocessed to produce as a sensor feature stream.

Table 6.3: A sample from the TV-sensor stream showing the measured luminosity of a TV diode. At reading number 4 the TV appears to have been switched off.

Timestamp	**Int 32**
1462353995977128982	34
1462353997001213073	33
1462353998025161027	34
1462353999049177885	9
1462354000073513984	4
1462354001096296072	3

Table 6.4: A sample from the couch sensor stream showing if the couch appears to be occupied or not. At reading number 4 the occupant no longer appears to be sitting in the couch.

Timestamp	**Boolean**
1462354002164657115	true
1462354003191551923	true
1462354004209553003	true
1462354005230532884	false
1462354006256860017	false
1462354007286078929	false

6.5.2 Ontologies

Unlike the two previous scenarios, the SmartHome Event ODP[19] was used as the starting point for our event representation. The pattern is aligned with both the DUL ontology and the SSN ontology. The ontology is still under

[19] `http://ontologydesignpatterns.org/wiki/Submissions:SmartHome_Event`

development and for the scenario we chose to leverage the recently released SSN core ontology (SOSA) described in the most recent version of the SSN ontology[20], rather than the full SSN.

The simplified event pattern is shown in Figure 6.10. A `Manifestation` represents an explicit event, which is roughly equivalent to what is referred to as a `SimpleEventObject` in the Event ODP (i.e., an event that has been observed rather than inferred).

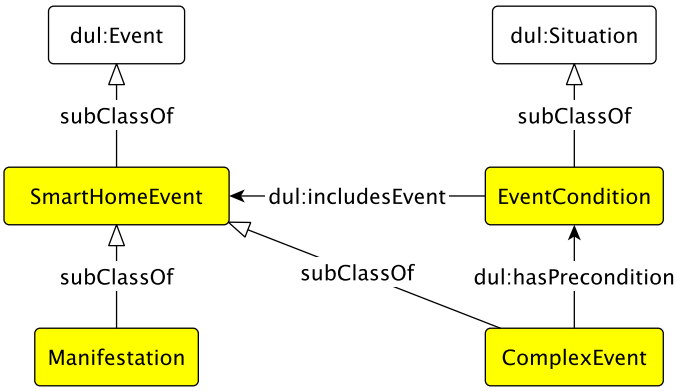


Figure 6.10: Illustration of the main parts of the SmartHome Event ODP.

6.5.3 Implementation

Apache Kafka[21] (hereafter referred to simply as Kafka) was used to manage the communication in the implementation. Kafka was since it is both scalable, provides fault-tolerance, and provides APIs for integration many programming platforms and environments. Kafka and Apache ZooKeeper[22,23] were run in separate Docker containers. The dockerized containers allow us to stick to single-machine deployment for the first experiments, while allowing us to transition to a full-scale cluster environment at a later time.

The main abstraction in Kafka is the topic and it uses the a publish-subscribe pattern. As with ActiveMQ, one or more producers can push data to a topic to which one or more consumers subscribe. Figure 6.11 illustrates a basic Kafka configuration. An interesting aspect of this framework is that it can persists all data, and that it scales beyond what fits into main memory without causing the typical a drop in performance typical for such frameworks.

[20] `https://www.w3.org/TR/vocab-ssn/`
[21] `https://kafka.apache.org/`
[22] `https://zookeeper.apache.org/`
[23] ZooKeeper is a distributed synchronization service required in recent version of Kafka

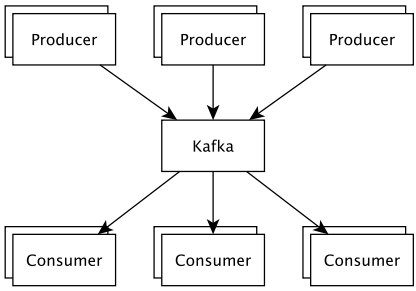


Figure 6.11: Diagram illustrating a simple Kafka configuration. Three groups of producers push messages to Kafka and consumed by three consumer groups.

Figure 6.12 shows the basic relationships between the different components in the event processing pipeline. All sensor data was replayed [24] and all sensor streams were converted into JSON, and the relevant streams were converted into RDF on-the-fly by the RDF adapter[25], and republished under a different topic. The RDF streams were modeled as sequences of RDF graphs; however, the C-SPARQL engine requires streams of RDF triples and the time-annotated structure proposed in RSP-QL cannot be supported. Instead, events were modeled within their individual graphs. Wrappers were created for the C-SPARQL engine, its internal stream representation, and a result listener was created to allow it to receive data from and push data to Kafka. An instance of the C-SPARQL engine[26] was created and set to listen the couch and TV sensor streams. Two C-SPARQL queries were set up to reduce the noise and volume of the data the data in the two streams. Finally, the partitioned data, along with the original streams were used by the ASP reasoner to infer current the activity of the occupant.

Event producers

The system includes a single event producer, which was used to internalize all sensor streams. In the internalization step, the data was converted into JSON based on the metadata provided for the sensor stream.

The sensor streams within a single occupant's home were published as single topic, and the key-value message structure of Kafka was used to associate each value with a particular stream URI via the key. The rationale behind this was that clashes between topics used in different homes can be prevented

[24] In the experimental setup, this is done from an intermediary database, allowing data to replayed from specific points in time.

[25] The metadata will be provided as part the ontology but was hard coded for the purpose of the first experiments.

[26] C-SPARQL engine version 0.9.7, compiled from the source in October 2016.

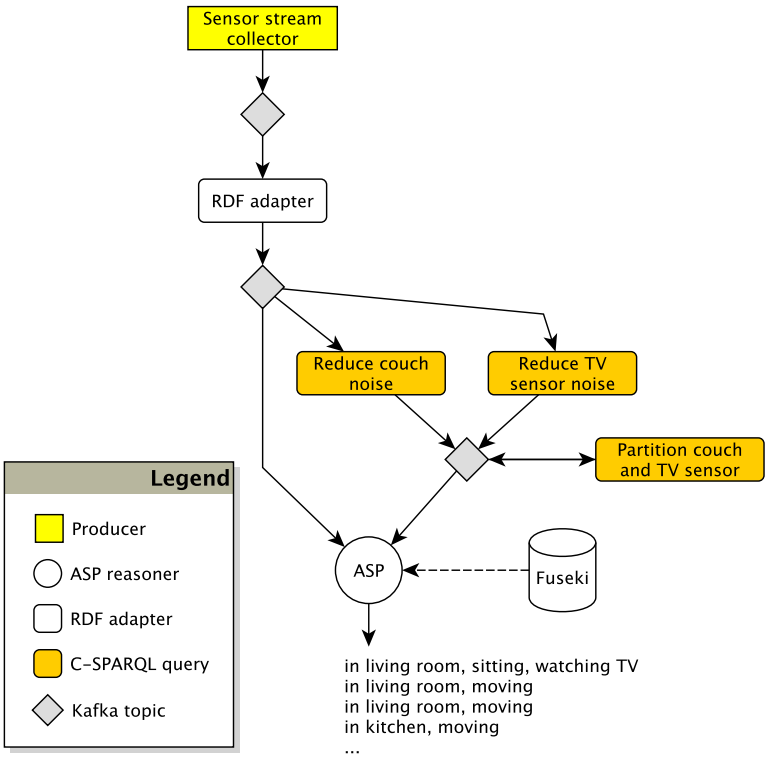


Figure 6.12: Diagram illustrating the event processing pipeline implemented in the electronic healthcare monitoring scenario.

in a multi-occupant setting, and it allows us to standardize the naming conventions of streams within a pipeline across multiple homes. An alternative approach would naturally be to publish each stream as a unique topic. This pattern was used throughout the entire pipeline, and all RSP components were set to publish to the same topic.

Event Processing Agents

The RDF adapter continuously translated the raw JSON stream into RDF. As a result of operating on the raw sensor stream, the translation was recognized as a potential bottleneck in the event processing pipeline. When tested, however, stable conversion rates were maintained at up to around 15,000 records per second, meaning that a single instance would easily be able to handle more than 50 experimental apartments in parallel. Listing 6.16 shows two examples from the TV and couch sensor streams represented as RDF.

```
@prefix : <http://ecare.org/sensor#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dul: <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#> .
@prefix time: <http://w3id.org/ecareathome/patterns/timeinterval.owl#> .
@prefix event: <http://w3id.org/ecareathome/patterns/event.owl#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .

[] a event:Manifestation ;
   dul:isObservableAt [ a dul:TimeInterval ;
                        time:hasUpperTimeStampValue 1462353996977128982 ;
                        time:hasLowerTimeStampValue 1462353995977128982 ] ;
   sosa:isObservedBy    :tv_sensor ;
   sosa:resultTime      "2016-05-04T11:26:35+0200"^^xsd:dateTime ;
   sosa:hasSimpleResult "34"^^xsd:int .

[] a event:Manifestation ;
   dul:isObservableAt [ a dul:TimeInterval ;
                        time:hasUpperTimeStampValue 1462354003164657115 ;
                        time:hasLowerTimeStampValue 1462354002164657115 ] ;
   sosa:isObservedBy    :couch_sensor ;
   sosa:resultTime      "2016-05-04T11:26:42+0200"^^xsd:dateTime ;
   sosa:hasSimpleResult true .
```

Listing 6.16: Two records from the TV and couch sensor streams represented as RDF.

The C-SPARQL engine was setup to listen to the TV and couch sensor streams. The aggregate queries used to simplify the two streams are represented as RSP-QL in Listings 6.17 and 6.18. The queries that resulted from the translation of the queries into C-SPARQL are included in Appendix B. The initial test showed consistent performance for these queries up to around 100 events per second (1300 triples/second), while higher stream rates caused the C-SPARQL engine to become unresponsive.

The two simplified streams were used to generate the time-series that was used by the ASP reasoner. The query is described as an RSP-QL query in Listing 6.19 and generates a new result every 10 seconds.

Finally, the ASP reasoner pulls the ontology and configuration information from the RDF store. The time-series stream is processed along with some sensor data from the original streams. At the time of writing, the full setup of this step in the process had only been partially completed and this step has not yet been verified.

```
PREFIX  :     <http://ecareathome.org/stream#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  dul:  <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>
PREFIX  time: <http://w3id.org/ecareathome/patterns/timeinterval.owl#>
PREFIX  event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX  sosa: <http://www.w3.org/ns/sosa/>

REGISTER STREAM :cleantv AS

CONSTRUCT RSTREAM
  {
    _:c0 rdf:type event:ComplexEvent ;
         dul:isObservableAt _:c1 ;
         sosa:isObservedBy ?sensor ;
         sosa:hasSimpleResult ?value .
    _:c1 rdf:type dul:TimeInterval ;
         time:hasUpperTimeStampValue ?maxTime ;
         time:hasLowerTimeStampValue ?minTime .
  }
FROM NAMED WINDOW :w ON :tv [RANGE PT10S STEP PT10S]
WHERE
  { { SELECT  ?sensor (AVG(?value) AS ?avg) (MAX(?upper) AS ?maxTime)
              (MIN(?lower) AS ?minTime)
      WHERE
        { WINDOW :w
            { _:b2  sosa:isObservedBy     ?sensor ;
                    sosa:hasSimpleResult  ?value ;
                    dul:isObservableAt    _:b1 .
              _:b1  time:hasUpperTimeStampValue  ?upper ;
                    time:hasLowerTimeStampValue  ?lower
            }
        }
      GROUP BY ?sensor
    }
    BIND(if(?avg < 20, false, true) AS ?value)
    BIND(now() AS ?time)
    FILTER(BOUND(?sensor))
  }
```

Listing 6.17: RSP-QL query generating a stream of Boolean values constructed using an aggregate over the TV sensor stream. If the average luminosity of the observations within the window was less than a 20 the query reported true (TV is on), and otherwise false (TV is off).

```
PREFIX  :     <http://ecareathome.org/stream#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  dul:  <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>
PREFIX  time: <http://w3id.org/ecareathome/patterns/timeinterval.owl#>
PREFIX  event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX  sosa: <http://www.w3.org/ns/sosa/>

REGISTER STREAM :cleancouch AS

CONSTRUCT RSTREAM
  {
    _:c0 rdf:type event:ComplexEvent ;
         dul:isObservableAt _:c1 ;
         sosa:isObservedBy ?sensor ;
         sosa:hasSimpleResult ?value .
    _:c1 rdf:type dul:TimeInterval ;
         time:hasUpperTimeStampValue ?maxTime ;
         time:hasLowerTimeStampValue ?minTime .
  }
FROM NAMED WINDOW :w ON :couch [RANGE PT10S STEP PT10S]
WHERE
  { { SELECT  ?sensor (MAX(?upper) AS ?maxTime)
              (MIN(?lower) AS ?minTime)
      WHERE
        { WINDOW :w
            { _:b0  sosa:isObservedBy     ?sensor ;
                    dul:isObservableAt    _:b1 .
              _:b1  time:hasUpperTimeStampValue  ?upper ;
                    time:hasLowerTimeStampValue  ?lower
            }
        }
      GROUP BY ?sensor
    }
    { SELECT  ?sensor (COUNT(?sensor) AS ?isTrue)
      WHERE
        { WINDOW :w
            { _:b2  sosa:isObservedBy     ?sensor ;
                    sosa:hasSimpleResult  true
            }
        }
      GROUP BY ?sensor
    }
    { SELECT  ?sensor (COUNT(?sensor) AS ?isFalse)
      WHERE
        { WINDOW :w
            { _:b3  sosa:isObservedBy     ?sensor ;
                    sosa:hasSimpleResult  false
            }
        }
      GROUP BY ?sensor
    }
    BIND(( ?isTrue / ( ?isTrue + ?isFalse ) ) AS ?avg)
    BIND(if(?avg >= 0.9, true, if(?avg < 0.1, false, ?avg)) AS ?value)
    BIND(now() AS ?time)
  }
```

Listing 6.18: RSP-QL query generating aggregated data from the couch sensor stream. If 90 % of the observations concur (true/false) within the window that value is reported. If there is no such clear majority the probability that the observation is true is reported instead.

```
PREFIX : <http://ecareathome.org/stream#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>

REGISTER STREAM :partitioned AS

CONSTRUCT RSTREAM {
   ?subject ?predicate ?object .
}
FROM NAMED WINDOW :w1 ON :tv_cleaned [RANGE PT10S STEP PT10S]
FROM NAMED WINDOW :w2 ON :couch_cleaned [RANGE PT10S STEP PT10S]
WHERE {
   ?event a event:ComplexEvent ;
          (!<>)* ?subject .
   ?subject ?predicate ?object .
}
```

Listing 6.19: RSP-QL query generating time-series for the ASP reasoner every 10 seconds.

```
PREFIX  :     <http://ecareathome.org/stream#>
PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX  event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  sosa: <http://www.w3.org/ns/sosa/>

REGISTER STREAM :partioned AS

CONSTRUCT RSTREAM {
   ?subject ?predicate ?object .
}
FROM NAMED WINDOW :w1 ON :tv_cleaned [RANGE PT10S STEP PT5S]
FROM NAMED WINDOW :w2 ON :couch_cleaned [RANGE PT10S STEP PT5S]
WHERE {
   ?event a event:ComplexEvent ;
          (!<>)* ?subject .
    ?subject ?predicate ?object .
 }
```

Listing 6.20: RSP-QL query generating aggregated data from the couch sensor stream. If 90% of the observations concur (true/false) within the window that value is reported. If there is no such clear majority, the probability of the observation being true is reported instead.

Event Consumers

The system presently has no sophisticated event consumer for the generated events. The intention is to have the high-level activities that are inferred by the ASP reasoner persisted in the RDF store or in a relational database, from which the data can be retrieved. The storage can then represent a simple persistent state, which can be continuously updated based on the results of the ASP reasoner. The inferred activities can also be given textual representations as seen in Listing 6.21.

```
in living room, sitting, watching TV
in living room, moving, watching TV
in living room, sitting, watching TV
in living room, moving
in bathroom, moving
```

Listing 6.21: Example of textual representations of inferences drawn by the ASP reasoner.

6.5.4 Summary and Discussion

In the scenario, the TV and couch sensor streams are aggregated over 10 second windows. This effectively removes around 90% of the streamed volume, and the queries also reduce the noise considerably within that time interval. However, it also impacts the precision of the values reported by the sensor processes. For example, if the TV is in fact turned on and off continuously for an extended period of time it could signify an interesting behavior, which would not be detectable in the cleaned stream. The original data stream, however, remains available to any component within the pipeline.

The first attempts at combining ASP and RSP was the StreamRule framework [39]. Despite preliminary investigations showing considerable promise. However, no detailed evaluation has been made available that tests the performance of this framework in practice [38].

The incremental ASP reasoner used in the E-care@home project provides the ability to convert between OWL-DL ontologies and the rules required for the ASP program. However, the expressiveness of the reasoner leads to high processing complexity, and if the reasoning cannot be completed before more data arrives performance deteriorates. Due to its inability to efficiently partition and filter the incoming data, the reasoner will therefore struggle when it comes to high-volume streams. RSP systems, on the other hand, can efficiently query, filter, enrich, and aggregate continuous streams of RDF data. In the future, we intend to benchmark the implemented infrastructure and components within it. In particular, we want to investigate the possibility of scaling processing by providing more than a single instance of a particular component (e.g., using more than one RSP engine instance or more than one ASP reasoner).

7 Discussion

Event models on the Semantic Web have not been developed with complex event processing (CEP) in mind. In response to this, we defined a set of requirements for modeling events in this context, which we then used as a basis for creating a new event model that was aligned with some of the existing event ontologies. The model was designed as a Content ODP, which can be viewed as a small reusable ontology that solves a specific domain-oriented problem, and it focused specifically on features such as payload support, relationships between events, event object encapsulation, and querying ability. The ontology itself can be used independently, without requiring any additional ontology imports. This allows applications to use a minimal set of base classes and properties for modeling of complex events. However, the goal of the model was to promote specialization and extensions for specific use-cases or scenarios.

The event model differentiates between events that occur in the real world, and the event objects, which are representations of events. This allows two parallel structures for events where models used by systems and humans can be very different. This separation can help manage the dissonance that often exists between reality and model of reality.

The event model also supports systems that need to distinguish between event headers (i.e., the known parts of the events), and event payloads. The payload (or body) allows data of unknown structure to be added to an event object. This allows the payload to be used further down the processing

pipeline, while having a minimal effect on processing performance. This is currently not supported by any other related event model.

Two of the requirements listed in relation to the model remain to be evaluated; namely, understandability and usability. The use of property domain and range restrictions, high class/property ratio, and supplementary materials, such as documentation and illustrations should promote the understandability of the model. Additionally, the model defines almost 90% fewer axioms than the SSN and the Event-Model-F ontologies. The reduced complexity should help reduce the semantic interactions within the model, which should make it easier to both understand and extend. We used the model in two of the scenarios described in Chapter 6, but an objective assessment of the model with respect to these requirements will be required in the future.

There are a number of additions to the model that should be considered for future iterations. Data of known structure will typically be added to the event via entity references, and the referenced entities are often not part of the event model itself. For example, in the VALCRI scenario references to crime reports and person records are not integrated with the event model, but references to such background knowledge from an event can, of course, be required. Providing a property for referencing such external content could be a valuable addition to the event model, especially with respect to query generalizability. Another possibility would be to extend the model to support temporal and spatial relationships between events, which could be accomplished by using, for example, the temporal properties in the OWL Time Ontology[1]. Finally, limiting the model to a single OWL 2 profile should, in hindsight, have been made a higher priority, since this would have reduced complexity further and provided better opportunities for efficient reasoning.

Taking event object boundaries into consideration is crucial in event processing. The issue becomes very prominent when using RSP systems that support only RDF triple streams. Although these systems consider triples with the same timestamps to have occurred at the same time, the triples arrive at the engine in sequential order, which can cause queries to match partial events. We have discussed two approaches that can be applied when working with RDF triple streams, and have showed that both can be applied in some of the current RSP implementations. RDF graph streams do not suffer from this issue to the same extent, since all the triples belonging to an event are delivered in a single transaction. This suggests that the future RSP-QL standard will cover most of these boundary issues, at least if we assume that the event objects are captured within a single named graph.

The time-annotated named graphs proposed in RSP-QL offer new ways of querying, encapsulating, and referencing event objects. If an event is limited to its own graph then matching the entire event object's structure is simply a matter of returning the entire event graph, regardless of the underlying

[1] `https://www.w3.org/TR/owl-time`

model complexity. However, this view may also force us to reconsider how the event objects themselves are referenced. If an event is captured in a named graph then a reference to a part of that event from a different event needs to reference not only the event identifier but also the named graph in which it is contained. Another alternative would be to use the named graph itself as the event identifier. The implications of the different approaches will need to be considered in future work.

Defining any non-trivial query for use in streaming contexts is both time-consuming and error prone. With underlying data shifting constantly, predicting the outcome is often difficult, and for queries that match only rarely, each iteration in the query development phase can take considerable time. In Chapter 5, we presented an extension to SPIN to support query templates for RSP. To the authors' knowledge, this is the first attempt at supporting query templates in the RSP domain. Previously, reuse of queries has primarily required manual query manipulation, although string replacements have occasionally been used to mimic some basic parameterization features. While there are many ways of implementing support for parameterized queries, such as general string-based template frameworks, RSP-SPIN provides support for both parameter constraints and sharing via standard Semantic Web technologies.

While RSP-QL was used as a starting point in defining the extended SPIN model, the RDF representation in itself is query language agnostic. This means that it can be translated into any of the query languages that can be modeled using RSP-QL. This can greatly reduce the effort required in moving between different RSP implementations. The RSP-QL draft has not yet been finalized and additions to the query language may be required to the RSP-SPIN vocabulary. We have also considered various extensions that go beyond what is currently supported in RSP-QL, such as support for temporal operators. Many of these additions can be made to the model without generating any conflicts with the current implementation.

Parameterization is presently the primary use of RSP-SPIN, but other potential application areas will be explored in the future. For example, query templates can help provide support for data access control by exposing information using predefined templates. The model can also be used to provide configuration information to RSP engines, such as setting the query evaluation policy, limiting maximum memory allocation, or provide time-out policies.

From a CEP perspective, the extensions that are currently being discussed relate to how query templates can be leveraged to create query pipelines declaratively. For example, queries could be designed to filter events in a high-volume stream and the results could be used in more computationally expensive queries. This is a common requirement in event processing using RSP, and a pattern that appeared in all three scenarios presented in Chapter 6. However, guaranteeing that compatibility exists between queries executed in

such pipelines, with engines providing different operational semantics, remains an open challenge.

Existing RSP implementations have several limitations with respect to query expressiveness. In standard SPARQL, extensions can often be provided to add support for specific tasks. A similar approach could also be applied in RSP; however, capturing all necessary CEP features in a single RSP system is presently not feasible. In Chapter 6, we proposed a general architecture for supporting semantic complex event processing. The typical structure of event processing networks as they are described in the CEP domain was used as a starting point. In this context, RSP systems can be viewed as generators of event processing agents (EPAs), where each query is viewed as a single semantically enabled component. This view differs considerably from previous approaches, where RSP systems have been viewed as CEP systems in their own right.

The conceptual architecture provides a basis for viewing RSP implementations in a broader context, allowing complementary components to be added as needed. We implemented the system for three different scenarios: (1) traffic-incident monitoring, (2) criminal-activity monitoring, and (3) electronic healthcare monitoring. The implementations differed mainly with respect to how message streams were communicated between components, but all scenarios demonstrated features that would have been difficult (or impossible) to implement using a single RSP engine in isolation. For example, in the traffic-incident monitoring scenario we added external components to calculate spatial and temporal distances, something that would not have been possible in any of the existing RSP systems. In the criminal-activity monitoring scenario, we leveraged the expressiveness of two separate RSP engines, where CQELS was used to filter a stream of vehicle observations, and C-SPARQL was used to process streams of considerably lower volumes. In the electronic healthcare monitoring scenario, we employed a separate component for performing temporal reasoning, and used C-SPARQL to clean and partition the sensor data. The development of this framework is still under active development in the E-care@home project, and performance comparisons with respect to throughput, latency, memory consumption, and scalability between RSP, ASP, and the two in combination have been planned for the immediate future.

Finally, the implementations illustrate that RSP systems do not need to be viewed in isolation and that other applications, systems, and tools can provide a great complement where RSP falls short. While this does not provide a solution for distributing and scaling the actual RSP processing, which is a problem in its own right, it enables RSP processing to be scaled up by supporting multiple RSP engine instances in a common pipeline. The intermediate messaging frameworks provided protection against burst of data by allowing the data to be buffered, and increased the potential scalability considerably.

8 Conclusion

Semantic Web technologies support integration of semantically-annotated data represented in heterogeneous formats; however, they have traditionally been focused on more or less static data. Recently, RSP systems have been developed in response to the increased need for processing of streaming information. The model underlying the different implementations differ in how they extend SPARQL for continuous processing and with respect to their underlying assumptions about the formats of streaming data. The RSP Group is currently in the process of defining a common model for producing, transmitting, and querying RDF streams. Although RSP-QL has not yet been proposed as a standard, it has been used as the reference language throughout this thesis.

The thesis work has focused on three main research questions. In response to the first research question (RQ1), a new event model called the Event ODP was introduced.

RQ1 How can events be modeled to support event abstraction and querying in RSP systems to assist in semantic complex event processing?

Unlike previous approaches, the Event ODP was based on requirements identified from the CEP domain, and developed specifically to support event modeling and querying in semantic complex event processing applications. We illustrated how the new model can be queried using standard SPARQL and how it supports event pattern generalization in queries. We also proposed

ways in which event object boundaries can be respected using different types of stream punctuation strategies, and showed how these can be used to provide workarounds in existing RSP implementations.

The second research question (RQ2) was answered by the development of an extension of SPIN to support RSP queries.

RQ2 How can RSP queries be abstracted to support reuse and maintenance of queries?

The extended model supported abstraction and parameterization of RSP queries and the model, which was fully compatible with standard SPARQL, supports all the constructs currently proposed as part of the RSP-QL syntax. The model supported efficient reuse of queries in the form of query templates, and was compatible with three popular RSP languages via a set of query serializers. The extension and the accompanying API were released as open-source[1] to promote the use of RSP technology within and outside the RSP community.

Finally, we presented an architecture for semantic complex event processing. Rather than viewing RSP engines as standalone event processing systems, the architecture lets us view RSP queries as EPAs. Three implementations of this architecture helped answer the final research question (RQ3).

RQ3 What are the limitations of current RSP technologies with respect to recurring decision-making tasks in the context of semantic complex event processing?

We showed that the architecture provided a straightforward method for creating loosely-coupled event processing systems. The RSP engines used were able to handle the stream velocities in the scenarios; however, their limited expressiveness presented challenges in some cases. For example, none of the RSP engines supported geographic querying, and they typically lacked support for datetime arithmetic. These features are essential in many event processing scenarios, and although they can be handled by implementing task-specific EPAs this adds to the complexity of the pipeline, and each step introduces additional latency. Managing the communication using message-oriented middleware made it easy to integrate several instances of independent RSP engines within the same application pipeline, and drastically increased the potential scalability of the system compared with a minimalistic pipeline.

[1] `https://w3id.org/rsp/spin`

Appendix A

RSP-SPIN Vocabulary

```
@prefix : <http://w3id.org/rsp/spin#> .
@prefix sp: <http://spinrdf.org/sp#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://w3id.org/rsp/spin>
  rdf:type owl:Ontology ;
  owl:imports <http://spinrdf.org/sp> ;
.

# Register as
:hasOutputStream
  rdf:type rdf:Property ;
  rdfs:comment "Set the name of the result stream of a query."^^xsd:string
      ↪ ;
  rdfs:domain sp:Query ;
  rdfs:range rdfs:Resource, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.

# Output stream operator
:hasOutputStreamOperator
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the output stream operator of a
      ↪ query."^^xsd:string ;
  rdfs:domain sp:Query ;
  rdfs:range :OutputStreamOperator ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:OutputStreamOperator
  rdfs:comment "Stream output operator."^^xsd:string ;
  rdf:type rdfs:Class ;
  rdfs:subClassOf sp:Element ;
.
:Dstream
  rdf:type rdfs:Class ;
  rdfs:comment "States that the output stream operator is Dstream (i.e.,
      ↪ return results that are in the previous window but not the current
      ↪ one)."^^xsd:string ;
  rdfs:subClassOf :OutputStreamOperator ;
.
:Istream
  rdf:type rdfs:Class ;
  rdfs:comment "States that the output stream operator is Istream (i.e.,
      ↪ return results that are in the current window but were not present
      ↪ in the previous one)."^^xsd:string ;
  rdfs:subClassOf :OutputStreamOperator ;
.
:Rstream
  rdf:type rdfs:Class ;
  rdfs:comment "States that the output stream operator is Rstream (i.e.,
      ↪ return all results that are in the current window)."^^xsd:string ;
  rdfs:subClassOf :OutputStreamOperator ;
.
```

```
# Named window
:fromNamedWindow
  rdf:type rdf:Property ;
  rdfs:comment "Property to define a named window for a
     ↪ query."^^xsd:string ;
  rdfs:domain sp:Query ;
  rdfs:range :NamedWindow ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:NamedWindow
  rdf:type rdfs:Class ;
  rdfs:comment "A named window over a stream."^^xsd:string ;
  rdfs:subClassOf sp:ElementGroup ;
.
:windowNameNode
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the name of a window."^^xsd:string ;
  rdfs:range rdfs:Resource ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:streamUri
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the stream of a window."^^xsd:string ;
  rdfs:domain :NamedWindow ;
  rdfs:range rdfs:Resource, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:windowUri
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the name of a window."^^xsd:string ;
  rdfs:domain :NamedWindow ;
  rdfs:range rdfs:Resource ;
  rdfs:subPropertyOf sp:systemProperty ;
.

# Logical window
:LogicalWindow
  rdf:type rdfs:Class ;
  rdfs:comment "A logical window defined using range and step expressed as
     ↪ durations."^^xsd:string ;
  rdfs:subClassOf :NamedWindow ;
.
:logicalRange
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the duration of a logical
     ↪ range."^^xsd:string ;
  rdfs:domain :LogicalWindow ;
  rdfs:range xsd:duration, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:logicalStep
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the duration of a logical
     ↪ step."^^xsd:string ;
  rdfs:domain :LogicalPastWindow, :LogicalWindow ;
  rdfs:range xsd:duration ;
  rdfs:range sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.

# Logical past window
:LogicalPastWindow
  rdf:type rdfs:Class ;
```

```
  rdfs:comment "A logical past window defined using a lower and an upper
      ↪ time bound."^^xsd:string ;
  rdfs:subClassOf :NamedWindow ;
.
:from
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the start of a window relative to the
      ↪ current time."^^xsd:string ;
  rdfs:domain :LogicalPastWindow ;
  rdfs:range xsd:duration, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:to
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the end of a window relative to the
      ↪ current time."^^xsd:string ;
  rdfs:domain :LogicalPastWindow ;
  rdfs:range xsd:duration, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.

# Physical window
:PhysicalWindow
  rdf:type rdfs:Class ;
  rdfs:comment "A physical window defined using size and step expressed as
      ↪ number elements."^^xsd:string ;
  rdfs:subClassOf :NamedWindow ;
.
:physicalRange
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the size of a window expressed as number
      ↪ of elements."^^xsd:string ;
  rdfs:domain :PhysicalWindow ;
  rdfs:range xsd:integer, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.
:physicalStep
  rdf:type rdf:Property ;
  rdfs:comment "Property to set the step of a window expressed as number
      ↪ of elements."^^xsd:string ;
  rdfs:domain :PhysicalWindow ;
  rdfs:range xsd:integer, sp:Variable ;
  rdfs:subPropertyOf sp:systemProperty ;
.
```

RSP-SPIN Example Query

```
@prefix :      <http://w3id.org/rsp/spin#> .
@prefix ex:    <http://example.org#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix debs:  <http://debs2015.org/onto#> .
@prefix prov:  <http://www.w3.org/ns/prov#> .
@prefix sp:    <http://spinrdf.org/sp#> .

[ a sp:Construct ;
    sp:templates       ( _:b9 _:b0 ) ;
    sp:where           ( _:b7 _:b6 _:b2 ) ;
    :fromNamedWindow   [ a               :PhysicalWindow ;
                         :physicalRange  "1000"^^xsd:int ;
                         :physicalStep   "1000"^^xsd:int ;
                         :streamUri      ex:trips ;
                         :windowUri      ex:w1
                       ] ;
    :fromNamedWindow   [ a             :LogicalPastWindow ;
                         :from         "PT3H"^^xsd:duration ;
                         :logicalStep  "PT1M"^^xsd:duration ;
                         :streamUri    ex:trips ;
                         :to           "PT1H"^^xsd:duration ;
                         :windowUri    ex:w3
                       ] ;
    :fromNamedWindow   [ a              :LogicalWindow ;
                         :logicalRange  "PT2H"^^xsd:duration ;
                         :logicalStep   "PT1M"^^xsd:duration ;
                         :streamUri     ex:trips ;
                         :windowUri     ex:w2
                       ] ;
    :hasOutputStream          ex:my-stream ;
    :hasOutputStreamOperator  :Istream
  ] .

_:b0    sp:object     [ sp:varName  "time" ] ;
        sp:predicate  prov:atTime ;
        sp:subject    [ sp:varName  "g" ] .

_:b1    sp:object     [ sp:varName  "time" ] ;
        sp:predicate  prov:atTime ;
        sp:subject    [ sp:varName  "g" ] .

_:b2    a                :NamedWindow ;
        sp:elements      ( _:b3 ) ;
        :windowNameNode  ex:w3 .

_:b3    sp:object     [ sp:varName  "time" ] ;
        sp:predicate  prov:atTime ;
        sp:subject    [ sp:varName  "g" ] .

_:b4    sp:object     [ sp:varName  "time" ] ;
        sp:predicate  prov:atTime ;
        sp:subject    [ sp:varName  "g" ] .

_:b5    sp:object     [ sp:varName  "o" ] ;
        sp:predicate  [ sp:varName  "p" ] ;
        sp:subject    [ sp:varName  "s" ] .

_:b6    a                :NamedWindow ;
        sp:elements      ( _:b1 ) ;
        :windowNameNode  ex:w2 .
```

```
_:b7    a                 :NamedWindow ;
        sp:elements       ( _:b8 _:b4 ) ;
        :windowNameNode   ex:w1 .

_:b8    a                  sp:NamedGraph ;
        sp:elements        ( _:b5 ) ;
        sp:graphNameNode   [ sp:varName  "g" ] .

_:b9    a                  sp:NamedGraph ;
        sp:elements        ( _:b10 ) ;
        sp:graphNameNode   [ sp:varName  "g" ] .

_:b10   sp:object      [ sp:varName  "o" ] ;
        sp:predicate   [ sp:varName  "p" ] ;
        sp:subject     [ sp:varName  "s" ] .
```

RSP-QL Sample Queries

```
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  debs: <http://debs2015.org/onto#>

REGISTER STREAM :query1 AS

SELECT (count(?ride) AS ?rideCount)
FROM NAMED WINDOW :win ON :trips [RANGE PT1H STEP PT1H]
WHERE
  { WINDOW :win
      { ?ride debs:distance ?distance
        FILTER ( ?distance > 2 )
      }
  }
```

```
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  debs: <http://debs2015.org/pred#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

REGISTER STREAM :query2 AS

SELECT  ?location (count(distinct ?taxi) AS ?taxinumber)
FROM NAMED WINDOW :w ON :trips [RANGE PT1H STEP PT30M]
WHERE
  { ?location <type> :dropoffLocation
    WINDOW :w
      { ?taxi debs:dropoff ?location }
  }
GROUP BY ?location
HAVING ( ?taxinumber >= 20 )
```

```
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  debs: <http://debs2015.org/pred#>

REGISTER STREAM :query3 AS

CONSTRUCT ISTREAM
  { ?location debs:profit ?totalamount }
FROM NAMED WINDOW :w ON :s [RANGE PT30M STEP PT15M]
WHERE
  { { SELECT  (sum(?amount) AS ?totalamount) ?location
      WHERE
        { WINDOW :w
            { ?taxi debs:pickup ?location .
              ?location debs:amount ?amount
            }
        }
      GROUP BY ?location
      ORDER BY DESC(?totalamount)
      LIMIT   3
    }
  }
```

```
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  geo:  <http://www.opengis.net/ont/geosparql#>
PREFIX  wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX  geof: <http://www.opengis.net/def/geosparql/function/>
PREFIX  time: <http://www.w3.org/2006/time#>
PREFIX  units: <http://unit#>
PREFIX  debs: <http://debs2015.org/onto#>
PREFIX  geodata: <http://linkedgeodata.org/ontology/addr%3A>

REGISTER STREAM :query4 AS

SELECT  ?time ?district
FROM NAMED WINDOW :wind ON :rides [RANGE PT6H STEP PT6H]
WHERE
  { WINDOW :wind
      { ?ride debs:dropoff_latitude ?lat .
        ?ride debs:dropoff_longitude ?lng .
        ?ride debs:dropoff_datetime ?time .
        ?time time:hour ?drop_hour .
        ?feature geo:hasGeometry ?dropGeom .
        ?feature wgs84:lat ?lat .
        ?feature wgs84:lng ?lng .
        ?feature geodata:district ?district
        FILTER ( ?drop_hour < 4 )
        FILTER ( 22 < ?drop_hour )
      }
    WINDOW :wind
      { ?ride debs:pickup_latitude ?lat .
        ?ride debs:pickup_longitude ?lng .
        ?ride debs:pickup_datetime ?time .
        ?time time:hour ?pick_hour .
        ?place geo:hasGeometry ?pickGeom .
        ?place wgs84:lat ?lat .
        ?place wgs84:lng ?lng .
        ?place geodata:district ?district
        FILTER ( ?pick_hour < 4 )
        FILTER ( 22 < ?pick_hour )
      }
    FILTER ( ( ?pick_hour - ?drop_hour ) > 1 )
    FILTER geof:distance(?dropGeom, ?pickGeom, units:mile, 0.1)
  }
```

```
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  debs: <http://debs2015.org/onto#>

REGISTER STREAM :query5 AS

SELECT  ?distance (( ( ( ?amount - ?tax ) - ?tips ) - ?tolls ) AS ?profit)
FROM NAMED WINDOW :wind ON :rides [RANGE PT1H STEP PT1H]
WHERE
  { WINDOW :wind
      { ?ride debs:trip_distance ?distance .
        ?ride debs:total_amount ?amount .
        ?ride debs:mta_tax ?tax .
        ?ride debs:tip_amount ?tips .
        ?ride debs:tolls_amount ?tolls
      }
  }
```

```
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  ex:   <http://example.org/>
PREFIX  debs: <http://debs2015.org/onto#>

REGISTER STREAM :query6 AS

SELECT  ?luckyRide
FROM NAMED WINDOW :win ON :rides [RANGE PT1H STEP PT1H]
WHERE
  { WINDOW :win
      { ?luckyRide debs:byTaxi ?taxi
        FILTER NOT EXISTS {?luckyRide ex:stoppedAt ?trafficLight }
      }
  }
```

```
# The original query contained several issues. This
# modified version should represent intent of the original.
PREFIX  :     <http://debs2015.org/streams/>
PREFIX  ex:   <http://example.org/>
PREFIX  debs: <http://debs2015.org/onto#>
PREFIX  geof: <http://www.opengis.net/def/geosparql/function/>
PREFIX  gn:   <http://www.geonames.org/ontology#>

REGISTER STREAM :query7 AS

SELECT  ?neighbourhood (( count(?newPickups) / count(?oldPickups) ) AS
     ↪ ?increase)
FROM gn:geonames
FROM NAMED WINDOW :newPickups ON :rides [RANGE PT1H STEP PT1H]
FROM NAMED WINDOW :oldPickups ON :rides [FROM NOW-PT2H TO NOW-PT1H STEP
     ↪ PT1H]
WHERE
  { ?oldPickups gn:neighbourhood ?neighbourhood .
    ?newPickups gn:neighbourhood ?neighbourhood
    WINDOW :newPickups
      { ?newPickups debs:pickup_latitude ?nlat .
        ?newPickups debs:pickup_longitude ?nlon
      }
    WINDOW :oldPickups
      { ?oldPickups debs:pickup_latitude ?olat .
        ?oldPickups debs:pickup_longitude ?olon
      }
  }
GROUP BY ?neighbourhood
HAVING ( ?increase >= 1.2 )
```

```
# This query could not be parsed since nested constructs
# are not supported in RSP-QL
PREFIX : <http://debs2015.org/streams/>
PREFIX debs: <http://debs2015.org/onto#>
PREFIX ogc: <http://www.opengis.net/ont/geosparql#>
PREFIX geom: <http://geovocab.org/geometry#>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX geonames: <http://linkedgeodata.org/ontology/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

REGISTER STREAM :query8 AS

SELECT ?strName ?profit
FROM <http://www.example.org/geonames>
WHERE {
  ?poi geonames:street ?strName ;
         geom:geometry ?point .
  { SELECT (SUM(?total - ?tax - ?tips) AS ?profit) ?point
    #FROM NAMED WINDOW :spot ON :stream [RANGE PT6H STEP PT6H]
    WHERE {
      WINDOW :spot {
        ?ride debs:fare_amount ?total ;
              debs:mta_tax ?tax ;
              debs:tip_amount ?tips .
        { CONSTRUCT {
            ?ride <http://debs2015.org/onto#dropPoint> ?point.
            ?point a geom:Geometry, geom:Point ;
                   ogc:asWKT ?wktLit ;
                   wgs84:lat ?lat ;
                   wgs84:long ?lng .
          }
          WHERE {
            ?ride debs:dropoff_latitude ?dropLat;
                  debs:dropoff_longitude ?dropLong;
            BIND(STRDT(?dropLat, xsd:double) AS ?lat)
            BIND(STRDT(?dropLong, xsd:double) AS ?lng)
          }
        }
      }
    }
    GROUP BY ?point
  }
}
```

CSRBench Queries

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query1 AS

SELECT ?sensor ?obs
FROM NAMED WINDOW :w ON :test [RANGE PT10S STEP PT10S]
WHERE {
  WINDOW :w {
    ?obs om-owl:observedProperty weather:_AirTemperature ;
      om-owl:procedure ?sensor ;
      om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value > 80)
}
```

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query2 AS

SELECT ?sensor ?obs
FROM NAMED WINDOW :w ON :test [RANGE PT1S STEP PT1S]
WHERE {
  WINDOW :w {
    ?obs om-owl:observedProperty weather:_AirTemperature ;
      om-owl:procedure ?sensor ;
      om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value > 80)
}
```

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query3 AS

SELECT ?sensor ?obs ?value
FROM NAMED WINDOW :w ON :test [RANGE PT4S STEP PT4S]
WHERE {
  WINDOW :w  {
    ?obs om-owl:observedProperty weather:_RelativeHumidity ;
      om-owl:procedure ?sensor ;
      om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value < 49)
  FILTER(?value > 24)
}
```

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query4 AS

SELECT (AVG(?value) AS ?avg)
FROM NAMED WINDOW :w ON :test [RANGE PT4S STEP PT4S]
WHERE {
  WINDOW :w {
    ?obs om-owl:observedProperty weather:_AirTemperature ;
      om-owl:procedure ?sensor ;
      om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value > 80)
}
```

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query5 AS

SELECT ?sensor ?obs
FROM NAMED WINDOW :w ON :test [RANGE PT5S STEP PT1S]
WHERE {
  WINDOW :w {
    ?obs om-owl:observedProperty weather:_AirTemperature ;
      om-owl:procedure ?sensor ;
      om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value > 80)
}
```

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query6 AS

SELECT ?sensor ?ob1 ?value1 ?obs
FROM NAMED WINDOW :w ON :test [RANGE PT5S STEP PT5S]
WHERE {
  WINDOW :w {
    ?ob1 om-owl:procedure ?sensor ;
      om-owl:observedProperty weather:_AirTemperature ;
      om-owl:result [om-owl:floatValue ?value1] .
    ?obs om-owl:procedure ?sensor ;
      om-owl:observedProperty weather:_AirTemperature ;
      om-owl:result [om-owl:floatValue ?value] .
  }
  FILTER(?value1 - ?value > 0.5)
  FILTER(?value > 75)
}
```

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX sens-obs: <http://knoesis.wright.edu/ssw/>
PREFIX : <http://ex.org/streams/>

REGISTER STREAM :query1 AS

SELECT ?sensor ?ob1
FROM NAMED WINDOW :w ON :test [RANGE PT5S STEP PT5S]
WHERE {
  WINDOW :w {
    ?ob  om-owl:procedure sens-obs:System_C1190 ;
      om-owl:observedProperty weather:_AirTemperature ;
      om-owl:result [om-owl:floatValue ?value] .
    ?ob1 om-owl:procedure ?sensor ;
      om-owl:observedProperty weather:_AirTemperature ;
      om-owl:result [om-owl:floatValue ?value1] .
  }
  FILTER(?value1 > ?value)
}
```

Appendix B

Scenario 1: CQELS queries

```
PREFIX  :     <http://traffic.org/data#>
PREFIX  gaz:  <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event: <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

CONSTRUCT
  {
    ?tweet rdf:type twitter:Status .
    ?tweet twitter:hasText ?text .
    ?tweet event:isSubEventObjectOf ?incident .
    ?tweet twitter:createdAt ?createdAt .
    ?tweet traffic:hasLocation ?location .
    ?location rdf:type gaz:NamedPlace .
    ?location rdfs:label ?locationLabel .
    ?location spatial:northing ?northing .
    ?location spatial:easting ?easting .
    ?location traffic:source "place" .
  }
WHERE
  { STREAM <http://traffic.org/data#twitter> [RANGE 2s] {
      ?tweet    rdf:type              twitter:Status ;
                twitter:hasText       ?text ;
                event:isSubEventObjectOf  ?incident ;
                twitter:createdAt     ?createdAt ;
                twitter:hasPlace      ?location .
      ?location  rdf:type             gaz:NamedPlace ;
                rdfs:label            ?locationLabel .
    }
    _:b0  rdf:type          gaz:NamedPlace ;
          rdfs:label        ?locationLabel ;
          spatial:easting   ?easting ;
          spatial:northing  ?northing
  }
```

```
PREFIX  :        <http://traffic.org/data#>
PREFIX  gaz:     <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event:   <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

CONSTRUCT
  {
    ?tweet rdf:type twitter:Status .
    ?tweet twitter:hasText ?text .
    ?tweet event:isSubEventObjectOf ?incident .
    ?tweet twitter:createdAt ?createdAt .
    ?tweet traffic:hasLocation ?location .
    ?location rdf:type gaz:NamedPlace .
    ?location rdfs:label ?locationLabel .
    ?location spatial:northing ?northing .
    ?location spatial:easting ?easting .
    ?location traffic:source "account" .
  }
WHERE
  { STREAM <http://traffic.org/data#twitter> [RANGE 2s] {
      ?tweet    twitter:hasText       ?text ;
                event:isSubEventObjectOf  ?incident ;
                twitter:createdAt     ?createdAt ;
                twitter:hasAccount    ?account .
      ?account  rdf:type              twitter:Account ;
                traffic:hasLocation   ?location .
      ?location  rdf:type             gaz:NamedPlace ;
                rdfs:label            ?locationLabel .
    }
    _:b0  rdf:type          gaz:NamedPlace ;
          rdfs:label        ?locationLabel ;
          spatial:easting   ?easting ;
          spatial:northing  ?northing
  }
```

```
PREFIX  :     <http://traffic.org/data#>
PREFIX  gaz:  <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event: <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

CONSTRUCT
  {
    ?tweet rdf:type twitter:Status .
    ?tweet twitter:hasText ?text .
    ?tweet event:isSubEventObjectOf ?incident .
    ?tweet twitter:createdAt ?createdAt .
    ?tweet traffic:hasLocation ?location .
    ?location rdf:type gaz:NamedPlace .
    ?location spatial:northing ?northing .
    ?location spatial:easting ?easting .
    ?location traffic:source "status" .
  }
WHERE
  { STREAM <http://traffic.org/data#twitter> [RANGE 2s] {
      ?tweet    rdf:type              twitter:Status ;
                twitter:hasText       ?text ;
                event:isSubEventObjectOf  ?incident ;
                twitter:createdAt     ?createdAt ;
                traffic:hasLocation   ?location .
      ?location  rdf:type             gaz:NamedPlace ;
                spatial:northing      ?northing ;
                spatial:easting       ?easting .
    }
  }
```

```
PREFIX  :     <http://traffic.org/data#>
PREFIX  gaz:  <http://data.ordnancesurvey.co.uk/ontology/50kGazetteer/>
PREFIX  twitter: <http://twitter.com/ontology#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  spatial: <http://data.ordnancesurvey.co.uk/ontology/
     ↪ spatialrelations/>
PREFIX  event: <http://www.ontologydesignpatterns.org/cp/owl/
     ↪ eventprocessing.owl#>
PREFIX  traffic: <http://trafficincident.com/ontology#>

SELECT  ?text ?createdAt ?easting ?northing ?contentSimilarity
     ↪ ?spatialDistance ?temporalDistance
WHERE
  { STREAM <http://traffic.org/data#stream4> [RANGE 5H] {
      ?tweet    twitter:hasText       ?text ;
                twitter:createdAt     ?createdAt ;
                traffic:hasLocation   ?location ;
                traffic:hasRelevance  ?relevance .
      ?location  spatial:easting      ?easting ;
                spatial:northing      ?northing .
      ?relevance  traffic:hasContentSimilarity  ?contentSimilarity ;
                traffic:hasSpatialDistance  ?spatialDistance ;
                traffic:hasTemporalDistance  ?temporalDistance .
    }
    FILTER ( ?contentSimilarity > 2 )
    FILTER ( ?spatialDistance < 50000 )
  }
```

Scenario 2: CQELS and C-SPARQL queries

```
PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
  {
    ?observation rdf:type anpr:VehicleObservation .
    ?observation anpr:hasCamera ?camera .
    ?observation anpr:hasObservationTime ?time .
    ?observation anpr:hasVehicle ?vehicle .
    ?vehicle rdf:type anpr:Vehicle .
    ?vehicle anpr:hasVehicleStatus anpr:Blacklisted .
    ?vehicle anpr:hasRegno ?regno .
  }
WHERE
  { STREAM <http://ns.valcri.org/stream/anpr> [RANGE 1s] {
      ?observation  rdf:type          anpr:VehicleObservation ;
                    anpr:hasCamera    ?camera ;
                    anpr:hasObservationTime  ?time ;
                    anpr:hasVehicle   ?vehicle .
      ?vehicle  rdf:type              anpr:Vehicle ;
                    anpr:hasRegno     ?regno .
    }
    _:b0  rdf:type              anpr:Vehicle ;
          anpr:hasRegno         ?regno ;
          anpr:hasVehicleStatus  anpr:Blacklisted
  }
```

```
PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
  {
    ?personObservation rdf:type anpr:PersonObservation .
    ?personObservation anpr:observationOfPerson ?person .
    ?personObservation anpr:derivedFrom ?observation .
    ?personObservation anpr:hasObservationTime ?time .
    ?observation rdf:type anpr:VehicleObservation .
    ?observation anpr:hasCamera ?camera .
    ?observation anpr:hasObservationTime ?time .
    ?observation anpr:hasVehicle ?vehicle .
    ?vehicle rdf:type anpr:Vehicle .
    ?vehicle anpr:hasRegno ?regno .
  }
WHERE
  { STREAM <http://ns.valcri.org/stream/anpr> [RANGE 1s] {
      ?observation  rdf:type          anpr:VehicleObservation ;
                    anpr:hasCamera        ?camera ;
                    anpr:hasObservationTime  ?time ;
                    anpr:hasVehicle       ?vehicle .
      ?vehicle  rdf:type              anpr:Vehicle ;
                    anpr:hasRegno         ?regno .
    }
    _:b0    rdf:type          anpr:Vehicle ;
            anpr:hasRegno     ?regno ;
            anpr:ownedBy      ?person .
    ?crime  rdf:type          anpr:Crime ;
            anpr:hasSuspect   ?person
    BIND(bnode() AS ?personObservation)
  }
```

```
REGISTER STREAM vehicleroutes AS

PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
  {
    _:c0 rdf:type anpr:VehicleRoute .
    _:c0 anpr:hasStartTime ?minTime .
    _:c0 anpr:hasEndTime ?maxTime .
    _:c0 anpr:hasVehicleObservation ?observation .
    ?observation rdf:type anpr:VehicleObservation .
    ?observation anpr:hasCamera ?camera .
    ?observation anpr:hasVehicle ?vehicle .
    ?observation anpr:hasObservationTime ?time .
    ?observation anpr:hasRegno ?regno .
  }
FROM STREAM <http://ns.valcri.org/stream/blacklisted> [RANGE 30m STEP 10s]
WHERE
  {
    { SELECT  ?regno (MAX(?time) AS ?maxTime) (MIN(?time) AS ?minTime)
      WHERE
        { ?event anpr:hasVehicle/anpr:hasRegno ?regno .
          ?event  anpr:hasObservationTime  ?time
        }
      GROUP BY ?regno
    } .
    ?observation  rdf:type          anpr:VehicleObservation ;
              anpr:hasCamera        ?camera ;
              anpr:hasVehicle       ?vehicle ;
              anpr:hasObservationTime  ?time .
    ?vehicle  anpr:hasRegno         ?regno .
  }
```

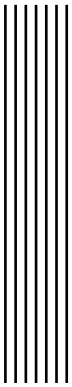
```
REGISTER STREAM rendezvous AS

PREFIX  :     <http://ns.valcri.org/stream/>
PREFIX  anpr: <http://ns.valcri.org/ontology/anpr#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
  {
    _:c0 rdf:type anpr:Rendezvous .
    _:c0 anpr:hasPersonObservation ?personObservation1 .
    _:c0 anpr:hasPersonObservation ?personObservation2 .
  }
FROM <http://ns.valcri.org/data>
FROM STREAM <http://ns.valcri.org/stream/personsofinterest> [RANGE 5m STEP
     ↪ 10s]
WHERE
  { ?personObservation1
              rdf:type              anpr:PersonObservation ;
              anpr:observationOfPerson  ?person1 ;
              anpr:derivedFrom      ?observation1 .
    ?observation1
              rdf:type              anpr:VehicleObservation ;
              anpr:hasCamera        ?camera1 .
    ?personObservation1
              rdf:type              anpr:PersonObservation ;
              anpr:observationOfPerson  ?person2 ;
              anpr:derivedFrom      ?observation2 .
    ?observation2
              rdf:type              anpr:VehicleObservation ;
              anpr:hasCamera        ?camera2 .
    ?crime    anpr:hasSuspect       ?person1 ;
              anpr:hasSuspect       ?person2
  }
```

Scenario 3: C-SPARQL queries

```
REGISTER STREAM cleantv AS

PREFIX  :     <http://ecareathome.org/stream#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX  dul:  <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>
PREFIX  time: <http://w3id.org/ecareathome/patterns/timeinterval.owl#>
PREFIX  event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX  sosa: <http://www.w3.org/ns/sosa/>

CONSTRUCT
  {
    _:c0 rdf:type event:ComplexEvent .
    _:c0 dul:isObservableAt _:c1 .
    _:c0 sosa:isObservedBy ?sensor .
    _:c0 sosa:hasSimpleResult ?value .
    _:c1 rdf:type dul:TimeInterval .
    _:c1 time:hasUpperTimeStampValue ?maxTime .
    _:c1 time:hasLowerTimeStampValue ?minTime .
  }
FROM STREAM <http://ecareathome.org/stream#tv> [RANGE 10s STEP 10s]
WHERE
  { { SELECT  ?sensor (AVG(?value) AS ?avg) (MAX(?upper) AS ?maxTime)
    ↪ (MIN(?lower) AS ?minTime)
     WHERE
       {
         _:b0  sosa:isObservedBy     ?sensor ;
               sosa:hasSimpleResult  ?value ;
               dul:isObservableAt    _:b1 .
         _:b1  time:hasUpperTimeStampValue  ?upper ;
               time:hasLowerTimeStampValue  ?lower .
       }
     GROUP BY ?sensor
    }
    BIND(if(( ?avg < 20 ), false, true) AS ?value)
    BIND(now() AS ?time)
    FILTER bound(?sensor)
  }
```

```
REGISTER STREAM cleancouch AS

PREFIX  :     <http://ecareathome.org/stream#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX  dul:  <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>
PREFIX  time: <http://w3id.org/ecareathome/patterns/timeinterval.owl#>
PREFIX  event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX  sosa: <http://www.w3.org/ns/sosa/>

CONSTRUCT
  {
    _:c0 rdf:type event:ComplexEvent .
    _:c0 dul:isObservableAt _:c1 .
    _:c0 sosa:isObservedBy ?sensor .
    _:c0 sosa:hasSimpleResult ?value .
    _:c1 rdf:type dul:TimeInterval .
    _:c1 time:hasUpperTimeStampValue ?maxTime .
    _:c1 time:hasLowerTimeStampValue ?minTime .
  }
FROM STREAM <http://ecareathome.org/stream#couch> [RANGE 10s STEP 10s]
WHERE
  { { SELECT  ?sensor (MAX(?upper) AS ?maxTime) (MIN(?lower) AS ?minTime)
      WHERE
        {
          _:b0  sosa:isObservedBy     ?sensor ;
                dul:isObservableAt    _:b1 .
          _:b1  time:hasUpperTimeStampValue  ?upper ;
                time:hasLowerTimeStampValue  ?lower .
        }
      GROUP BY ?sensor
    }
    { SELECT  ?sensor (COUNT(?sensor) AS ?isTrue)
      WHERE
        {
          _:b2  sosa:isObservedBy     ?sensor ;
                sosa:hasSimpleResult  true .
        }
      GROUP BY ?sensor
    }
    { SELECT  ?sensor (COUNT(?sensor) AS ?isFalse)
      WHERE
        {
          _:b3  sosa:isObservedBy     ?sensor ;
                sosa:hasSimpleResult  false .
        }
      GROUP BY ?sensor
    }
    BIND(( ?isTrue / ( ?isTrue + ?isFalse ) ) AS ?avg)
    BIND(if(( ?avg >= 0.9 ), true, if(( ?avg < 0.1 ), false, ?avg)) AS
      ↪ ?value)
    BIND(now() AS ?time)
  }
```

```
REGISTER STREAM partitioned AS

PREFIX  :     <http://ecareathome.org/stream#>
PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX  event: <http://w3id.org/ecareathome/patterns/event.owl#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  sosa: <http://www.w3.org/ns/sosa/>

CONSTRUCT
  {
    ?subject ?predicate ?object .
  }
FROM STREAM <http://ecareathome.org/stream#tv_cleaned> [RANGE 10s STEP 10s]
FROM STREAM <http://ecareathome.org/stream#couch_cleaned> [RANGE 10s STEP
     ↪ 10s]
WHERE
  { ?event  rdf:type  event:ComplexEvent .
    ?event (!<>)* ?subject .
    ?subject  ?predicate  ?object
  }
```



Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. "Aurora: A New Model and Architecture for Data Stream Management". In: *The VLDB Journal* 12.2 (Aug. 2003), pp. 120–139. ISSN: 1066-8888. DOI: 10.1007/s00778-003-0095-z. URL: http://dx.doi.org/10.1007/s00778-003-0095-z.

[2] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. "CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets". In: *Proceeding of the 14th International Semantic Web Conference - Part II*. Ed. by Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab. ISWC '15. Springer International Publishing, 2015, pp. 374–389. ISBN: 978-3-319-25010-6. DOI: 10.1007/978-3-319-25010-6_25. URL: https://doi.org/10.1007/978-3-319-25010-6_25.

[3] Marjan Alirezaie and Amy Loutfi. "Reasoning for Sensor Data Interpretation: an Application to Air Quality Monitoring". In: *Journal of Ambient Intelligence and Smart Environments* 7.4 (2015), pp. 579–597.

[4] Marjan Alirezaie, Jennifer Renoux, Uwe Köckemann, Annica Kristoffersson, Lars Karlsson, Eva Blomqvist, Nicolas Tsiftes, Thiemo Voigt, and Amy Loutfi. "An Ontology-based Context-aware System for Smart Homes: E-care@home". In: *Sensors* 17.1586 (2017).

[5] James F. Allen. “Maintaining Knowledge About Temporal Intervals”. In: *Communications of the ACM* 26.11 (Nov. 1983), pp. 832–843. ISSN: 0001-0782. DOI: 10.1145/182.358434. URL: http://doi.acm.org/10.1145/182.358434.

[6] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. “EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning”. In: *Proceedings of the 20th International Conference on World Wide Web*. WWW ’11. Hyderabad, India: ACM, 2011, pp. 635–644. ISBN: 978-1-4503-0632-4. DOI: 10.1145/1963405.1963495. URL: http://doi.acm.org/10.1145/1963405.1963495.

[7] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. “Stream Reasoning and Complex Event Processing in ETALIS”. In: *Semantic Web Journal* 3.4 (Oct. 2012), pp. 397–407. ISSN: 1570-0844. URL: http://dl.acm.org/citation.cfm?id=2590208.2590214.

[8] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. 2nd ed. The MIT Press, 2008. ISBN: 0262012421, 9780262012423.

[9] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. *STREAM: The Stanford Data Stream Management System*. Technical Report. Stanford InfoLab, 2004.

[10] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL Continuous Query Language: Semantic Foundations and Query Execution”. In: *The VLDB Journal* 15.2 (June 2006), pp. 121–142. ISSN: 1066-8888. DOI: 10.1007/s00778-004-0147-z. URL: http://dx.doi.org/10.1007/s00778-004-0147-z.

[11] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. “Incremental Reasoning on Streams and Rich Background Knowledge”. In: *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part I*. ESWC’10. Heraklion, Crete, Greece: Springer-Verlag, 2010, pp. 1–15. ISBN: 3-642-13485-8, 978-3-642-13485-2. DOI: 10.1007/978-3-642-13486-9_1. URL: http://dx.doi.org/10.1007/978-3-642-13486-9_1.

[12] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. “Querying RDF Streams with C-SPARQL”. In: *ACM SIGMOD Record* 39.1 (Sept. 2010), pp. 20–26. ISSN: 0163-5808. DOI: 10.1145/1860702.1860705. URL: http://doi.acm.org/10.1145/1860702.1860705.

[13] Christian Bizer and Andreas Schultz. “The Berlin SPARQL Benchmark”. In: *International Journal on Semantic Web and Information Systems* 5.2 (2009), pp. 1–24.

[14] Eva Blomqvist and Mikko Rinne. “Event Processing ODP”. In: *WOP 2013 Workshop on Ontology and Semantic Web Patterns Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns co-located with 12th International Semantic Web Conference (ISWC 2013) Sydney, Australia, October 21, 2013*. Vol. 1188. Aachen, Germany: CEUR-WS.org, 2013.

[15] Andre Bolles, Marco Grawunder, and Jonas Jacobi. “Streaming SPARQL Extending SPARQL to Process Data Streams”. In: *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*. ESWC’08. Tenerife, Canary Islands, Spain: Springer-Verlag, 2008, pp. 448–462. ISBN: 3-540-68233-3, 978-3-540-68233-2. URL: http://dl.acm.org/citation.cfm?id=1789394.1789438.

[16] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. “Enabling Ontology-Based Access to Streaming Data Sources”. In: *The Semantic Web – ISWC 2010: 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*. Ed. by Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm. Berlin, Heidelberg: Springer Berlin Heidelberg, Nov. 2010, pp. 96–111. ISBN: 978-3-642-17746-0. DOI: 10.1007/978-3-642-17746-0_7. URL: https://doi.org/10.1007/978-3-642-17746-0_7.

[17] Michael Compton, Payam Barnaghi, Luis Bermudez, RaúL GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. “The SSN Ontology of the W3C Semantic Sensor Network Incubator Group”. In: *Web semantics: science, services and agents on the World Wide Web* 17 (2012), pp. 25–32.

[18] Gianpaolo Cugola and Alessandro Margara. “Processing Flows of Information: From Data Stream to Complex Event Processing”. In: *ACM Computing Surveys (CSUR)* 44.3 (June 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677.

[19] Minh Dao-Tran and Danh Le-Phuoc. “Towards Enriching CQELS with Complex Event Processing and Path Navigation”. In: *Proceedings of the 1st Workshop on High-Level Declarative Stream Processing co-located with the 38th German AI conference (KI 2015)*. HiDeSt ’15. Dresden, Germany: CEUR-WS.org, Sept. 2015, pp. 2–14.

[20] Daniele Dell’Aglio, Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. “On Correctness in RDF Stream Processor Benchmarking”. In: *Proceedings of the 12th International Semantic Web Conference - Part II*. ISWC’13. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 326–342. ISBN: 978-3-642-41337-7. DOI:

10.1007/978-3-642-41338-4_21. URL: http://dx.doi.org/10.1007/978-3-642-41338-4_21.

[21] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. "Towards a Unified Language for RDF Stream Query Processing". In: *Revised Selected Papers of the ESWC 2015 Satellite Events on The Semantic Web: ESWC 2015 Satellite Events - Volume 9341*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 353–363. ISBN: 978-3-319-25638-2. DOI: 10.1007/978-3-319-25639-9_48. URL: http://dx.doi.org/10.1007/978-3-319-25639-9_48.

[22] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. "RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems". In: *International Journal on Semantic Web and Information Systems* 10.4 (Oct. 2014), pp. 17–44. ISSN: 1552-6283. DOI: 10.4018/ijswis.2014100102. URL: http://dx.doi.org/10.4018/ijswis.2014100102.

[23] Opher Etzion and Peter Niblett. *Event Processing in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN: 1935182218, 9781935182214.

[24] Ixent Galpin, Christian Y. A. Brenninkmeijer, Alasdair J. G. Gray, Farhana Jabeen, Alvaro A. A. Fernandes, and Norman W. Paton. "SNEE: a query processor for wireless sensor networks". In: *Distributed and Parallel Databases* 29.1 (Feb. 1, 2011), pp. 31–85. ISSN: 1573-7578. DOI: 10.1007/s10619-010-7074-3. URL: https://doi.org/10.1007/s10619-010-7074-3.

[25] Aldo Gangemi and Valentina Presutti. "Ontology Design Patterns". In: *Handbook on Ontologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 221–243. ISBN: 978-3-540-92673-3. DOI: 10.1007/978-3-540-92673-3_10. URL: https://doi.org/10.1007/978-3-540-92673-3_10.

[26] Shen Gao, Daniele Dell'Aglio, Soheila Dehghanzadeh, Abraham Bernstein, Emanuele Della Valle, and Alessandra Mileo. "Planning Ahead: Stream-Driven Linked-Data Access Under Update-Budget Constraints". In: *The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I*. Ed. by Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil. Cham: Springer International Publishing, 2016, pp. 252–270. ISBN: 978-3-319-46523-4. DOI: 10.1007/978-3-319-46523-4_16. URL: https://doi.org/10.1007/978-3-319-46523-4_16.

[27] Robert L. Glass, V. Ramesh, and Iris Vessey. “An Analysis of Research in Computing Disciplines”. In: *Communications of the ACM* 47.6 (2004), pp. 89–94.

[28] Mark Graham, Scott A Hale, and Devin Gaffney. “Where in the world are you? Geolocation and language identification in Twitter”. In: *The Professional Geographer* 66.4 (2014), pp. 568–578.

[29] Sven Groppe, Jinghua Groppe, Dirk Kukulenz, and Volker Linnemann. “A SPARQL Engine for Streaming RDF Data”. In: *Proceedings of the 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*. SITIS ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 167–174. ISBN: 978-0-7695-3122-9. DOI: 10.1109/SITIS.2007.22. URL: http://dx.doi.org/10.1109/SITIS.2007.22.

[30] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. “Temporal RDF”. In: *Proceedings of the Second European Conference on The Semantic Web: Research and Applications*. ESWC’05. Heraklion, Greece: Springer-Verlag, 2005, pp. 93–107. ISBN: 3-540-26124-9, 978-3-540-26124-7. DOI: 10.1007/11431053_7. URL: http://dx.doi.org/10.1007/11431053_7.

[31] Karl Hammar. “Content Ontology Design Patterns: Qualities, Methods, and Tools”. PhD thesis. Tekniska Högskolan i Jönköping, 2017, p. 238. ISBN: 9789176854549. DOI: 10.3384/diss.diva-139584. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-138724.

[32] Jesper Hoeksema and Spyros Kotoulas. “High-performance Distributed Stream Reasoning using S4”. In: *1st International Workshop on Ordering and Reasoning at ISCW 2011*. Oct. 2011. URL: http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Workshops/OrdRing/paper_8.pdf.

[33] Robin Keskisärkkä and Eva Blomqvist. “Event Object Boundaries in RDF Streams: A Position Paper”. In: *Proceedings of the 2nd International Workshop on Ordering and Reasoning Co-located with the 12th International Semantic Web Conference (ISWC 2013) Sydney, Australia, October 22nd, 2013*. Vol. 1059. Aachen, Germany: CEUR-WS.org, Oct. 2013, pp. 37–42.

[34] Maxim Kolchin, Peter Wetz, Elmar Kiesling, and A Min Tjoa. “YABench: A Comprehensive Framework for RDF Stream Processor Correctness and Performance Assessment”. In: *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Springer International Publishing, 2016, pp. 280–298. ISBN: 978-3-319-38791-8. DOI: 10.1007/978-3-319-38791-8_16. URL: https://doi.org/10.1007/978-3-319-38791-8_16.

[35] Srdjan Komazec, Davide Cerri, and Dieter Fensel. "Sparkwave: Continuous Schema-enhanced Pattern Matching over RDF Data Streams". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS '12. Berlin, Germany: ACM, 2012, pp. 58–68. ISBN: 978-1-4503-1315-5. DOI: 10.1145/2335484.2335491. URL: http://doi.acm.org/10.1145/2335484.2335491.

[36] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[37] David Luckham, W. Roy Schulte, Jeff Adkins, Pedro Bizarro, Hans-Arno Jacobsen, Albert Mavashev, Brenda M. Michelson, Peter Niblett, and David Tucker. *Event Processing Glossary - Version 2.0*. July 2011. URL: http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/.

[38] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. "Streaming the Web: Reasoning over Dynamic Data". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 25 (Mar. 2014), pp. 24–44. ISSN: 1570-8268. DOI: http://dx.doi.org/10.1016/j.websem.2014.02.001. URL: http://www.sciencedirect.com/science/article/pii/S1570826814000067.

[39] Alessandra Mileo, Ahmed Abdelrahman, Sean Policarpio, and Manfred Hauswirth. "StreamRule: A Nonmonotonic Stream Reasoning System for the Semantic Web". In: *Web Reasoning and Rule Systems: 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013*. Ed. by Wolfgang Faber and Domenico Lembo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 247–252. ISBN: 978-3-642-39666-3. DOI: 10.1007/978-3-642-39666-3_23. URL: https://doi.org/10.1007/978-3-642-39666-3_23.

[40] Ingemar Nordin. *Using Knowledge: On the Rationality of Science, Technology, and Medicine*. 1st ed. Maryland USA, London UK: Lexington Books, 2017.

[41] Samir Okasha. *Philosophy of Science: A Very Short Introduction*. 1st ed. Oxford University Press, 2002.

[42] Rafail Ostrovsky and Boaz Patt-Shamir. "Optimal and Efficient Clock Synchronization Under Drifting Clocks". In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '99. Atlanta, Georgia, USA: ACM, 1999, pp. 3–12. ISBN: 1-58113-099-6. DOI: 10.1145/301308.301316. URL: http://doi.acm.org/10.1145/301308.301316.

[43] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. "A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data". In: *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*. ISWC'11. Bonn, Germany: Springer-Verlag, 2011, pp. 370–388. ISBN: 978-3-642-25072-9. URL: http://dl.acm.org/citation.cfm?id=2063016.2063041.

[44] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. "Linked Stream Data Processing Engines: Facts and Figures". In: *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*. ISWC'12. Boston, MA: Springer-Verlag, 2012, pp. 300–312. ISBN: 978-3-642-35172-3. DOI: 10.1007/978-3-642-35173-0_20. URL: http://dx.doi.org/10.1007/978-3-642-35173-0_20.

[45] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. "Linked Stream Data Processing Engines: Facts and Figures". In: *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*. ISWC'12. Boston, MA: Springer-Verlag, 2012, pp. 300–312. ISBN: 978-3-642-35172-3. DOI: 10.1007/978-3-642-35173-0_20. URL: http://dx.doi.org/10.1007/978-3-642-35173-0_20.

[46] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. "Elastic and Scalable Processing of Linked Stream Data in the Cloud". In: *Proceedings of the 12th International Semantic Web Conference - Part I*. ISWC '13. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 280–297. ISBN: 978-3-642-41334-6. DOI: 10.1007/978-3-642-41335-3_18. URL: http://dx.doi.org/10.1007/978-3-642-41335-3_18.

[47] Mikko Rinne, Eva Blomqvist, Robin Keskisärkkä, and Esko Nuutila. "Event Processing in RDF". In: *WOP 2013 Workshop on Ontology and Semantic Web Patterns Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns co-located with 12th International Semantic Web Conference (ISWC 2013) Sydney, Australia, October 21, 2013*. Vol. 1188. Aachen, Germany: CEUR-WS.org, 2013.

[48] Mikko Rinne and Esko Nuutila. "Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL". In: *Journal on Data Semantics* 6.2 (June 2016), pp. 57–69. ISSN: 1861-2040. DOI: 10.1007/s13740-016-0073-4. URL: https://doi.org/10.1007/s13740-016-0073-4.

[49] Mikko Rinne and Esko Nuutila. "User-Configurable Semantic Data Stream Reasoning Using SPARQL Update". In: *Journal on Data Se-*

mantics (Feb. 20, 2017). ISSN: 1861-2040. DOI: 10.1007/s13740-017-0076-9. URL: https://doi.org/10.1007/s13740-017-0076-9.

[50] Mikko Rinne, Esko Nuutila, and Seppo Törmä. "INSTANS: High-performance Event Processing with Standard RDF and SPARQL". In: *Proceedings of the ISWC 2012 Posters & Demonstrations Track*. Vol. 914. Boston, USA: CEUR-WS.org, Nov. 2012, pp. 101–104.

[51] Ansgar Scherp, Thomas Franz, Carsten Saathoff, and Steffen Staab. "A Core Ontology on Events for Representing Occurrences in the Real World". In: *Multimedia Tools and Applications* 58.2 (May 2012), pp. 293–331. ISSN: 1380-7501. DOI: 10.1007/s11042-010-0667-z. URL: http://dx.doi.org/10.1007/s11042-010-0667-z.

[52] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 Requirements of Real-time Stream Processing". In: *ACM SIGMOD Record* 34.4 (Dec. 2005), pp. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504. URL: http://doi.acm.org/10.1145/1107499.1107504.

[53] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. "Exploiting Punctuation Semantics in Continuous Data Streams". In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (Mar. 2003), pp. 555–568. ISSN: 1041-4347. DOI: 10.1109/TKDE.2003.1198390. URL: http://dx.doi.org/10.1109/TKDE.2003.1198390.

[54] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. "It's a Streaming World! Reasoning Upon Rapidly Changing Information". In: *IEEE Intelligent Systems* 24.6 (Nov. 2009), pp. 83–89. ISSN: 1541-1672. DOI: 10.1109/MIS.2009.125. URL: http://dx.doi.org/10.1109/MIS.2009.125.

[55] Ruben Verborgh, Olaf Hartig, Ben Meester, Gerald Haesendonck, Laurens Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Walle. "Querying Datasets on the Web with High Availability". In: *Proceedings of the 13th International Semantic Web Conference - Part I*. ISWC '14. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 180–196. ISBN: 978-3-319-11963-2. DOI: 10.1007/978-3-319-11964-9_12. URL: http://dx.doi.org/10.1007/978-3-319-11964-9_12.

[56] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. "Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web". In: *Journal of Web Semantics* 37–38 (Mar. 2016), pp. 184–206. ISSN: 1570-8268. DOI: doi:10.1016/j.websem.2016.03.003. URL: http://linkeddatafragments.org/publications/jws2016.pdf.

[57] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. “SRBench: A Streaming RDF/SPARQL Benchmark”. In: *Proceedings of the 11th International Conference on The Semantic Web - Volume Part I*. ISWC'12. Boston, MA: Springer-Verlag, 2012, pp. 641–657. ISBN: 978-3-642-35175-4. DOI: 10.1007/978-3-642-35176-1_40. URL: http://dx.doi.org/10.1007/978-3-642-35176-1_40.

Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.
No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
No 598 **Rego Granlund:** C^{3}Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel och Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.
No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006.
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 2008.
No 1361 **Martin Karresand:** Completing the Picture - Fragments and Back Again, 2008.
No 1363 **Per Nyblom:** Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
No 1371 **Fredrik Lantz:** Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
No 1401 **Annika Öhgren:** Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
No 1410 **Rickard Holsmark:** Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
No 1421 **Sara Stymne:** Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
No 1450 **Fabian Segelström:** Visualisations in Service Design, 2010.
No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.
No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011.
No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011.
No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.
No 1485 **Loove Broms:** Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011.
FiF-a 101 **Johan Blomkvist:** Conceptualising Prototypes in Service Design, 2011.
No 1490 **Håkan Warnquist:** Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011.
No 1503 **Jakob Rosén:** Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011.
No 1504 **Usman Dastgeer:** Skeleton Programming for Heterogeneous GPU-based Systems, 2011.
No 1506 **David Landén:** Complex Task Allocation for Delegation: From Theory to Practice, 2011.
No 1507 **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.
No 1509 **Mariusz Wzorek:** Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011.
No 1510 **Piotr Rudol:** Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011.
No 1513 **Anders Carstensen:** The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011.
No 1523 **Jody Foo:** Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012.
No 1550 **Anders Fröberg:** Models and Tools for Distributed User Interface Development, 2012.
No 1558 **Dimitar Nikolov:** Optimizing Fault Tolerance for Real-Time Systems, 2012.
No 1582 **Dennis Andersson:** Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013.
No 1586 **Massimiliano Raciti:** Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013.
No 1588 **Banafsheh Khademhosseinieh:** Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013.
No 1589 **Amy Rankin:** Resilience in High Risk Work: Analysing Adaptive Performance, 2013.
No 1592 **Martin Sjölund:** Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013.
No 1606 **Karl Hammar:** Towards an Ontology Design Pattern Quality Model, 2013.
No 1624 **Maria Vasilevskaya:** Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013.
No 1627 **Ekhiotz Vergara:** Exploiting Energy Awareness in Mobile Communication, 2013.
No 1644 **Valentina Ivanova:** Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014.
No 1647 **Dag Sonntag:** A Study of Chain Graph Interpretations, 2014.
No 1657 **Kiril Kiryazov:** Grounding Emotion Appraisal in Autonomous Humanoids, 2014.
No 1683 **Zlatan Dragisic:** Completing the Is-a Structure in Description Logics Ontologies, 2014.
No 1688 **Erik Hansson:** Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014.
No 1715 **Nicolas Melot:** Energy-Efficient Computing over Streams with Massively Parallel Architectures, 2015.
No 1716 **Mahder Gebremedhin:** Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models, 2015.
No 1722 **Mikael Nilsson:** Efficient Temporal Reasoning with Uncertainty, 2015.
No 1732 **Vladislavs Jahundovics:** Automatic Verification of Parameterized Sytems by Over-Approximation, 2015.
FiF 118 **Camilla Kirkegaard:** Adding Challenge to a Teachable Agent in a Virtual Learning Environment, 2016.
No 1758 **Vengatanathan Krishnamoorthi:** Efficient and Scalable Content Delivery of Linear and Interactive Branched Videos, 2016.
No 1771 **Andreas Löfwenmark:** Timing Predictability in Future Multi-Core Avionics Systems, 2017.
No 1777 **Anders Andersson:** Extensions for Distributed Moving Base Driving Simulators, 2017.
No 1780 **Olov Andersson:** Methods for Scalable and Safe Robot Learning, 2017.
No 1782 **Robin Keskisärkkä**: Towards Semantically Enabled Complex Event Processing, 2017.
No 1783 **Daniel de Leng**: Spatio-Temporal Stream Reasoning with Adaptive State Stream Generation, 2017.