Efficiënte verwerking van heterogene IoT-data aan de hand van expressieve redeneertechnieken

Efficient Processing of Heterogeneous IoT Data through Expressive Reasoning Techniques

Pieter Bonte

Promotoren: prof. dr. ir. F. De Turck, dr. F. Ongenae
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. B. Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-214-1
NUR 984
Wettelijk depot: D/2019/10.500/23

Dankwoord

Het schrijven van dit dankwoord is iets wat ik de voorbije weken op de lange baan geschoven heb. Nu de finale deadline steeds dichterbij komt en er ook maar geen einde lijkt te komen aan de rest van mijn werk, begint er toch geleidelijk aan wat paniek op te borrelen. Lichte paniek, de ideale drijfveer. Sommigen onder jullie verdenken mij ervan dat ik slechts over twee emotionele toestanden beschik: neutraal en eerder blij. Na het schrijven van dit dankwoord kan ik deze personen gerust stellen en met zekerheid melden dat er nog een extra emotie aanwezig is in mijn assortiment: de emotie dankbaarheid.

Gezien ik van nature relatief kort van stof ben en de kans bestaat dat ik enkele mensen vergeten ben, wil ik alvast iedereen waarmee ik de laatste jaren in contact gekomen ben bedanken. Hierna volgen enkele dankbetuigingen aan de personen die net iets meer invloed gehad hebben op het voltooien van dit boek.

Het schrijven van dit boek is een hele reis geweest. Om iedereen te bedanken die mij tijdens deze tocht geholpen heeft, stoppen we op enkele cruciale locaties. Onze eerste halte ligt in Gent waar ik eerst en vooral mijn promotoren wil bedanken die deze reis mogelijk hebben gemaakt. Filip De Turck, bedankt voor het vertrouwen en de kans om aan dit avontuur te beginnen. Mijn co-promotor Femke Ongenae wil ik bedanken om altijd het goede voorbeeld te tonen. Als protegé was het een plezier om in je voetsporen te mogen treden. Ook wil ik Piet Demeester bedanken om mijzelf en mijn collega's alle mogelijkheden te bieden. Mijn doctoraatsjury, Sofie Van Hoecke, Emanuele Della Valle, Davy Preuveneers en Bruno Volckaert wil ik bedanken om het proefschrift kritisch te lezen en nuttige feedback te geven. Voorzitter Luc Taerwe wil ik bedanken om de verdediging in goede banen leiden.

Je job graag doen hangt grotendeels af van de inhoud, maar ook een heel groot deel van je collega's. Ik heb het geluk gehad om tijdens mijn reis in contact te komen met fantastische collega's die het werk nog aangenamer gemaakt hebben. Eerst en vooral wil ik Wannes Kerckhove en Thomas Dupont bedanken om tijdens mijn eerste jaren altijd klaar te staan om mijn technische problemen op te lossen. Ex-collega Jeroen Schaballie wil ik bedanken voor alle technische ondersteuning tijdens mijn eerste projecten. Extra dank gaat uiteraard naar mijn huidige bureaugenoten Mathias De Brouwer, Gilles Vandewiele, Joris Heyse, Bram Steenwinckel, Stephanie Carlier, Philip Leroux, Femke Ongenae en Femke De Backere. Dankzij jullie is het elke dag een plezier om naar het werk te komen!

Ook de collega's die niet in de bureau aanwezig zijn, maar steeds zorgen voor een interessante middagpauze of een leuke babbel op de gang mogen bedankt worden: Pieter Van Molle, Elias De Coninck, Jeroen van der Hooft, Ozan Catal, Sam Leroux, Tim Verbelen, Bert Vankeirsbilck, Stijn Verstichel, Johannes Nauta en nog vele anderen. Naast de collega's in dezelfde bureau, hetzelfde verdiep of gebouw, wil ik ook nog collega's in de 'andere' toren bedanken voor de interessante discussies en leuke conferenties: Dörthe Arndt, Ben De Meester, Pieter Heyvaert, Pieter Colpaert, Anastasia Dimou, Sander Vanden Hautte, Miel Vander Sande, Ruben Verborgh, Sofie Van Hoecke, Erik Mannens en vele anderen.

Naast mijn eigen bureau, verdiep, toren of andere toren, moeten we nog even de landsgrens over naar Italië om Riccardo Tommasini te bedanken voor al zijn enthousiasme, overtuiging en vastberadenheid in het onderzoeksdomein. Emanuele Della Valle wil ik bedanken om mij enkele maanden onder zijn vleugels te nemen en zijn visie op de toekomst van Stream Reasoning met mij te delen.

Laat ons even terugkeren naar Gent om mijn dichtste vrienden, de 'Gent Groep' te bedanken. Eerst en vooral bedankt om onze studententijd te optimaliseren, om mee te helpen zoeken naar de optimale balans tussen 'student zijn' en studeren. Bedankt voor de wekelijkse, of wat minder frequente, pils-avond die toe laat even te reflecteren over het leven en te relativeren. Bedankt voor alle memorabele momenten gasten!

Ik denk dat het nu tijd geworden is om naar mijn geboortestad Roeselare af te zakken om enkele heel belangrijke mensen te bedanken. Mijn ouders, Joos & Adelheid, bedankt voor de warme jeugd die jullie mij gegeven hebben, om altijd in mij te geloven zelfs al was ik niet altijd de beste student of lukte niet altijd alles even vlot. Ik draag altijd jullie stempel mee, en daar ben ik trots op. Mijn zussen, Lien & Nel, bedankt om altijd zo goed voor jullie kleine broer te zorgen. Ik kan mij geen betere zussen voorstellen! Voor we onze reis verder zetten, wil ik nog even mijn grootouders bedanken: mijn meme Augusta om mij te leren relativeren en te genieten van het leven, mijn pepe Hector om zo'n wijze man te zijn, mijn opa Lucien om mij mee te nemen in de wereld van techniek en mijn oma Irma, om ons te leren dat het belangrijk is om fit te zijn.

Nu we ons in het verre West-Vlaanderen bevinden, moeten we even stoppen aan onze mooie Belgische kust, want daar woont een prachtig gezin die ik wil bedanken om mij altijd met open armen te ontvangen: mijn schoonouders Reginald & Karlien en schoonbroer Ewoud.

Het wordt stilaan tijd om naar huis te gaan want daar wacht de belangrijkste persoon in mijn leven: mijn vriendin Charlotte. Charlotte heeft de laatste jaren alles van dichtbij meegemaakt, alle goede momenten, maar ook alle stresserende momenten die soms bij het schrijven van dit boek durfden komen kijken. Bedankt Charlotte om er altijd voor mij te zijn, om mij op te beuren als ik het even niet zag zitten of in te springen als het extra druk werd. Maar als we heel eerlijk zijn zou dit boek zonder jou nu nog niet af zijn. Bedankt Charlotte om voor mij te beslissen om jouw eigen doctoraat in te dienen, zodat ik dankzij lichte paniek alles op alles heb kunnen zetten om nog voor jou te verdedigen. Maar onafhankelijk van het schrijven van dit boek moet ik jou, Charlotte, bedanken voor de persoon die je bent! Je aanstekelijk enthousiasme, verwondering en warm karakter maken van elke dag een zonnige dag. Bedankt!

Gent, April 2019
Pieter Bonte

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

AAL Ambient Assisted Living.

ABox Assertion Box.

ACCIO Ambient-Aware Continuous Care Ontology.

ARM Assocation Rule Mining.

ASP Answer Set Programming.

B

BFS Breadth-First Search.

BGP Basic Graph Pattern.

C

CELOE Class Expression Learning for Ontology Engineering.

CEP Complex Event Processing.

D

DL Description Logic.

DRL Drools Rule Language.

DSMS Data Stream Management System.

DYAMAND DYnamic, Adaptive MAnagement of Networks and Devices.

E

EL Existential quantification Language.

EPL Event Processing Language.

G

GUI	Graphical User Interface.
I	
IFP	Information Flow Processor.
ILP	Inductive Logic Programming.
IoT	Internet of Things.
J	
JSON	JavaScript Object Notation.
L	
LRU	Least Recently Used.
LSM	Linked Sensor Middleware.
M	
MASSIF	ModulAr, Service, Semantic & Flexible platform.
O	
OCCS	Organizing Home Care Using a Cloud-based Platform.
OSGi	Open Services Gateway initiative.
OWL	Web Ontology Language.
Q	
QA	Query Answering.
QL	Query Language.
R	
R.A.M.P.	Real-time Automation of Media Production.
RA	Relational Algebra.
RDF	Resource Description Framework.
RDFS	RDF Schema.
RFID	Radio Frequency Identification.

RL	Rule Language.
RSP	RDF Stream Processing.
S	
SCB	Semantic Communication Bus.
SPARQL	SPARQL Protocol and RDF Query Language.
SR	Stream Reasoning.
SSN	Semantic Sensor Network.
SWRL	Semantic Web Rule Language.
T	
TBox	Terminological Box.
U	
URI	Uniform Resource Identifier.
W	
WSN	Wireless Sensor Network.

Samenvatting
– Summary in Dutch –

In het Internet of Things (IoT) zijn objecten (of things) verbonden via een computernetwerk. Deze objecten bestaan uit sensoren, die een deel van hun omgeving observeren, en actuatoren, die hun omgeving kunnen aanpassen. De opkomst van het IoT zorgde voor een sterke toename in onderzoek en innovatieve toepassingen zoals smart homes, smart cities, pervasive health en smart transport. In elk van deze toepassingen verzamelen de geconnecteerde objecten data die onmiddellijk verwerkt dienen te worden, zodat mensen kunnen worden ondersteund in hun dagelijkse activiteiten. Men verwacht dat het aantal IoT-apparaten zal toenemen van 8.3 miljard geconnecteerde toestellen in 2019 tot 21.5 miljard in 2025. Verder wordt voorspeld dat tegen 2025 het marktaandeel dat het IoT inneemt zal toenemen tot meer dan een biljoen dollar. Daarom is het belangrijk dat IoT-serviceproviders het toenemende aantal aangesloten apparaten en de enorme hoeveelheden geproduceerde data kunnen blijven verwerken. Wegens de doorgaans hoge dataproductie is het opslaan van alle gegenereerde data vaak onmogelijk. Bovendien wordt in het IoT vaak verwacht dat acties onmiddellijk kunnen ondernomen worden, waardoor onmiddellijke verwerkingstechnieken nodig zijn. De geproduceerde data zijn typisch ook erg heterogeen omdat deze afkomstig zijn uit verschillende bronnen. Het is een uitdaging om zinvolle inzichten uit deze heterogene IoT-data af te leiden. Omdat de geproduceerde sensordata op zich vrij betekenisloos zijn, moeten deze vaak gecombineerd worden met andere bronnen, om de context van de observaties te kunnen capteren. Bovendien is ook de integratie van achtergrondkennis en informatie over het domein noodzakelijk. Achtergrondkennis biedt aanvullende informatie, terwijl domeinkennis het domein zelf beschrijft. Bijvoorbeeld, in een eHealth scenario, kan de achtergrondkennis informatie verschaffen omtrent de patiënten, hun pathologie, en meer, terwijl de domeinkennis beschrijft dat patiënten met een hersenschudding gevoelig zijn voor geluid en licht.

Semantische webtechnologieën, zoals de Web Ontology Language (OWL) en het Resource Description Framework (RDF), maken het mogelijk om domeinkennis te modelleren en hebben zich reeds bewezen als gepaste techniek om heterogene data te integreren. Een ontologie beschrijft op een formeel niveau concepten, hun eigenschappen en hun relaties, binnen een bepaald domein. Door de relaties tussen verschillende concepten te definiëren, kan een model de kennis over een bepaald domein opnemen. Formele redeneertechnieken kunnen impliciete feiten over de gegenereerde data afleiden, in overeenstemming met de gedefinieerde domeinkennis. Hoe gedetailleerder het beschreven domein, des te expressiever de redeneertechnieken moeten zijn. Bovendien hebben veel bestaande ontologieën expressieve redeneertechnieken nodig om hun domein correct te interpreteren. Er is echter nog steeds een tegenstrijdigheid tussen de frequentie waaraan data in IoT-omgevingen worden geproduceerd en de beperkte snelheid waarmee expressieve redeneertechnieken data maar kunnen verwerken. Dit komt omdat de complexiteit van expressieve redeneertechnieken exponentieel is in functie van de grootte van de data. Daarom is er nood aan een schaalbare oplossing die toelaat de heterogeniteit inherent aan het IoT te verwerken en onmiddellijke resultaten te produceren.

De belangrijkste onderzoeksbijdrage van dit proefschrift is dan ook het ontwerp van een semantisch IoT-platform dat verschillende optimalisaties biedt om expressief te redeneren over stromen van IoT-data, zodat reactieve en slimme services aangeboden kunnen worden aan de eindgebruiker. Meer concreet vertaalt dit zich in de volgende vier bijdragen.

De eerste uitdaging voor IoT-services die heterogene data verwerken, is het ophalen van de juiste data. Data worden vaak geproduceerd op verschillende niveaus van granulariteit, waardoor serviceontwikkelaars moeten onderzoeken met welk detail de data door elke databron worden geproduceerd. Als eerste bijdrage wordt daarom een semantisch publiceer/abonneer systeem voorgesteld waarmee services de geabstraheerde data kunnen ontvangen. Om de data te abstraheren, worden expressieve redeneertechnieken gebruikt om het domein correct te interpreteren. Hierdoor kunnen services zich abonneren op ontologische concepten op hoog niveau, zonder zich zorgen te hoeven maken over de details van de data op een lager niveau. Het gebruik van de ontologie legt een gemeenschappelijke semantiek op zodat verschillende services kunnen samenwerken.

Omdat er een tegenstrijdigheid is tussen de snelheid waarmee data in het IoT worden geproduceerd en de snelheid waarmee redeneertechnieken data kunnen verwerken, wordt als tweede bijdrage een hiërarchisch redeneerplatform voorgesteld. Hiërarchisch redeneren is een visie die een hiërarchie van verschillende verwerkingslagen voorstelt. Op het laagste niveau worden data verwerkt met technieken met lage verwerkingscomplexiteit. Elke laag selecteert de relevante delen van de data voor verdere verwerking en naarmate men stijgt in de hiërarchie, neemt de complexiteit van de verwerking toe. Aangezien data door de verschillende lagen worden geselecteerd en geëlimineerd, hoeven de meer complexe lagen slechts een selectie van de geproduceerde data te verwerken. Hierdoor kunnen veel hogere throughputs worden bereikt, aangezien de hoeveelheid data uit de stroom die verwerkt dienen te worden tot een minimum wordt beperkt. Dit proefschrift stelt een hiërarchisch redeneerplatform voor dat het mogelijk maakt om expressief en temporeel te redeneren over IoT-datastromen. Om relevante delen van de datastromen te kunnen selecteren, worden RDF Stream Processing (RSP) engines gebruikt. Deze engines kunnen onbegrensde stromen van RDF-data verwerken, door gebruik te maken van windowing-mechanismen om stukken uit de data te extraheren en queries continu te evalueren over de datastroom. De geselecteerde data van de RSP engines worden doorgestuurd naar meer expressieve redeneertechnieken, die de data abstraheren. Zodra de data geabstraheerd zijn, kunnen temporele redeneertechnieken worden toegepast bovenop deze abstracties.

Omdat de RSP engines, die in de onderste lagen van ons hiërarchisch redeneerplatform worden gebruikt, de data selecteren die nodig zijn voor verdere verwerking, bepaalt hun verwerkingscapaciteit de totale throughput van het platform. Bovendien zal het filteren exacter zijn naarmate deze engines meer redeneermogelijkheden beziten. Bij het inzetten van verschillende onderdelen in de edge, namelijk dicht bij de databronnen, zijn de beschikbare middelen vaak beperkt. Daarom wordt als derde bijdrage een geoptimaliseerd algoritme voorgesteld dat in staat is om te redeneren over hiërarchieën, dit betekent het redeneren over subklassen en subproperties, in lineaire tijd in functie van het aantal queries dat parallel geëvalueerd dient te worden. De voorgestelde oplossing is minstens twee keer zo snel als vergelijkbare state-of-the-art RSP engines, terwijl slechts een beperkte geheugenvoetafdruk gebruikt wordt.

Naast redeneren over datastromen, dient veel statische achtergrondkennis gecombineerd te worden met de datastromen vooraleer correcte conclusies kunnen genomen worden. Aangezien meer data langzamere redeneertijden impliceert, is het een uitdaging om expressief te redeneren over deze vaak veranderende data in combinatie met grote hoeveelheden statische achtergrondkennis. Het selecteren van relevante delen uit de datastromen biedt geen oplossing in dit scenario, omdat de hoeveelheid statische data groot blijft. Daarom wordt als vierde contributie een benaderingstechniek voorgesteld die een deel van de data uit de achtergrondkennis extraheert, om zo efficiënt te redeneren over de datastromen die gecombineerd dienen te worden met deze achtergrondkennis. Dit laat toe om het redeneerproces tot 10 keer te versnellen voor kleine datasets en tot 1000 keer voor grotere datasets.

Deze vier contributies hebben geleid tot een semantisch IoT-platform dat om kan met de heterogeniteit die inherent is aan het IoT en reactieve oplossingen op een onmiddellijke en schaalbare manier kan verwezenlijken.

Summary

In the Internet of Things (IoT), ubiquitous objects or things are connected through a computer network. These objects or things typically consist of sensors, which capture a part of their environment, and actuators, which can modify their environment. The rise of the IoT enables a plethora of new applications, such as smart homes, smart cities, pervasive health, and smart transport. In each of these applications, the interconnected things collect data which should be processed as fast as possible, to support humans in their daily activities. The number of IoT devices is predicted to increase from 8.3 billion in 2019 to 21.5 billion in 2025, and the IoT market is expected to further grow with more than a trillion dollars by 2025. Therefore, it is important that IoT service providers can cope with the increasing amount of connected devices and the huge amounts of produced data. As data production is typically high, storing all the generated data is often not possible. Furthermore, many IoT domains require immediate actions, thus requiring reactive processing techniques. The produced data is typically very heterogeneous as well, as it results from a wide variety of sources. It is challenging to extract meaningful insights from this heterogeneous IoT data. As the produced sensor data itself is rather meaningless, it often needs to be combined with other sources to capture the context the sensor readings are observed in. Furthermore, the integration of background knowledge and information regarding the domain is necessary. Background knowledge provides additional information. For example, in an eHealth scenario, the background knowledge could provide information regarding the patients and their pathology. The domain knowledge describes the domain itself. For example, patients with concussions are sensitive to sound and light.

Semantic web technologies, such as the Web Ontology Language (OWL) and the Resource Description Framework (RDF), allow to model domain knowledge and have proven to be an ideal tool to integrate heterogeneous data. An ontology formally describes concepts, properties, and their relations within a certain domain. By defining the relations between various concepts, a model can incorporate the knowledge about a certain domain. Reasoners can infer implicit facts about the generated data, conform with the defined domain knowledge. The more detailed the described domain, the more expressive the reasoning techniques are required to be. Furthermore, many existing ontologies require expressive reasoning techniques to correctly interpret their domain. However, there is still a mismatch between the frequency at which data is produced in the IoT and the rate at which expressive reasoners can process data, as the complexity of expressive reasoning techniques is exponential to the size of the data. There is still a need for a solution that can deal with the heterogeneity of IoT sources in a scalable and reactive fashion.

The main research contribution of this dissertation is the design of a semantic IoT platform that provides various optimizations to perform expressive reasoning over volatile IoT data in order to provide reactive and smart IoT services to the end users. This translates in the following four contributions:

The first challenge for IoT services processing heterogeneous data is the retrieval of the correct IoT data. Data is often produced at different levels of granularity, requiring service developers to investigate the detail at which data is produced by each data source. Therefore, as a first contribution, a semantic publish/subscribe system is proposed that allows services to subscribe to abstracted data. To abstract the IoT data, expressive reasoning is employed to correctly interpret the domain. This allows services to subscribe to high-level ontological concepts without the need to worry about the lower level details. The use of ontologies imposes a common semantics, whereby various services can collaborate.

As there is a mismatch between the rate at which data is produced in the IoT and the rate expressive reasoners can process data, as a second contribution, a cascading reasoning platform is proposed. Cascading reasoning is a vision that proposes a hierarchy of various layers of processing. At the lowest level, data is processed with techniques with low processing complexities. Each layer selects the parts of the data that are relevant for further processing and while going up in the hierarchy, the complexity of processing increases. As data is selected and eliminated throughout the various layers, the more complex layers only need to process a selection of the produced data, allowing to achieve much higher throughputs as the amount of data from the stream is minimized. This dissertation proposes a cascading reasoning platform that allows to perform expressive and temporal reasoning over volatile data streams. To be able to select relevant parts of the data streams, we utilize RDF Stream Processing (RSP) engines. These engines allow to process unbounded streams of RDF data, by employing windowing mechanisms to create processable chunks of data and by continuously evaluating queries over the streaming data. The selected data from the RSP engines is forwarded to more expressive reasoning techniques, which allow to abstract the data. Once the data is abstracted, temporal reasoning techniques can be employed on top of these abstractions. This enables services to subscribe to only those parts of the volatile data streams that should be reacted upon, by exploiting expressive and temporal reasoning over data streams.

As the RSP engines used in the lower layers of our cascading reasoning approach select the data that is required for further processing, their throughput defines the total throughput of the platform. Furthermore, the more reasoning capabilities these engines posses, the more fine-grained their filtering can be. When deploying various parts in the edge, i.e. close to the data sources, available resources are often limited. Therefore, as a third contribution, an optimized algorithm is proposed that is able to perform hierarchical reasoning, i.e. subclass and subproperty reasoning, in linear time, proportional to the number of queries that need to be evaluated in parallel. The proposed RSP engine is at least twice as fast as the state-of-the-art while employing a limited memory footprint.

Besides reasoning over volatile data streams, large amounts of static background knowledge often need to be combined with the data streams. As more data implies slower reasoning times, performing expressive reasoning over frequently changing data in combination with large amounts of background knowledge is challenging. Selecting only relevant parts from the data streams does not provide a solution in this scenario, as the amount of static data that needs to be considered stays large. Therefore, as a fourth contribution, an approximation approach is proposed, which approximates a subset of data, from the large knowledge base, to efficiently perform reasoning over the streaming data that needs to be combined with the large knowledge base. This allows to speed up the reasoning process up to 10 times for small datasets and up to 1000 times for larger dataset.

By combining these four contributions, a semantic IoT platform is proposed that is able to handle the heterogeneity of data imposed by the IoT by providing a reactive solution that allows to extract actionable insights in a reactive and scalable manner.

1 Introduction

"The beginning is the most important part of the work."

–Plato

1.1 The Rise of the IoT

The Internet of Things (IoT) is the upcoming computing paradigm that enables ubiquitous objects or things to be connected through a computing network [1]. These things are typically invisibly embedded in the environment and mainly consist of sensors and actuators [2]. The sensors each sense a specific part of our environment, for example, temperature, movement, light, or sound. The actuators can interact with and adapt the environment. Examples of actuators are window, door or blind automations, smart traffic lights or intelligent braking systems.

Figure 1.1 clearly shows an increase in the number of connected IoT devices. The number of IoT devices has grown significantly since 2015 and according to the predictions, these numbers will triple by 2025. Other sources confirm these numbers, such as Gartner[2] and Cisco[3]. Figure 1.2 shows that the IoT market size keeps growing and is predicted to grow exponentially in the next years. The rise of the IoT enables a plethora of new applications, such as smart homes, smart cities, eHealth, smart transport and logistics [1, 3].

[1]https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

[2]https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016

[3]https://blogs.cisco.com/datacenter/internet-of-things-iot-data-continues-to-explode-exponentially-who-is-using-that-data-and-how

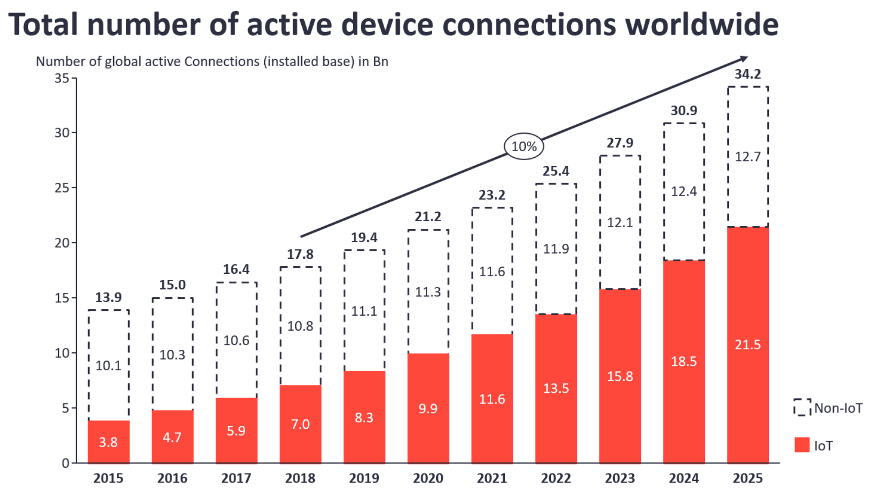


Figure 1.1: Number of connected devices worldwide, both IoT as non-IoT devices, in number of billion devices (Bn). Source: IoT analytics[1]

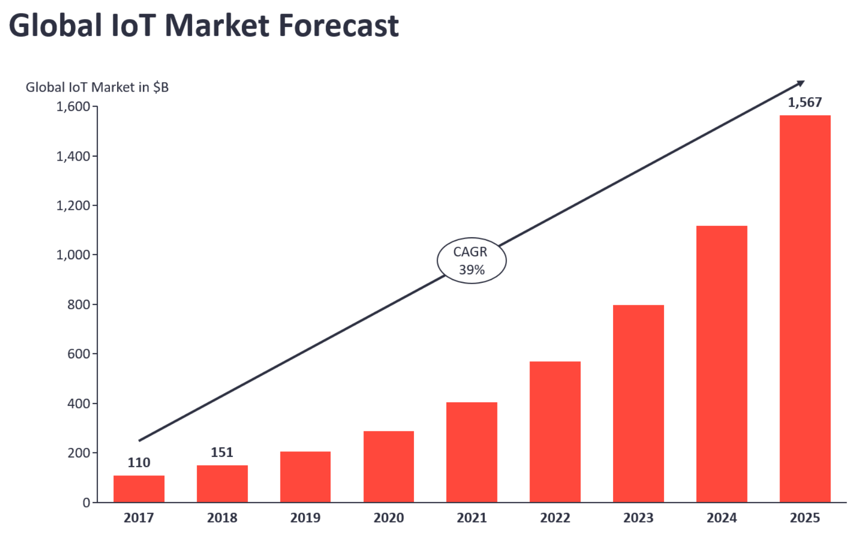


Figure 1.2: The IoT market forecast, in billion of dollars. Source: IoT analytics[1]

However, processing this IoT data introduces some additional challenges:

1. Heterogeneity: To make meaningful decisions regarding the produced IoT data, data from various heterogeneous sources needs to be combined [3], each source potentially having a different format and data encoding. Data resulting from various sensors needs to be combined, but also integrated with contextual information. For example, in an eHealth scenario, simply processing the values produced by light or sound sensors is not enough. It makes sense to consult the patient record in order to capture each patient's diseases. Furthermore, data is often produced at different levels of granularity. For example, this means that a light sensor might publish its observations as a "light sensor observation" or an "observation that observes the property light". Semantically they denote the same, however, on a data-level they describe different kinds of data.

2. Actionable Insights: Many IoT domains focus on automated decision making based on the sensed sensor data [2]. To obtain actionable insights, meaning should be attached to the raw IoT data and it should be integrated with context data, such as the patient information from the previous example, as insights are typically context dependent. For example, patients with a concussion, who are, according to the domain knowledge, sensitive to light and sound, should not be exposed to sound or light stimuli above a certain threshold. If this does happen, actions should be taken to relieve the patient from this stressful situation. Patients who do not have a concussion can be exposed to bright light, at least during the day. As we can see from the example above, raw data is often meaningless, unless it is properly processed to actionable insights taking the context into account.

3. Scalability: The increasing number of devices requires solutions that can easily integrate new devices, ranging from additional devices of the same type or completely new devices. More devices imply more data, scalable solutions thus have a clear advantage. This also implies large numbers of services that process IoT data. It is thus important that a large number of services can collaborate in an efficient and scalable manner. In our patient example, we would like to provide personalized care for all patients, each with different pathologies and different needs, requiring a multitude of sensors to be monitored and large amounts of context to be considered.

4. Reactiveness: As the generated IoT data can grow to numbers that are impossible to store, this data should be processed as fast as possible. Furthermore, many IoT domains require reactive decision-making. For instance, in the concussion example, light levels should be immediately reduced when a patient who is sensitive to light is exposed to it. In each IoT domain, the interconnected things collect data that requires reactive processing, so that humans can be supported in their daily activities. In this dissertation we aim to provide answers or execute actions within a period that users do not have the feeling they have to wait, typically within seconds.

In this dissertation, we focus on finding meaning in IoT data and extracting actionable insights by inferring implicit facts regarding the data, through the use of expressive reasoning techniques.

Meaning can be attached by annotating the data through the use of Semantic Web technologies [3]. Semantic Web technologies, such as ontologies [4], are the preferred model for the integration of heterogeneous IoT data with background knowledge [5, 6]. Therefore, in the next section, a short introduction to ontologies and reasoning is given, in order to provide the technical background to clearly convey the current challenges in this domain.

1.2 Background on Ontologies, Reasoning, and Stream Reasoning

Gruber [4] defined an ontology as "an explicit specification of a conceptualization". An ontology formally describes concepts, properties, and their relations, within a certain domain, which can easily be reused in different settings [7, 8]. This allows to model knowledge and to make data machine-readable. In IoT settings, an ontology serves as a common semantic model, allowing the integration of heterogeneous data [3]. The Web Ontology Language (OWL) [9] is a web standard and is the most popular language to describe ontologies. Ontologies form an excellent model to integrate data, exchange knowledge, and to reason upon. The ontology's Terminology Box (TBox) describes the concepts and their relations, while the Assertion Box (ABox) contains the data instances with respect to the TBox. The concepts are also called classes and the relations between classes are called object properties. Data properties describe the relation of a concept to a specific data value. The ABox consists of entities of the defined classes (the individuals), their relations, and their data properties (the literals). OWL builds on the Resource Description Framework (RDF) [10]. RDF allows to link concepts, while OWL allows to represent knowledge through complex class definitions, make distinctions between object and data properties, and define restrictions. RDF Schema (RDFS) is less expressive than OWL and allows to define ontologies describing class hierarchies and properties. In this dissertation, we make a distinction between context, background knowledge and domain knowledge. Context describes the current situation of a certain individual, e.g. a patient that is currently in the hospital and has been diagnosed with a concussion. Background knowledge consists of the collection of context information over all observed individuals, note that individuals are not restricted to persons. The domain knowledge describes the domain itself, e.g. patients with a concussion are sensitive to sound and light.

Figure 1.3 depicts an example of an ontology, modeling light sensor observations. The rounded squares represent the various concepts in the domain, i.e. Observations, Sensors, Diseases, etc. The thick arrows depict the object properties between the concepts, e.g. an Observation is observed by a certain Sensor and a System is linked to a certain Person. The dashed arrows indicate inheritance, e.g. a LightSensor is a type of Sensor and a Concussion is a type of Disease. The thin arrows indicate instantiations, e.g. lightSensor_001.03 is of the type LightSensor. The ovals indicate data properties, e.g. a SensorOutput can have a certain observed value.

A reasoner [11] allows to infer logical consequences based on the definitions in the ontology. This allows to infer implicit facts from the data. For example, in Figure 1.3, a reasoner will infer that lightSensor_001.03, which is a LightSensor, is also a Sensor and a System. This is very basic

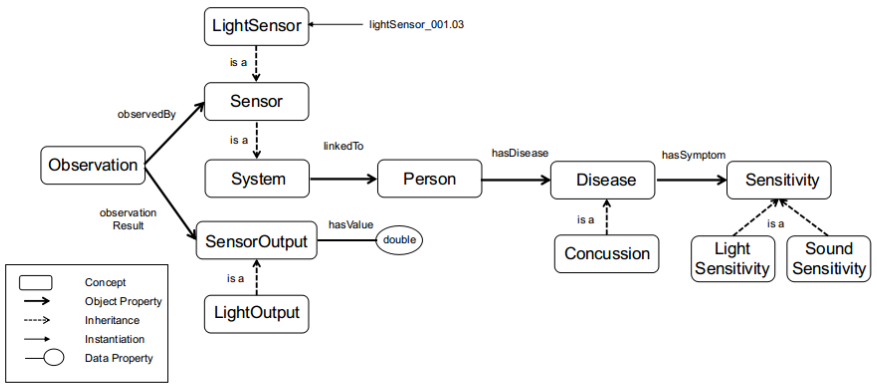


Figure 1.3: Example of an ontology, modeling sensor observations.

hierarchical reasoning, more complex definitions can be made in order to represent the domain knowledge. For example, one could define that an AlarmingLightObservation is an Observation that was observed by a LightSensor and that has a LightOutput with a value above 500 lumen, where the sensor is linked to a Person that has a disease with light sensitivity symptoms. Formally this could be defined as:

$$\text{AlarmingLightObservation} \equiv \text{Observation} \wedge \exists \text{observationResult}.(\exists \text{hasValue} > 500) \wedge \exists \text{observedBy}.(\text{LightSensor} \wedge \exists \text{linkedTo}.(\text{Person} \wedge \exists \text{hasDisease}.(\exists \text{hasSymptom}.\text{LightSensitivity})))$$

In an IoT context, this allows to model situations that need to be detected for further processing. Furthermore, reasoning allows to largely simplify querying the data, as one can simply query all the Sensors or all AlarmingObservations, without the need to worry about the exact definitions. The more accurately one wants to define the domain, the more expressive the required reasoning has to be to correctly interpret the domain. However, higher expressivity of reasoning requires higher complexity of processing [12]. RDFS reasoning has a low expressivity, allowing to infer hierarchies of concepts and properties, while Description Logic (DL) reasoning is very expressive, allowing to model complex domains.

OWL, which uses DL as it logical-based formalism, contains multiple sublanguages, each with different levels of expressivity. The sublanguages are listed below with increasing expressivity [13]:

1. OWL-Lite: supports classification and simple restriction functions.

2. OWL-DL: the largest subset without the loss of computational completeness, i.e. the system will eventually come up with an answer, with no guarantees regarding runtime or memory.

3. OWL-Full: maximal expressivity and syntactical freedom. However, the reasoning process might not be computable.

Table 1.1: Overview of the reasoning capabilities of existing RSP engines.

	Reasoning Capabilities
C-SPARQL [28]	RDFS
MORPH$_{Stream}$ [30]	None
EP-SPARQL [27]	RDFS
CQELS [26]	None
Sparkwave [31]	RDFS
INSTANS [32]	None

The popularity of OWL has led to the creation of three OWL 2 profiles [14], each offering a specific subset of the overall expressivity to obtain advantages in specific applications. Each profile is a subset of the OWL DL language.

1. OWL 2 Existential quantification Language (EL) is useful in applications utilizing an ontology that contains a large number of properties and/or classes.

2. OWL 2 Query Language (QL) is created for applications where query answering is the main task.

3. OWL 2 Rule Language (RL) provides scalable reasoning without sacrificing too much expressiveness.

Besides the profiles, OWL 2 now also provides extended support for data types, annotations, restrictions on property definitions and syntactic sugar to make certain patterns easier.

Reasoners for each profile exist, exploiting some of the limitations in expressivity to obtain higher performance. However, many of the designed ontologies still require OWL 2 DL expressivity, as the profiles are not expressive enough to describe these domains. We note that 78% of the IoT labeled ontologies in the Linked Open Vocabularies[4] repository require the OWL 2 DL expressivity to infer all concepts correctly[5]. However, there is still a mismatch between expressive reasoning and reactiveness requirements [15]. Expressive reasoning techniques, such as DL reasoning, can have up to NEXPTIME complexity [16], resulting in slow reasoning times with growing datasets [17, 18]. For example, reasoning over well-known ontologies such as GALEN [19] and DOLCE [20] takes up to 30 minutes [21]. The LUCADA ontology [22] provides guidelines for lung cancer treatment. Reasoning over the LUCADA ontology increases from less than 1 second for a single patient to 70 seconds for 30 patients [22].

Expressive reasoners have mostly focused on the processing of static data [23] or slowly changing data [24]. Many advances have been made in the Stream Reasoning domain [25–27] to combine data from multiple streams with static background knowledge. Generated IoT data produced by various sensors can be considered as data streams. To be able to process these unbounded streams of data, stream reasoning techniques consider the data within a defined time frame, i.e., a window. By using windows, multiple data items can be processed simultaneously.

[4] lov.linkeddata.es

[5] Only the ontologies that were accessible at the time of writing were considered.

RDF Stream Processing (RSP), a sub-domain of Stream Reasoning, focuses on the integration of highly volatile RDF streams with background knowledge and can continuously answer queries while performing simple reasoning. The reasoning capabilities of these systems are typically absent or very low compared to the expressive OWL 2 DL reasoners because they need to be able to handle volatile data streams. The current state-of-the-art in RSP has mainly focused on query answering over RDF streams [26, 28]. However, to provide generic query answering, RSP engines should provide some reasoning capabilities [29]. Even simple hierarchical reasoning capabilities, such as subclass and subproperty reasoning, increase the expressivity of the query extensively and simplify data integration. Table 1.1 provides an overview of the reasoning capabilities of the most prominent RSP engines.

1.3 Open Challenges & Overall Goals

There are still many open challenges to process IoT data in a meaningful way:

- **Challenge 1:** There is still a need for a platform that abstracts the fact that IoT data is heterogeneous and might be produced at different levels of granularity. The platform should be flexible such that services can easily filter the data they need and collaborate. Furthermore, it should be user-friendly, allowing services to subscribe and obtain actionable insights in a declarative manner, eliminating the need to write code.

- **Challenge 2:** Expressive reasoning provides a solution to align the heterogeneous data, interpret the domain and infer actionable insights. However, there is still a mismatch between the rate at which data is produced in an IoT setting and the time it takes to perform expressive reasoning. Therefore, solutions are needed that perform expressive reasoning over these volatile data streams produced in IoT settings.

- **Challenge 3:** Many applications that process data streams need to detect temporal dependencies between fragments in the stream. For example, in a patient pick-up scenario, the patient is first scanned to verify that the correct patient will be transported. When, within a certain amount of time, presence sensors detect that the patient is leaving the room, it can be expected that the patient is being transported to the scheduled destination. To infer that the patient is being transported to the scheduled destination, there is a temporal dependency between the scanning of the patient and the detection by the presence sensors. However, detecting temporal patterns is still challenging in complex domains [15].

- **Challenge 4:** In the edge computing paradigm, the processing of the produced data is performed as close as possible to its source, such that the data does not (completely) need to be transmitted and processed in the cloud [33]. However, this implies that the available resources at the edge are typically limited compared to cloud computing [34]. In IoT settings where many devices produce data, it makes sense to process the data close to the source, i.e. at the edge. However, as the available resources at the edge are limited, solutions to reason upon the produced IoT streams should be as performant as possible while utilizing limited resources.

- **Challenge 5:** To make meaningful decisions and infer actionable insights, services often need to combine IoT data with large amounts of background knowledge. For example, light observations need to be combined with information about the sensors, the room in which they are employed, information regarding the patient in the room, etc. However, reasoning is required to enable actionable insights. For example, deducing that the lights should be turned off again, because the patient has a concussion and is thus sensitive to bright light, requires a thorough understanding of the domain and can be achieved by expressive reasoning. However, reasoning over large amounts of background knowledge is still a challenge as the reasoning time increases exponentially with the size of the data [16].

The overall goal of this dissertation is to investigate a semantic-enabled IoT data processing platform that allows services to efficiently filter the produced IoT data and obtain actionable insights, without the need to worry about the heterogeneity of the data. The goal is to provide this functionality in a user-friendly fashion, by enabling declarative definitions of the service subscriptions and actions. This is done by enabling expressive reasoning over the data streams. This dissertation discusses various reasoning techniques in order to efficiently process IoT data in a reactive and scalable manner.

1.4 Research Questions & Hypotheses

To enable such a flexible semantic-enabled IoT data processing platform, some technical challenges need to be tackled:

In order to allow services to utilize IoT data, there is a need for a way to describe the data they are interested in. Processing all IoT data is often infeasible, as data production in IoT settings can be extremely high. Furthermore, as data is produced by various heterogeneous sources, data is often produced at different levels of granularity. There should be a way for services to subscribe in a flexible way to the data they are interested in. This leads to the first research question:

Research Question 1. How can we provide IoT services fine-grained access to IoT data in a flexible manner?

To answer this first question, we assume that services typically subscribe to data utilizing a publish/subscribe paradigm, as it allows to decouple the interaction between data producers and consumers [35]. However, matching the produced data with the subscription definitions is often not straightforward, as data might be incomplete or produced at different levels of granularity. Therefore, implicit facts should be automatically inferred based on the defined domain knowledge. Furthermore, data subscriptions might be complex, making it hard to pinpoint the specific data a service is interested in. It is thus important that the domain can be correctly interpreted by taking reasoning capabilities into account. This allows services to subscribe to high-level ontological

Table 1.2: An overview of the challenges tackled by each research question (RQ) and the different chapters.

	Challenge 1	Challenge 2	Challenge 3	Challenge 4	Challenge 5	
RQ 1	●					Chapter 1
RQ 2	●	●	●			Chapter 2
RQ 3		●		●		Chapter 2
RQ 4			●		●	Chapter 4

concepts that are not necessarily present in the data, but largely simplify the subscription definition.

Hypothesis 1. Using an ontology-enabled publish/subscribe platform will allow semantic and flexible service subscription.

As the data grows, a mismatch occurs between the rate at which data is produced and the time it takes to perform expressive and temporal reasoning. Given that the reasoning process takes longer than the rate at which data is produced by the sensors streams, data starts piling up and the system eventually crashes [12]. However, in many IoT settings, the domain is very complex, e.g. in the eHealth domain, and data is typically produced at high frequencies. This leads to the next research questions:

Research Question 2. Can expressive and temporal reasoning be performed over highly volatile data streams?

A cascading approach might provide a solution. The idea of cascading reasoning [12] provides a layered approach, which reduces in each layer the data that needs to be processed and increases in each layer the complexity of processing. This allows to filter out most of the data utilizing efficient processing techniques, so that only the relevant parts of the data are used for the expensive processing steps, such as expressive and temporal reasoning.

Hypothesis 2. Using a cascading reasoning system will improve the efficiency of expressive OWL 2 DL reasoning and temporal reasoning over volatile data streams, enabling to process up to hundreds of events per second.

As the layered approach depends on the lower layers consisting of less expressive RSP engines performing the selection of the relevant parts in the data stream, these RSP engines should be as performant as possible, in order to keep up with the data stream rate. When distributing certain parts of the cascade to the edge, where the available resources are limited, efficient processing techniques are required. However, these engines still need some reasoning capabilities. This leads to the following research questions:

Research Question 3. Can RSP engines efficiently reason over highly volatile data streams?

To answer this question, we focus on hierarchical reasoning, i.e. subconcept and subproperty reasoning. Event processing systems, which do not understand semantics, support subscriptions to hierarchical definitions of events by utilizing hierarchical encoding of event hierarchies. However, these encodings have never been exploited in RSP and have no formalization. This leads to the following hypothesis:

Hypothesis 3. Using a hierarchical encoding of concepts will improve the throughput while performing hierarchical reasoning with at least a factor two and maintaining a minimal memory footprint, compared to the state-of-the-art.

Even after efficiently filtering the data streams, there might still be a large static knowledge base that needs to be combined with the sensor data to reason upon. As the complexity of expressive reasoning systems is typically exponential to the size of the data and large knowledge bases need to be combined to extract correct decisions, the following research question can be formulated:

Research Question 4. Can expressive reasoning over event data, that needs to be combined with large static knowledge bases, be employed in time-critical use cases?

As this static data does not change very often, it is possible to precompute the data and approximate how much of the static data is necessary to reason upon the event data and the static data together, without the need to reason upon the complete large static knowledge base. This leads to the following hypothesis:

Hypothesis 4. Using an approximation technique that extracts a subset of data to reason upon, we can speed up the expressive OWL 2 DL reasoning process at least 10 times, compared to the state-of-the-art.

Table 1.2 indicates how the different challenges from Section 1.3 are tackled by each research question and chapter.

1.5 Outline

This dissertation is composed of a number of publications that were realized within the scope of this PhD. The selected publications provide an integral and consistent overview of the work performed. The complete list of peer-reviewed publications that resulted from this work is presented in Section 1.6.

Within this section, we give an overview of the remainder of this dissertation and explain how the different chapters are linked. Fig. 1.4 positions the different contributions that are presented in each chapter.

Chapter 2 presents the framework that allows services to subscribe to heterogeneous IoT data. It describes a semantic publish/subscribe mechanism facilitating reasoning capabilities, which allows services to describe the data they would like to subscribe to, utilizing high-level ontological concepts.

Chapter 3 describes and formalizes how the framework can be further extended to allow the subscription to highly volatile data streams. This is achieved by enabling a cascading reasoning

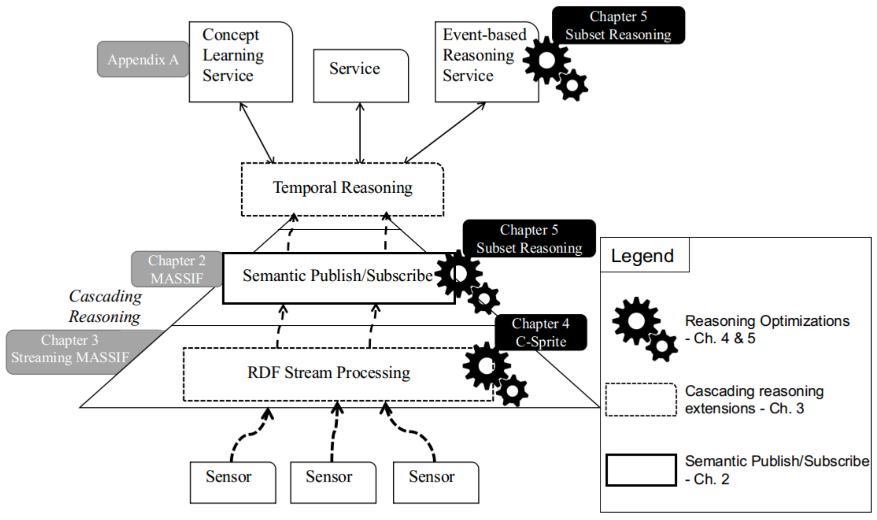

Appendix A
Concept Learning Service
Service
Event-based Reasoning Service
Chapter 5 Subset Reasoning
Temporal Reasoning
Chapter 5 Subset Reasoning
Chapter 2 MASSIF
Semantic Publish/Subscribe
Cascading Reasoning
Chapter 3 Streaming MASSIF
Chapter 4 C-Sprite
RDF Stream Processing
Sensor
Sensor
Sensor
Legend
Reasoning Optimizations - Ch. 4 & 5
Cascading reasoning extensions - Ch. 3
Semantic Publish/Subscribe - Ch. 2

Figure 1.4: Schematic overview of the different chapters in this dissertation.

approach, consisting of multiple layers of processing, each with different complexities of processing. The idea is that data is selected through the less complex layers, so that only a relevant selection remains to be processed with more complex techniques, such as expressive and temporal reasoning techniques. It extends the semantic publish/subscribe platform from Chapter 2 with two additional layers: an RDF Stream Processing layer to select the relevant data from the data streams and a temporal reasoning layer to detect temporal dependencies between the data.

Chapter 4 describes a reasoning optimization for the lower layer of the cascading approach from Chapter 3. As the lowest layers have to process large amounts of data, their performance is critical. We research and formalize how a hierarchical encoding of concepts can optimize hierarchical reasoning over highly volatile data streams.

Chapter 5 describes and formalizes how expressive reasoning can be enabled over more slowly changing data that needs to be integrated with large static knowledge bases. This reasoning optimization can be employed in the semantic publish/subscribe mechanism to abstract the data, or by the services that utilize the cascading reasoning platform to filter the IoT data.

Nowadays we see an increasing demand for combining data-driven techniques into the knowledge driven cascading reasoning hierarchy. These data-driven techniques allow to constantly expand the domain with new knowledge. A first step towards realizing this integration is discussed **Appendix A**.

1.6 Publications

The research results obtained during this PhD research have been published in scientific journals and presented at a series of international conferences. The following list provides an overview of the publications during my PhD research.

1.6.1 Publications in international journals (listed in the Science Citation Index [6])

1. Femke De Backere, Femke Ongenae, Floris Van den Abeele, Jelle Nelis, **Pieter Bonte**, Eli Clement, Matthew Philpott, Jeroen Hoebeke, Stijn Verstichel, and Filip De Turck. Towards a social and context-aware multi-sensor fall detection and risk assessment platform. Published in Computers in biology and medicine, Volume 64, Issue 1, pages 307–320, September 2015.

2. **Pieter Bonte**, Femke Ongenae, Femke De Backere, Jeroen Schaballie, Dörthe Arndt, Stijn Verstichel, Erik Mannens, Rik Van de Walle and Filip De Turck. The MASSIF platform : a modular and semantic platform for the development of flexible IoT services. Published in Knowledge and Information Systems, Volume 51, Issue 1, pages 89–126, July 2016.

3. Femke De Backere, **Pieter Bonte**, Stijn Verstichel, Femke Ongenae and Filip De Turck. The OCarePlatform : a context-aware system to support independent living. Published in Computer Methods and Programs in Biomedicine, Volume 140, pages 111–120, March 2017.

4. Mathias De Brouwer, Femke Ongenae, **Pieter Bonte** and Filip De Turck. Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions . Published in Sensors, Volume 18, Issue 10, October 2018.

5. **Pieter Bonte**, Riccardo Tommasini, Emanuele Della Valle, Filip De Turck and Femke Ongenae. Streaming MASSIF: Cascading Reasoning for Efficient Processing of IoT Data Streams . Published in Sensors, Volume 18, Issue 11, November 2018.

6. **Pieter Bonte**, Femke Ongenae and Filip De Turck. Towards Optimizing Hospital Patient Transports by Automatically Identifying Interpretable Causes of Delays . Accepted to International Journal of Software Engineering and Knowledge Engineering.

7. Christof Mahieu, Femke Ongenae, Femke De Backere, **Pieter Bonte**, Filip De Turck and Pieter Simoens . Semantics-based platform for context-aware and personalized robot interaction in the Internet of Robotic Things . Accepted for Journal of Systems and Software.

8. **Pieter Bonte**, Femke Ongenae and Filip De Turck. Subset Reasoning for Event-Based Systems . Revisions submitted to IEEE Access, January 2019.

[6]The publications listed are recognized as 'A1 publications', according to the following definition used by Ghent University: A1 publications are articles listed in the Science Citation Index Expanded, the Social Science Citation Index or the Arts and Humanities Citation Index of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper.

1.6.2 Publications in international conferences (listed in the Science Citation Index [7])

1. Dörthe Arndt, Ben De Meester, **Pieter Bonte**, Jeroen Schaballie, Jabran Bhatti, Wim Dereuddre, Ruben Verborgh, Femke Ongenae, Filip De Turck, Rik Van de Walle and Erik Mannens. Ontology reasoning using rules in an eHealth context. Published in proceedings of International Symposium on Rules and Rule Markup Languages for the Semantic Web, Berlin, Germany, pages 465 – 472, 2015.

2. Femke Ongenae, **Pieter Bonte**, Jeroen Schaballie, Bert Vankeirsbilck and Filip De Turck. Semantic context consolidation and rule learning for optimized transport assignments in hospitals. Published in Posters & Demonstrations Track co-located with the Extended Semantic Web Conference (ESWC), Heraklion, Greece, pages 88–92 2016.

3. **Pieter Bonte**, Femke Ongenae, Jelle Nelis, Thomas Vanhove and Filip De Turck. User-friendly and scalable platform for the design of intelligent IoT services: a smart office use case. Published in Posters & Demonstrations Track co-located with the International Semantic Web Conference (ISWC), Kobe, Japan, 2016.

4. Riccardo Tommasini, **Pieter Bonte**, Emanuele Della Valle, Erik Mannens, Filip De Turck and Femke Ongenae. Towards Ontology-Based Event Processing. Published in proceedings of the 13th International Workshop on OWL - Experiences and Directions (OWLED) / 5th International Workshop on OWL Reasoner Evaluation (ORE), Bologna, Italy, 2016.

5. **Pieter Bonte**, Femke Ongenae, Jeroen Schaballie, Wim Vancroonenburg, Bert Vankeirsbilck, Filip De Turck. Context-Aware and Self-learning Dynamic Transport Scheduling in Hospitals. Published in Posters & Demonstrations Track co-located with the Extended Semantic Web Conference (ESWC), Portoroz, Slovenia, pages 88–92, 2017.

6. Riccardo Tommasini, **Pieter Bonte**, Emanuele Della Valle, Femke Ongenae and Filip De Turck. A Query Model for Ontology-Based Event Processing over RDF Streams. Published in the proceedings of Knowledge Engineering and Knowledge Management (EKAW), Nancy, France, pages 439–453 2018.

7. **Pieter Bonte**, Riccardo Tommasini, Femke Ongenae, Filip De Turck and Emanuele Della Valle. C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing. Submitted to the Extended Semantic Web Conference (ESWC), Portoroz, Slovenia, 2019

[7] The publications listed are recognized as 'P1 publications', according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index - Science or Conference Proceedings Citation Index - Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1.

1.6.3 Publications in other international conferences

1. Floris Van den Abeele, Jeroen Hoebeke, Femke De Backere, Femke Ongenae, **Pieter Bonte**, Stijn Verstichel, Tommy Carlier, Pieter Crombez, K De Gryse, S Danschotter, Ingrid Moerman and Filip De Turck. OCareClouds: improving home care by interconnecting elderly, care networks and their living environments. Published in proceedings of the eight International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth),Oldenburg, Germany, pages 1–2, 2014.

2. **Pieter Bonte**, Femke Ongenae, Jeroen Schaballie, Ben De Meester, Dörthe Arndt, Wim Dereuddre, Jabran Bhatti, Stijn Verstichel, Ruben Verborgh, Rik Van de Walle, Erik Mannens and Filip De Turck. Evaluation and Optimized Usage of OWL 2 Reasoners in an Event-based eHealth Context. Published in proceedings of the forth OWL reasoner evaluation (ORE) workshop, Athene, Greece, pages 1–7, 2015.

3. **Pieter Bonte**, Femke Ongenae, Jeroen Schaballie, Dörthe Arndt, Robby Wauters, Philip Leroux, Ruben Verborgh, Rik Van de Walle, Erik Mannens and Filip De Turck. Semantic intelligence for real-time automated media production. Published in Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC), Bethlehem, USA, pages 1–4, 2015.

4. Ben De Meester, Dörthe Arndt, **Pieter Bonte**, Jabran Bhatti, Wim Dereuddre, Ruben Verborgh, Femke Ongenae, Filip De Turck, Erik Mannens and Rik Van de Walle. Event-Driven Rule-Based Reasoning using EYE. Published in proceedings of the 14th International Semantic Web Conference (ISWC), Bethlehem, USA, pages 1–4, 2015.

5. Dörthe Arndt, Ben De Meester, **Pieter Bonte**, Jeroen Schaballie, Jabran Bhatti, Wim Dereuddre, Ruben Verborgh, Femke Ongenae, Filip De Turck, Rik Van de Walle and Erik Mannens. Improving OWL RL reasoning in N3 by using specialized rules. Published in proceedings of International Experiences and Directions Workshop on OWL, Bethlehem, USA, pages 93–104, 2015.

6. **Pieter Bonte**, Femke Ongenae and Filip De Turck. Learning semantic rules for intelligent transport scheduling in hospitals. Published in proceedings of the Extended Semantic Web Conference (ESWC), Heraklion, Greece, 2016.

7. **Pieter Bonte**, Femke Ongenae, Eveline Hoogstoel and Filip De Turck. Mining semantic rules for optimizing transport assignments in hospitals. Published in proceedings of the International Semantic Web Conference (ISWC), Kobe, Japan, 2016.

8. **Pieter Bonte**, Femke Ongenae and Filip De Turck. Generic semantic platform for the user-friendly development of intelligent IoT services. Published in proceedings of the International Semantic Web Conference (ISWC), Kobe, Japan, 2016.

9. Dörthe Arndt, **Pieter Bonte**, Alexander Dejonghe, Ruben Verborgh, Filip De Turck, Femke Ongenae. SENSdesc: Connect Sensor queries and Context. Published in proceedings of 11th International Joint Conference on Biomedical Engineering Systems and Technologies, Funchal, Portugal, 2018.

10. **Pieter Bonte**, Femke Ongenae and Filip De Turck. Context-aware patient monitoring through sensor streams Published in proceedings of the International Semantic Web Conference (ISWC), Monterey, CA, USA, 2018.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. Future generation computer systems, 29(7):1645–1660, 2013.

[2] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. Computer Networks, 54(15):2787–2805, oct 2010. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1389128610001568, arXiv:arXiv:1011.1669v3, doi:10.1016/j.comnet.2010.05.010.

[3] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. Semantics for the Internet of Things: early progress and back to the future. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012.

[4] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. Knowl. Acquis., 5:199–220, 1993.

[5] S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano. Semantic Integration of Heterogeneous Information Sources Using a Knowledge-Based System. Data & Knowledge Engineering, 36:215–249, 2001.

[6] T. Strang and C. Linnhoff-Popien. A context modeling survey. In Workshop on advanced context modelling, reasoning and management, UbiComp, volume 4, pages 34–41, 2004.

[7] H. S. Pinto and J. P. Martins. Ontologies: How can They be Built? KAIS, 6(4):441–464, 2004.

[8] E. Simperl. Reusing ontologies on the Semantic Web: A feasibility study. Data & Knowledge Engineering, 68:905–925, 2009.

[9] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. Technical report, W3C, http://www.w3.org/TR/owl-ref/, February 2004.

[10] E. Miller. An introduction to the resource description framework. Bulletin of the American Society for Information Science and Technology, 25(1):15–19, 1998.

[11] S. J. Russell and P. Norvig. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.

[12] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen. Towards Expressive Stream Reasoning. In Semantic Challenges in Sensor Networks, 24.01. - 29.01.2010, 2010.

[13] D. L. McGuinness, F. Van Harmelen, et al. OWL Web Ontology Language Overview. W3C recommendation, 10:2004, 2004.

[14] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language Profiles. W3C recommendation, 27:61, 2009.

[15] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. Data Science, (Preprint):1–24, 2017.

[16] U. Hustadt, B. Motik, and U. Sattler. Data complexity of reasoning in very expressive description logics. In IJCAI, volume 5, pages 466–471, 2005.

[17] L. Al-Jadir, C. Parent, and S. Spaccapietra. Reasoning with large ontologies stored in relational databases: The OntoMinD approach. Data & Knowledge Engineering, 69(11):1158–1180, 2010.

[18] A. Hogan, A. Harth, and A. Polleres. Saor: Authoritative reasoning for the web. The Semantic Web, pages 76–90, 2008.

[19] A. L. Rector, J. Rogers, P. E. Zanstra, and E. Van Der Haring. OpenGALEN: open source medical terminology and tools. In AMIA Annual Symposium Proceedings, volume 2003, page 982. American Medical Informatics Association, 2003.

[20] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening ontologies with DOLCE. In International Conference on Knowledge Engineering and Knowledge Management, pages 166–181. Springer, 2002.

[21] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. HermiT: an OWL 2 reasoner. Journal of Automated Reasoning, 53(3):245–269, 2014.

[22] M. B. Sesen, E. Jiménez-Ruiz, R. Banares-Alcántara, and M. Brady. Evaluating OWL 2 Reasoners in the context of Clinical Decision Support in Lung Cancer Treatment Selection. In ORE, pages 121–127, 2013.

[23] R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In OWLED, volume 432, page 91, 2008.

[24] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. RDFox: A Highly-Scalable RDF Store. In ISWC 2015 , Proceedings, Part II, pages 3–20, 2015.

[25] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. SIGMOD Record, 2010.

[26] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data, pages 370–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[27] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. pages 635–644, 2011.

[28] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. SIGMOD Record, 39(1):20–26, 2010. doi:10.1145/1860702.1860705.

[29] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[30] J.-P. Calbimonte, O. Corcho, and A. J. Gray. Enabling ontology-based access to streaming data sources. In International Semantic Web Conference, pages 96–111. Springer, 2010.

[31] S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pages 58–68. ACM, 2012.

[32] M. Rinne, E. Nuutila, and S. Törmä. INSTANS: high-performance event processing with standard RDF and SPARQL. In Proceedings of the 2012th International Semantic Web Conference: Posters & Demonstrations Track-Volume 914, pages 101–104. Citeseer, 2012.

[33] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. IEEE Internet of Things Journal, 3(5):637–646, 2016.

[34] X. Sun and N. Ansari. EdgeIoT: Mobile edge computing for the Internet of Things. IEEE Communications Magazine, 54(12):22–29, 2016.

[35] N. Alhakbani, M. M. Hassan, and M. Ykhlef. An Effective Semantic Event Matching System in the Internet of Things (IoT) Environment. Sensors, 17(9):2014, 2017.

2
The MASSIF Platform: a Modular & Semantic Platform for the Development of Flexible IoT Services

Semantic Web technologies, such as ontologies, have been proposed as an ideal candidate to model IoT data. Ontologies allow to integrate the IoT data, that is heterogeneous by nature. However, there is still a need for platforms that provide IoT services easy access to IoT data. Data is often produced at different levels of granularity, requiring service developers to investigate the underlying structure and level of granularity by which data is produced by each data source. In this chapter, we propose a semantic enabled publish/subscribe platform that allows IoT services to subscribe to the data they need, by defining their data need on an ontological level. The platform proposes reasoning capabilities, allowing services to subscribe to implicit data and which largely simplifies the subscription process as services can subscribe to high-level ontological concepts, eliminating the need to worry about the lower level details. The platform is more than a semantic publish/subscribe system. It provides mechanisms to annotate raw data if data is not yet mapped to the semantic model. It enables scalability by decoupling the data producers and consumers, and improves performance by providing each consumer its own context model. This is done while guaranteeing the correctness of the filtered data. Chapter 3 builds upon this platform and further extends it to allow services to subscribe to volatile streams and incorporate time-aware definitions. Chapter 5 provides approximate reasoning capabilities over large knowledge bases, a technique that can be utilized by the services that subscribe to the platform discussed in this chapter or within the semantic publish/subscribe mechanism. This chapter investigates Research Question 1: “How can we provide IoT services fine-grained access to IoT data in a flexible manner?" and validates Hypothesis 1: "Using an ontology-enabled publish/subscribe platform will allow semantic and flexible service subscription.".

⋆ ⋆ ⋆

P. Bonte, F. Ongenae, F. De Backere, J. Schaballie, D. Arndt, S. Verstichel, E. Mannens, R. Van de Walle and F. De Turck.

Published in Knowledge and Information Systems, Volume 51, Issue 1, pages 89–126, July 2016

Abstract

In the Internet of Things (IoT), data-producing entities sense their environment and transmit these observations to a data-processing platform for further analysis. Applications can have a notion of context-awareness by combining this sensed data, or by processing the combined data. The processes of combining data can consist both of merging the dynamic sensed data, as well as fusing the sensed data with background and historical data. Semantics can aid in this task, as they have proven their use in data integration, knowledge exchange and reasoning. Semantic services performing reasoning on the integrated sensed data, combined with background knowledge, such as profile data, allow extracting useful information and support intelligent decision making. However, advanced reasoning on the combination of this sensed data and background knowledge is still hard to achieve. Furthermore, the collaboration between semantic services allows to reach complex decisions. The dynamic composition of such collaborative workflows that can adapt to the current context, has not received much attention yet.

In this paper, we present MASSIF, a data-driven platform for the semantic annotation of and reasoning on IoT data. It allows the integration of multiple modular reasoning services, that can collaborate in a flexible manner to facilitate complex decision making processes. Data-driven workflows are enabled by letting services specify the data they would like to consume. After thorough processing, these services can decide to share their decisions with other consumers. By defining the data these services would like to consume, they can operate on a subset of data, improving reasoning efficiency. Furthermore, each of these services can integrate the consumed data with background knowledge in its own context model, for rapid intelligent decision making. To show the strengths of the platform, two use cases are detailed and thoroughly evaluated.

2.1 Introduction

2.1.1 Background

In the Internet of Things (IoT) paradigm, numerous things are connected to the Internet [1]. Through interactions with these connected things, specific goals can be reached that support our daily tasks. The data these things transmit, originates from numerous heterogeneous sources, each sensing a part of the environment. Combining data from different sources facilitates applications to support context awareness [2]. This enables applications to understand the given situation.

For example, to allow elderly people to stay at their own home as long as possible, fall detection systems combine multiple sensors with background knowledge, such as the profile of the elderly. A high fall detection precision can be achieved by combining the profile, habits and whereabouts of the elderly with multiple sensors, such as motion and pressure sensors. The IoT aims at creating intelligent systems that can support people as much as possible during their daily activities. To achieve this awareness, understanding the raw sensor data is necessary. Collection, modeling, reasoning, and distribution of context in relation to sensor data plays a critical role in order to tackle this challenge [3].

Context-aware systems can acquire, interpret and use context information to adapt their behavior to the current context [4]. They have played an important role in tackling this challenge in previous paradigms. Their proven previous success makes them a solution that is ought to be successful in the IoT paradigm as well [3]. According to Perera, et al. [3], one of the important design principles for context-aware systems is scalability and extensibility. Gartner[1] expects 20 billion connected things to be in use worldwide by 2020. Thus, it should be straightforward to add new sensors and devices to a context-aware system. Additional sensors and devices produce new data that might need to be processed differently. Consequently, it should be possible to easily add extra processing services to extract high-level knowledge. Following this thought, the number of services and applications handling the produced IoT-data will increase rapidly. Consequently, context-aware platforms for the IoT should be easily extensible.

According to Strang, et al. [5], semantics are the preferred mechanism of managing and modeling context. Semantics can aid in the integration of the generated heterogeneous IoT data by enabling interoperability between different sources and providing a uniform model [5, 6]. For example, the profile information of the elderly, the floor plan of the house and the sensor readings have different sources and data formats. Combining them within the semantic model enables interoperability. A concise introduction to semantics can be found in Section 2.1.2.3. However, analysis and mapping of data to the semantic model has to be handled for each source individually. Combining the domain information, such as the sensor readings, with profile information, allows to make personalized decisions.

Semantic reasoning allows to compute logical consequences defined in the semantic model. For example, the model could define an alarming fall as a sensor reading from a fall detection sensor with an accuracy above a certain threshold (e.g., 78%) resulting from a sensor in the home of a resident with a profile that states that the patient is in a wheelchair. When such a sensor reading is detected by the reasoner, it will know it has detected a fall and someone should be called to assist the patient, even when it is not explicitly stated in the sensor data. Utilizing semantic reasoning enables transforming the integrated low-level data into high-level knowledge, allowing accurate and intelligent decisions. However, expressive logics such as Description Logic (DL)-Reasoning have EXPTIME complexity [7], resulting in slow reasoning times with growing datasets [8, 9]. This is inconsistent with the vision that events in the IoT should be processed in a timely manner [2].

[1]http://www.gartner.com/

2.1.2 Related Work

The following section describes existing context-aware and IoT frameworks. Ontologies are discussed and considered to be the most used semantic model in current practice. The discussion of the context-aware and IoT frameworks will focus on four important aspects:

1. The capability to semantically annotate raw data. To be able to extract useful knowledge from the IoT-data, the data needs to be semantically annotated first [10].

2. Inference techniques. The extraction of knowledge is an important instrument in the IoT [2]. More advanced techniques allow to extract more complex knowledge.

3. Context model. Platforms can utilize a central context model that contains all context information in one central knowledge base for easy access, a duplicated context model for resilience or a distributed context model for efficiency.

4. Service Collaboration. Service composition provides functionality to build a specific (IoT) application, which is composed of various independent services [3]. By allowing services to collaborate, more complex tasks can be tackled.

2.1.2.1 Context-Awareness

A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task [11, 12]. Context-awareness frameworks typically support acquisition, representation, delivery and reaction [13].
Various methods have been proposed to model context information. The six most popular are: key-value modeling, markup scheme modeling, graphical modeling, object-based modeling, logic-based modeling and ontology-based modeling. According to many surveys in context-aware computing, ontologies are the preferred mechanism of managing and modeling context [3, 5].
Over 30 distinctive context-aware systems have been developed, each providing a different kind of system. An exhaustive analysis of these systems can be found in Perera, et al. [3] and Li, et al. [14]. The most recent and related semantic context-aware platforms are elaborated upon in the following paragraphs.

SeCoMan [15] is a context-aware platform, designed to provide privacy-preserving solutions in the design of context-aware services. These services or applications are in charge of managing their own context for security issues. The privacy handling itself is implemented using a rule-based approach. However, the platform is aware of locations only, other sensor data cannot be semantically annotated and incorporated.

CoCaMAAL [16] is a cloud-oriented context-aware middleware solution for Ambient Assisted Living (AAL). It is able to annotate and abstract raw data from the AAL systems, based on pre-designed ontologies. Service providers enable specific applications based on the context-aware middleware. They can subscribe to the context-aware middleware by providing service rules. However, these service providers do not collaborate. The context model utilized in the context-aware middleware is duplicated in the cloud, however, it is not possible to isolate the context model for the various services providers.

CASF [17] is a framework for context-aware service discovery and integration. It consist of 3 layers: (i) a physical sensor layer which captures the raw sensor data, (ii) a public context layer that processes the sensor data and administers various context providers, and (iii) a context service layer, which consumes context information from one or more context providers. The context services can consume the provided context, but are not capable of sharing conclusions. The platform uses Web Services for automatic discovery and integration of context information.

Although these frameworks look promising, they do not provide advanced reasoning capabilities, such as description logics, and lack the capability to coordinate high-level workflows.

2.1.2.2 IoT Frameworks

IoT frameworks serve as a middleware solution to provide connectivity for sensors and actuators to the Internet. Numerous IoT frameworks exist, most of them focus on the integration of the devices and sensors, less attention is given to intelligent data processing of IoT data [2]. The following paragraphs discuss recent attempts to process IoT data, more specifically through the use of semantics.

Patkos et al. propose an ambient intelligence framework that combines rule-based reasoning with causality-based reasoning, to reason about actions and causalities [18]. The proposed framework does not provide capabilities to annotate raw IoT data.

The LinkSmart platform [19] was designed to support interoperability and integration of various devices, sensors and services. It provides an abstraction of the devices and sensors as regular programming objects towards the application layer. It allows to compose workflows through the use of business rules to optimize the service composition.

Gray et al., propose a system to annotate and integrate heterogeneous streaming data with stored data, through the use of ontologies [20]. Their approach focuses on the discovery and integration of data sources, both static as streaming data. Reasoning and service collaboration techniques are not presented.

Sense2Web [21] is a multilayer platform, allowing to annotate and integrate sensor data in the form of Linked Data and makes it available to other Web applications via SPARQL endpoints. Its data source layer is modeled using expressive ontologies, allowing high-level data retrieval. However, service collaboration and advance reasoning capabilities are not provided for the service and application layer.

XGSN [22] is an end-to-end, semantic-enabled IoT platform that allows to semantically annotate sensors and processes the produced sensor stream using the Linked Sensor Middleware (LSM). The stream can be archived or processed using stream processors. External application can query the context through Web Services. However, these applications cannot share conclusions back to the context layer.

Ali et al. propose an IoT-enabled communication system that allows the annotation of sensory data through the use of XGSN and the continuous analysis of data streams through the use of a stream query processing module for the detections of events [23]. These events can then be further processed in the Stream Reasoning component that can infer implicit semantic statements. Applications can subscribe to events generated in the centralized stream processing and reasoning layer. However, they cannot collaborate to achieve complex workflows.

OpenIoT [24] is an open source IoT platform enabling the semantic interoperability of IoT services in the cloud. It allows the integration and annotation of virtually any sensor. LSM is utilizes and acts as a cloud database which enables the storages of the annotated data streams. Services can access the the annotated data through the use of SPARQL queries. However, service collaboration and advanced reasoning capabilities are lacking.

SOFIA2 [25] is an ontology-based Big Data IoT middleware, allowing interoperability and semantic annotation of multiple heterogeneous devices. It facilitates Complex Event Processing (CEP) to orchestrate the context-data between context consumers. However, besides context subscription based on CEP, there is no real semantic reasoning possible.

These frameworks can annotate data semantically and draw conclusions in an efficient reactive manner. Efficient processing of data is often provided, however advanced reasoning capabilities are still missing. Furthermore, these platforms fail to mutually collaborate in a high-level manner. An overview of the discussed ontology-based context-aware and IoT platforms is summarized in Table 2.1.

Table 2.1: Comparison of existing ontology-based context-aware and IoT platforms. With SP = Stream Processing and SR = Stream Reasoning.

	Year	Semantic Annotation	Inference	Context Model	Service Collaboration
Patkos et al. [18]	2010	/	Rules & Causality	Central	/
LinkSmart	2011	✓	Rules	Central	✓
Gray, et al. [20]	2011	✓	/	Central	/
Sense2Web	2012	✓	/	Central	/
SeCoMan	2013	locations	Rules	Distributed	/
CoCaMAAL	2014	✓	Rules	Duplicated	/
CASF	2013	✓	/	Distributed	/
SOFIA2	2014	✓	/	Central	✓
XGSN	2014	✓	Basic Stream Processing	Central	/
Ali et al. [23]	2015	✓	SP & SR	Central	/
OpenIoT	2015	✓	/	Central	/
MASSIF	2016	✓	OWL DL	Distributed	✓

2.1.2.3 Ontology

Gruber [26] defined an ontology as “an explicit specification of a conceptualization”. An ontology formally describes concepts, properties and their relations, within a certain domain, that can easily be reused [27, 28] in different settings. This allows to model knowledge and make data machine-readable. The Web Ontology Language (OWL) [29] is the most popular language to describe ontologies. Ontologies form an excellent model to integrate data, exchange knowledge and to reason upon that enhanced information. The ontology’s Terminology Box (TBox) describes the concepts and their relations, while the Assertion Box (ABox) contains the data instances with respect to the TBox. The concepts are also called classes and the relations between classes are called object properties. Data properties describe the relation of a concept to a specific data value. The ABox consists of entities of the defined classes (the individuals), their relations and their data properties (the literals). An OWL ontology is described as a collection of axioms, e.g., the class axioms describe the concepts in the ontology in a formal manner. OWL contains multiple sublanguages, each of which varies between expressivity and complexity. The sublanguages are listed below with increasing expressivity [30]:

1. OWL-Lite: supports classification and simple restriction functions.

2. OWL-DL: the largest subset without the loss of computational completeness.

3. OWL-Full: maximal expressivity and syntactical freedom. However, the reasoning process might not be computable.

The popularity of OWL has led to the creation of three OWL 2 profiles [31], each offering a specific subset of the overall expressivity to obtain advantages in specific applications. Each profile is a subset of the OWL DL sublanguage.

1. OWL 2 Existential quantification Language (EL) is useful in applications utilizing an ontology that contains a large number of properties and/or classes.

2. OWL 2 Query Language (QL) is created for applications where query answering is the main task.

3. OWL 2 Rule Language (RL) provides scalable reasoning without sacrificing too much expressiveness.

2.1.3 Objectives

There are still many challenges left to tackle in the IoT-paradigm. Over the past years, efforts in IoT have mainly focused on developing infrastructures to collect and communicate IoT data. Less attention was given to intelligent data processing of this data [2]. The aim of this research is to propose a platform for reactive and real-time data processing, that complies with the following objectives:

- **Semantic Annotation**: To be able to extract useful knowledge from the IoT-data, the data needs to be semantically annotated first [10]. Since it cannot be expected that all sensors and devices generate semantically annotated data natively, it should be possible to enrich raw (sensor) data according to the semantic model.
- **Knowledge Extraction**: The extraction of knowledge is an important instrument in the IoT [2]. It allows to analyze the data, infer new data and to abstract the data for easy data consumption [32]. Utilizing advanced reasoning capabilities, such as description logics, allows the extraction of intelligent high-level conclusions and the execution of intelligent decisions.
- **Extensibility**: To be able to cope with the ever growing amount and types of connected sensors and devices, IoT platforms should be extensible [3]. This allows adding new functionality without altering the existing components. Thus, a plug-in architecture is mandatory.
- **Performance**: Due to the fact that the produced data in the IoT is only temporary valid, a timely processing is required [2].
- **Scalability**: Cisco[2] expects there will be more than 50 billion devices connected to the Internet by 2050. The processing of a growing number of connected devices requires a scalable platform.
- **High-level Workflows**: IoT data consumers are often interested in the high-level concepts, such as concepts higher in the class hierarchy of ontologies or implicit concepts requiring reasoning to infer [2]. Service composition provides functionality to build a specific (IoT) application, which is composed of various independent services. The composition of these services, through the use of workflows, should be possible base on these high-level concepts [3].
- **Real-time Processing:** Many solutions provide IoT analysis tools [33]. However, to detect and immediately react to events, real-time processing of the IoT data is necessary [2].

2.1.4 Paper Organization

The remainder of this paper is structured as follows. Section 2.2 highlights our contribution and provides an overview of the proposed platform. Section 2.3 elaborates on the implementation-specific details of the platform. Two use cases are explained in Section 2.4 to illustrate the capabilities of the platform to derive useful information in a timely manner. These use cases handle data originating from a home care and media setting. The limitations of the platform are discussed in Section 2.5. Section 2.6 highlights the conclusions and introduces tracks for the future work.

2.2 The MASSIF Platform

This section highlights the novelty of the presented work and provides an architectural overview of the platform.

[2] http://www.cisco.com/

2.2.1 Our Contribution

In this paper, we present ModulAr, Service, Semantic & Flexible platform (MASSIF) . It is designed for rapid enrichment and reasoning on IoT data. It uses ontologies to represent context information and different kinds of reasoning can be enabled to derive high-level knowledge. To be able to cope with any kind of input data, the platform allows to semantically annotate raw data. Once the data is annotated, it can be combined with static context data. The use of semantic reasoning allows data consumers to extract useful knowledge, make intelligent decisions and take appropriate actions. Furthermore, the data consumers that extract this knowledge can become data producers and share their findings with other data consumers. This allows the creation of workflows. Since abstractions and high-level concepts are mandatory to create complex workflows, reasoning is performed to coordinate the data flow. To allow dynamic and context dependent workflows, we propose a data-driven workflow composition, where data consumers define the data they would like to receive, based on high-level concepts. When data is processed by the platform, it will check which data consumers are interested in the particular type of data. These types of workflows allow loosely-coupled modular services, which enable extensibility and scalability. Furthermore, a distributed context model is utilized. Each of the data consumers manages its own context. The distributed context model, combined with the data-driven workflows allows each data consumer to operate on a subset of data. Minimizing the dataset enables more effective reasoning, even when utilizing logics such as description logics.

MASSIF is a reactive data-driven platform, in the sense that it reacts to the received data and handles accordingly. This eliminates the need for active polling the various MASSIF components for updates or actions.

2.2.2 The Platform Architecture

The platform consists of five types of components, whereof two of them can be extended to provide specific functionality in each use case. These are called API-components and can be distinguished with the dotted lines in Figure 2.1.

The API-components consist of the Context Adapters, which can semantically annotate the data and Services, which process the semantic data to retrieve high-level knowledge.

The various components that make up the MASSIF Platform are discussed below. The explanation will start with the Semantic Communication Bus (SCB) [34], since it regulates the data flow within the platform.

1. The SCB provides a publish-subscribe mechanism based on high-level ontology concepts. The services can subscribe by defining what kind of data they would like to consume. These definitions are called semantic filter rules and are in fact OWL class expressions. The services can be both consumers and producers. They can decide to share their conclusions by publishing their findings on the SCB. The SCB has its own context model and utilizes reasoning on the subscribed filter rules and the published data to determine which services subscribed to the published data. Note that through the use of reasoning, services can define their input data in

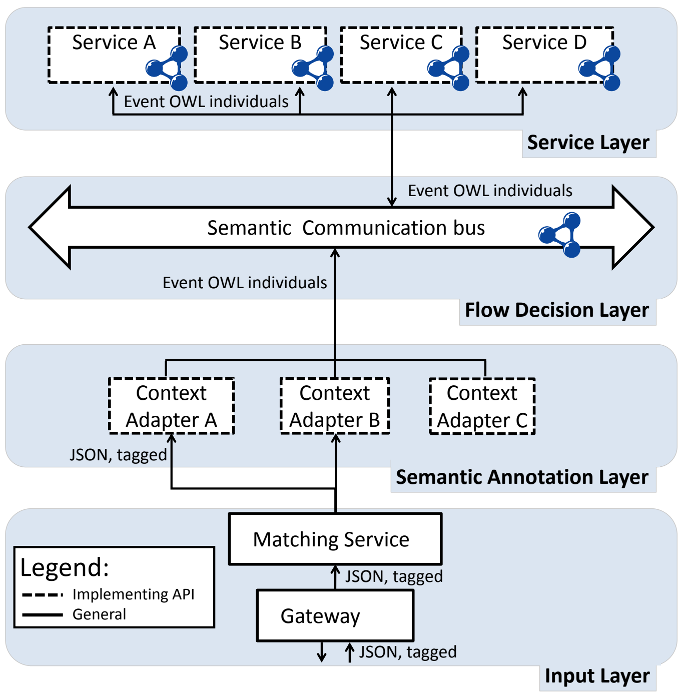


Figure 2.1: Conceptual Architecture of the MASSIF platform.

an abstract and high-level manner.

2. The Gateway serves as the primary communication interface of the platform. It allows both input and output with external devices.

3. The Matching Service inspects the raw data that have been sent from an external source to the Gateway. It selects a Context Adapter that is able to annotate the low-level data according to the semantic model.

4. A Context Adapter receives low-level data from the Matching Service and semantically annotates it. Multiple Context Adapters can be active to annotate numerous kinds of raw data. Once the data is converted to OWL individuals, it is pushed on the SCB. The platform also allows Virtual Context Adapters. These context adapters do not receive data from the Matching Service, but annotate data they capture from existing sources, such as Twitter streams.

5. A Service subscribes to the SCB with one or more filter rules. These filter rules describe the data that the Service would like to consume. Each Service performs a distinct reasoning task or algorithm and can share its inferred knowledge with other Services, over the SCB.

Note that the SCB and the Services each contain their own ontology model and can preload it with background knowledge, such as profile data.

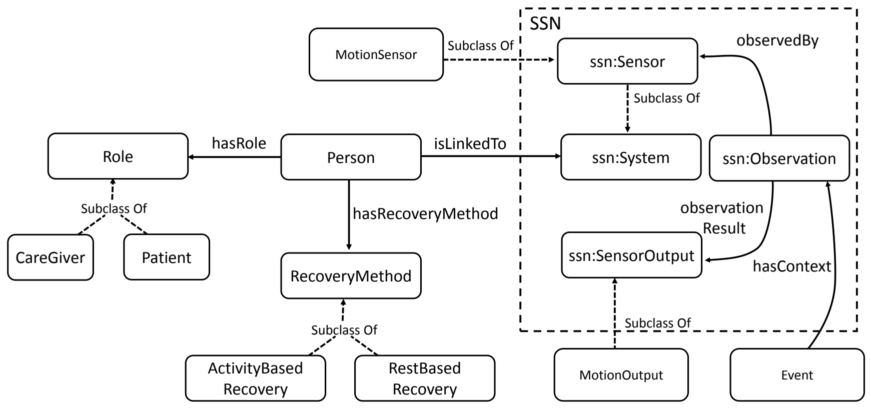


Figure 2.2: Overview of used extension of the SSN ontology.

2.3 Implementation Details

The following section explains the introduced components from Section 2.2 in greater detail. First, a running example is presented that will be used to clarify the further details of the components in the platform. Second, additional implementation details about the used technology are given in Section 2.3.2. Third, more clarification is provided in Section 2.3.3 on how the ontologies are internally represented. Finally, each component is described in great detail in Section 2.3.4, based on the provided information from the first three subsections.

2.3.1 Running Example

To further explain the various components, a running example is introduced to provide practical insights. In the example, a motion sensor is integrated in the home of a patient. The patient should be active for a certain amount of time, to fully recover. The exact upper and lower bound of the allowed active time is patient- and situation-dependent and should not be exceeded. The sensor will capture the activity of the patient and send it to the platform, that will compare it against the background knowledge, which describes the profile of the patient. If the platform detects that not enough/too much activity has been reached, an alarm is triggered.

To model the different concepts and relations for the running example, the Semantic Sensor Network (SSN) [35] ontology is utilized. Figure 2.2 shows the TBox of the extended SSN ontology, describing the designed concepts and their relations. The MotionSensor, MotionOutput and Observation concept are used to model the motion sensor readings.

2.3.2 OSGi

The platform has been developed utilizing the Open Services Gateway initiative (OSGi) [36], which is an extra layer on top of the Java Virtual Machine, enabling modularity. It allows components to be dynamically added, even at run time. This enables our platform to be extended with additional Context Adapters or Services, even when the platform is fully operative. Additionally to the enabled extensibility, OSGi allows straightforward scalability. All components are OSGi Services, which can be distributed utilizing Distributed OSGi [36].

2.3.3 Ontology Representation

The ontologies are internally represented using the OWL API [37]. Data is shared, over the SCB, as a set of OWLAxioms, describing the data semantically. The OWL API provides an OWLReasoner-interface, which is implemented by numerous popular reasoners [38] such as Pellet [39], Hermit [40], Fact++ [41], JFact [42], Chainsaw [43] and RacerPro [44]. This allows services to choose which reasoner to utilize. Various reasoners provide different functionality, complexity and efficiency.

2.3.4 MASSIF Implementation

Since it cannot be expected that all data-producing entities transmit their sensory observations semantically annotated, the platform can annotate raw sensor data itself. It is assumed that each sensor is capable of transmitting its raw data in JavaScript Object Notation (JSON) [45], annotated with a certain tag, which provides some extra information about the origin of the data. The tag itself is added by the sensor gateway. The modular structure of the platform allows to extend the platform in order to accept different input formats. If sensors are able to transmit semantic data, the semantic annotation step can be skipped.

2.3.4.1 Gateway

Low-level data, in JSON format, enters the platforms through the Gateway, as depicted at the bottom of Figure 2.1. Since the platform is fully data-driven, devices can push their data to the platform. The Gateway serves as an entry point and forwards the data to the Matching Service.

2.3.4.2 Matching Service

Listing 2.1: Raw data fragment in JSON format.

```
{
"prefixes": {
"ssn": "http://purl.oclc.org/NET/ssnx/ssn"
},
"userID": "00001",
"data": {
"n": "motion_sensor",
"v": "0.85f",
"tag": "MotionSensor"
}
}
```

The Matching Service analyses the data and decides which Context Adapter can semantically annotate the received data. The decision is made based on the tag in the arriving JSON message. The tag indicates which type of device sent the low-level data. An example of this low-level data can be seen in Listing 2.1. The data describes a motion sensor reading with a precision of 85%.

2.3.4.3 Context Adapters

When a Context Adapter is added to the platform, it provides the types of sensors, i.e., tags of low-level data, it is able to annotate semantically. Each Context Adapter can annotate a specific kind of received data to the semantic model. The result of the annotation phase is semantic annotated data, i.e., ontological individuals. Since each Context Adapter indicates the type of data it is able to annotate itself, additional adapters can easily be added to cope with new types of raw data, such as additional sensors. The Context Adapters enrich the data to a set of OWLAxioms, which are pushed on the SCB. Each Context Adapter provides a mapping, describing how the raw data translates to the semantic data. Listing 2.2 shows an extract of the created OWLAxioms in the annotation phase for the fragment in Listing 2.1.

Listing 2.2: Enriched data as OWLAxioms

```
ClassAssertion(pre:Event pre:event_1),
ClassAssertion(ssn:Observation ssn:observation_1),
ClassAssertion(pre:MotionSensor pre:motionSensor_1),
ClassAssertion(pre:MotionOutput pre:motionOutput_1),
ObjectPropertyAssertion(pre:hasContext pre:event_1 pre:observation_1),
ObjectPropertyAssertion(ssn:observedBy pre:observation_1 pre:motionSensor_1),
ObjectPropertyAssertion(ssn:observation_result pre:observation_1 pre:
     motionOutput_1),
DataPropertyAssertion(ssn:hasValue pre:motionOutput_1 "0.85f"^^xsd:float)
```

The fragment illustrates various assertions: ClassAssertions, ObjectPropertyAssertions and a DataPropertyAssertion. For example, the first ClassAssertion states that the event_1 individual is a member of the class Event. It indicates that an Event has been created that is linked to an Observation. The Observation states the kind of sensor that made the observation and the output of the observation, combined with the actual measured value.

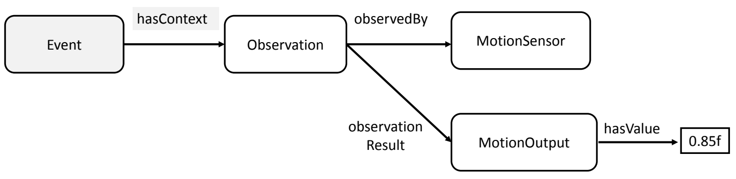


Figure 2.3: Conceptual representation of the events in the platform.

All semantic data passing through the platform are Events. Figure 2.3 shows an example of how the Event is linked to the data. From now on, all data flowing through the platform will be called events. Each event has a starting point of the type Event in the graph-like data structure. With the relation hasContext it is linked to the actual data. Note that the Event concept and the hasContext relation are pictured in grey, since they are required by the platform. All other types and relations, such as the Observation and the MotionSensor are use case specific and are not obliged by the platform.

Virtual Context Adapters are a special type of Context Adapters that do not receive data from the Matching Service, and thus do not register a tag. These adapters annotate data they capture from external sources, such as Facebook and Twitter streams.

2.3.4.4 Semantic Communication Bus

The SCB supports communication and collaboration between the different components. The different components publish their data on the SCB in the form of OWLAxioms. Each component can subscribe to the SCB by passing a filter rule in the form of an OWL class expression. The class describes the kind of data the component wants to consume, on a semantic level. Upon subscription, the registered classes are added to the ontology model of the SCB. Note that the SCB loads its ontology, describing its domain, at startup. When data gets published on the SCB, the published data is temporarily added to the ontology model. The SCB's ontology model now contains the loaded concepts from startup, the registered filter rules as OWL class expressions and the temporary data as OWLAxioms. Through the use of semantic reasoning on the ontology, the type of the published event is retrieved, which matches the subscribed filter rules of those services that would like to consume the data. This way, high-level data coordination is achieved. When the data is forwarded to the selected services, it is removed from the ontology.

In axiom (2.1), an example filter rule is depicted. It shows that the MotionFilter is an Event with a relation hasContext to an Observation and that the Observation should have a relation observedBy with a MotionSensor.

$$\begin{aligned} MotionFilter \equiv\ & Event \\ & \wedge \exists hasContext.(Observation \wedge (\exists observedBy.MotionSensor)) \end{aligned} \tag{2.1}$$

A direct match can be seen between the Event in Figure 2.3 and the filter. When the reasoner in the SCB is asked for the type of the event, it will also return the MotionFilter.

Assume that the following classes are also present in the ontology:

$$MotionSensor \sqsubseteq Sensor \tag{2.2}$$

$$MotionObservation \sqsubseteq Observation \wedge \exists observation\_result.MotionOutput \tag{2.3}$$

The class definition in (2.2) defines the MotionSensor as a subclass of Sensor. To subscribe to all Sensors, including the MotionSensor, one can easily subscribe the filter rule in (2.4).

$$\begin{aligned} MotionFilter_2 \equiv\ & Event \\ & \wedge \exists hasContext.(Observation \wedge (\exists observedBy.Sensor)) \end{aligned} \tag{2.4}$$

To show the added value of the reasoning in the SCB, an additional filter is added in (2.5).

$$MotionFilter_3 \equiv Event \wedge \exists hasContext.MotionObservation \tag{2.5}$$

This rule makes use of the axiom in (3). Even though the MotionObservation is not explicitly defined in the received data, the reasoner knows what a MotionObservation is and will return the filter rule as the type of the event. This enables collaboration between services based on high-level concepts.

The SCB enables intelligent collaboration and distribution of retrieved knowledge between Services.

2.3.4.5 Services

Each Service subscribes to the SCB through the use of one or more filter rules. After processing the consumed data, inferred knowledge can be published to the SCB, to notify other Services about its findings. Each Service contains its own ontology and reasoner. The use of filter rules limits the data each Service receives, resulting in more efficient reasoning, since each Service only needs to incorporate a subset of data. Note that reasoning might become slow when the size of the dataset increases.

Lets assume a new Service, the MotionService, which loads profile information in its ontology at startup and subscribes to all motion data with axiom (2.5). Compared to the event data, big datasets are typically loaded directly into the ontology model of the Services, through the use of tools such as Ontop[3] and D2R[4]. The loading of such static background data allows to combine the low-level event data with background knowledge. This combination facilitates the extraction of high-level knowledge. At run time, the sensor data from the motion sensor is combined with the profile data to check the activity of recovering patients. Note that the time period a patient needs to be active is person-dependent. When aberrant activity has been detected, a Task is generated indicating that someone should check on that person.

A second Service captures all Tasks and tries to assign the most suited person to perform these Tasks. The Service could subscribe to all tasks with the following filter rule:

$$TaskFilter \equiv Event \wedge \exists hasContext.Task \tag{2.6}$$

Additional Services can easily be added to provide extra functionality. Let us assume that the lifetime of certain sensors is limited and when a sensor fails, it starts to produce random values. A Service can be added that captures all sensor values and analyses them to see if they start to produce aberrant values. It can then choose to share this knowledge with the other Services over the SCB.

When ontologies have been constructed with modularity in mind, each Service can load only the necessary parts of the whole ontology, again improving performance. A modular ontology is

[3]http://ontop.inf.unibz.it/
[4]http://d2rq.org/

an ontology that consists of multiple stand-alone ontology modules which improves reasoning efficiency [46]. When a part of the ontology holds a specific profile [47] (e.g., OWL 2 EL, OWL 2 QL, OWL 2 RL) instead of OWL 2 DL, different reasoners can be utilized in each service to optimize performance. Even when no specific profile can be used or no modularity can be detected, different reasoners or techniques can still be used on the whole ontology in each Service to optimize the performance of the task at hand.

Each Service can share its inferred knowledge through the SCB, which might be of interest to other Services which can also process the data and share its knowledge. Combining and passing the results of each Service allows the creation of complex reasoning chains. Since each Service only defines its data of interest and shares its conclusions, dynamic workflows can be created without the need to predefine a static workflow. A special Service, the Notification Service, gathers all final knowledge and sends it to all interested parties outside the platform, through the Gateway. Each service can decide if the data is ready and label it as final knowledge. The flexibility of the platform allows multiple implementations for the Service components. Multiple types of reasoners can be used and other techniques such as data mining and machine learning can easily be applied within these Services to process and retrieve knowledge.

2.3.5 Policy Management

This section elaborates on how inconsistencies are handled in the platform. Since the platform allows users to define their own mapping, describing how raw data should be semantically annotated in the Context Adapters and which data their Services should consume through registering OWL class expressions, inconsistencies can occur. The following scenarios can occur:

1. A Service subscribes with an OWL class expression which makes the SCB's ontology inconsistent. To resolve this, the SCB will check at the time of subscription whether the registered OWL class expression is consistent with the remainder of the ontology. If this is not the case, the subscription of the expression is deemed unsuccessful and it is not added to the ontology. A warning is sent to the service provider that the subscription has failed.
2. A Context Adapter or Service publishes data on the SCB and when reasoned upon in the SCB, a realization inconsistency occurs. This inconsistency can have two causes:

    (a) A Context Adapter or Service failed to format the semantic data correctly. When the malformed data is reasoned upon in the SCB, a realization inconsistency occurs. For example, the types of an individual have been modeled as two classes that are in fact disjoint with each other.

    (b) A Service has recently added an OWL class expression that does not make the ontology inconsistent upon consistency checking, but does causes problems upon realization. For example, a Service can subscribe a new class expression that is disjoint with a previously subscribed filter rule.

When these types of inconsistencies occur, the cause of the problem is first determined. To achieve this, all class expressions that have been registered by the Services are removed and a realization consistency check is performed on the SCB's core ontology and the published data. When this causes inconsistencies, this means that we are dealing with inconsistencies of type 2.a. The platform can then track which component published the data, deactivate it and send a warning.
If this does not cause any problems, this means that we are dealing with inconsistencies of type 2.b. The platform needs to trace the Service that subscribed the OWL class expression that is causing problems. Therefore, the subscribed class expressions are added one by one to the SCB's core ontology and for each addition a consistency check is performed on the published data. The expression that causes the inconsistency is removed from the ontology and a warning is sent to the subscriber.

2.3.6 Supporting Components

The platform provides multiple additional services such as logging, back-up of the knowledge in the Services and visualization of the workload.

2.3.6.1 Journaling

Tailored logging is provided in the core of the platform to track the data flow between the different components. Each component logs its output data, if any, and its destination in the platform. The Services typically log their inferred knowledge they would like to share and indicate the SCB as the destination. After determining which Services are interested in the data, the SCB will log to which Services the data is sent. Logging is important to provide accountability. It allows to justify the choices made at a given time.

2.3.6.2 Backup

Since robustness and resilience is an important aspect in the IoT, a specialized backup system is provided to minimize the data loss upon failure. All messages between components are logged using the journaling and each service makes a backup of its current knowledge base at discrete intervals to further minimize data loss.

2.3.6.3 Cached SCB

The SCB is the central communication link between different components and can thus easily become a bottleneck. The performance of the SCB is dependent on the used ontology, because reasoning is used to determine the services that are interested in the arriving data. To optimize the performance of the SCB, intelligent caching is introduced to match similar events without the need to reason.

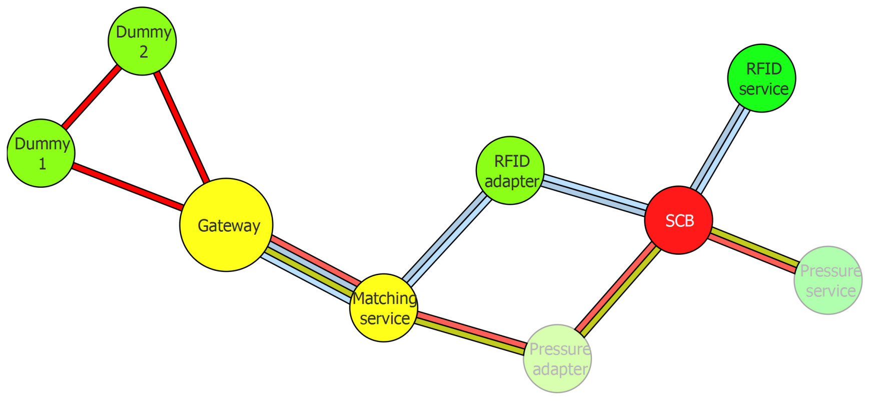


Figure 2.4: Visualization of a data flow in the MASSIF platform, visualized as a graph. The vertices represent the components and the edges represent the dataflows between them.

Since the platform is closed, all data traveling through the SCB has been produced in the platform, it can be assumed that each Context Adapter or Service can only produce a finite number of conceptually distinct messages. Note that only the difference in structure of these messages on a TBox level is considered, not the specific ABox initializations.

The filter rules in the SCB define the high level structure of the expected data. When a filter is triggered, it is possible to map the specific part of the message, responsible for the match, on the filter rule. If this is done on a high level, the presence of the specific structure can be checked with other messages to decide if there is a match.

To determine the data in the message, responsible for the match, the reasoning needs to be reversed and investigated to see which axioms led the reasoner to infer the data as the selected rule, which is an OWL class. The structure of the responsible data is saved in a cache, enabling a simple look-up the next time a similar message passes by. The cache utilizes the Least Recently Used (LRU) strategy, discarding the least used entries first.

2.3.6.4 Visualization

Visualization tools are available to monitor the data flow through the platform. Keeping a global overview of the data flow when operating in a data-driven, service-oriented environment can become complicated. The use of visual aids simplifies this process.

Figure 2.4 depicts how the flow through the platform can be monitored on a graphical level. The workflow is visualized as a graph. The vertices represent the services and the edges represent the dataflows between them. Each color represents a different flow of data.

The workload inside the platform is visualized in Figure 2.5. For each component, the number of messages that are being processed are visualized. This provides a visual understanding of the work distribution. Clicking on one of the components enables a deeper inspection. As visualized in Figure 2.6, the initial JSON-message is shown and the executed SPARQL Protocol and

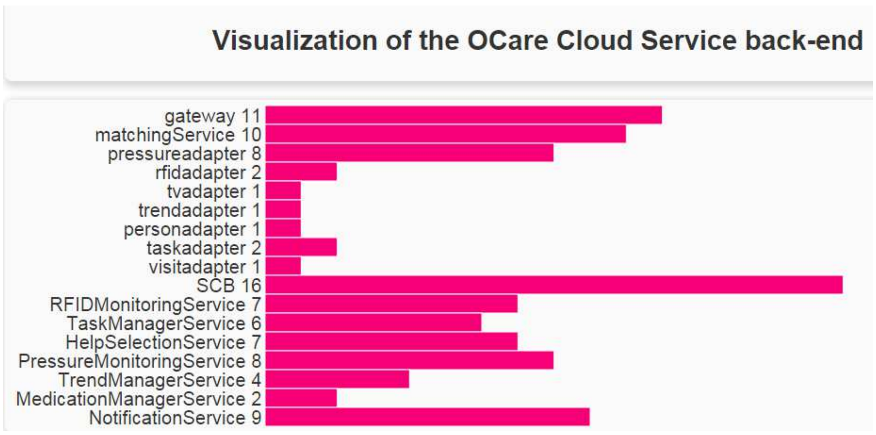
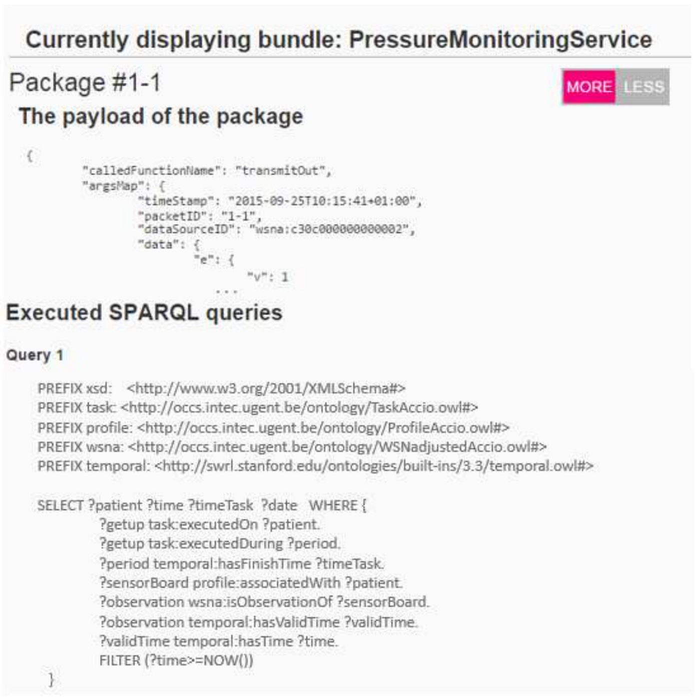


Figure 2.5: Visualization of the platform workload. The number of the current processed messages is shown for each component.



Currently displaying bundle: PressureMonitoringService

Package #1-1

The payload of the package

```
{
        "calledFunctionName": "transmitOut",
        "argsMap": {
                "timeStamp": "2015-09-25T10:15:41+01:00",
                "packetID": "1-1",
                "dataSourceID": "wsna:c30c000000000002",
                "data": {
                        "e": {
                                "v": 1
                        ...
```

Executed SPARQL queries

Query 1

```
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
PREFIX task: <http://occs.intec.ugent.be/ontology/TaskAccio.owl#>
PREFIX profile: <http://occs.intec.ugent.be/ontology/ProfileAccio.owl#>
PREFIX wsna: <http://occs.intec.ugent.be/ontology/WSNadjustedAccio.owl#>
PREFIX temporal: <http://swrl.stanford.edu/ontologies/built-ins/3.3/temporal.owl#>

SELECT ?patient ?time ?timeTask  ?date   WHERE {
          ?getup task:executedOn ?patient.
          ?getup task:executedDuring ?period.
          ?period temporal:hasFinishTime ?timeTask.
          ?sensorBoard profile:associatedWith ?patient.
          ?observation wsna:isObservationOf ?sensorBoard.
          ?observation temporal:hasValidTime ?validTime.
          ?validTime temporal:hasTime ?time.
          FILTER (?time>=NOW())
  }
```

Figure 2.6: A detailed inspection of the PressureMonitoringService, showing the initial JSON-message and the executed SPARQL-queries in the Service.

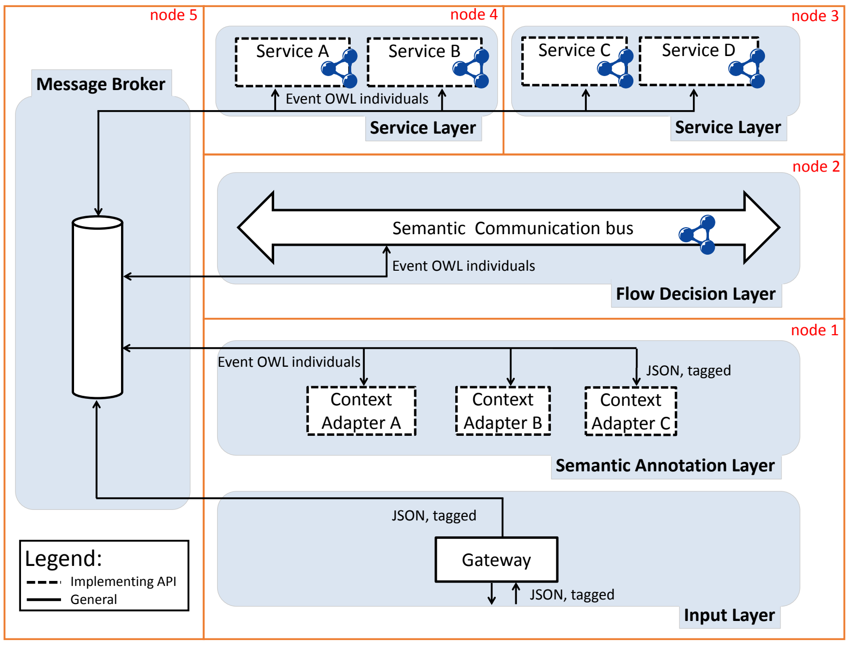


Figure 2.7: Architecture of MASSIF with integrated message broker.

RDF Query Language (SPARQL) [48] query in the selected component. These tools allow a better understanding of the internal flow of messages in the platform.

2.3.6.5 Message Broker

To enable resilient distributed communication, MASSIF allows the integration of highly efficient message brokers such as RabbitMQ [5] and Kafka [6]. The integration of a message broker allows to communicate with non-semantic services. Communication wrappers have been provided, such that nothing needs to be changed in the implementation of existing Context Adapters or Services. These wrappers function as an additional layer between the existing components and the message broker and handle all communication. Furthermore, one can choose how to distribute the components over various distributed nodes, e.g., Services requiring huge amounts of processing power can be run on separate nodes of a processing cluster. Figure 2.7 visualizes the message broker integration in MASSIF. As depicted in Figure 2.7, one can choose to distribute the Gateway and Context Adapters on one node, since they require low processing, the SCB on another node and to distribute the Services over two node since they require most processing resources.

[5]https://www.rabbitmq.com/
[6]http://kafka.apache.org/

2.4 Two Use Cases

MASSIF has been evaluated in the Organizing Home Care Using a Cloud-based Platform (OCCS)[7] project and the R.A.M.P.[8] project. R.A.M.P. is short for Real-time Automation of Media Production for interactive radio and conferences. The use cases illustrate the strengths and the performance of the platform in two real-life scenarios. Both cases combine low-level data with background knowledge to extract high-level knowledge.

2.4.1 eHomeCare

The following sections describe the realization of the OCCS project and give a general overview, a description of the used ontology, an overview of the created Services and Adapters and finally an evaluation of the created system.

2.4.1.1 General Overview

The OCCS project presents a pervasive health use case, demonstrating how healthcare can benefit from the IoT. The OCCS project tackles problems in the home care environment, enabling care organization through cloudy-like services. Hospitals and residential care homes need to cope with the increasing elderly population and the shift from acute to chronic diseases, resulting in a reduced number of available places. An elaborative project description can be found in De Backere et al. [49].

The care receiver's home has been provided with a small amount of discrete sensors and a specialized TV or tablet to interact with. Each caregiver has a smartphone to interact with the platform. By combining the low-level data from the sensors and the smartphones through semantic reasoning, high-level knowledge can be retrieved. From this high-level knowledge, it can, for example, be decided that a care receiver has not been able to get out of bed alone, since the bed pressure sensor is abnormally long active. The system can then decide who might be the designated person to help the care receiver. The selection of the best suited caregiver is based the on the type of relationship the patient has with his helping staff, the competences of the caregivers and their status.

2.4.1.2 The Ontology

The Ambient-Aware Continuous Care Ontology (ACCIO) [50] is used to model all data in the home care environment. Figure 2.8 shows that it is a modular ontology, containing multiple ontology layers. This allows each distinct Service to load only the necessary part of the ontology. An extensive elaboration of the ontology can be found in Ongenae, et al [50].

[7] http://www.iminds.be/en/projects/2014/04/07/ocareclouds
[8] http://www.iminds.be/en/projects/2014/03/05/ramp

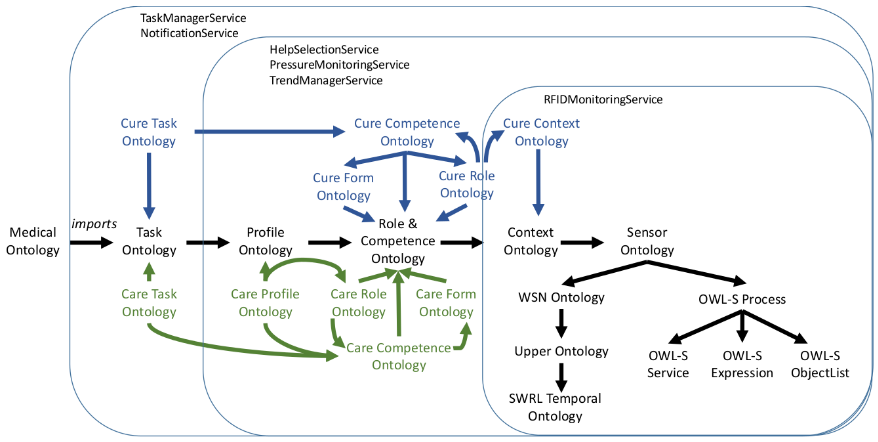


Figure 2.8: Import schema of the used ontology.

The ambient-aware continuous care ontology describes the eHealth sector. It has been constructed in collaboration with stakeholders, such as nurses, caregivers and physicians, social scientists and ontology engineers.

- The Upper ontology describes general classes, relations and axioms. The Upper ontology allows data to be related with a unique ID.
- The Sensor & Observation ontology allows the filtering of data. It imports the Wireless Sensor Network (WSN) ontology and extends it with additional eHealth related concepts.
- The Context ontology describes the contextual information regarding the environment. It contains all localization information.
- The Profile ontology models all profile information about the patients and the staff members. Each Person is linked to a Profile, that can either be a basic profile or a risk profile. This has been described as axioms, allowing the risk patients to be automatically inferred.
- The Role & Competence ontology describes roles and competences in the eHealth sector. Roles, based on competences, can be automatically inferred through the use of axioms.
- The Task ontology models the task and call handling.
- The Medical ontology models medical concepts such as observed parameters and pathologies.

Table 2.2 summarizes the metrics of the proposed ontology.

2.4.1.3 Designed Adapters

Multiple Context Adapters have been implemented, to semantically annotate the raw data. There are seven Adapters present in the OCCS project. Figure 2.9 shows the created Adapters and Services.

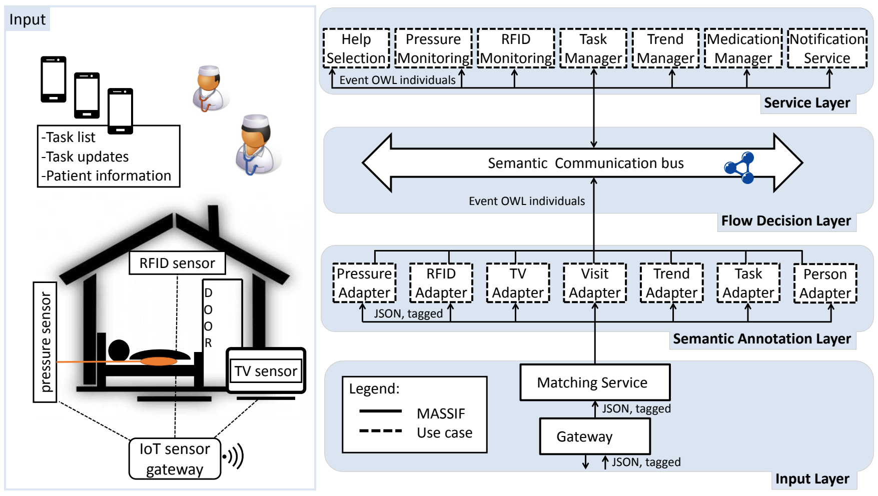


Figure 2.9: An visual overview of the designed Adapters and Services for the OCCS use case. Multiple sensors characterize the environment, enabling context-awareness in the home of the care receiver. These sensors transmit their observation through an IoT sensor gateway to the MASSIF platform.

Table 2.2: Ontology metrics for the used ontology in the eHomeCare use case.

#Axioms	3065
#Logical Axioms	1736
#Individuals	147
#Classes	409
#Object Properties	185
#Data Properties	53
DL Expressivity	SHOIQ(D)

- A PressureAdapter, enriches all data transmitted by a pressure sensor.
- A RFIDAdapter, translates all received Radio Frequency Identification (RFID)s from caregivers logging in, indicating their presence in the home of the care receiver.
- A TVAdapter, annotates all data sent by the TV, capturing the activity of the patient, such as volume and channel changes.
- The TaskAdapter handles all data regarding tasks: newly created tasks, task updates, finished tasks. Some examples of possible tasks are: doing groceries, helping the care receiver out of bed, cooking and cleaning.
- The VisitAdapter handles all data regarding planned visits: new visits, updated visits and canceled visits.

- The PersonAdapter enriches all data describing relations between caregivers and care receivers. For example, when a new caregiver will aid a care receiver, the caregiver is first added to the trust circle of the care receiver. The trust circle indicates which persons the care receiver is familiar with.

- The TrendAdapter shows how the data-driven platform can handle specific requests. Trend information about the activity can be requested, based on multiple sensors. The TrendAdapter only enriches the request data. The processing of the trend data is done in the TrendManagerService. The trend information can give an indication of how active a care receiver is. Some care receivers should do at least some movement during the day.

2.4.1.4 Specific OCCS Services

Multiple Services have been implemented, these are thoroughly discussed below. Note that the Services load static data from a database at startup. This data contains the profile information about the caregivers and the care receivers, information about the numerous sensors and devices and recurrent tasks. This data is parsed to the semantic model and loaded in the individual ontologies of these Services.

- The RFIDMonitoringService registers itself to all semantic RFID-data passing over the SCB. The Service receives this particular data by registering the filter rule depicted in Listing 2.3.

  Listing 2.3: Example filter rule for retrieving RFID data.

  ```
  RFIDFilter ≡ Event and (hasContext some
                      (isObservationOf some (hasSensorPart some RFIDSensor)))
  ```

  The service captures the RFID-data and retrieves the person who is linked to the received RFID, which is not available in the raw sensor data. Finally, it sends an event to the SCB, stating that a specific person has logged in at a given location. Other services, e.g., the TaskManagerService, might be interested in this kind of information. Listing 2.4 shows the generated event from the Service.

  Listing 2.4: Resulting Event from the RFIDMonitoringService

  ```
  ClassAssertion(upper:Event regEvent)
  ClassAssertion(careTask:RegistrationTask regTask)
  ClassAssertion(profile:Person regPatient)
  ClassAssertion(profile:Person regHelper)
  ...
  ObjectPropertyAssertion(upper:hasContext regEvent regTask)
  ObjectPropertyAssertion(task:assignedTo regTask regHelper)
  ObjectPropertyAssertion(task:checkedInWith regHelper regPatient)
  ...
  ```

  The last axiom indicates that the regHelper, which resembles the registered caregiver, has checked in with the patient.

- The TaskManagerService handles all task information. The initial tasks are loaded from the database into the ontology of the TaskManagerService. These tasks cover all activities a caregiver performs to aid the patient. When a location update, originating from the RFIDMonitoringService, arrives, all the tasks that this specific person should perform will be calculated and send, through the NotificationService, to the device of the caregiver. These tasks are derived based on the profile of the caregiver. New, updated and finished task information will also arrive in this service, to be able to keep an up-to-date overview of the task lists.

- The HelpSelectionService receives all activated tasks and determines who is the most suited person to execute these tasks, depending on location, relation with the care receiver and profile. It links the selected person to the task, labels it as final and sends it to the SCB. A thorough discussion of the task assignment can be found in Bonte, et al [51].

- The PressureMonitoringService receives data originating from a pressure sensor, located in the bed of a care receiver. The sensor data indicates the activity of the pressure sensor and thus the presence of the patient. If a patient is longer in bed than usual, without being able to get out alone, a task is generated to get this person out of bed. Listing 2.5 shows an example query that determines if the patient is longer in bed than usual. Note that the time a patient sleeps on average is calculated before and inserted in the query at query time.

  Listing 2.5: Example query

```
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX task: <http://occs.intec.ugent.be/ontology/TaskAccio.owl#>
PREFIX profile: <http://occs.intec.ugent.be/ontology/ProfileAccio.owl#>
PREFIX wsna: <http://occs.intec.ugent.be/ontology/WSNadjustedAccio.owl#>
PREFIX temporal: <http://swrl.stanford.edu/ontologies/built-ins/3.3/temporal
     .owl#>

SELECT ?patient ?time ?timeTask  ?date
WHERE {
?getup task:executedOn ?patient.
?getup task:executedDuring ?period.
?period temporal:hasFinishTime ?timeTask.
?sensorBoard profile:associatedWith ?patient.
?observation wsna:isObservationOf ?sensorBoard.
?observation temporal:hasValidTime ?validTime.
?validTime temporal:hasTime ?time.
FILTER (?time>=AVGSLEEP())}
```

  To better involve the patient in the process, the care receiver is asked if help is necessary. The option to stay in bed is still possible. If help is required, the task is updated, picked up by the HelpSelectionService and the most suitable person is asked to help the care receiver out of bed.

- The TrendManagerService obtains all kinds of sensor data, enabling trend analysis. Compared to the filter in Listing 2.3, the TrendManagerService registers a filter describing the interest in all types of sensor data, as shown in Listing 2.6.

  Listing 2.6: Example filter rule for retrieving all sensor data.

```
SensorFilter ≡ Event and (hasContext some
                    (isObservationOf some (hasSensorPart some Sensor)))
```

The Service stores all sensor data in an external triplestore[9]. This allows the analysis of the activity of the care receiver. The Service allows to request a summary of the activity of one or more sensors. The caregivers can interpret the summary to determine how active the patient has been in a given time interval.

- The MedicationManagerService will not receive any data, but will inform care receivers when they should take their medication and the specific amount. This information is retrieved from Vitalink[10], which is an online government platform, containing a complete patient information dossier. The Service will analyze the dossier and determine which medications should be taken at what intervals. It will send reminders what medication to take at given time intervals to stimulate the patients to take their medication.

- The NotificationService captures knowledge from the SCB that is ready to be shared with the outside world. Other services can indicate that their knowledge can be sent to the caregivers by making it an instance of the Notification ontology concept. It will analyze the arriving event and make sure the information is sent to the correct device. The NotificationService will register to all Notifications on the SCB, as depicted in Listing 2.7.

Listing 2.7: Example filter rule for retrieving all Notifications.

```
NotificationFilter ≡ Event and (hasContext some Notification)
```

Additional services can be added to the platform, providing the caregivers and care receivers with supplementary information.

2.4.1.5 Results

To evaluate the performance of the platform in the described use case, the time necessary to complete one scenario is presented. According to Alshareef et al. [52], an eHealth help system should be able to respond within 5 seconds. The scenario consists of an automatic trigger from a pressure sensor, informing the system that the patient is still in bed. The system will notice that the patient is longer in bed than usual and will automatically send a notification to the patient. The patient can inform the system if help is needed. This way, the patient is involved in the automated decision process. If the patient decides that help is required, the system will search for the most suited caregivers to aid the patient. They automatically receive a message with the question if they could go and help the patient out of bed. Note that the message automatically disappears if one of the caregivers accepts the task.

The scenario was evaluated 35 times. The first three and the last two results were dropped to eliminate the influence of the warm-up and cooling down period. The averages are calculated over the remaining 30 iterations. The evaluation was done on a Ubuntu 14.04 server with an Intel Xeon CPU E5520 (16 cores) @ 2.27GHz with 12 GB of memory.

[9] http://stardog.com/
[10] http://www.vitalink.be/

Table 2.3: An overview off all average processing times for each component in each system call. The averages (μ) and the standard deviation (σ) are both given in milliseconds.

	Pressure Sensor		Help Needed		Task Accepted	
	μ(ms)	σ(ms)	μ(ms)	σ(ms)	μ(ms)	σ(ms)
Gateway	<1	<1	<1	<1	<1	<1
MatchingService	0.62	0.49	0.54	0.50	0.54	0.50
ObservationAdapter	5.77	1.22	-	-	-	-
TaskAdapter	-	-	3.15	0.82	3.31	0.87
SCB-0	0.62	0.49	0.46	0.50	0.62	0.49
SCB-1	0.50	0.50	0.54	0.50	0.54	0.50
SCB-2	0.54	0.50	-	-	-	-
HelpSelectionService	55.50	14.18	75.96	14.23	57.54	11.24
PressureMonitoringService	230.50	23.17	-	-	-	-
TaskManagerService	32.65	15.96	31.50	13.25	31.62	8.96
NotificationService	19.38	2.76	22.81	12.99	19.62	1.62

As presented in Table 2.3, it is clear that the platform overhead (Gateway, Adapters, SCB) has limited influence. The table shows multiple SCB entries, this is because the Services need to exchange their inferred knowledge and thus messages pass the SCB multiple times for each call.

Figure 2.10 visualizes the time spent in the Services, grouped for each call. The time spent in the various Services differs, this is due to the fact that each Service performs a different reasoning task. The PressureMonitoringService performs the most complex task in this scenario, as it needs to decide if help should be requested to aid the patient. The HelpSelectionService takes longest in the second call, where the system needs to select the best suited caregivers to aid the patient in the current situation.

Figure 2.11 visualizes the performances of the Cached SCB as discussed in Section 2.3.6.3, compared to the normal SCB. The graph shows the time needed for both buses to execute 15 successive calls. Thus, the x-axis can be seen as a timeline. Note that the warm-up period was not omitted to investigate the performance of the cache over time. At first, the cached SCB is less efficient, because additional reasoning needs to be performed to select the dominant parts of data that should be cached. After a few misses in the cache, only hits occur and the needed time declines to less than 1 millisecond. The normal SCB also starts performing better after a few calls, this is due to the performed optimizations inside the reasoner. Eventually the time spent in the Cached SCB gets down to less than one millisecond.

2.4.2 Media

The following sections describe the realization of the media use case and give a general overview, a description of the used ontology, an overview of the created Services and Adapters and finally an evaluation of the created system.

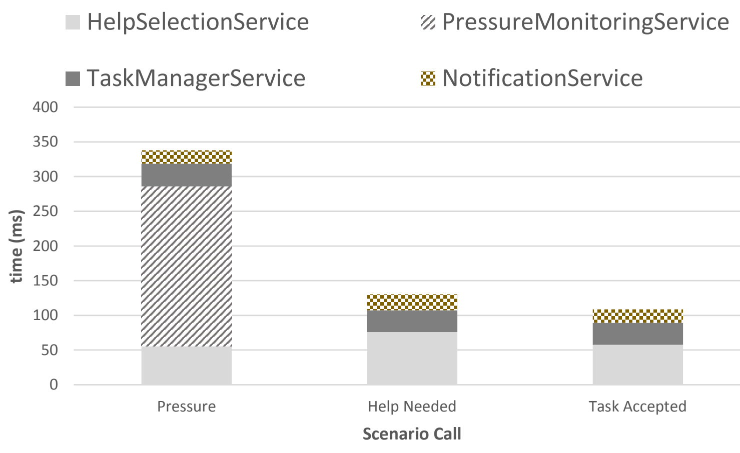


Figure 2.10: Evaluation of the designed Services in the OCCS use case. The scenario consists of three calls. Pressure: the pressure sensor indicates that the patient is still in bed. If the system sees that the patient is longer in bed than usual, it asks the patient if help is needed. Help Needed: the patient indicates that help is welcome and the system will select the most suited caregivers for this task. Task Accepted: one of the caregivers accepts the task.

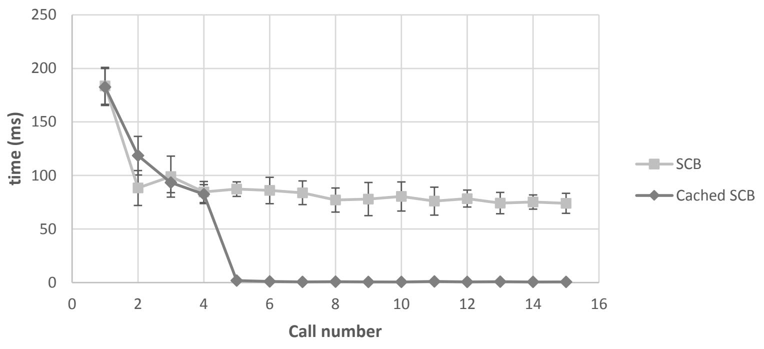


Figure 2.11: Evaluation of the Cached SCB compared to the normal SCB. After a few cache misses, only hits occur and the time spent in the Cache SCB gets down to less than one millisecond.

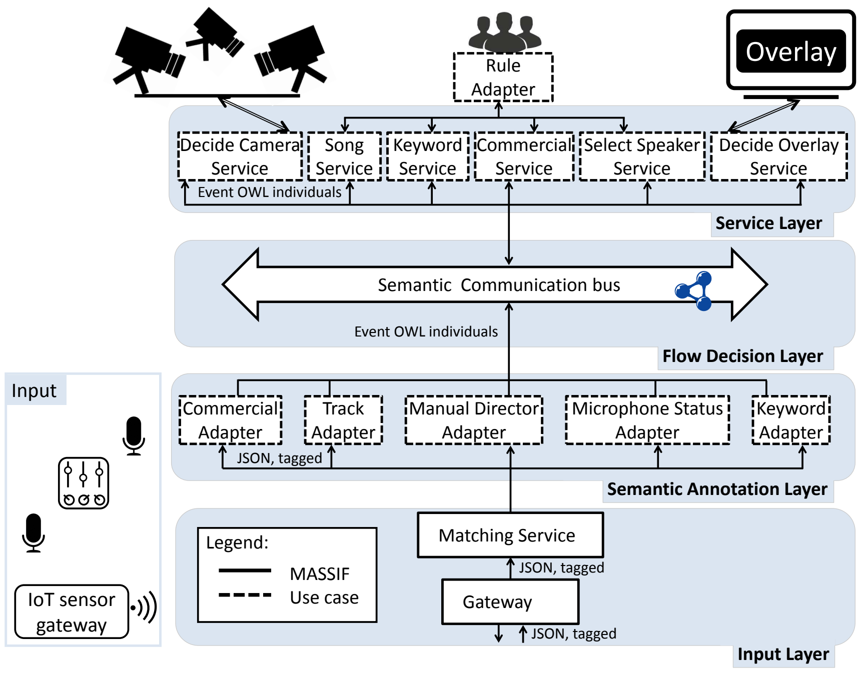


Figure 2.12: A visual overview of the designed Adapters and Services for the R.A.M.P. use case. Multiple sensors characterize the environment, enabling context-awareness in the studio. These sensors transmit their observation through an IoT sensor gateway to the MASSIF platform.

2.4.2.1 General Overview

The general idea behind the media use case is elaborated below, or more specifically the Real-time Automation of Media Production (R.A.M.P.) project. Both a visual radio and conference use case have been designed, however only the radio case will be elaborated upon. The idea behind both cases is equal: the studio or conference room has been equipped with multiple ubiquitous devices, producing contextual data continuously. Combining these with background knowledge allows to infer high-level knowledge to control various cameras and video overlays. A mapping of the created system to the MASSIF platform is depicted in Figure 2.12.

2.4.2.2 Radio

The radio scenario aims at providing visual radio with interactive content, based on the subject of the radio show or the played song in an automated manner, with minimal input from the DJ. The MASSIF platform is the brain of the designed system. It captures and integrates all available data regarding the show and the events inside the studio. The different triggers contain the activity of each microphone, the status of the played songs and commercials, information about certain detected keywords and the configuration of the studio and the cameras. These triggers result in

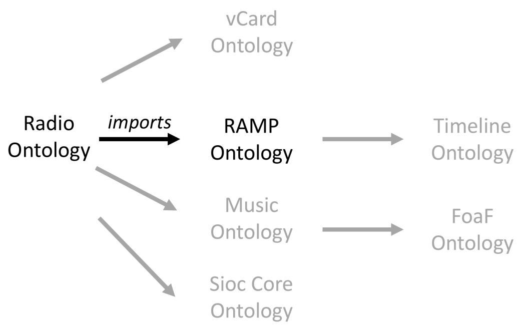


Figure 2.13: Import schema of the used ontology. The external imported ontologies are depicted in grey.

the selection of the best suited camera or the activation of certain visual overlays.

2.4.2.3 The Ontology

The created radio ontology makes use of existing ontologies, as shown in Figure 2.13. The used ontologies are:

- The Music Ontology[11] describes music related concepts.
- Friend of a Friend[12] (FoaF) defines people-related terms.
- Sioc Core Ontology[13] is an ontology for describing the information in online communities.
- The vCard Ontology[14] describes electronic business cards. They contain names, addresses, phone numbers, email addresses, etc.
- The Time ontology[15] describes the temporal content.

Table 2.4 summarizes the metrics of the proposed ontology.

2.4.2.4 Designed Adapters

To allow extracting useful information from the data originating from multiple sources, the data is first semantically annotated in one of the various created Context Adapters.

- The CommercialAdapter receives data describing the start and the stop of a commercial and enriches it to the semantic model.

[11] http://musicontology.com/
[12] http://xmlns.com/foaf/spec
[13] rdfs.org/sioc/ns
[14] http://www.w3.org/2006/vcard/ns-2006.html
[15] http://www.w3.org/TR/owl-time

Table 2.4: Ontology metrics for the used ontology in the media use case.

#Axioms	4989
#Logical Axioms	1582
#Individuals	109
#Classes	247
#Object Properties	384
#Data Properties	203
DL Expressivity	SHOIQ(D)

- The TrackAdapter annotates data regarding the start and stop of a music track.
- The MicrophoneStatusAdapter receives data describing the microphone activity of one of the speakers. This alternates between active and inactive.
- The KeywordAdapter receives the detected keywords during the show and enriches the data. The keywords are detected through the use of speech recognition. The detector listens for a list of keywords that have been extracted from the DJ preparation. The DJ prepares a document a few minutes before the start of the show, containing a summary of the various topics handled during the show.
- The ManualDirectorAdapter is used for the semantic annotation of the data resulting from the overruling mechanism allowing to show a specific person. This is further explained in Section 2.4.2.6.

2.4.2.5 Specific R.A.M.P. Services

The various Services react on the received data and decide to manipulate the cameras or visualize additional data if necessary. Each Service reasons on the integrated data and generates a Sequence of shots that could be shown in the video stream. The creation is based on some precondition, e.g., when the DJ's microphone is active. Semantic Web Rule Language (SWRL)-rules [53] are used to create these Sequences. Listing 2.8 shows this first type of rules in (1). The use of rules allows easy adaptation of the automated process. The rule creates a Sequence when the microphone of the DJ is active.

Listing 2.8: SWRL-rule examples

```
(1) Microphone(?m),capability(?m,DJ),unitState(?m, On)
                                -> Sequence(Sequence_dj)
(2) Track(?t),capability(?g,MainGuest)
                                -> member(Sequence_dj,Shot1),show(Shot1,?g)
```

A second type of rule in (2) generates Shots to be shown in the Sequence. It shows how the Shot is added to the created Sequence. The Shot can show the main guest and can only be added if there is such a guest. The separation of the two types of rules allows multiple combinations in each Service, where multiple rules of each type can be active. When an event arrives at one of the Services, the reasoner retrieves all instances of the type Sequence and Shot and their attributes, leaving the creation of the Sequences and Shots up to the reasoner.

The created Services are elaborated below:

- The Select Speaker Service gathers all information regarding the microphone activity, the configuration of the room and the information of who is sitting on which seat. Combining the low-level microphone activity with the room configuration allows to determine which exact person is speaking. Note that only the microphone activity arrives as an event at run time, the room configuration and the profile information is loaded into the ontology at startup. Based on the defined rules, the outcome of the reasoning task determines who should be selected to show.
- The Decide Camera Service captures the possible shots, e.g., from the Select Speaker Service and determines what the best suited cameras and camera positions are to show a given person. The service loads the camera configuration at startup, stating the possible presets for each camera. The presets define which seats can be shown with a given certainty and quality. The Decide Camera Service will use reasoning to determine the best camera preset for a specific shot in a selected sequence. The sequence gets selected based on its priority. When a sequence of shots arrives with a higher priority, the current sequence will be interrupted and the new sequence will be shown. To provide fluent camera switching towards the viewer, the service will make sure that the same camera with a different preset is never selected sequentially. If this would be the case, the viewer would witness the repositioning of the camera. Therefore, after the selection of a new camera shot, the service will wait until the repositioning has finished before switching shots. Reasoning is performed to enable the camera selection which facilitates fluent camera positioning.
- The Decide Overlay Service captures the various activities in the studio and selects the correct overlay based on those activities. For example, different overlays are shown based on the fact that someone is speaking, a keyword has been detected or a song is playing.
- The Commercial Service contains all the information regarding the played commercials. It provides rules that can specify how to react on the starting or stopping of a commercial. These rules define who should be shown in case of the described event.
- The Song Service captures the information about the played songs. The rules can define how to react when a song is started or stopped.
- The Keyword Service is a bit different since it does not allow any manipulation through the use of rules. It receives a spoken keyword as input and will determine which overlay should be shown upon detecting the keyword. This means that the outcome cannot be adapted.

2.4.2.6 Reasoning Manipulation

To allow control over the automated video composition, a visual Rule Adapter is provided which allows end users to adapt the reasoning decisions in the Services.

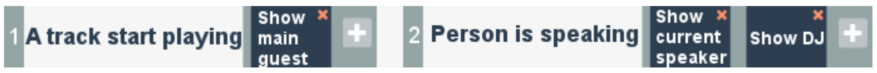


Figure 2.14: User interface for the adaptation of the rules by non-technical users.

The rules in each Service can be adapted to manipulate the automated process. The manipulation of the rules has been abstracted, eliminating the need for the end user to have specific knowledge regarding rules or the ontology. Each Service provides high-level generic rules, which can be made more specific. The provided rules are based on the possible events, during the show or conference. As shown in Figure 2.14, a possible rule might be the fact that a track starts playing and multiple predefined actions can be chosen for that fact. Listing 2.9 shows an example of a high-level rule with the possible event as the antecedent in (2) and the predefined actions as consequences in (5) and (7).

Listing 2.9: High-level rule example

```
(1) {"description": "A track starts playing",
(2)  "antecedent": {
(3)    "rule": ["Track(?t), q:isActive(?t,true) -> Sequence(Sequence_Song)"
     ]},
(4)  "consequences": [
(5)    {"subRule":"Track(?t), capability(?guest, MainGuest)
             -> member(Sequence_Song, Shot1), show(Shot1, ?guest)",
(6)     "description":"Show the main guest"},
(7)    {"subRule":"Track(?t), capability(?dj, DJ)
             -> member(Sequence_Song, Shot2), show(Shot2, ?dj)",
(8)     "description":"Show the DJ"}]}
```

The descriptions in (1), (6) and (8) show how the rules are mapped to readable sentences, alleviating the non-technical user from the technical details.

The antecedent and the consequences contain a (sub)rule, which is a valid SWRL-rule. When the end user selects one (or more) of the predefined consequences, the antecedent rule (3) and the selected subrules (5) or (7) are added to the ontology, allowing the reasoner to incorporate the users preferences.

To provide the viewer a more natural experience, additional properties can be set.

- Priorities: Since multiple rules can be activated at the same time, the video composition can be fine-tuned by specifying which rules have a higher priority than others. For example, the fact that the DJ is speaking might be more important than the fact that a track starts playing.

- Randomness: As depicted in Figure 2.14, multiple actions (subrules) can be selected for one antecedent. The order in which these actions occur can be specified. However, one can add a randomness factor to mix up the order and provide a more natural flow.

- Timing: The time period each camera shot is selected, can be specified. For example, one could opt to capture the DJ longer than his guests.

The Decide Camera Service takes all these options into account when deciding what to show and when to show it. When a high-priority Sequence arrives, the current camera shot should be interrupted, even when the specified timing has not been passed.

2.4.2.7 Reasoning Overruling

A Manual Director tool was designed to explicitly overrule the automatic camera selection. This provides the end users control over the automated process. Through the use of the Manual Director tool, one can manually select the seat that should be captured on camera. This overrules the selection made by the automated reasoning process. A special Context Adapter can enrich this data and directly create a shot sequence of one shot with the highest possible priority. This shot sequence will be captured by the Decide Camera Service. This Service will show the desired seat directly since this sequence has the highest possible priority. In the manual director, it is possible to define how many seconds the manual director should overrule the system, which corresponds to the time period a shot should be shown in a shot sequence. After the defined period of time, the automatic selection of shots continues.

2.4.2.8 Results

To evaluate the performance of the platform in the described use case, the time the system needs to coordinate a typical radio show was evaluated. Since this coordination should be visually appealing, the system should react within 100 milliseconds. According to Card et al. [54], a delay of 100 milliseconds is not troublesome for the human perception. The show consists of the following events:

1. The DJ turns on his microphone and talks for 5 seconds.
2. A special keyword is detected.
3. The DJ turns off his microphone.
4. A new track starts playing
5. The track stops playing.
6. A commercial starts playing.
7. The commercial stops.

The scenario was evaluated 35 times. The first three and last two results were dropped to eliminate the influence of the warm-up and cooling down period. The averages are calculated over the remaining 30 iterations. The evaluation was done on a Ubuntu 14.04 server with an Intel Xeon CPU E5520 (16 cores) @ 2.27GHz with 12 GB of memory.

Table 2.5: An overview off all average processing times for each component in each system call. The averages (μ) and the standard deviation (σ) are both given in milliseconds.

	Mic On		Keyword Detected		Mic Off		Track Start		Track Stop		Commercial Start		Commercial Stop	
	μ(ms)	σ(ms)	μ(ms)	σ(ms)	μ(ms)	σ(ms)	μ(ms)	σ(ms)	μ(ms)	σ(ms)	μ(ms)	σ(ms)	μ(ms)	σ(ms)
Gateway	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
MatchingService	0.31	0.46	0.38	1.00	0.23	0.42	0.38	0.49	0.31	0.46	0.19	0.39	0.23	0.42
MicrophoneAdapter	1.85	0.82	-	-	1.81	0.83	-	-	-	-	-	-	-	-
KeywordAdapter	-	-	1.58	0.57	-	-	-	-	-	-	-	-	-	-
TrackAdapter	-	-	-	-	-	-	1.50	0.57	1.65	1.00	-	-	-	-
CommercialAdapter	-	-	-	-	-	-	-	-	-	-	1.50	0.57	1.88	0.97
SCB-0	0.46	0.50	0.42	0.49	0.42	0.49	0.50	0.50	0.69	0.72	0.42	0.49	0.42	0.57
SCB-1	0.58	0.49	-	-	0.54	0.50	0.69	0.46	-	-	0.81	0.39	-	-
SelectSpeakerService	13.92	7.52	-	-	11.27	2.30	-	-	-	-	-	-	-	-
SongService	-	-	-	-	-	-	13.81	2.67	12.23	2.55	-	-	-	-
CommercialService	-	-	-	-	-	-	-	-	-	-	13.31	3.11	9.92	1.47
KeywordService	-	-	1.23	0.50	-	-	-	-	-	-	-	-	-	-
DecideCameraService	18.27	3.61	-	-	2.46	0.63	20.92	8.13	1.12	0.51	20.62	5.37	-	-
DecideOverlayService	1.00	<1	1.02	<1	1.08	0.47	1.46	0.50	1.19	0.62	-	-	-	-

Table 2.5 presents the overall average times for all components in each call. It is clear that the Services have the greatest impact on the system, since these components perform reasoning. Figure 2.15 visualizes the average time spent in each Service in each step of the scenario. The other components have negligible impact and have thus been omitted. It shows that the Decide Camera Service needs about 20 milliseconds when the microphone gets turned on, a track starts or when a commercial starts. However, when the microphone is turned off, only a small fraction of time is spent in the Decide Camera Service. This is due to the fact that the system is configured (through the use of the rules) to show an overview camera shot when nobody is speaking. The overview shots do not change and are cached for performance measures. In the other scenario steps (Keyword detected, Track stops and Commercial stops), there is no camera activity, because these Services have been configured not to manipulate the cameras as a results of its incoming triggers. Note that this can be easily changed by updating the rules. It is notable that the Decide

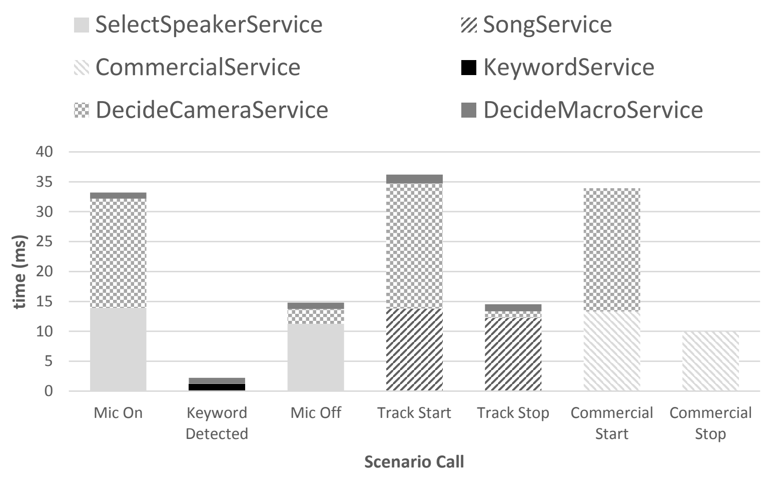


Figure 2.15: Evaluation results of the designed Services for the R.A.M.P. use case. The scenario consist of seven calls, which are explained in the beginning of this Section.

Overlay Service is quite efficient. This is due to the fact that most of the overlay extraction has been done as a preprocessing step. During the show this is only a simple look-up.

Furthermore, it is clear that the system is efficient and causes a maximum delay of less than 35 milliseconds. This is more than acceptable, since a delay of less than 100 milliseconds is not troublesome for the human perception [54]. It is important for visual radio that delays are minimized as much as possible to provide a fluent flow to the end user. The high performance results in a scalable platform that enables the possibility for multiple concurrent scenarios.

2.5 Discussion

We have presented the MASSIF platform, a platform that can successfully enable dynamic and high-level coordination between IoT services. Through two use cases, we have shown that the platform can efficiently handle generated IoT data and also allows to perform complex reasoning. However, there are some known limitations that will be addressed in our future work.

MASSIF is an event-based system, in the sense that it can perform advanced reasoning on event data. Processing of continuous flows of streaming data is currently not the focus of MASSIF. Examples of such streaming data are Facebook and Twitter streams or current measurements of streaming sensors. Each of these streams would be annotated by its own Context Adapter. Currently a single adapter can annotate more than 100 messages per second, as presented in Table 2.3 and 2.5. However, more than 500 messages per second would flood the Context Adapter. Since each component can run on a separate node of a processing cluster, the rest of the system might not suffer from this congestion. Once the data is annotated, the Services need to process the data, these can again congest if the arrival rate is higher than the processing rate. Similarly, if the

MASSIF platform is deployed in a distributed fashion, the rest of the platform will not be hindered when one service congests, at least if they are not dependent on the outcome of the congested service. One way to mediate this congestion is to subscribe the filter rules, which indicate the data the services will consume, more intelligently. One could opt to filter most of the data in the SCB, limiting the data arrival rate in the services. The uptake of stream reasoning is a high priority in our future work. However, current stream reasoning systems do not support complex reasoning.

A second limitation is the use of the tag in the low-level sensor data. In the presented use cases, the sensor integration is supported by the DYnamic, Adaptive MAnagement of Networks and Devices (DYAMAND) platform [55], which was developed at the IBCN[16] research group. The DYAMAND platform allows the detection and integration of sensors and devices. It utilizes a model to make a distinction between sensors. The tag is a result of this model. Similar functionality could be offered by mapping the internal model used by other sensor gateways on the tags. In the future, we wish to offer an API to sensor (gateway) developers that allows them to request the available tags in the system. These tags would be accompanied by a human-readable description. This would allow these developers to choose the appropriate tags for the data they send to the platform. Offering such an API enables the integration of data into the platform that has not been semantically annotated in an easy and straightforward way. When the data does not have a tag, it is filtered by the gateway and thus not processed by the MASSIF Platform. In the future, we wish to make the Matching Service more intelligent by incorporating machine learning and text analysis algorithms that allow to automatically process the incoming data and choose the most appropriate Context Adapter. As such, the platform would be able to handle data that is not tagged.

2.6 Conclusions & Future Work

The number of connected devices will know a rapid increase due to the rising popularity of the IoT. The need to capture, transform and process the produced data by these devices grows. Moreover, the number of services processing the produced data will also increase.

In this paper, the MASSIF platform is presented. It allows semantic annotation of IoT data and the high-level coordination between semantic IoT-services. The platform is fully data-driven and by representing the data semantically, Services can indicate their input data on an abstract level. Each Service can process the received data and share its gained knowledge with other Services through the use of the Semantic Communication Bus. This allows the creation of data-driven workflows that can fulfill complex reasoning chains. By defining their input data, Services operate on a subset of the available data, achieving more efficient reasoning. The applicability of the platform has been shown by presenting two concrete use cases: an eHomeCare case and a media case. The platform has also been thoroughly evaluated by means of the same two use cases. These use cases demonstrate the performance of the platform. Furthermore, they indicate that the platform can be extended to cope with additional data producers and new Services to provide

[16] https://www.ibcn.intec.ugent.be/

extra processing capabilities.

In our future work, stream reasoning techniques will be incorporated for the efficient processing of data streams for less complex reasoning scenarios. To be able to annotate unknown sensor data, machine learning techniques will be investigated enabling the platform to learn how to annotate unknown data. Furthermore, load balancing techniques and automated duplication of the Services will be investigated to provide a truly scalable system.

Acknowledgment

OCCS was partly funded by iMinds and the IWT project nr. 110113. It involves Boone, Familiehulp Gent, Telecom-IT, Televic Healthcare and TP Vision.

R.A.M.P. was partly funded by iMinds and involves Media Innovatie Centrum, Small Town Heroes, Televic Conference and Vlaamse Media Maatschappij.

Authors' contributions

PB carried out the study, developed the platform, ran the experiments and drafted the manuscript. FO and FDB laid the foundations of the SCB. JS contributed to the development of the eHealth use case. DA contributed to the development of the RAMP use case. SV, EM, RVDW and FDT supervised the study, participated in its design and coordination and helped to draft the manuscript. All authors read and approved the final manuscript.

References

[1] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. Comput. Netw., 54:2787–2805, October 2010.

[2] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. Semantics for the Internet of Things: Early Progress and Back to the Future. Int. J. Semant. Web Inf. Syst., 8:1–21, January 2012.

[3] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context Aware Computing for The Internet of Things: A Survey. IEEE Commun Surv. Tut., 16:414–454, 2014.

[4] H. E. Byun and K. Cheverst. Utilizing Context History To Provide Dynamic Adaptations. Appl. Artif. Intell., 18:533–548, 2004.

[5] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. In Workshop Proceedings, 2004.

[6] S. Bergamaschi and et al. Semantic Integration of Heterogeneous Information Sources Using a Knowledge-Based System. Data & Knowledge Engineering, 36:215–249, 2001.

[7] U. Hustadt, B. Motik, and U. Sattler. Data Complexity of Reasoning in Very Expressive Description Logics. IJCAI International Joint Conference on Artificial Intelligence, pages 466–471, 2005.

[8] A. Hogan, A. Harth, and A. Polleres. Saor: Authoritative reasoning for the web. In The Semantic Web, pages 76–90. Springer Berlin Heidelberg, 2008.

[9] L. Al-Jadir, C. Parent, and S. Spaccapietra. Reasoning with large ontologies stored in relational databases: The OntoMinD approach. Data & Knowledge Engineering, 69:1158–1180, 2010.

[10] F. Wang, L. Hu, J. Zhou, and K. Zhao. A Survey from the Perspective of Evolutionary Process in the Internet of Things. International Journal of Distributed Sensor Networks, 2015:9, 2015.

[11] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a Better Understanding of Context and Context-Awareness. In Handheld and ubiquitous computing, pages 304–307. Springer, 1999.

[12] E. Baralis, L. Cagliero, T. Cerquitelli, P. Garza, and M. Marchetti. CAS-Mine: providing personalized services in context-aware applications by means of generalized rules. KAIS, 28(2):283–310, 2011.

[13] A. K. Dey, G. D. Abowd, and D. Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. Comput.-Hum. Interact., 16:97–166, 2001.

[14] X. Li, M. Eckert, J.-F. Martinez, and G. Rubio. Context Aware Middleware Architectures: Survey and Challenges. Sensors, 15(8):20570–20607, 2015.

[15] A. Huertas Celdran, G. Clemente, J. Felix, M. Gil Perez, and G. Martinez Perez. SeCoMan: A semantic-aware policy framework for developing privacy-preserving and context-aware smart applications. 2013.

[16] A. Forkan, I. Khalil, and Z. Tari. CoCaMAAL: A cloud-oriented context-aware middleware in ambient assisted living. Future Generation Computer Systems, 35:114–127, 2014.

[17] J. Kang and S. Park. Context-Aware Services Framework Based on Semantic Web Services for Automatic Discovery and Integration of Context. International Journal of Advancements in Computing Technology, 5(4), 2013.

[18] T. Patkos and et al. A Reasoning Framework for Ambient Intelligence. In Artificial Intelligence: Theories, Models and Applications, pages 213–222. Springer, 2010.

[19] P. Kostelnik, M. Sarnovsk, and K. Furdik. The semantic middleware for networked embedded systems applied in the Internet of Things and Services domain. Scalable Computing: Practice and Experience, 12(3):307–316, 2011.

[20] A. J. Gray and et al. A Semantically Enabled Service Architecture for Mashups over Streaming and Stored Data. In The Semanic Web: Research and Applications, pages 300–314. Springer, 2011.

[21] S. De, T. Elsaleh, P. Barnaghi, and S. Meissner. An Internet of Things Platform for Real-World and Digital Objects. Scalable Computing: Practice and Experience, 13(1):45–58, 2012.

[22] J.-P. Calbimonte, S. Sarni, J. Eberle, and K. Aberer. XGSN: An Open-source Semantic Sensing. Middleware for the Web of Things. . Terra Cognita and Semantic Sensor Networks, page 51, 2014.

[23] M. I. Ali, N. Ono, M. Kaysar, K. Griffin, and A. Mileo. A Semantic Processing Framework for IoT-Enabled Communication Systems. In The Semantic Web-ISWC 2015, pages 241–258. Springer, 2015.

[24] J. Soldatos and et al. OpenIoT: Open Source Internet-of-Things in the Cloud. In Interoperability and Open-Source Solutions for the Internet of Things, pages 13–25. Springer, 2015.

[25] Indra. IoT Interoperability Platform with a Big Data approach, March 2016. Available from: sofia2.com.

[26] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. Knowl. Acquis., 5:199–220, 1993.

[27] H. S. Pinto and J. P. Martins. Ontologies: How can They be Built? KAIS, 6(4):441–464, 2004.

[28] E. Simperl. Reusing ontologies on the Semantic Web: A feasibility study. Data & Knowledge Engineering, 68:905–925, 2009.

[29] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. Technical report, W3C, http://www.w3.org/TR/owl-ref/, February 2004.

[30] D. L. McGuinness, F. Van Harmelen, et al. OWL Web Ontology Language Overview. W3C recommendation, 10:2004, 2004.

[31] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language Profiles. W3C recommendation, 27:61, 2009.

[32] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. Communications Surveys & Tutorials, IEEE, 17(4):2347–2376, 2015.

[33] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma. A gap analysis of Internet-of-Things platforms. arXiv, pages 1–7, 2015.

[34] J. Famaey, S. Latré, J. Strassner, and F. De Turck. An Ontology-Driven Semantic Bus for Autonomic Communication Elements. In Lecture Notes in Comput. Sci., volume 6473, pages 37–50. Springer Verlag Berlin, 2010.

[35] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The SSN Ontology of the W3C Semantic Sensor Network Incubator Group. Web Semantics: Science, Services and Agents on the World Wide Web, 17:25–32, 2012.

[36] OSGi Alliance. OSGi Service Platform Release 4. http://www.osgi.org/. Accessed 9 September 2015, 2009.

[37] M. Horridge and S. Bechhofer. The OWL API: A Java API For OWL Ontologies. Semantic Web, 2:11–21, 2011.

[38] I. Palmisano. Reasoners, OWL API Support, papers about the OWL API. https://github.com/owlcs/owlapi/wiki/Reasoners,-OWL-API-Support,-papers-about-the-OWL-API. Accessed 23 April 2015, 2014.

[39] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web, 5:51–53, 2007.

[40] R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In OWLED, volume 432, 2008.

[41] D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In International Joint Conference on Automated Reasoning, pages 292–297, Berlin, Heidelberg, 2006. Springer-Verlag.

[42] I. Palmisano. JFact DL Reasoner. http://jfact.sourceforge.net/. Accessed 1 July 2015, 2014.

[43] D. Tsarkov and I. Palmisano. Chainsaw: a Metareasoner for Large Ontologies. In ORE, volume 858. CEUR Workshop Proceedings, 2012.

[44] V. Haarslev, K. Hidde, R. Möller, and M. Wessel. The RacerPro Knowledge Representation and Reasoning System. Semantic Web, 3:267–277, 2012.

[45] D. Crockford. The Application/JSON Media Type For Javascript Object Notation (JSON). Internet informational RFC 4627, 2006.

[46] F. Ensan and W. Du. A knowledge encapsulation approach to ontology modularization. KAIS, 26(2):249–283, 2011.

[47] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language: Profiles. W3C recommendation, 27:61, 2009.

[48] E. Prud'Hommeaux, A. Seaborne, et al. SPARQL Query Language for RDF. W3C recommendation, 15, 2008.

[49] F. De Backere, F. Ongenae, F. Vannieuwenborg, J. V. Ooteghem, P. Duysburgh, A. Jansen, J. Hoebeke, K. Wuyts, J. Rossey, F. Van den Abeele, et al. The OCareCloudS project: Toward organizing care through trusted cloud services. Informatics for Health and Social Care, (0):1–19, 2014.

[50] F. Ongenae, L. Bleumers, N. Sulmon, M. Verstraete, M. van Gils, A. Jacobs, S. D. Zutter, P. Verhoeve, A. Ackaert, and F. D. Turck. Participatory Design of a Continuous Care Ontology - Towards a User-driven Ontology Engineering Methodology. In KEOD, pages 81–90. SciTePress, 2011.

[51] P. Bonte, F. Ongenae, J. Schaballie, B. De Meester, D. Arndt, W. Dereuddre, J. Bhatti, S. Verstichel, R. Verborgh, R. Van de Walle, et al. Evaluation and Optimized Usage of OWL 2 Reasoners in an Event-based eHealth Context. In OWL Reasoner Evaluation Workshop, volume 4. CEUR, 2015.

[52] H. Alshareef and D. Grigoras. First responder help facilitated by the mobile cloud. In Cloud Technologies and Applications (CloudTech), 2015 International Conference on, pages 1–8. IEEE, 2015.

[53] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, World Wide Web Consortium, 2004.

[54] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The Information Visualizer, an Information Workspace. In Proceedings of the SIGCHI Conference on Human factors in computing systems, pages 181–186. ACM, 1991.

[55] J. Nelis, T. Verschueren, D. Verslype, and C. Develder. DYAMAND: DYnamic, Adaptive MAnagement of Networks and Devices. In Local Computer Networks (LCN), 2012 IEEE 37th Conference on, pages 192–195. IEEE, 2012.

3 Streaming MASSIF: Cascading Reasoning for Efficient Processing of IoT Data Streams

IoT data is often very volatile, however, there is still a mismatch between expressive reasoning and volatile data. The data frequency is often too high to enable traditional reasoning techniques. In this chapter, we investigate a cascading reasoning approach, consisting of various layers of processing. The lower layers consist of low complexity processing, such that they can handle large amounts of data. When going up in the hierarchy of layers, the complexity of processing rises and the size of data decreases as each layer only selects the relevant parts of the data. This allows to perform expressive reasoning in the top layers. This chapter extends the platform from Chapter 2, by allowing services to subscribe using high-level ontological concepts to data that is also very volatile. Furthermore, we define how temporal dependencies within the data can be detected. The platform in Chapter 2 was not able to handle data streams, nor was it able to model temporal dependencies between the data. The platform from Chapter 2 is extended by two additional layers in order to efficiently process volatile data streams and detect temporal dependencies. Chapter 4 is an optimization of one of the lower RSP layers of the cascading reasoning approach. Chapter 5 can be utilized to efficiently abstract the events to high-level concept when dealing with large knowledge bases. This chapter investigates Research Question 2: "Can expressive reasoning be performed over highly volatile data streams?" and validates Hypothesis 2: "Using a cascading reasoning system will improve the efficiency of expressive OWL 2 DL reasoning over volatile data streams, enabling to process up to hundreds of events per second.".

⋆ ⋆ ⋆

P. Bonte, R. Tommasini, E. Della Valle, F. De Turck and F. Ongenae.

Published in Sensors, Volume 18, Issue 10, pages 33, November 2018

Abstract In the Internet of Things (IoT), multiple sensors and devices are generating heterogeneous streams of data. To perform meaningful analysis over multiple of these streams, stream processing needs to support expressive reasoning capabilities to infer implicit facts and temporal reasoning to capture temporal dependencies. However, current approaches cannot perform the required reasoning expressivity while detecting time dependencies over high frequency data streams. There is still a mismatch between the complexity of processing and the rate data is produced in volatile domains. Therefore, we introduce Streaming MASSIF, a Cascading Reasoning approach performing expressive reasoning and complex event processing over high velocity streams. Cascading Reasoning is a vision that solves the problem of expressive reasoning over high frequency streams by introducing a hierarchical approach consisting of multiple layers. Each layer minimizes the processed data and increases the complexity of the data processing. Cascading Reasoning is a vision that has not been fully realized. Streaming MASSIF is a layered approach allowing IoT service to subscribe to high-level and temporal dependent concepts in volatile data streams. We show that Streaming MASSIF is able to handle high velocity streams up to hundreds of events per second, in combination with expressive reasoning and complex event processing. Streaming MASSIF realizes the Cascading Reasoning vision and is able to combine high expressive reasoning with high throughput of processing. Furthermore, we formalize semantically how the different layers in our Cascading Reasoning Approach collaborate.

3.1 Introduction

Due to the rise of the Internet of Things (IoT) and the popularity of Social Media, huge amounts of frequently changing data are continuously produced [1, 2]. This data can be considered as unbounded streams. In order to extract meaningful insights from these streams, they should be combined and integrated with background knowledge [3]. For example, in the Smart City of Aarhus [4], sensors have been integrated into multiple aspects of the city: traffic sensors to measure the traffic density, sensors to capture the occupation of parking spots, and pollution sensors to measure the pollution values over the city. Since the sensory data typically only describe the sensor readings, it needs to be combined with additional data, e.g., the type of measurement linked to the sensor and the location of the sensor. Combining streams and integrating background knowledge introduces more context and ensures more accurate results. Semantic Web technologies proved to be an ideal tool to fulfill these requirements [5–7]. Ontology languages, such as Web Ontology Language (OWL), allow one to model a certain domain and formally specify its domain knowledge. Expressive reasoning and Complex Event Processing (CEP) techniques allow one to extract implicit facts from the streams, enabling meaningful analysis [8–10]. Expressive reasoning, such as Description Logic (DL) reasoning [11], which can be used to reason about ontology models, allow one to infer implicit facts conform to the domain knowledge defined in the model. We focus on DL reasoning, as it is a web standard and widely adopted.

For example, a street can be considered to be a 'high traffic street' when there have been at least two high traffic observations and each type of street has other thresholds and requirements in order to accurately label an observation as a high traffic observation. In order to accurately interpret the traffic stream, a well-defined background model is necessary. The more accurately one wants to define its domain, the more expressive the required reasoning has to be to correctly interpret the domain. However, higher expressivity of reasoning requires higher complexity of processing [9]. If we want to detect decreasing levels of traffic, we need to detect a temporal relation between low traffic observations and high traffic observations. More specifically, we need to detect when high traffic observations are followed by low traffic observations within a certain amount of time. Furthermore, we need to be able to filter out only those traffic updates going from high to low occurring in the same location.

RDF Stream processors (RSPs) [12–14] tackle the problem of combining various streams, integrating background knowledge and processing the data. They focus on efficiency of processing streams and only allow low expressive reasoning or no reasoning at all. Existing work on expressive DL reasoning has focused on static [15] or slowly changing [16] data. The problem of performing expressive reasoning over high velocity streams is, however, still not resolved [17]. Furthermore, temporal DL tends to become easily undecidable [18], making it even harder to perform temporal reasoning over high velocity streams. Through the use of CEP engines, temporal dependencies can be defined in various patterns.

However, CEP engines struggle to integrate complex domains, which makes it difficult to define complex patterns [10].

Stuckenschmidt et al. [9] envisioned the possibility to trade off complexity of processing and data change frequency in order to perform expressive reasoning over high velocity streams. They named this vision Cascading Reasoning, presenting various layers of processing, each with different complexities. To the best of our knowledge, this vision inspired several RSP works, but this paper reports the first attempt to realize the vision and offers blueprints for practitioners willing to exploit it in alternative implementations.

To allow the development of services that can provide intelligent decision making based on heterogeneous streaming data, we set the following objectives:

1. Combine various data streams: To make meaningful analysis we need to combine streams from various sensors.

2. Integrate background knowledge: Since the sensory data typically only describe the sensor readings, we need to be able to link additional data, e.g., the type of measurement linked to the sensor and the location of the sensor.

3. Integrate complex domain knowledge: In order to correctly interpret the domain, domain knowledge needs to be integrated. The more accurate the domain definition, the more complex the domain knowledge and the higher the required reasoning expressivity.

4. Detect temporal dependencies: Understanding the temporal domain is often necessary

when processing streaming data, as many events in data streams have temporal dependencies.

5. Easy subscription: To allow service to subscribe to the data of their interest, they should be able to define their information need in a straightforward and declarative manner.

To tackle the challenges of performing expressive reasoning and detecting temporal dependencies over high velocity streams, we introduce Streaming MASSIF, a layered Cascading Reasoning realization. Streaming MASSIF allows IoT services to subscribe to high-volatile streams using high-level concepts and temporal dependencies, which can be evaluated using expressive reasoning techniques. This allows one to tackle highly complex domains, while keeping the subscription definitions simple. For example, since more and more employees have flexible working hours, we would like to create a service that notifies them when it is a good time to go home. More specifically, that is when traffic near their offices starts decreasing. This notification should only be considered if the office allows flexible working hours. To enable this, multiple streams need to be combined and integrated with background knowledge, complex domain knowledge needs to be considered in order to correctly interpret the observations, and temporal dependencies need to be detected to observe the decrease in traffic.

Our Cascading Reasoning approach combines RDF Stream Processing (RSP), expressive DL reasoning, and CEP, in order to perform expressive and temporal reasoning over high volatile streams. We seamlessly combine DL and CEP, enabling the definition of patterns using high-level concepts. This enables the use of complex domain models within CEP and integrates a temporal notion in DL. The integration of RSP tackles the high velocity aspect of the streams. Furthermore, we introduce a query language that bridges the gap between stream processing, expressive reasoning, and complex event processing. This allows the service to easily define the data they would like to subscribe to. Furthermore, we formalize semantically how the different layers collaborate.

We show that Streaming MASSIF is able to handle expressive reasoning and complex event processing over high velocity streams, up to hundreds of events per second.

The paper is structured as follows: Section 3.2 describes the related work. Section 3.3 describes all the required background knowledge to understand the remainder of the paper. Section 3.4 introduces the Streaming MASSIF platform, while Section 3.5 describes the implications of combining layers more formally. Section 3.6 details the evaluation of our platform. Section 3.7 discusses the results, the limitations of the platform, and how our platform compares to the state of the art. The conclusion and our outlook and direction for future work is elaborated in Section 3.8.

3.2 Related Work

We now elaborate on the related work in the literature and the drawbacks of these previous approaches.

EP-SPARQL [13] is an RSP engine that focuses on event processing over basic graph patterns using Allen's Algebra for detecting temporal dependencies. However, the reasoning expressivity is low (RDFS) and the definition of event patterns is complex.

StreamRule [19] is a two-layered platform that combines RSP with Answer Set Programming (ASP) [20]. However, there is no support for additional layers such as CEP and the two layers are not integrated in a unifying query language for easy usage.

Ali et al. [21] proposed an IoT-enabled communication implemented on StreamRule that performs event-condition-actions rules in ASP. This allows one to define action rules on specific events detected in the stream.

In the CityPulse project [22], the combination of RSP, CEP and expressive reasoning through ASP is presented. The combination of RSP and ASP is supported by StreamRule. In order to handle CEP rules, the system can be extended programmatically, which makes the definition and overview of event patterns complex.

To the best of our knowledge, existing Semantic Complex Event Processing (SCEP) solutions focus on enriching events with semantic technologies.

Teymourian et al. [10] proposed a knowledge-based CEP approach where events are enriched using external knowledge bases. The enrichment is defined using multiple SPARQL queries. However, the system is event-based, there is no support for streaming data, and reasoning is only provided in the external knowledge base that is used for the event enrichment. Thus, no reasoning on the events themselves is possible.

Taylor et al. [23] proposed a SCEP approach that allows one to generalize query definition for CEP engines, enabling interoperability. This is done by defining the event processing operators as ontology concepts. These generalized queries can then be translated into a target language, for example in Event Processing Language (EPL). However, reasoning and streaming data are not taken into account.

Gillani et al. [24] extended the SPARQL query language to include CEP operators. However, reasoning is not taken into account. The benefits of decoupling expressive ontological and temporal reasoning through the use of CEP has been shown in Tommasini et al. [25] and Margara et al. [26].

The MASSIF platform [27] is an event-driven platform IoT platform, allowing service subscription using high-level ontological concepts. MASSIF facilitates the annotation of raw sensor data to semantic data and allows the development and deployment of modular semantic reasoning services which collaborate in order to allow scalable and efficient processing of the annotated data. Each one of the services fulfills a distinct reasoning task and operates on a different ontology model. The Semantic Communication Bus (SCB) facilitates collaboration between services. Services indicate in which types of data they are interested in, referring to high-level ontology concepts. The SCB can coordinate the data on a high-level through the use of semantic reasoning.

Although MASSIF is an event-driven platform, it processes one event at a time and is thus not able to process streams nor capture temporal dependencies between events.

3.3 Background on Cascading Reasoning

In this section, we introduce the necessary knowledge to understand the content of the paper. First, we introduce the original cascading reasoning vision and all the frameworks contained in its layers.

3.3.1 The Original Cascading Reasoning Vision

Stuckenschmidt et al.'s vision of **Cascading Reasoning** [9] consisted of four layers: Raw Stream Processing, RDF Stream Processing (RSP), Logic Programming (LP), and Description Logics (DL), as depicted in Figure 3.1. Starting from the bottom, each of the layers increases in complexity of processing and reduces the amount of data that is forwarded to the next level. By reducing the data in each layer, higher complexity layers receive fewer data and can still be utilized efficiently.

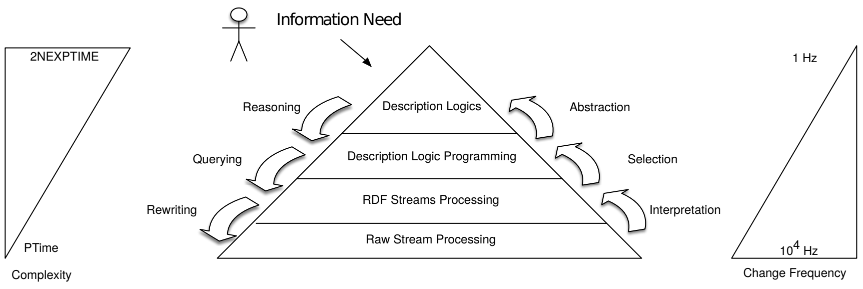


Figure 3.1: Cascading Reasoning.

3.3.1.1 Raw Stream Processing

This application domain comprises the bottom layer of the Cascading Reasoning pyramid and refers to those systems capable of processing large amounts of information in a timely fashion.

Raw Stream processing or **Information Flow Processing** (IFP) [28] describes how to timely process unbounded sequences of information, also called streams. IFP systems are divided into Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) engines.

DSMSs extend traditional Data Base Management Systems to answer continuous queries that are registered and continuously evaluated over time.

CEP Engines [29] are able to capture time dependencies between events. Complex events can be defined through event patterns consisting of various event operators. Examples of these event operators are the time-aware extensions of boolean operators (AND, OR) and the sequencing of events (SEQ). In the following, we present a list of the most prominent CEP operators, guards, and modifiers:

- AND is a binary operator: A AND B matches if both A and B occur in the stream and turns true when the latest of the two occurs in the stream. In Figure 3.2, A AND B matches at t_2 in both Stream 1 and Stream 2.
- OR is a binary operator: A OR B matches if either A or B occurs in the stream. In Figure 3.2, A OR B matches at t_1 in both Stream 1 and Stream 2.

- SEQ is a binary operator that takes temporal dependencies into account. A SEQ B matches when B occurs after A, in the time-domain. In Figure 3.2, A SEQ B matches at t_3 in Stream 1 and at t_2 in Stream 2.
- NOT is a unary operator: NOT A matches when A is not present in the stream. NOT A matches at t_1 in Stream 1 and t_2 at Stream 2.
- WITHIN is a guard that limits the scope of the pattern within the time domain. A SEQ A WITHIN 2s matches in Figure 3.2 at t_3 in Stream 2 and has no match in Stream 1.
- EVERY is a modifier that forces the re-evaluation of a pattern once it has matched. EVERY A SEQ B matches at t_3 in Stream 1 and at t_2 & t_5 in Stream 2 for (A_2, B_2) and (A_3, B_2).

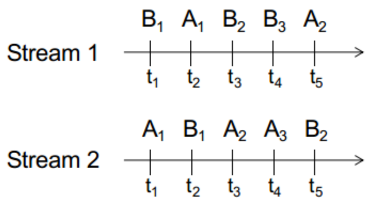


Figure 3.2: Two example streams to illustrate the various event operators. Each of the streams produces events of the type A or B at different time steps, indicated by t_i.

For example, we can define a decreasing traffic observation as every high traffic observation followed by a low traffic observation within a certain amount of time, with the following event pattern:

DecreasingTraffic = EVERY HighTraffic SEQ LowTraffic WITHIN 10m.
However, it is not straightforward in CEP to define what a HighTraffic or LowTraffic exactly is.

For a comprehensive list of operators, we point the reader to Luckham [29]. Note that more advanced temporal relations exist, such as the ones presented in Allen's interval algebra [30].

3.3.1.2 RDF Stream Processing

RDF Stream Processing (RSP) [3] is an extension of IFP that can cope with heterogeneous data streams by exploiting semantic technologies. Resource Description Framework (RDF) streams are semantically annotated data streams encoded in RDF. RSP-QL [31] is a recent query language formalization that unifies the semantics of the existing approaches with a special emphasis on the operational semantics. In the following, we introduce some of RSP-QL definitions that are relevant to understand the next sections:

Definition 1. An RDF Stream S is a potentially infinite multiset of pairs (G$_i$, t$_i$), with G$_i$ an RDF Graph and t$_i$ a timestamp:

$$S = (g_1, t_1), (g_2, t_2), (g_3, t_3), (g_4, t_4), \; \ldots.$$

Since a stream S is typically unbounded, a window is defined upon the stream in which the processing takes place.

Definition 2. A Window $W(S)$ is a multiset of RDF graphs extracted from a stream S. A time-based window is defined through two time instances o and c that are respectively the opening and closing time instants of each window: $W^{(o,c]}(S) = \{(g,t)|(g,t) \in S \wedge t \in (o,c]\}$.

Note that physical windows, based on the number of triples in the window, also exist [12].

Definition 3. A time-based sliding window $\mathbb{W}$ consumes a stream S and produces a time-varying graph $\overline{G}_{\mathbb{W}}$. $\mathbb{W}$ operates according to the parameters (α, β, t^0): it starts operating at t^0, and it has a window width (α) and sliding parameter (β).

We now introduce the concepts of time-varying graphs and instantaneous graphs. The former captures the evolution of the graph over time, while the latter represents the content of a graph at a fixed time instant.

Definition 4. A time-varying graph $\overline{G}_{\mathbb{W}}$ is a function that selects an RDF graph for all time instants $t \in T$ where $\mathbb{W}$ is defined:

$$\overline{G}_{\mathbb{W}} : T \to \{G|G_i \; an \; RDF \; graph\}.$$

The RDF graph identified by the time-varying graph, at the time instant t, is called an instantaneous graph $\overline{G}_{\mathbb{W}}(t)$.

A dataset used within RSP-QL is defined as follows:

Definition 5. An RSP-QL dataset SDS is a set consisting of an (optional) default graph and n named graphs describing the static background data and m named time-varying graphs resulting from applying time-based sliding windows over $o \leq m$ streams, with $m, n \geq 0$.

Example 1. In our example, the SDS is defined as

$$SDS = \{G_0 = \\ G_{sensors}, (w_1, \mathbb{W}_1(S_{traffic_1})), (w_2, \mathbb{W}_2(S_{traffic_2})), ...(w_n, \mathbb{W}_n(S_{traffic_n}))\}.$$

$G_{sensors}$ describe the domain knowledge and the static data about the sensors such as their kinds and their locations. $S_{traffic_i}$ describes the traffic observations and is windowed in $\mathbb{W}_i$. w_i is the window name.

To be able to query the SDS dataset, we define an RSP-QL query:

Definition 6. An RSP-QL query Q is defined as (SE, SDS, ET, QF) where

- SE is an RSP-QL algebraic expression;
- SDS is an RSP-QL dataset;
- ET is a sequence of time instants on which the evaluation of the query occurs;
- QF is the Query Form (e.g., Select or Construct)

3.3.1.3 Description Logic Programming

The reasoning application domain consists of the top two layers of the Cascading Reasoning pyramid. It refers to systems capable of deriving implicit knowledge from the input data combined with rules and domain models. The first reasoning layer in the original Cascading Reasoning vision was Logic Programs.

Logic Programs (LPs) are sets of rules of the form head $\leftarrow$ body that can be read as head "if" body. The original vision of Cascading Reasoning referred to a specific fragment of LPs, called Description Logic Programs (DLPs) [32], which consists of the intersection between Description Logics and those LPs also expressible in First Order Logics. DLPs can be seen of an ontological sub-language of DL that can be encoded in rules.

3.3.1.4 Description Logics

The popularity of OWL has led to the design of OWL2, defining the foundations of OWL2 DL reasoning.

Description Logics [11], the second reasoning layer of the Cascading Reasoning pyramid, are the logical-based formalisms on which OWL2 DL has been built. We introduce the syntax of a simplified DL, explaining the basic notions to understand the remainder of the paper. We refer the reader to Horrocks et al. [33] for a more thorough description of the OWL2 DL logic ($\mathcal{SROIQ}$) and its semantics.

DL languages contain concepts names $A_1, A_2, \ldots$, role names $P1, P2, \ldots$ and individual names $a_1, a_2, \ldots$ A role R is either a role name P_i, its inverse P_i^-, or a complex role $R_1 \circ \cdots \circ R_n$ consisting of a chain of roles. Concrete roles (or data properties) are roles with datatype literals (D) in the second argument. Concepts C are constructed from two special primitive concepts $\bot$ (bottom) and $\top$ (top) or concepts names and roles using the following grammar:

$$C ::= A_i|\top|\bot|\neg C|C_1 \sqcap C_2|C_1 \sqcup C_2|\exists R_1.C_1|\forall R_1.C_1|\exists R_1.D_1|\forall R_1.D_1.$$

Note that the two last concepts are called, respectively existential ($\exists$) and universal ($\forall$) quantifiers.

A Terminological Box (TBox) $\mathcal{T}$ is a finite set of concept (C) and role (R) inclusion axioms of the form

$$C_1 \sqsubseteq C_2 \text{ and } R_1 \sqsubseteq R_2$$

with C_1, C_2 concepts and R_1, R_2 roles. A concept equation ($C_1 \equiv C_2$) denotes that both C_1 and C_2 include each other:

$$C_1 \sqsubseteq C_2 \text{ and } C_2 \sqsubseteq C_1.$$

An Assertion Box (ABox) $\mathcal{A}$ is a finite set of concept and role assertions of the form

$$C(a) \text{ and } R(a, b)$$

with C a concept, R a role, and a and b individual names. We call the concepts assigned to an individual the types of the individual. A Knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ combines $\mathcal{T}$ and $\mathcal{A}$. $\mathcal{I}$ is an interpretation for $\mathcal{K}$. $\mathcal{I}$ is a model of $\mathcal{K}$ if it satisfies all concept and role inclusions of $\mathcal{T}$ and all concept and role assertions of $\mathcal{A}$. This can be written as $\mathcal{I} \models \mathcal{K}$.

OWL2 contains three profiles, each limiting the expressivity power in a different way, to ensure efficiency of reasoning:

- OWL2 RL, which does not allow existential quantifiers on the right-hand side of the concept inclusion, eliminating the need to reason about individuals that are not explicitly present in the knowledge base. Furthermore, it does not allow quantified restriction, e.g., a minimum number of roles, a maximum number of roles or exactly a specific number of quantified roles. This profile is ideal to be executed on a rule-engine.

- OWL2 EL, which mainly provides support for conjunctions and existential quantifiers. This profile is ideal for reasoning over large TBoxes that do not contain, among others, universal quantifiers, quantified restrictions or inverse object properties.

- OWL2 QL, which does not allow, among others, existential quantifiers to a class expression or a data range on the left-hand side of the concept inclusion. This makes the profile ideal for query rewriting techniques.

Note that each of these profiles is a subset of OWL2 DL.

Example 2. In the ontology used to model our domain from our example in Section 3.1, we assign each Office various Policies. Based on these Policies, an Office can be considered a FlexibleOffice or not:

$$\begin{aligned}
&\text{NoFixedHoursOffice} \equiv \text{Office} \sqcap \exists hasPolicy.\text{FlexibleHours},\\
&\text{NoFixedHoursOffice} \sqsubseteq \text{FlexibleOffice},\\
&\text{StartEarlyOffice} \equiv \text{Office} \sqcap \exists hasPolicy.\text{StartEarly},\\
&\text{StartEarlyOffice} \sqsubseteq \text{FlexibleOffice},\\
&\text{StopEarlyOffice} \equiv \text{Office} \sqcap \exists hasPolicy.\text{StopEarly},\\
&\text{StopEarlyOffice} \sqsubseteq \text{FlexibleOffice}.
\end{aligned}$$

To model the observations that capture the various sensor readings across the city, we use the SSN Ontology [34]. We first model observations near flexible offices, and we then model observations near flexible offices that also capture congestion levels:

$$\begin{aligned}
&\text{FlexibleOfficeObservation} \equiv \text{Observation}\\
&\sqcap (\exists observedFeature.(\exists isLocationOf.\text{FlexibleOffice}))\\
&\text{CongestionFOObservation} \equiv \text{FlexibleOfficeObservation}\\
&\sqcap \exists observedProperty.\text{CongestionLevel}
\end{aligned}$$

We can now model for each type of street, which is located near a flexible office, when it should be considered congested. With the congestion level defined as the number of detected vehicles divided by the street length (in meters):

$$\begin{aligned}
&\text{HighTrafficMainRoadNearFlexibleOffice} \equiv\\
&\quad \text{CongestionFOObservation}\\
&\quad \sqcap \exists observedProperty.\text{MainRoad}\\
&\quad \sqcap \exists hasLocation.\text{Location}\\
&\quad \sqcap \exists hasValue > 0.025,\\
&\text{LowTrafficMainRoadNearFlexibleOffice} \equiv\\
&\quad \text{CongestionFOObservation}\\
&\quad \sqcap \exists observedProperty.\text{MainRoad}\\
&\quad \sqcap \exists hasLocation.\text{Location}\\
&\quad \sqcap \exists hasValue < 0.01.
\end{aligned}$$

Note that similar constructions can be made for different types of streets and that all these constructs are also subclasses of the concepts HighTrafficObservation or LowTrafficObservation.

Example 3. In Example 2, we have modeled the TBox. Let us consider a minimal ABox $\mathcal{A}$ describing the office, the road, and their property:

$$\begin{aligned}
&Office(office), hasPolicy(office, pol1),\\
&StopEarly(pol1), MainRoad(road),\\
&CongestionLevel(prop), propertyOf(prop, road),\\
&isLocationOf(road, office).
\end{aligned}$$

The observation capturing the current congestion level can be modeled as

$$\begin{aligned}
&Observation(obs_i), observedProperty(obs_i, prop),\\
&hasValue(obs_i, 0.03).
\end{aligned}$$

By applying reasoning, we can infer from $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ that

$$\begin{aligned}
&\mathcal{K} \models FlexibleOffice(office),\\
&\mathcal{K} \models FlexibleOfficeObservation(obs1),\\
&\mathcal{K} \models CongestionFOObservation(obs1),\\
&\mathcal{K} \models HighTrafficMainRoadNearFlexibleOffice(obs1),\\
&\mathcal{K} \models HighTrafficObservation(obs1).
\end{aligned}$$

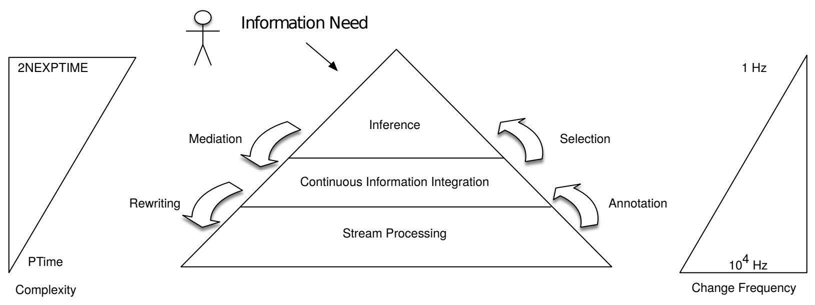


Figure 3.3: A generalization of Cascading Reasoning.

3.3.2 Cascading Reasoning Generalization

Since Stuckenschmidt et al.'s vision of Cascading Reasoning was proposed, several new approaches populated the Stream Reasoning state of the art [17]. We slightly generalize the vision such that it is up to date with the latest developments within the Stream Reasoning domain.

The initial scope of reasoning frameworks was focused mainly on DL and DLP. Recently, temporal logics, non-monotonic LPs and technique for reasoning about time were proposed beside the traditional Stream Reasoning research areas. In the future, we also imagine the integration of online machine learning application, which already showed appealing results, and the combination of deductive and inductive reasoning [35, 36].

In the original cascading reasoning pyramid, the role of RSP was limited to streaming data integration. Although this is utterly meaningful in combination with DL reasoning, data integration is a much more general problem to investigate when data are continuously changing. Moreover, RSP, but also stream processing, can support reasoning tasks (e.g., RSP under entailment or query rewriting).

The updated and generalized cascading stream reasoning pyramid is depicted in Figure 3.3. As in the original vision, it aims at presenting the trade-off between expressiveness and rate of changes in the data.

We now detail each of the layers:

1. **Stream Processing:** At the lowest level, the data streams are processed. Different processing techniques can be used accordingly to the levels above, e.g., which information integration technique is used (if any). This layer can implement stream processing techniques like DSMSs and CEPs or use RSP when dealing with semantically annotated data. Moreover, this level can also solve part of the analytic needs, since it is able to compute descriptive analysis of the streaming data.

2. **Continuous Information Integration:** In order to achieve a high-level view on the streaming data, we need an information integration layer that offers a homogeneous view over the streams. The Continuous Information Integration layer combines data from heterogeneous streams into a common semantic space by the means of mapping assertions that populate

a conceptual model. Two approaches are then possible to access the data: (i) Data Annotation (a.k.a. data materialization), i.e., data are transformed into a new format closer to the information need (ii) Query Rewriting (a.k.a. data visualization), i.e., the information need is rewritten into sub tasks that are closer to each of the original data formats.

3. **Inference:** In a cascading approach, an information need (IN) is formulated accordingly to a high-level view of the data. To enable efficient IN resolution, we need an inference layer that mediates the IN with domain-specific knowledge to the lower layers. Computational tasks at this level have a high complexity. This reduces the volume of data this level can actually process. Therefore, it is necessary to select, from the lower layers, the relevant parts of the streams that this layer has to interpret to infer hidden data. Possible inference implementations range from expressive reasoning, such as DL, ASP, metric temporal logic (MTL), or CEP, to machine learning techniques such as Bayesian Networks (BN) or hidden Markov models (HMM).

The original vision—which consists of raw stream processing, RSP, DL, and logic programming—fits this more general view: the raw stream processing is contained in our Stream Processing layer, RSP is contained in the continuous information integration layer, and DL & logic programming are part of the inference layer.

3.4 Cascading Reasoning with Streaming MASSIF

In this section, we explain how we realized Cascading Reasoning with Streaming MASSIF. We introduce the architecture of Streaming MASSIF and present a Domain Specific Language (DSL) that allows one to target the different layers of the cascading approach.

3.4.1 Layer Design

In the following sections, we design a stream reasoning architecture that fulfills the Objectives and fits the generalized Cascading Reasoning vision. As depicted in Figure 3.4b, our approach consists of two layers that perform four tasks, starting from the bottom: (i) An RSP layer selects the parts of the streams that are relevant. (ii) It also integrates data from different streaming and static sources. (iii) An inference layer enriches the output of the previous layer by deriving implicit data using DL reasoning. (iv) It also performs temporal reasoning via CEP on the inferred abstractions.

3.4.2 Architecture

As discussed in Section 3.2, MASSIF is an event-driven platform that processes one event at a time and is, thus, not able to process streams nor capture temporal dependencies between events. However, its layered architecture and the ability to perform service composition over high-level concepts offer a good base to extend it into a Cascading Reasoning approach. We note that other

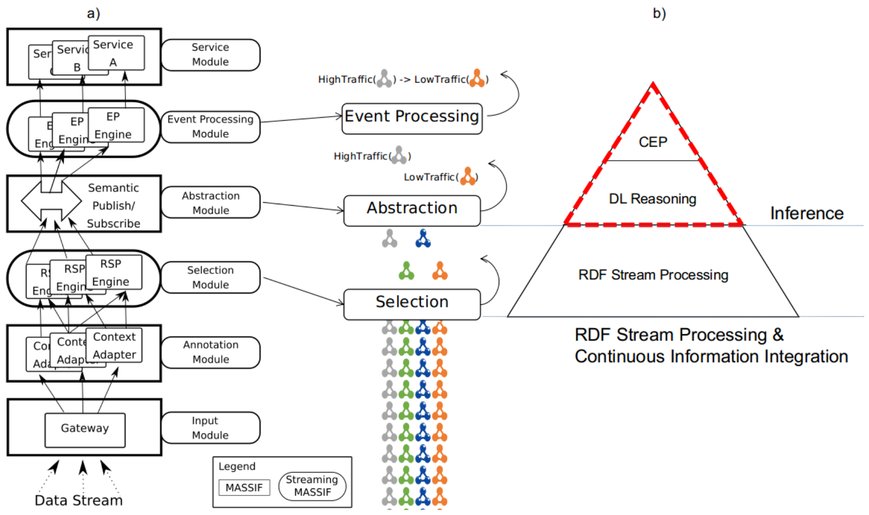


Figure 3.4: Streaming MASSIF architecture and the alignment of the inference (consisting of the DL and CEP layer) and stream processing layer with the event processing, abstraction, and selection modules.

platforms could have been used to realize our Cascading Reasoning approach; however, the layered architecture of MASSIF and enabled service subscription made it an ideal candidate.

To realize our cascading stream reasoning approach, two additional modules have been added on the MASSIF platform, as depicted by the rounded blocks in Figure 3.4a, i.e., a Selection and Event Processing Module. We named the resulting platform Streaming MASSIF. Compared to the original MASSIF platform, the Selection Module allows one to handle streaming data and select only the parts from the data stream that are relevant for further processing. These selections then can be abstracted in the Abstraction Module. The Event Processing Module allows one to detect temporal dependencies between events. Thus, the MASSIF platform allows services to subscribe to high-level events. Streaming MASSIF allows services to subscribe to data streams, extract high-level events, and detect temporal dependencies between those events. Furthermore, this can all be declaratively defined in a unifying language, which is further elaborated in Section 3.4.3.

3.4.2.1 Selection Module

The **Selection Module** implements both the Stream Processing and the Continuous Information Integration Layer of the Cascading Reasoning approach and selects, through RSP, those parts of the RDF stream that are relevant. As depicted in Figure 3.4, the goal of this layer is to minimize that data stream and select only those parts of the stream that are relevant for further processing. We utilized YASPER [37], i.e., an RSP engine recently developed, that fully implements RSP-QL [31] semantics and can consumes RSP-QL queries. YASPER, differently from C-SPARQL [12] or CQELS [14] consumes time-annotated graphs instead of time-annotated triples. Only the selected data are forwarded to the next module. Note that multiple RSP engines can optionally run in parallel, for

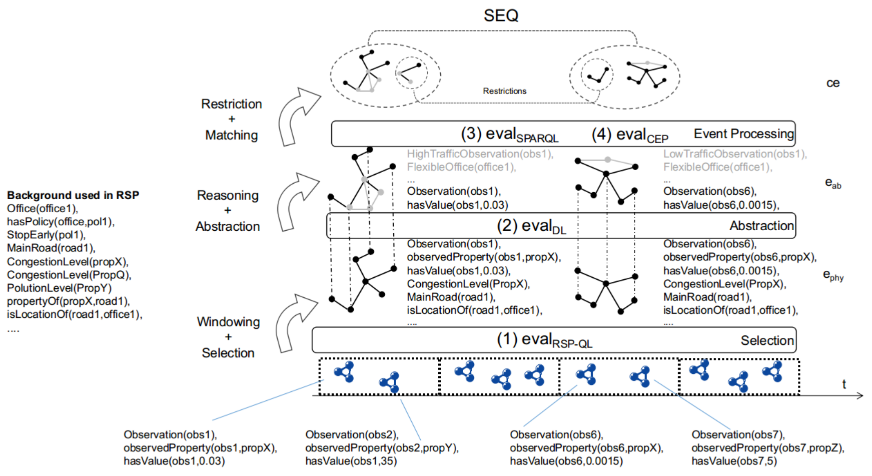

SEQ
Restrictions
ce
Restriction + Matching
(3) eval$_{SPARQL}$
(4) eval$_{CEP}$
Event Processing
HighTrafficObservation(obs1), FlexibleOffice(office1), ...
LowTrafficObservation(obs1), FlexibleOffice(office1), ...
e$_{ab}$
Background used in RSP
Office(office1), hasPolicy(office,pol1), StopEarly(pol1), MainRoad(road1), CongestionLevel(propX), CongestionLevel(PropQ), PolutionLevel(PropY) propertyOf(propX,road1), isLocationOf(road1,office1), ...
Reasoning + Abstraction
Observation(obs1), hasValue(obs1,0.03)
Observation(obs6), hasValue(obs6,0.0015),
(2) eval$_{DL}$
Abstraction
Observation(obs1), observedProperty(obs1,propX), hasValue(obs1,0.03), CongestionLevel(PropX), MainRoad(road1), isLocationOf(road1,office1), ...
Observation(obs6), observedProperty(obs6,propX), hasValue(obs6,0.0015), CongestionLevel(PropX), MainRoad(road1), isLocationOf(road1,office1), ...
e$_{phy}$
Windowing + Selection
(1) eval$_{RSP-QL}$
Selection
t
Observation(obs1), observedProperty(obs1,propX), hasValue(obs1,0.03)
Observation(obs2), observedProperty(obs2,propY), hasValue(obs1,35)
Observation(obs6), observedProperty(obs6,propX), hasValue(obs6,0.0015)
Observation(obs7), observedProperty(obs7,propZ), hasValue(obs7,5)

Figure 3.5: Processing steps of the Streaming MASSIF Cascading Reasoning Approach.

example, to distribute the load of various queries or handle multiple data streams.

For example, at the bottom of Figure 5.6, a traffic observation data stream is visualized. As in a realistic situation, the events in the stream only describe that they are observations (e.g., Observation(obs1)), that they observe a certain property (e.g., observedProperty(obs1,propX)) and that a specific value has been observed (e.g., hasValue(obs1,0.03)). Note that these observations need to be combined with background knowledge to figure out if the event was observing congestion levels. Since we are only interested in traffic observations that can be considered high-traffic, we select only the congestion level observations in the stream with a value above 0.03 or below 0.01, as indicated in the domain knowledge. However, to determine that an observation is, in fact, a congestion level, we need to integrate with static background data describing the sensors. We also extract the information regarding the office near the location where the observation comes from, so we can determine later if these are flexible offices or not. Listing 3.1 shows a query Q that selects the relevant portion of the stream.

In Figure 5.6, this query will select Observation(obs1) and Observation(obs6) from the stream. It will also add some additional data to the event, such as information regarding the road and the offices that can be used in the next layer for the expressive reasoning step.

Example 4. (cont'd) One of the selected events describes the first observation in the stream:

$$\begin{aligned}&Observation(obs1),\\&observedProperty(obs1, propX),\\&hasValue(obs1, 0.03).\end{aligned}$$

In the Selection Module, it has also been enriched with the following data:

$$\begin{aligned}&CongestionLevel(propX),\\&Office(office1),\\&MainRoad(road1),\\&isLocationOf(road1, office1),\\&StopEarly(pol1),\\&hasPolicy(office1, pol1).\end{aligned}$$

Listing 3.1: Example of the RSP-QL Query used in the Selection Module.

```
CONSTRUCT {
        ?obs_X a ssn:Observation.
        ?obs_X ssn:observedBy ?sensor_X.
        ?obs_X ssn:observedProperty ?property_X.
        ?property_X a CongestionLevel.
        ?obs_X hasValue ?value.
        ?property_X isPropertyOf ?foi.
        ?foi isLocationOf ?loc.
        ?loc hasPolicy ?pol. }
FROM NAMED WINDOW :traffic [RANGE 5m, SLIDE 1m] ON STREAM :Traffic
WHERE {
        ?property_X a CongestionLevel.
        ?property_X isPropertyOf ?foi.
        ?foi isLocationOf ?loc.
        ?loc hasPolicy ?pol.
        WINDOW ?w {
                ?obs_X a ssn:Observation.
                ?obs_X ssn:observedBy ?sensor_X.
                ?obs_X ssn:observedProperty ?property_X.
                ?obs_X hasValue ?value.
                FILTER(?value > 0.03 || ?value < 0.01)
        }
}
```

3.4.2.2 The Abstraction Module

The **Abstraction Module** implements the DL inference sub-layer. It receives the selected events from the Selection Module and abstracts them to high-level concepts. The Abstraction Module consists of a semantic publish/subscribe mechanism and allows the subscription to abstracted events, through high-level concepts. Each service in the service module can subscribe to events by defining event descriptions.

Technically, the Abstraction Module operates on an OWL reasoner, i.e., the HermiT [15] reasoner (note that, due to the modularity of the platform, other reasoners can easily be plugged in). Each time events have been selected in the Selection Module, they are added to the ontology in the Abstraction Module. Through the use of reasoning, we check which inferred types of the individuals are the types that one of the services subscribed to. When these types are found, the abstracted events are constructed using the found types, the underlying event, and the processing time. The abstracted event is then forwarded to those services that subscribed to the found types. Lastly, the events are removed from the ontology ABox. When new events have been selected by the underlying module, they are added to the ontology and the types of new events can be checked.

Example 5. (cont'd) The selected events from the Selection Module can now be abstracted according to the defined ontology in Example 2. Through reasoning we obtain that

$$
\begin{aligned}
&HighTrafficStreet(obs1),\\
&HighTrafficMainRoadNearFlexibleOffice(obs1),\\
&HighTrafficObservation(obs1),\\
&FlexibleOffice(office1).
\end{aligned}
$$

Let us assume that a service is interested in all HighTrafficObservations. The selected event, enriched with the inferred types, is forwarded to that service or to its Event Processing Module.

3.4.2.3 The Event Processing Module

The **Event Processing Module** implements the temporal reasoning sub-layer. When event processing is necessary, the Event Processing Module receives the abstracted events from the Abstraction Module. Each of the received abstracted events is checked if it matches an event pattern, through the use of the Esper CEP engine (http://www.espertech.com/esper/). We choose Esper since it supports the declarative language EPL. Note that, when multiple abstracted events are inserted at once, they are first ordered according to their timestamp. We allow one to define additional filter restrictions, such that the patterns can be matched on a fine-grained level.

In CEP, filter restrictions can be defined on the event values, e.g., Event A (speed = 45) has the property speed with a value of 45, and one can restrict events to have speed values above a certain threshold. Join restrictions can be defined over events, e.g., if each event type has a location A (location = loc1) and B (location = loc1), then we can impose the restriction that Events A and B should have the same location. We allow one to define additional queries to specify both restrictions.

Example 6. (cont'd) Let us assume that the pattern defined in the Event Processing Module is looking for all HighTrafficObservations followed by LowTrafficObservation within 10 min, which detects decreasing traffic. This can be defined through the pattern:

EVERY HighTrafficObservation SEQ LowTrafficObservation WITHIN $10\,m$.

We need to add additional restrictions to ensure that both the HighTrafficObservation and low TrafficObservation occurred in the same street. This can be done by filtering on the location. We know from Example 2 that each HighTrafficObservation should have a hasLocation relation. Therefore, we can enforce that they should be linked to the same location. In Section 3.4.3, we show how this can easily be defined.

When filter restrictions have been defined, these restrictions are checked first before adding the event to the CEP engine. When a join-restriction has been detected (e.g., the TrafficObservations should have the same location), the bindings of those variables are used within the CEP engine to perform the joins. When an event pattern matches, it is forwarded to the associated service.

3.4.2.4 The Remaining Modules

The MASSIF platform also consists of an **Input Module** that serves as the entry point of the platform and an **Annotation Module**, where raw data can be semantically annotated if necessary.

Finally, the **Service Module** receives the processed data and can perform additional analysis. Through the **Service Module**, information needs formulated using our DSL (see Section 3.4.3) can be issued to Streaming Massif. Therefore, services can subscribe to all underlying modules with one query.

Listing 3.2: Syntax of the Streaming MASSIF DSL.

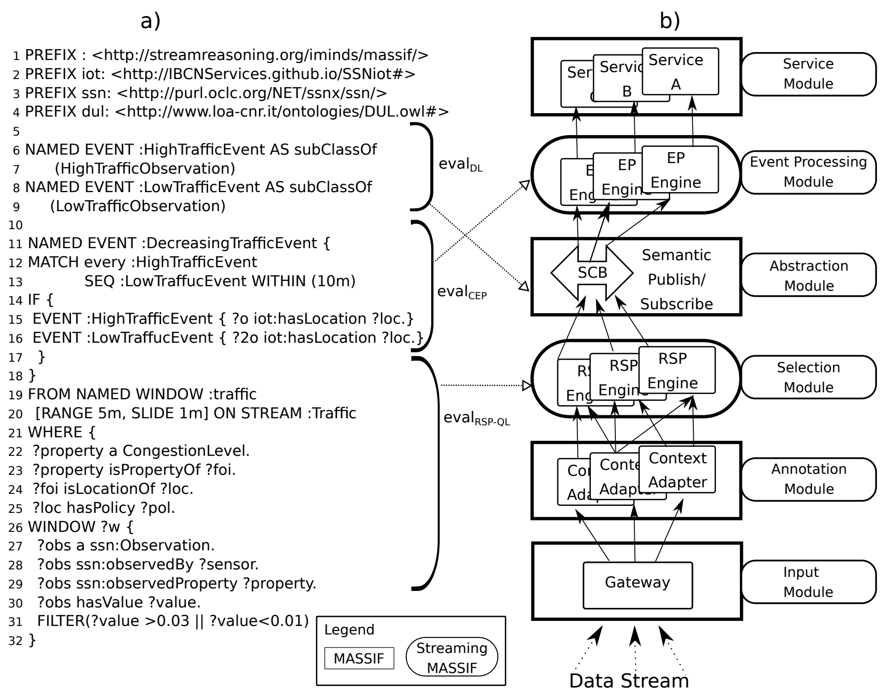
```
DSL —> NameSpace* EventDecl* RSPQL?
EventDecl —> 'NAMED EVENT' EventName (AbstractEvent |
     ComplexEvent)
AbstractEvent —> 'AS' DLDescription
ComplexEvent —> 'MATCH' (Modifier)? EventPattern (Guard)? (
     IFClause)?
EventPattern —> EventPattern EventOperator EventPattern|
     AbstractEvent|'NOT' EventPattern
IFClause —> 'IF' '{' ('EVENT' AbstractEvent '{' BGP '}')* '}'
EventOperator —> 'AND' | 'OR'| 'SEQ'
Modifier —> 'EVERY' | 'FIRST' | 'LAST'
Guard —> 'WITHIN' Num '(' TIMEUNIT ')'
TIMEUNIT —> 's' | 'm' | 'h' | 'd'
EventName —> String
Num —> [0—9]+
NameSpace —> SPARQL PREFIX SYNTAX
DLDescription —> MANCHESTER SYNTAX
BGP —> SPARQL BGP SYNTAX
RSPQL —> RSP—QL SYNTAX
```

3.4.3 A Domain Specific Language for Streaming MASSIF

In this section, we introduce a DSL that allows users to formulate information needs by using the proposed Cascading Reasoning approach. In order to explain the DSL, we provide an example of information need and we explain how each part of the query maps to the different module described in Section 5.6.

Listing 3.2 describes the grammar of the proposed query language. Note that for conciseness reasons, we did not incorporate the following sub-grammars:

1. DLDescription: The definition of the abstract event types is based on the Manchester syntax. For more information regarding this syntax, we refer the reader to the Manchester W3C page



```
1 PREFIX : <http://streamreasoning.org/iminds/massif/>
2 PREFIX iot: <http://IBCNServices.github.io/SSNiot#>
3 PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn/>
4 PREFIX dul: <http://www.loa-cnr.it/ontologies/DUL.owl#>
5
6 NAMED EVENT :HighTrafficEvent AS subClassOf
7       (HighTrafficObservation)
8 NAMED EVENT :LowTrafficEvent AS subClassOf
9     (LowTrafficObservation)
10
11 NAMED EVENT :DecreasingTrafficEvent {
12 MATCH every :HighTrafficEvent
13            SEQ :LowTraffucEvent WITHIN (10m)
14 IF {
15  EVENT :HighTrafficEvent { ?o iot:hasLocation ?loc.}
16  EVENT :LowTraffucEvent { ?2o iot:hasLocation ?loc.}
17   }
18 }
19 FROM NAMED WINDOW :traffic
20   [RANGE 5m, SLIDE 1m] ON STREAM :Traffic
21 WHERE {
22  ?property a CongestionLevel.
23  ?property isPropertyOf ?foi.
24  ?foi isLocationOf ?loc.
25  ?loc hasPolicy ?pol.
26 WINDOW ?w {
27   ?obs a ssn:Observation.
28   ?obs ssn:observedBy ?sensor.
29   ?obs ssn:observedProperty ?property.
30   ?obs hasValue ?value.
31   FILTER(?value >0.03 || ?value<0.01)
32 }
```

Figure 3.6: (a) Example of the Streaming MASSIF DSL and how it targets the Streaming MASSIF achitecture (b).

(https://www.w3.org/TR/owl2-manchester-syntax/).

2. BGP: In the definition of the complex events, one can define Basic Graph Pattern (BGP) for restricting the validity of the events. We did not incorporate the explanation of the syntax of BGP in this proposal.

3. RSPQL: For targeting the RSP module, we utilize RSP-QL. The full syntax of RSP-QL has not been incorporated in our syntax proposal, more information regarding RSP-QL can be found in Dell'Aglio et al. [31].

As defined in Listing 3.2, an information need comprises multiple namespaces (NameSpace), multiple event declarations (EventDecl) and an optional RSPQL declaration. Figure 3.6a shows an information need from the example use-case. We now explain how this DSL targets each module of the cascading stream reasoner.

3.4.3.1 DSL Fragment for the RSP Layer

From Line 19 in Figure 3.6a, the RSP-QL syntax is used for selecting the relevant events from various streams. Note that there is no query form defined, since we restrict the use to the construct query form. The construct query template is generated from the BGP in the WHERE clause. This part of the query targets the Selection Module of the Streaming MASSIF architecture. Note that the definition of the RSP-QL clause is optional in the language. In the absence of the RSP-QL clause, all streaming data is directly processed by the next layer (i.e., the abstraction layer). In this case, each event in the stream is processed one by one.

3.4.3.2 DSL Fragment for the DL Sub-Layer

An information need typically requires one to define multiple events. An event declaration (EventDecl) starts with the declaration of a NAMED EVENT, a name for the event (EventName), and either the definition of an abstract event (AbstractEvent) or a complex event (ComplexEvent). The abstract event definition start with the 'AS' keyword to indicate how the event name should be interpreted, followed by a declaration in Manchester DL syntax. This is shown in Figure 3.6a on Lines 6–9. We chose the Manchester Syntax[1] for the definition of these events since its very concise and expressive.

The defined abstracted event definitions are used in the Abstraction Module to indicate the high-level concepts that should be abstracted and forwarded to the next layers.

3.4.3.3 DSL Fragment for the CEP Sub-Layer

Besides the AbstractEvents, the EventDecl clause can also define complex events (ComplexEvents). These are declared with the 'MATCH' keyword, followed by a modifier (Modifier), an event pattern (EventPattern), a guard (Guard), and an optional restriction clause (IFClause). The EventPattern is constructed from various abstract events and event operators (EventOperators). These declarations are used within the Event Processing Module. Figure 3.6a shows an example event pattern defined over high and low traffic abstractions on Lines 11–13.

The restrictions (IFClause) are declared using the 'IF' keyword, followed by the abstract event name used in the pattern that needs to be restricted. The restriction itself is defined in a BGP. Both filter and join restrictions can be modeled in this manner. An example on how to define join restrictions over multiple events can be found in Figure 3.6a on Lines 14–16. The restriction states that the high and low traffic abstractions should occur in the same location. Note that the variable name 'loc' is the same in both restrictions.

Listing 3.3: DSL Event Restriction Clause Example.

```
NAMED EVENT :DecreasingTrafficEvent {
MATCH EVERY :HighTrafficEvent
SEQ :LowTrafficEvent WITHIN (10 m)
IF {
EVENT :HighTrafficEvent { ?o timeStamp ?time.
FILTER(hours(?time) > 15) }
EVENT :LowTrafficEvent { ?o2 timeStamp ?time2.
FILTER(hours(?time)>15) } }
}
```

[1] https://www.w3.org/TR/owl2-manchester-syntax/

We can also define restrictions to filter individual events. Listing 3.3 shows a filter restriction example over the high and low traffic abstractions that restricts observations to be after 3 o'clock in the afternoon.

Note that the SPARQL FILTER clause is optional. When the defined BGP does not match the underlying event, i.e., no results are returned, the event is filtered out and not considered in the complex event processing.

3.5 Streaming MASSIF's Formalization

Now that we have described the architecture we formalize how the different layers of Streaming MASSIF collaborate. We do this by focusing on the Cascading Reasoning pyramid that abstracts Streaming MASSIF, as shown in Figure 3.4b. The cascading approach consists of CEP and DL as inference methods and RSP for continuous information integration.

3.5.1 RDF Stream Processing Layer

The RSP layer receives RDF streams (as defined in Definition 1) as input and answers continuous queries written in RSP-QL (see Definition 3.3.1.2). A given RSP-QL query Q is evaluated against a RSP-QL dataset SDS (as defined in Definition 5). The result of the defined queries is forwarded to the next layer. Therefore, we fix the Query Form to the CONSTRUCT query form.

3.5.2 Continuous Information Integration Layer

As we previously mentioned, we assume that data streams arrive directly encoded as RDF streams. This assumption allows us to perform stream processing and continuous information integration in the RSP layer by means of a common vocabulary.

Notably, we do not consider the annotation task (a.k.a. the data materialization task) as part of the approach. If the data are not natively RDF streams, approaches such as TripleWave [38], which rely on mapping techniques such as RML [39] and R2RML[2], can be utilized. Note that the Annotation Module in the Streaming MASSIF architecture can be used for this goal.

3.5.3 The Inference Layer

The inference layer of our architecture consists of two sub-layers: (i) Description Logics, since we want to infer information not explicitly available in the streams, and (ii) Temporal Logics, because we aim at deducing information based on temporal relations between the data.

In the following, we explain how we link those sub-layers together.

First, we need to make a distinction between physical events and abstract events:

[2] https://www.w3.org/TR/r2rml/

Definition 7. A physical event e_{phy} is an event that occurs directly in the input stream S or is a result of the RSP layer. Note that, in the latter, the event may also include background data. A collections of physical events is defined as E_{phy}.

In Figure 5.6, multiple physical events are depicted in a stream. Four physical events are detailed. Example 4 describes the physical event that contains the first observation in the stream. Note that, in the RSP layer, the physical events can still be enriched with additional information.

Definition 8. A subscription TBox $\mathcal{E}$ consists of those TBox concepts that have been used as high-level concepts in a service subscription. It bridges the gap between domain ontology and the physical events. $\mathcal{E}$ contains all the NAMED EVENTS defined for the Abstraction Module.

The use of $\mathcal{E}$ allows us to select only those physical events that services are actually interested in. From these physical events we derive abstract events:

Definition 9. An abstract event e_{ab} consists of one or more physical events e_{phy} and hides their low-level details. An abstracted event e_{ab} can be inferred under an entailment Σ from a collection of physical events E_{phy} iff $\exists e_i \in E_{phy} : (\mathcal{T}^+, \mathcal{A}^+) \models C_{\mathcal{E}}(e_i)$ with $C_{\mathcal{E}} \in \mathcal{E}$ and $\mathcal{K} = (\mathcal{T}^+, \mathcal{A}^+)$, with $\mathcal{K}$ the knowledge based used in the reasoning process. $\mathcal{T}^+ = \mathcal{T} \cup \mathcal{E}$ is the TBox and $\mathcal{A}^+$ the ABox, with $\mathcal{A}^+ = \mathcal{A} \cup E_{phy}$. We can now define the abstracted event as the triple $e_{ab} = (C_{\mathcal{E}}, E'_{phy}, t)$, with E'_{phy} the collections of physical events in E_{phy} that lead to infer $C_{\mathcal{E}}(e_i)$ and t the processing time at which the first physical event in E'_{phy} was produced. E_{ab} represents a collection of abstracted events. This is the case when multiple abstracted events can be abstracted.

Example 7. (cont'd) In the DSL defined in Figure 3.6a, the services subscribed to HighTrafficObservations by defining the named event HighTrafficEvent. Let us assume that only HighTrafficEvent is contained in $\mathcal{E}$. The physical events can now be abstracted according to the defined ontology in Example 2. Through reasoning we obtain that

$$\begin{aligned}
&HighTrafficObservation(obs1),\\
&HighTrafficMainRoadNearFlexibleOffice(obs1),\\
&FlexibleOffice(office1),\\
&HighTrafficEvent(obs1).
\end{aligned}$$

This results only in the abstracted event $(HighTrafficEvent, e_{phy}, t_i)$ with e_{phy}, the physical event, and t_i, the time e_{phy} is produced, since only HighTrafficEvent is defined in $\mathcal{E}$.

We now want to identify temporal dependencies between the abstracted events provided by the DL sub-layer.

We build upon the definitions from CEP to detect the temporal dependencies between abstracted events provided by the DL sub-layer.

Definition 10. An event pattern EP is a statement of the form

$$[\triangledown](E_1 \wedge \cdots \wedge E_k)|(E_1 \vee \cdots \vee E_k)[\triangle]$$

with E_i either (i) an event type, (ii) a complex event using AND, OR, NOT, or SEQ, or (iii) another event pattern (recursively). $\triangledown$ is an optional modifier, e.g., EVERY, and $\triangle$ is an optional guard, e.g., WITHIN.

We use these patterns to instantiate complex events that represent inferred information.

Definition 11. A complex event ce definition is a triple $ce = (h, p, R)$ with

- h as the complex event type,
- p as the pattern defined using operators, modifiers, and guards, and
- R as a set of restrictions.

h is instantiated when p and R are satisfied.

The set of abstracted events (i.e., the collection of triples $(C_{\mathcal{E}}, e_{phy}, t)$) is used in the event pattern matching. More specifically, each type $C_{\mathcal{E}}$ is checked if it matches the event types within the pattern. Additionally, the restrictions $R = (CR_{\mathcal{E}}, q_{SPARQL})$ can be defined on each event type in an event pattern. $CR_{\mathcal{E}}$ is an event type (i.e., defined in $\mathcal{E}$) and q_{SPARQL} is a SPARQL query. The SPARQL query is evaluated over each e_{phy} contained in the abstracted event ($e_{ab} = (C_{\mathcal{E}}, e_{phy}, t)$), where $C_{\mathcal{E}} == CR_{\mathcal{E}}$. Restrictions over multiple events in the event pattern can be achieved by creating multiple restrictions R with the same variable names in the q_{SPARQL}. The variable bindings are extracted and used for joining the events. This is shown in the restrictions of Example 8 through the use of the reoccurring variable name "?loc".

Example 8. (cont'd) To detect the decreasing traffic, we need to monitor for a high amount of traffic near flexible offices followed by low amounts of traffic near the same flexible offices within a certain time range. This can be done by defining the complex event definition triple: $ce = (CE_{\mathcal{E}}, p, R)$ with

- $CE_{\mathcal{E}}$ as the complex event type $DecreasingTraffic$,
- p as the pattern describing EVERY $HighTraffic\ Abstraction\ SEQ$ $LowTrafficAbstraction\ WITHIN\ 10m$, and
- R as a set of restrictions of the form $(CR_{\mathcal{E}}, q_{SPARQL})$ consisting of
  * $(HighTrafficAbstraction, q_1)$ with $q_1 =$

```
Select * WHERE {
?o ssniot:hasLocation ?loc.}
```

  * $(LowTrafficAbstraction, q_2)$ with $q_2 =$

```
Select * WHERE {
?o2 ssniot:hasLocation ?loc.}
```

Note that the restrictions state that high and low traffic events need to have the same location. The value in $?loc$ will be used to restricts the complex events, since its the only variable with the same name in q_1 and q_2.

3.5.4 Unified Evaluation Functions

In the following, we explain, by means of Figure 5.6, how to combine the different layers into a single evaluation framework. At the lowest level, we have the evaluation of the RSP layer. Let us consider an RSP-QL query Q. The evaluation of Q over dataset SDS is defined as

$$\Omega(t) = eval(SDS, SE, t), \text{with } t \in ET$$

where ET represents all the time instances where SDS is defined, and Ω is a time-varying multiset of solution mappings that maps time T to the set of solution mappings multisets [31]:

$$\Omega : T \to \{\omega | \omega \text{ is a multiset of solution mappings}\}.$$

We consider only the CONSTRUCTS query form; therefore, the solution mappings still need to be substituted in a graph template defined in the query (as defined in the SPARQL 1.1. specification: https://www.w3.org/TR/sparql11-query/#construct):

$$G_{\Omega}(t) = \sigma(G_{template}, \Omega(t)).$$

with σ the substitution function and $G_{template}$ the graph template defined in Q. The solution $G_{\Omega}(t)$, for each $t \in ET$, is a subset of the data in SDS and is sent to the next layer in the cascading reasoner for further processing. We can define the evaluation of the RSP layer as

$$eval_{RSP-QL}(SDS, Q) = G_{\Omega}(t), \forall t \in ET.$$

Each time the RSP layer produces results, they are sent to the DL layer as a set of physical events E$_{phy} = G_{\Omega}(t)$. The DL layer converts the physical events E_{phy} to a set of abstracted events E_{ab} under a certain entailment Σ.

$E_{ab} = \{C_{\mathcal{E}}(e_i) | \exists e_i \in E_{phy} : (\mathcal{T}^+, \mathcal{A}^+) \models C_{\mathcal{E}}(e_i) \wedge C_{\mathcal{E}} \in \mathcal{E}$ with $\mathcal{T}^+ = \mathcal{T} \cup \mathcal{E}$ and $\mathcal{A}^+ = \mathcal{A} \cup E_{phy}\}$. The $eval_{DL}$ reasoning step is defined as

$$eval_{DL}(E_{phy}, \mathcal{E}, \mathcal{O}, \Sigma) = E_{ab}$$

where E_{ab} is the set of abstracted events and the quadruple $< E_{phy}, \mathcal{E}, \mathcal{O}, \Sigma >$ comprises the following:

- E_{phy}—a set of one or more selected physical events contained in $G_{\Omega}(t)$.
- $\mathcal{O}$—the ontology describing the domain knowledge. $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ with $\mathcal{T}$ the TBox and $\mathcal{A}$ the ABox describing $\mathcal{O}$.
- $\mathcal{E}$—an ontology TBox that bridges the domain ontology $\mathcal{O}$ and the physical events E_{phy}. This describes formally the abstraction based on $\mathcal{O}$. Only the concepts in $\mathcal{E}$ will be considered as abstracted events.
- Σ—the entailment regime under which the reasoner has to extract the abstract events from E_{phy}.

Finally, we define the result of the evaluation of the CEP layer as a set of abstract events:

$$eval_{CEP+}(CE, E_{ab}) = \{(CE_{\mathcal{E}}, \bigcup e_{phy}, t)\}$$

with $CE_{\mathcal{E}}$ the complex event type of the complex event $ce \in CE$ that matched, e_{phy} the physical events in E_{ab} that cause the patterns to trigger and t the processing time at which the patterned triggered. In the resulting complex event, the union of the underlying physical events is taken and the complex event type is assigned.

Since complex events are still physically represented as RDF graphs, in order to evaluate restrictions we can simply extend $eval_{CEP}$ with $eval_{SPARQL}$ that evaluates the restrictions describes as SPARQL queries.

To ensure termination, we restrict to non-recursive pattern definitions, i.e., $\forall p \in CE, \nexists E \in p : CE_{\mathcal{E}} == E$. The complex event type is thus not allowed in the definition of the pattern.

3.5.5 Summary

To conclude, we described a stream reasoning stack that is able to (a) select the relevant portions of the stream using RSP, (b) abstract the selected RDF graphs using expressive reasoning techniques and selecting only those that match the expected abstractions, and (c) apply complex event processing over these abstractions to detect temporal dependencies.

3.6 Evaluation

To evaluate Streaming MASSIF, we extended the City Bench benchmark [4] with expressive ontology concepts, as those described in Example 2. We also extended the ABox and added various offices located near the monitored streets, each with a set of random policies. Among these office policies is the possibility to start early, to stop early, and to have flexible work hours and the presence of childcare. To further increase the complexity, we also added some complex roles which are used within the high and low traffic modeling, e.g.,

$$observedFeature \sqsubseteq observedProperty \circ isPropertyOf.$$

For streaming the City Bench data, we utilized RSP Lab (https://github.com/streamreasoning/rsplab) and ran the streamers on a different node. The evaluation was conducted on a 16 core Intel Xeon E5520 @ 2.27 GHz CPU with 12 GB of RAM running on Ubuntu 16.04.

We first show the need for Cascading Reasoning when dealing with high-volatile streams.

3.6.1 The Need for Cascading Reasoning

To illustrate the need for Cascading Reasoning, we first show that current approaches have problems performing expressive reasoning over high-volatile streams. For now, we do not consider the temporal aspect. Reasoning techniques exist with different trade-offs between expressivity

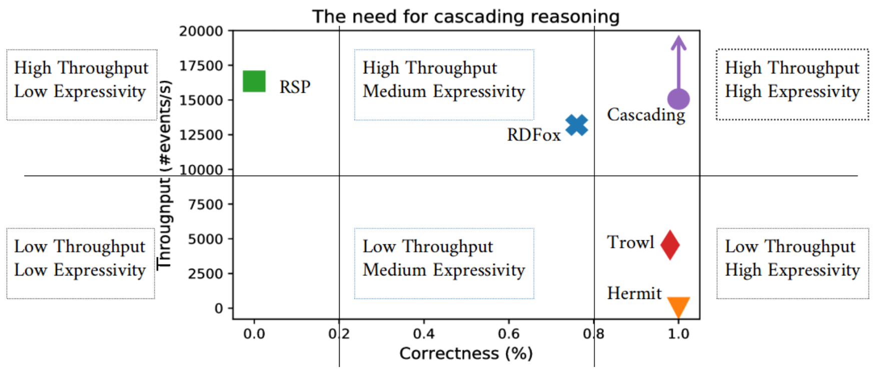


Figure 3.7: The comparison in terms of throughput and correctness for various approaches. A monolithic approach needs to trade off correctness for performance, whereas combined Cascading Reasoning approaches can cover both. This allows one to achieve high throughput and high expressivity.

of reasoning and complexity of processing. Very low expressive reasoners are more performant as their complexity of processing is lower. We compare various reasoners within the spectrum of expressivity.

3.6.1.1 Setup

To show the need for Cascading Reasoning, we provide batches of events, ranging between different number of events, to various reasoning techniques and measured the time it took each engine to process a specific number of events. The events themselves were captured from a City Bench event stream and the extended City Bench ontology was utilized to perform the reasoning. As the expressivity of each reasoning approach differs, we calculated the correctness of each engine. The correctness is measured as the percentage of concepts in the ontology that can be correctly calculated considering the expressivity of the reasoner.

The throughput is calculated by serving batches of 1, 10, 100, 1000, and 10,000 traffic observation events and calculating how long each approach takes, on average, to process the events. The batches are considered, as data is typically windowed when considering streaming data.

3.6.1.2 Results

Figure 3.7 shows a comparison of various reasoners in terms of throughput and correctness, while Table 3.1 provides the processing time for each reasoner in function of the number of events processed in each batch. HermiT was not able to handle batch sizes larger than 10,000 events due to out-of-memory exception. Therefore, the averages in Figure 3.7 are taken over batch sizes between 1 and 10,000 events.

Table 3.1: Evaluation of the processing time (in ms) of the different reasoning techniques. By combining very expressive reasoning with very efficient processing, we can achieve high expressivity and high throughput.

Engine/#Events	1	10	100	1000	10,000	25,000	50,000	80,000
RSP	15	15.1	23.1	127.3	398.3	973.5	2011.4	3291.3
RDFox	21.2	21.6	27.7	130.7	500.15	1230.7	2453.3	4405.5
TrOWL	440.3	455.9	415.7	702.8	1292.45	3205.6	7083.3	14,153.0
Hermit	12,895.0	12,972.0	13,440.0	27,885.0	170,532.5			
Cascading	74.2	76.9	74.4	147.5	443.3	1040.0	2303.9	3754.4

RSP engines typically have very low to no reasoning capabilities, as they specifically aim at processing high volatile streams. As depicted in Figure 3.7, they have a high throughput, but very low expressivity as their correctness is very low. RDFox [16] is the fastest reasoner currently available. It supports OWL2 RL reasoning, a subset of OWL2 DL reasoning. It does not consider various ontology construction in order to achieve high performance. As can be seen in Figure 3.7, its throughput is rather high but it is not completely correct, as it lacks the expressivity to reason about all concepts correctly. Hermit [15] is a fully fledged OWL2 DL reasoner consisting of the needed expressivity to reason correctly about OWL2 DL ontologies. However, due to this expressivity, it is rather slow. TrOWL [40] is an OWL2 DL reasoner that allows one to perform approximation to enable stream reasoning over ontologies. Its throughput is higher than HermiT but lower than RDFox. However, its expressivity is higher than RDFox's but lower than HermiT's, as it does not support all OWL2 DL concepts. We show that, by combining the highest throughput approach, i.e., RSP, with the highest expressivity approach, i.e., HermiT, we can achieve both a high throughput and high expressivity approach. This is depicted as Cascading in Figure 3.7. The processing time is not simply the addition of the two layers. The speedup is achieved because of two reasons. The first reason is that the RSP layer can select only relevant parts of the streams to be processed with the expressive reasoner, resulting in fewer events being processed in the second layer. The second reason is that a lower amount of background data is necessary in the second layer, as the tasks of integrating the sensor data with the background data can now be performed in the RSP layer. From there, the relevant information for further processing can be selected and used in the second layer. The arrow indicates that possible higher throughputs can be achieved by duplicating and distributing the various parts of the Cascading Reasoning approach. For example, multiple streams can first be processed with its dedicated RSP engine before the results are combined and processed with a higher expressivity approach. This scalability is not possible with the other approaches, as they are monolithic systems. We also note that 78% of the IoT-labeled ontologies in the Linked Open Vocabularies repository (lov.linkeddata.es) (we only considered the ontologies

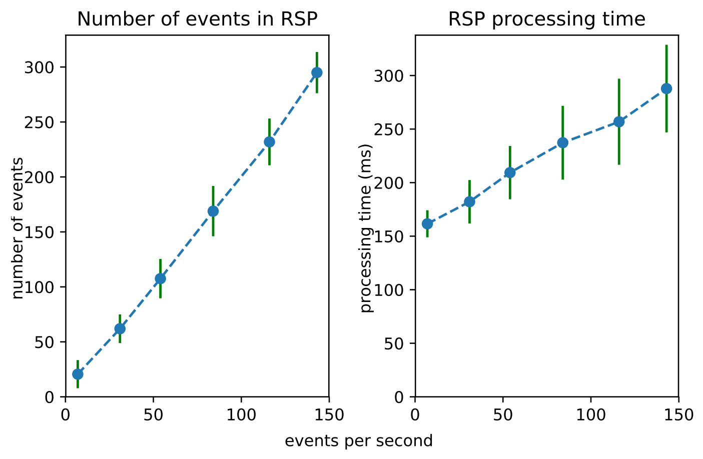


Figure 3.8: The influence of an increasing event rate on the number of the to-be-processed events (left) and the performance (right) of the Selection Module.

that were accessible at the time of writing) require the OWL2 DL expressivity to infer all concepts correctly.

3.6.2 Test 1: Increasing Event Rate

To test the scalability of the Streaming MASSIF itself, we first artificially sped up the traffic streams to see how many events the platform can handle. Each stream in City Bench produces data every 5 min. We sped up the stream to produce multiple events per second. Figures 3.8–3.10 visualize for each component the number of processed events and the processing time for a specific event rate. The RSP processing time in Figure 3.8 denotes the time taken to select the events within the window, the Abstraction time in Figure 3.9 denotes the time taken to abstract the received events from the RSP layer to high-level concepts, and the CEP processing time in Figure 3.10 measures how long it takes for the pattern to match, when the last event that causes the pattern to match arrives. On the x-axis, for all of the figures, we plotted the (rounded) actual event rate as they entered the platform. Note that, each time the stream produces data, five observations are produced: the average speed, the vehicle count, the measured time, the estimated time, and the congestion level. However, it is not stated explicitly in the stream what kind of observation is transmitted. Integrating with background knowledge is thus required to filter out the congestion level observations. This is performed in the RSP layer. We evaluated our results over eight streams and calculated the averages over the first 120,000 events. To easily calculate the processing time in each layer, we used a tumbling window (the sliding parameters is the same as the window width) of 2 s for each event rate. Using a tumbling window, each event only occurs once, and this simplifies the processing time calculations. To perform the evaluation, we used the example query from Figure 3.6a.

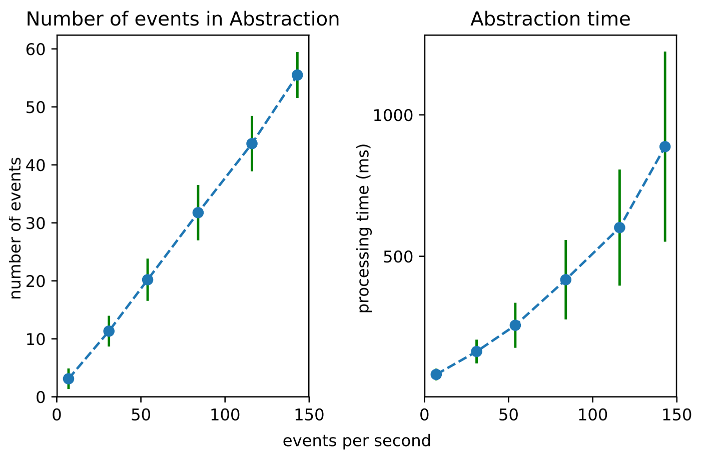


Figure 3.9: The influence of increasing event rate on the number of the to-be-processed events (left) and the performance (right) of the Abstraction Module.

From Figure 3.8, we can see that the greatest selection of events happens in the RSP layer, while fewer events are selected in the abstraction layer. This is clear, since the number of events in the abstraction decreases when forwarded to the Event Processing Module, as depicted in Figure 3.10. Furthermore, the processing time in the Abstraction layer rises more quickly than it does in the other layers, which can be expected of an expressive reasoning process. However, we see that, when abstracting even more than 50 events, the abstraction time is lower than 1 s. The total latency of the abstraction remains well below 2 s (the size of the window), and the system thus stays reactive even when processing 300 events per second.

Figure 3.11 shows the influence of the event rate on the different layers combined. It is clear that the abstraction is most influenced by the event rate. This is because more events need to be abstracted. As the time for the event processing is very low, it is hardly visible in the graph.

3.6.3 Test 2: Increasing Window Size

The performance of each layer is clearly dependent on the number of considered events. We investigated the processing time of each layer when the window size in the RSP layer increases. This forces the processing of an increasing number of events in each layer. Figures 3.12–3.14 visualize the number of processed events and the processing time for each layer when the window size increases from 1 to 100 s. We see a clear increase in the processing time of each layer. The Abstraction time increases exponentially, which can be expected of an expressive reasoning process. However, abstracting up to 100 events takes about 15 s, still much faster than the 100 s it takes for the window to slide.

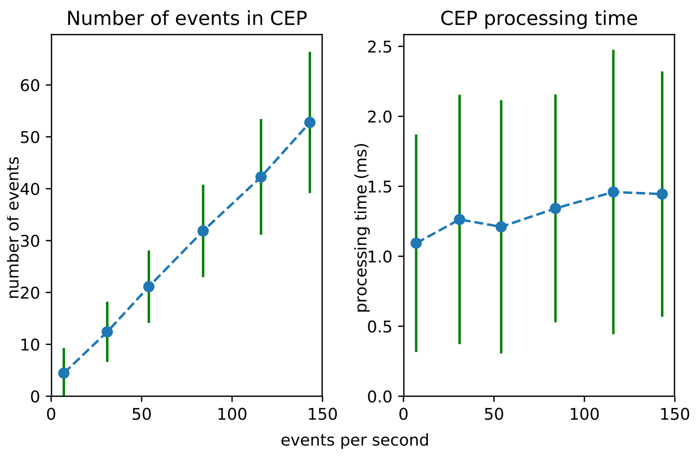


Figure 3.10: The influence of increasing event rate on the number of the to-be-processed events (left) and the performance (right) of the Event Processing Module.

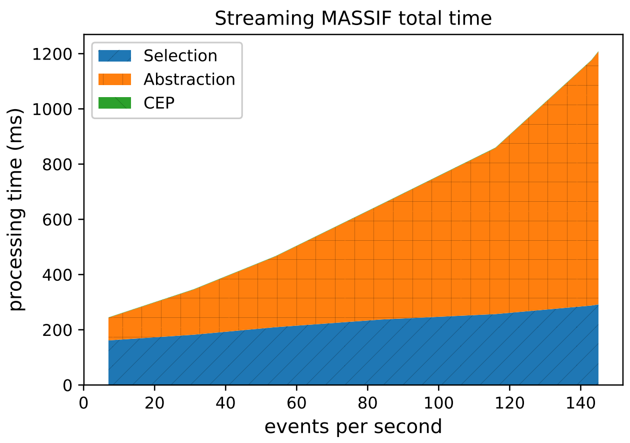


Figure 3.11: The influence of increasing event rate on total processing time of Streaming MASSIF and its components.

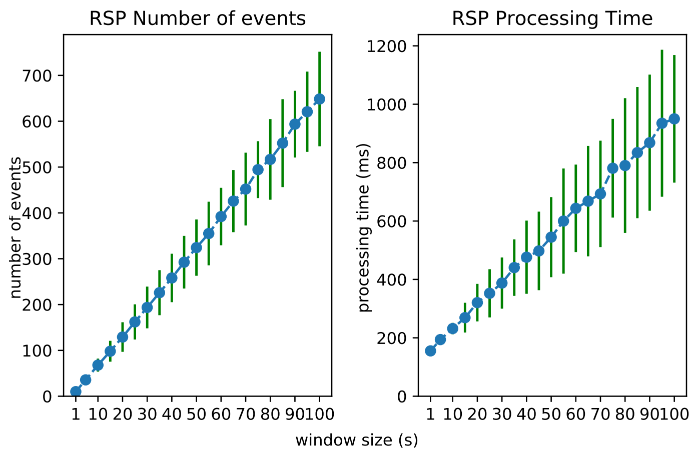


Figure 3.12: The influence of increasing the window size on the number of the to-be-processed events (left) and the performance (right) of the Selection Module.

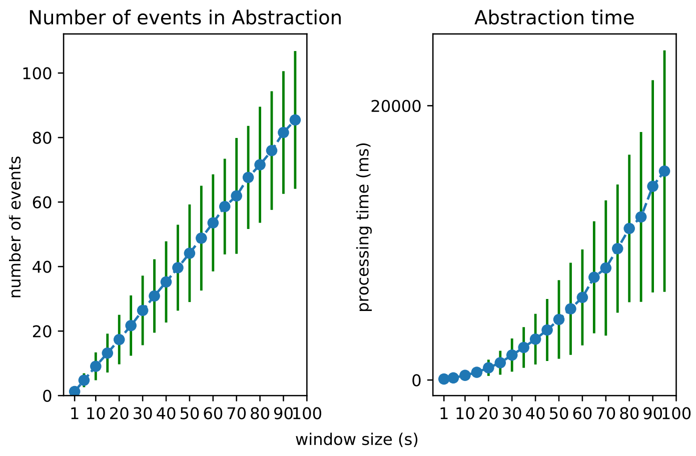


Figure 3.13: The influence of increasing the window size on the number of the to-be-processed events (left) and the performance (right) of the Abstraction Module.

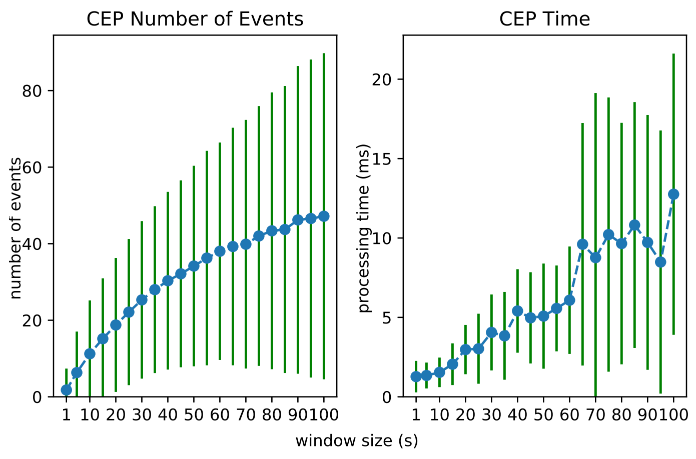


Figure 3.14: The influence of increasing the window size on the number of the to-be-processed events (left) and the performance (right) of the Event Processing Module.

3.6.4 Test 3: The Influence of the Selection Rate

Besides the size of the window, the percentage of events that are selected in the Selection Module influences the abstraction time in the Abstraction Module. This is because the more events that are selected, the more events that need to be abstracted through expressive reasoning, and the expressive reasoning is expensive. Figure 3.15 shows the influence of a decreasing selection rate on the abstraction time. When the selection rate decrease, the number of events that need to be abstracted decreases and has thus less influence on the reasoning time. It is thus important that the Selection Module carefully selects only the relevant events.

3.6.5 Test 4: Comparison with MASSIF

Since we extended the MASSIF platform to implement the adapted Cascading Reasoning vision, we also measure how fast the MASSIF platform could process the event stream. Note, however, that the MASSIF platform needs to perform the abstraction on all the background data, consisting of all the information of all the sensors, the streets, the offices, etc. All of the background data contain more than 60,000 statements. In the RSP layer, we select the relevant portion from the stream but also select the relevant data from the background knowledge. This eliminates the need for the Abstraction layer to contain the whole background knowledge. The TBox is most important there. Without this selection step, the abstraction of a single event in the MASSIF platform takes up to 20 s. Figure 3.16 shows the comparison for different event rates. The figure shows that the layered approach is much more scalable. The first reason for the speedup is because, compared to MASSIF, Streaming MASSIF can process events in windows, while MASSIF processes each event one by one. The second reason is that events are filtered in Streaming MASSIF before they are exposed to the

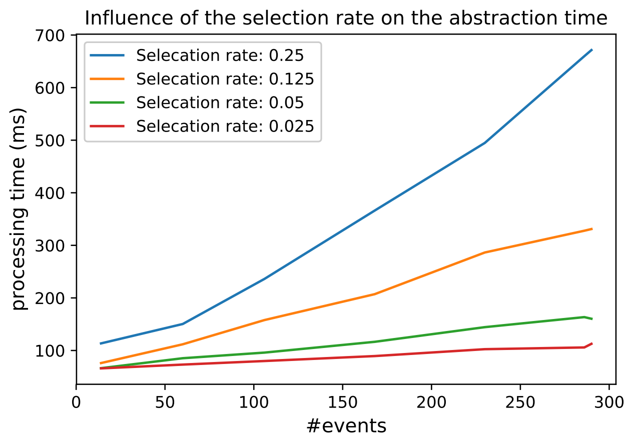


Figure 3.15: The influence of the selection rate on the abstraction time.

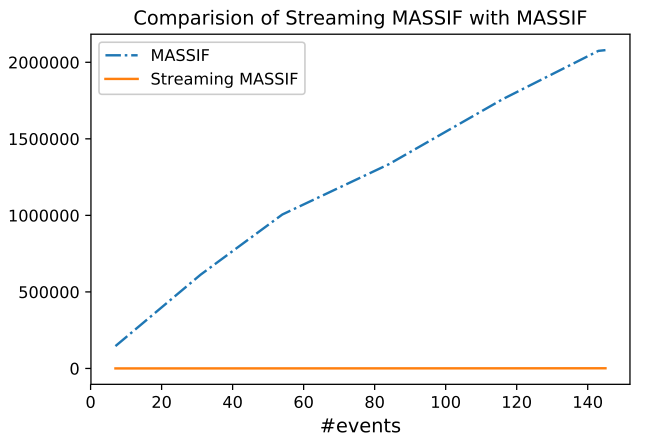


Figure 3.16: Throughput of Streaming MASSIF and MASSIF.

expressive reasoning, while each event in MASSIF goes through the expressive reasoning step. The last reason is that the whole background needs to be used for the reasoning step in MASSIF, while in Streaming MASSIF the background is spread between the Selection and Abstraction Module.

3.7 Discussion

The aim of this research was to design a layered Cascading Reasoning realization that can perform both expressive and temporal reasoning over volatile data streams. The evaluation sections shows that Streaming MASSIF is able to perform high expressive OWL DL reasoning with high throughput. Note that other reasoning approaches exist, such as ASP [41], but we opted for DL since it is a web standard and widely adopted.

3.7.1 Objectives Discussion

Looking back at the Objectives set in Section 3.1, we can now discuss how Streaming MASSIF tackles the various objectives:

1. Combine various data streams: Streaming MASSIF can combine various heterogeneous data streams by utilizing a common semantic model, i.e., an ontology, that can be understood throughout the platform. The Selection Module allows one to process multiple streams, combining them together, while keeping a common semantics. From these various streams, the Selection Module selects those parts that are relevant for further processing.

2. Integrate background knowledge: Streaming MASSIF allows the integration with background knowledge both in the Selection Module and in the Abstraction Module. The integration in the Selection Module allows one to combine the data streams with more static data, in order to retrieve more information about the observations in the streams, which typically do not describe the full context they observe. The integration in the Abstraction Module allows one to take more context into account to perform the expressive reasoning. The tight coupling between these two modules also allows parts of the static background to be selected in the Selection Module, such that it can be used in the Abstraction Module.

3. Integrate complex domain knowledge: The integration of complex domain knowledge is achieved by allowing expressive reasoning in the Abstraction Module in order to correctly interpret the domain. The domain knowledge itself is modeled in the ontology.

4. Detect temporal dependencies: Streaming MASSIF can detect temporal dependencies between abstracted events. This is achieved by first abstracting selected observations in the data streams and performing CEP over these abstractions. This allows one to efficiently introduce a temporal aspect in ontology reasoning and integrate complex domain knowledge in CEP. This is more efficient as the direct integration of the temporal domain in DL, i.e., temporal DLs, as they easily become undecidable [18] and CEP is unable to model complex domains [17].

5. Easy subscription: Streaming MASSIF allows services to easily subscribe to the data streams, enabling filtering, abstraction, and temporal reasoning, through the use of its unifying query language. This allows services to define their information need in a declarative way, without the need for writing code.

Table 3.2: Related work based on the set objectives. (1: not for streams, 2: complex definitions, 3: only programmatically, 4: no unifying subscription language, 5: only incremental changes, 6: using approximations)

	Data Streams	**Background knowledge**	**Complex Domains**	**Temporal Dependencies**	**Unifying QL**	**Service Subscription**
EP-SPARQL [13]	X	/	RDFS	Allen Algebra	X[2]	/
StreamRule [19]	X	/[1]	ASP	/	/	/
Ali et al. [21]	X	/[1]	/	ASP	/	/
CityPulse [22]	X	/[1]	ASP	CEP[3]	/	X[4]
HermiT [15]	/	X	OWL2 DL	/	/	/
RDFox [16]	/[5]	X	OWL2 RL	/	/	/
TrOWL [40]	/[5]	X	OWL2 DL[6]	/	/	/
MASSIF [27]	/	X	OWL2 DL	/	/	X
Streaming MASSIF	**X**	**X**	**OWL2 DL**	**CEP**	**X**	**X**

3.7.2 Related Work Comparison

Table 3.2 compares the related work and the engines used in the evaluation based on the objectives. We also added a column Service Subscription, as most engines typically focus on data processing and do not provide mechanisms for service subscription. We note that, even though EP-SPARQL has a query language, the definition of the temporal patterns is complex compared to the pattern definition over abstracted events, as provided by Streaming MASSIF. Furthermore, the reasoning expressivity of EP-SPARQL is low, i.e., RDFS. StreamRule, Ali et al. and CityPulse do not allow the integration of background knowledge when handling the data streams, since they utilize the CQELS RSP engine, which does not allow the integration of static data. Furthermore, their expressive reasoning is done through ASP, while we opted for DL, since it is a web standard and widely adopted. CityPulse also allows the definition of temporal dependencies through CEP; however, the patterns need to be defined programmatically, which further complicate the definition of the information need. Streaming MASSIF integrates CEP and DL reasoning both syntactically, through the use of its unifying query language, and semantically.

Compared to HermiT, RDFox, and TrOWL, Streaming MASSIF is a cascading approach, able to combine the streaming domain with complex and temporal domains, while HermiT, RDFox, and TrOWL focus on performing expressive reasoning on static or slow-moving data. It is clear that Streaming MASSIF targets all objectives.

Table 3.3 provides an overview of the systems discussed in the related work and how they fit the generalized Cascading Reasoning vision. We see that StreamRule, Ali et al., and CityPulse utilize the CQELS RSP engine for Stream Processing. As seen in Table 3.2, this is the reason they fail to integrate background knowledge in the stream processing, as CQELS is not able to integrate static data. Most of the approaches use annotations to convert data to the common semantic model, while EP-SPARQL is able to rewrite but only from a prolog statement. The inference entailments differ for each platform. Streaming MASSIF is able to achieve the highest expressivity by com-

Table 3.3: Overview of the related work and how they relate to our generalized Cascading Reasoning vision. (1: only programmatically, 2: via CEP rules)

Name	Stream Processing	Continuous Information Integration	Inference Entailment	Unifying QL
EP-SPARQL	Etalis	Rewriting	RDFS & Allen Algebra	✓
StreamRule	CQELS	Annotation	ASP	None
Ali et al.	CQELS	Annotation	Action-Rules in ASP	None
CityPulse	CQELS	Annotation	ASP & CEP^1	None
Streaming MASSIF	Yasper	Annotation	OWL2 DL & CEP^2	✓

bining OWL2 DL with CEP. We note that the Allen Algebra utilized in EP-SPARQL for the temporal detections, is typically broader than the temporal patterns allowed in CEP. However, the ontology reasoning supported in EP-SPARQL is low (RDFS) and the definition of the temporal patterns is rather complex compared to Streaming MASSIF.

3.7.3 Evaluation Discussion

In Section 3.6.1, we evaluated the throughput of different reasoning systems and compared them to a cascading approach. The evaluation shows that, by combining different approaches, both high throughput and high expressivity can be achieved, what is not possible with a monolithic approach. The achieved throughput and expressivity is depending on the components used in the layered approach. Different throughputs can be achieved by combining different engine complexities. For example, if we would combine RSP with RDFox (instead of HermiT), the throughput would be even higher; however, the expressivity would be lower as the expressivity of RDFox is lower as the on of HermiT. Furthermore, higher throughputs can be achieved by duplicating certain components in the layered approach. For example, by distributing multiple RSP engines that each handle different streams and select the relevant parts from their streams, higher throughputs can be achieved. Furthermore, by more intelligently selecting the relevant parts and decreasing the selection rate, the throughput can also be increased, as fewer data need to be processed by the more complex layers.

In the evaluation, we can see that the Abstraction Module can easily become the bottleneck with a high number of events, so incremental reasoning techniques should be further researched. Currently, there are no efficient expressive incremental reasoning techniques that also incorporate data property reasoning. We could easily perform the abstraction in parallel and load balance eventsto increase the performance. This is possible in the cases that the events are independent of each other. When multiple physical events should be abstracted together, the query in the lower RSP layer could be adapted to link them together. This would allow one to scale the abstraction module even more since the abstraction of a low number of events is still rather quick, i.e., less than half a second for 30 events. We also note that the higher the selection rate, i.e., the fewer events are selected in the Selection Module, the higher the throughput of the complete system, as the abstraction time is still the most time-consuming. It is thus important that only the relevant events can be selected and forwarded.

3.7.4 Streaming MASSIF Limitations & Future Work Directions

One of the limitations of Streaming MASSIF is the fact that the user still manually needs to define a query over all the layers. The query mediation and query rewriting process [42] are currently not researched yet. By enabling the query mediation and rewriting, the query could be defined on a high-level and the query necessary for the selection of the relevant events in the stream could automatically be constructed. This would result in an even easier subscription language.

Another limitation, as discussed above is that the Abstraction Module can become the bottleneck when the selection rate is high, i.e., many events still need to be abstracted. As the throughput of the abstraction layer is typically lower than the selection layer, the more data there are that still need to be abstracted, the lower the total throughput will be.

The fact that each layer currently consists of a single engine can be limited as well. As we discussed above, distributing various parts of each layer can further increase the throughput.

In future work, we wish to investigate query mediation and rewriting techniques, such that the query definition can be defined on high-level event definitions and that the parts for the selection over the data streams can be automatically defined. This would further simplify event definitions and service subscriptions. To target the abstraction bottleneck, incremental reasoning techniques or efficient caching techniques need to be further investigated. This would further improve the performance of the expressive reasoning layer and thus improve the total performance of the cascading platform. This is especially necessary when the selection layer is unable to select only a small portion of the stream, and many events need to be abstracted. To further increase the throughput, distribution techniques should be investigated in order to distribute and duplicate various components and layers.

3.7.5 Applicability for Real-World Use-Cases

Table 3.4 describes some of the sensors and their frequencies in two Smart Cities, i.e., the City of Things in Antwerp (www.imec-int.com/en/cityofthings) and the Aarhus City Lab. (www.smartaarhus.eu/). We see that the frequency of most sensors is typically low and thus Streaming MASSIF can easily process these data streams. Compared to Aarhus, the city of Antwerp transmits all the sensor changes without aggregating them. This means that each Traffic Count sensor transmits the observation of a passing vehicle, while in Aarhus the exact number of measured vehicles since the last transmission is provided. The busier the road, the higher the transmission frequency of the sensor. This shows the strength of Streaming MASSIF and the use of a declarative language. The Selection Module utilizes an RSP engine which can easily perform aggregations. The change between the two different types of traffic count sensors would thus only result in the addition of a Count statement in the query language. Even if the number of Traffic Count sensors would be very high and they transmit data very frequently, since the data first need to be aggregated, this would result in a very low selection rate, filtering only a very select fragment of aggregated events. The same goes for the Traffic Lights in the smart city of Antwerp. There are eight sensors per intersection transmitting five observations each second. These data typically first need to be aggregated, which results in a very low selection rate; thus, a very small number of events even-

Table 3.4: Overview of the produced IoT sensor data in two real-life Smart Cities.

	Frequency	Single Sensor Events/s	#Sensors	Total Events/s
Antwerp				
Air Quality	1 per 30s	0.034	22	0.667
Temperature	100 per day	0.001	2	0.002
Rain	250 per day	0.003	4	0.012
Traffic Count	$\pm$800 per day	$\pm$0.01	$\pm$115	$\pm$1.15
Traffic Lights	5 per second	5	8	40
Aarhus				
Air Quality	1 per 5 min	0.0034	449	1.5
Weather	1 per 5 min	0.0034	9	0.03
Parking	1 per 5 min	0.0034	449	1.5
Traffic Count	3 per hour	0.0008	1	0.008

tually needs to be abstracted. Data from other Smart Cities tell the same story, e.g., in the city of Padova in Italy, data are transmitted once every 10 min [43] by each device. These findings allow us to conclude that Streaming MASSIF can handle cases of real-life smart-city use.

3.8 Conclusions and Future Work

In this paper, we presented Streaming MASSIF, a Cascading Reasoning approach that allows one to perform expressive and temporal reasoning over volatile data streams. Special attention was given to ensure that the platform could combine various data streams, integrate background knowledge, integrate complex domain knowledge, detect temporal dependencies and allow for the easy subscription of services. In order to tackle these objectives, we propose a cascading reasoning approach, consisting of various layers, each specialized in specific tasks. We defined semantically how these layers collaborate.

Streaming MASSIF is thus a Cascading Reasoning realization consisting of stream processing, continuous information integration, and inference layers. The layers are instantiated by combining RSP, DL reasoning, and CEP to enable expressive reasoning and event processing over high-velocity streams. We described a query languages that combines these various layers, allowing easy querying of the whole reasoning stack without the need to write any code. Our approach can perform expressive reasoning and event processing over high-velocity streams by selecting only the relevant events from the stream.

We have shown that Streaming MASSIF is able to combine high expressive reasoning with a high throughput of processing by combining techniques with different complexities in a layered approach. Furthermore, we have defined on a semantic level how the layers in our cascade cooperate, allowing one to assess the correctness of the approach.

However, when the RSP layer is not able to make this selection from the stream and huge numbers of events need to be abstracted, the platform might become slow. In our future work, we will try to tackle this issue by incorporating load balancing and caching techniques.

We will also investigate query mediation and rewriting to automatically construct the queries on the lower levels, based on the defined concepts on the highest layer. This will further simplify the query definition and bring Stream Reasoning closer to the masses.

Acknowledgment

This research was funded by the VLAIO Strategic Fundamental Research (SBO) grant number 150038 (DiSSeCt).

Authors' contributions

PB carried out the study, investigated the various layers, developed the platform, ran the experiments and drafted the manuscript. RT designed the query language, aided in the experimental design and proofread the paper. EDV, FDT and FO supervised the study, participated in its design and coordination and helped to draft the manuscript. All authors read and approved the final manuscript.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. IEEE Communications Surveys & Tutorials, 17(4):2347–2376, 2015.

[2] X. Su, J. Riekki, J. K. Nurminen, J. Nieminen, and M. Koskimies. Adding semantics to internet of things. Concurrency and Computation: Practice and Experience, 27(8):1844–1860, 2015.

[3] E. Della Valle and et al. Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In Proceedings of DEBS, 2016.

[4] M. I. Ali, F. Gao, and A. Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In International Semantic Web Conference, pages 374–389. Springer, 2015.

[5] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. Semantics for the Internet of Things: early progress and back to the future. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012.

[6] A. Margara, J. Urbani, F. Van Harmelen, and H. Bal. Streaming the web: Reasoning over dynamic data. Web Semantics: Science, Services and Agents on the World Wide Web, 25:24–44, 2014.

[7] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. IEEE Communications Surveys & Tutorials, 16(1):414–454, 2014.

[8] E. Della Valle, S. Schlobach, M. Krötzsch, A. Bozzon, S. Ceri, and I. Horrocks. Order matters! harnessing a world of orderings for reasoning over massive data. Semantic Web, 4(2):219–231, 2013.

[9] H. Stuckenschmidt and et al. Towards Expressive Stream Reasoning. In Semantic Challenges in Sensor Networks, 24.01. - 29.01.2010, 2010.

[10] K. Teymourian. A Framework for Knowledge-Based Complex Event Processing. PhD thesis, Free University of Berlin, 2014.

[11] R. Kontchakov and M. Zakharyaschev. An introduction to description logics and query rewriting. In Reasoning Web International Summer School, pages 195–244. Springer, 2014.

[12] D. F. Barbieri and et al. Querying RDF streams with C-SPARQL. SIGMOD Record, 2010.

[13] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. pages 635–644, 2011.

[14] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data, pages 370–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Available from: http://dx.doi.org/10.1007/978-3-642-25073-6{_}24.

[15] R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In OWLED, volume 432, page 91, 2008.

[16] Y. Nenov and et al. RDFox: A Highly-Scalable RDF Store. In ISWC 2015 , Proceedings, Part II, pages 3–20, 2015.

[17] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. Data Science, (Preprint):1–24, 2017.

[18] S. Batsakis, E. G. Petrakis, I. Tachmazidis, and G. Antoniou. Temporal representation and reasoning in OWL 2. Semantic Web, 8(6):981–1000, 2017.

[19] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. Streamrule: a nonmonotonic stream reasoning system for the semantic web. In International Conference on Web Reasoning and Rule Systems, pages 247–252. Springer, 2013.

[20] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, and T. Schaub. Evaluation Techniques and Systems for Answer Set Programming: a Survey. In IJCAI, pages 5450–5456, 2018.

[21] M. I. Ali, N. Ono, M. Kaysar, K. Griffin, and A. Mileo. A Semantic Processing Framework for IoT-Enabled Communication Systems. The Semantic Web-ISWC 2015, pages 241–258, 2015. doi:10.1007/978-3-319-25007-6.

[22] D. Puiu, P. Barnaghi, R. Tonjes, D. Kumper, M. I. Ali, A. Mileo, J. Xavier Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T.-L. Pham, C.-S. Nechifor, D. Puschmann, and J. Fernandes. CityPulse: Large Scale Data Analytics Framework for Smart Cities. IEEE Access, 4:1086–1108, 2016. Available from: http://ieeexplore.ieee.org/document/7447851/, doi:10.1109/ACCESS.2016.2541999.

[23] K. Taylor and et al. Ontology-Driven Complex Event Processing in Heterogeneous Sensor Networks. In The Semanic Web: Research and Applications - ESWC 2011, Proceedings, Part II, pages 285–299, 2011.

[24] S. Gillani, A. Zimmermann, G. Picard, and F. Laforest. A query language for semantic complex event processing: Syntax, semantics and implementation. Semantic Web, (Preprint):1–41, 2017.

[25] R. Tommasini, P. Bonte, E. Della Valle, E. Mannens, F. De Turck, and F. Ongenae. Towards Ontology-Based Event Processing. In OWL: Experiences and Directions–Reasoner Evaluation, pages 115–127. Springer, 2016.

[26] A. Margara, G. Cugola, D. Collavini, and D. Dell'Aglio. Efficient Temporal Reasoning on Streams of Events with DOTR. In European Semantic Web Conference, pages 384–399. Springer, 2018.

[27] P. Bonte, F. Ongenae, F. De Backere, J. Schaballie, D. Arndt, S. Verstichel, E. Mannens, R. Van de Walle, and F. De Turck. The MASSIF platform: a modular and semantic platform for the development of flexible IoT services. KAIS, 2016.

[28] G. Cugola and et al. Processing flows of information: From data stream to complex event processing. ACM Comput. Surv., 44(3):15, 2012.

[29] D. Luckham. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. In Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML, Orlando, FL, USA, 2008.

[30] J. F. Allen. Maintaining knowledge about temporal intervals. Communications of the ACM, 26(11):832–843, 1983.

[31] D. Dell'Aglio, E. Della Valle, J. Calbimonte, and Ó. Corcho. RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. Int. J. Semantic Web Inf. Syst., 10(4):17–44, 2014.

[32] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In Proceedings of the 12th international conference on World Wide Web, pages 48–57. ACM, 2003.

[33] I. Horrocks, O. Kutz, and U. Sattler. The Even More Irresistible SROIQ. Kr, 6:57–67, 2006.

[34] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The SSN ontology of the W3C semantic sensor network incubator group. Web semantics: science, services and agents on the World Wide Web, 17:25–32, 2012.

[35] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, Y. Huang, V. Tresp, A. Rettinger, and H. Wermser. Deductive and Inductive Stream Reasoning for Semantic Social Media Analytics. IEEE Intelligent Systems, 25(6):32–41, 2010.

[36] M. Balduini, I. Celino, D. Dell'Aglio, E. D. Valle, Y. Huang, T. K. Lee, S. Kim, and V. Tresp. Reality mining on micropost streams - Deductive and inductive reasoning for personalized and location-based recommendations. Semantic Web, 5(5):341–356, 2014.

[37] R. Tommasini and E. Della Valle. Challenges & Opportunities of RSP-QL Implementations. 2017.

[38] A. Mauri, J.-P. Calbimonte, D. Dell'Aglio, M. Balduini, M. Brambilla, E. Della Valle, and K. Aberer. Triplewave: Spreading RDF streams on the web. In International Semantic Web Conference, pages 140–149. Springer, 2016.

[39] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In Proceedings of the 7th Workshop on Linked Data on the Web, April 2014. Available from: http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_01.pdf.

[40] J. Z. Pan, Y. Ren, N. Jekjantuk, and J. Garcia. Reasoning the FMA ontologies with TrOWL. In Informal Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation (ORE-2013). CEUR-WS, 2013.

[41] H. Beck, M. Dao-Tran, and T. Eiter. LARS: A Logic-based framework for Analytic Reasoning over Streams. Artificial Intelligence, 261:16–70, 2018.

[42] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev. Ontology-based data access: a survey. IJCAI, 2018.

[43] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. IEEE Internet of Things journal, 1(1):22–32, 2014.

4 C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing

This chapter discusses how RSP engines that process large amounts of RDF data, can efficiently provide lower expressive reasoning capabilities, such as hierarchical reasoning. As the amount of produced data streams keeps rising, stream processors need to be able to keep up with the data rates. Furthermore, in order to integrate data and combine background knowledge to provide insights into a certain domain, these engines should propose some reasoning capabilities. This chapter focuses on efficient hierarchical reasoning over volatile RDF data streams. The solution proposed in this chapter can be utilized in the lower layers of the cascading reasoning approach in Chapter 3. The performance of a cascading approach depends on the efficiency of the lower layers. Furthermore, the more reasoning capabilities can efficiently be executed by these lower layers, the more specific their filtering can be. This chapter investigates Research Question 3: " Can RSP engines efficiently reason over highly volatile data streams?" and validates Hypothesis 3: " Using a hierarchical encoding of concepts will improve the throughput and performing hierarchical reasoning with at least a factor two, compared to the state of the art.".

⋆ ⋆ ⋆

P. Bonte, R. Tommasini, F. Ongenae, F. De Turck and E. Della Valle .

Submitted to the Extended Semantic Web Conference (ESWC), December 2018.

Abstract

Many domains, such as the Internet of Things and Social Media, demand to combine data streams with background knowledge to enable meaningful analysis in real-time. When background knowledge takes the form of taxonomies and class hierarchies, Semantic Web technologies are valuable tools and their extension to data streams, namely RDF Stream processing (RSP), offers the opportunity to integrate the background knowledge with RDF streams. In particular, RSP Engines can continuously answer SPARQL queries while performing reasoning. However, current RSP engines are at risk of failing to perform reasoning at the required throughput. In this paper, we formalize continuous hierarchical reasoning. We propose an optimized algorithm, namely C-Sprite, that operates in constant time and scales linearly in the number of continuous queries (to be evaluated in parallel). We present two implementations of C-Sprite: one exploits a language feature often found in existing Stream Processing engines while the other is an optimized implementation. The empirical evaluation shows that the proposed solution is at least twice as fast as current approaches.

4.1 Introduction

Data stream intensive domains, such as the Internet of Things (IoT) and social media, are still gaining popularity. Huge amounts of frequently changing data are continuously produced [3, 7]. However, to extract meaningful insights from multiple heterogeneous data streams, these streams should be combined and integrated with domain knowledge [10].

For instance, industrial IoT is about deploying sensors on production lines to continuously monitor temperature, pressure, vibrations and hundreds of other types of observations about the production tools deployed along the line[1]. On those industrial settings, it is easy to observe throughputs of MB per second (which means GB per hour)[2]. Both the observations and the tools are often classified using taxonomies. For instance, a taxonomy may tell that a pneumatic drill is as a power drill, a drill, a tool, an instrumentation, etc. All this background knowledge is useful to meaningfully analyze the time-series of observations at-rest, but it challenges real-time analytics. A real-time analysis, willing to aggregate observations about drills, implicitly requires to collect also observations about power drills and pneumatic drills. Naïve implementations may simply register multiple queries and union the resulting stream, but this is a resource-aggressive and human-intensive approach. It would be better that the user declares only the most abstract query (e.g., observations about drills) and a system takes care of efficiently solving the task (e.g., looking for all the specific types of drills).

Semantic Web technologies are valuable tools to combine various heterogeneous data and integrate it with the domain knowledge [7, 19, 23]. Stream Reasoning (SR) is the research domain that investigates how to infer implicit facts about rapidly changing data through reasoning tech-

[1]Interested readers can learn more on https://opcdatahub.com/WhatIsOPC.html

[2]A typical process industry deployment with 200 sensors, which record 20 measurements in 32 bytes messages every 200 ms, generates 0.61 MB/sec (2 GB/hour) per machine. In the oil & gas industry, the number of sensors can easily grow up to hundreds of thousands considering all the machines.

niques, such as found in the Semantic Web [11]. RDF Stream Processing (RSP), a sub-domain of SR, focuses on the integration of highly volatile RDF streams with background knowledge and can continuously answer SPARQL queries while performing simple reasoning.

The need for SR is rising as data stream production increases and the need for real-time analytics over heterogeneous streams keeps growing. The current state-of-the-art in RSP has mainly focused on query answering over RDF streams [6, 18] while more expressive incremental reasoners [22, 26] have focused on providing expressive reasoning capabilities over slower changing data. However, to provide generic query answering, RSP engines should provide some reasoning capabilities [14]. Even simple hierarchical reasoning capabilities, such as subclass and subproperty reasoning increase the expressivity of the query extensively and simplifies data integration.

Currently, there are three approaches to perform reasoning, each with their own drawbacks:

- Materialization: the process of computing all possible inferences, such that the query can be evaluated without reasoning. Therefore, it also produces data that is not relevant for the Query Answering (QA), resulting in many unnecessary computations and redundant statements. Incremental approaches allow to maintain the materialization in a streaming context, however, this can be very expensive depending on the number of changes in the data [5]. The approach pays off when multiple queries consume the materialized stream.

- Goal driven: relies on backward reasoning to infer only what is relevant for the QA. However, backward chaining causes the same intermediate results to be produced over and over again, resulting in redundant computations [24]. Furthermore, in a streaming context, many recomputations occur since there is no incremental approach possible over the data stream.

- Query Rewriting: is the process of injecting the logic inside the query. This results in a query with multiple UNION clauses. However, UNION is not supported in most DSMS on which RSP engines rely. To solve this problem, multiple parallel queries are registered [8]. However, the number of queries is inversely proportional to the throughput.

So even for simple reasoning tasks, such as instance checking over hierarchies of classes and properties, each of these techniques has some serious drawbacks. Furthermore, as data stream production keeps rising, current RSP engines are at risk of failing to perform the reasoning at the required throughput[3].

Information Flow Processors (IFPs), such as Complex Event Processing (CEP) engines and Data Stream Management System (DSMS), often support hierarchical reasoning as a standard language feature. Note that the language feature defines hierarchies over relational data and is not related to RDF. However, if the underlying system inherently understands hierarchies, this could be beneficial for each of the mentioned approaches. Namely, it would result in less unnecessary statements, less recomputations and less queries for the materialization, goal driven and query rewriting approaches respectively.

[3] In the industrial IoT example, each machine produces 20k measurements per second. Each measurement is typically described by at least two RDF triples. In the evaluation, we will see that current RSP engines have a maximum throughput of about 60k triples per second.

This language feature was never before exploited in stream reasoning, since current approaches either pipelined the IFP with a SPARQL engine [6] or integrated limited amount of stream processing inside the reasoner [22]. Furthermore, even though this feature seems very interesting to exploit, its semantics were never formalized, only defined by implementation.

Therefore, in this paper we formalize continuous hierarchical reasoning and introduce C-Sprite, an optimized hierarchical reasoning algorithm that operates in constant time and scales linearly in the number of continuous queries. We present two implementations of C-Sprite: one exploiting the hierarchical features found in existing IFP and one fully optimized based on the theoretical formalization.

In this paper we tackle the following **Research Question:**

1. Can we formalize continuous hierarchical reasoning?
2. Can we exploit the formalization to speed up continuous RSP querying under hierarchical entailment?

We summarize the main **Contributions** as:

1. Continuous Taxonomy-based Relational Algebra (C-TRA), the continuous extension of the existing Taxonomy-based Relational Algebra (TRA) model.
2. The formalization of the hierarchical reasoning through the means of C-TRA.
3. An optimized hierarchical reasoning algorithm, i.e. C-Sprite.
4. An empirical study that validates the approach.

Paper organization: Section 4.2 introduces an example that will be used throughout the paper. In Section 4.3 all necessary background is introduced to understand the remainder of the paper. Section 4.4 and 4.5 formalize the approach, while in Section 4.6 we provide a possible data structure and algorithm to efficiently perform the hierarchical reasoning in continuous query answering. Section 4.7 discusses the related work and in Section 4.8 we provide the empirical study to show the feasibility of the approach. Section 4.9 discusses the contributions and Section 4.10 elaborates on the limitations of the approach and concludes the paper.

4.2 Running Example

Suppose we are interested in retrieving all Wikipedia changes in creative work-related articles. Wikipedia exposes the changes that have been made as a data stream, detailing the changes to each article and the category it is contained in. These categories are very specific, e.g. videogame, novel, article, etc. However, it is not straightforward to target the categories that should be considered creative works.

By introducing a hierarchical description of the various categories, its is possible to define how the categories relate on a hierarchical level. Utilizing a system that understands this hierarchy, one can query the changes stream for changes in creative work-related articles and retrieve

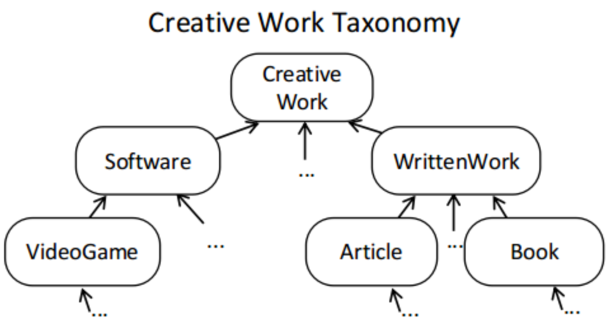


Figure 4.1: Hierarchical structure of Wikipedia categories

the specific underlying changed articles, without the need to query all the categories separately. Figure 4.1 visualizes the hierarchy for the creative work categories.

Throughout the remainder of the paper, we will introduce examples based on this creative work taxonomy.

4.3 Background

This section introduces the background material necessary to understand the remainder of the paper.

4.3.1 RDFS entailment

In our approach, we focus on simple hierarchical reasoning, i.e. reasoning over hierarchies of classes and properties. RDF Schema (RDFS) entailment defines 13 rules[4] to express, among others, hierarchical reasoning but also domain/range reasoning and schema reasoning. We focus specifically on the entailment rules rdfs7 and rdfs9 since they specify the hierarchical reasoning we are interested in:

- rdfs7 states that if p is a subproperty of q and a and b are connected through a property p, then the property q holds between a and b:
  rdfs7: $\frac{(p \quad subPropertyOf \quad q) \quad (a \quad p \quad b)}{(a \quad q \quad b)}$
- rdfs9 states that if A is a subclass of B and a is of the type A, then it holds that a is of the type B:
  rdfs9: $\frac{(A \quad subClassOf \quad B) \quad (a \quad type \quad A)}{(a \quad type \quad B)}$

We note that there exists rules with respect to the transitive properties of the subPropertyOf/subClassOf. However, they are not important in this context as the transitivity can be obtained by the execution of a sequence of rdfs7/9 rules.

[4]https://www.w3.org/TR/rdf11-mt/#rdfs-entailment

Example 9. As shown in Figure 4.1 VideoGame is a subclass of Software and Software is a subclass of CreativeWork. When we have an instance, i.e. Doom, of the type VideoGame (Doom type VideoGame) and execute the rdfs9 rule we obtain that Doom is also a Software:

$$\frac{(VideoGame \quad subClassOf \quad Software)(Doom \quad type \quad VideoGame)}{(Doom \quad type \quad Software)}$$

Since Software is also a subclass of CreativeWork and we now know that Doom is a Software, we obtain through similar means that Doom is also a CreativeWork:

$$\frac{(Software \quad subClassOf \quad CreativeWork)(Doom \quad type \quad Software)}{(Doom \quad type \quad CreativeWork)}$$

Definition 12. The materialization of a knowledge base under RDFS entailment is the process of computing and storing all the inferred facts derived from the executing of the RDFS rules. The materialization stops when no new facts can be derived from the execution of the RDFS rules.

Example 10. (cont'd) The materialization of the knowledge base containing the triple $(Doom\ type\ VideoGame)$ according the the schema depicted in Figure 4.1 results in the triples: $(Doom\,type\,VideoGame)$, $(Doom\,type\,Software)$, $(Doom\,type\,CreativeWork)$.

4.3.2 SPARQL under RDFS entailment

SPARQL is the query language for RDF data[5], different from other QA systems, it can match data that is not explicitly stated, but can be derived under a certain entailment[6]. More specifically, implicit data can be derived from the given data, the ontology and an ontological language (or entailment).

Definition 13. We define the evaluation of SPARQL under entailment as $eval(G, BGP, O, RDFS9)$ with $RDFS9$ the entailment regime, O the ontology, BGP the basic graph pattern used in the SPARQL query and G the RDF dataset.

Example 11. Lets consider again our simple dataset containing the single triple $G = \{(Doom\ type\ VideoGame)\}$. We are interested in querying for all CreativeWork concepts, as defined in the ontology hierarchy in Figure 4.1. The BGP consists thus of "?w type CreativeWork". The ontology O is the ontology represented by the hierarchical definition of concepts as depicted in Figure 4.1 and the entailment is the RDFS entailment consisting of the rdfs9 rule. When evaluating the query without the ontology and the entailment regime only the explicit data can be queried and no matches are found:
$eval(G, BGP, \emptyset, \emptyset) = \emptyset$
When considering the ontology and the entailment regime, we can find a match through the derivation of the implicit data as described in Example 9 (we derive that $Doom$ is a $CreativeWork$) while executing the query:
$eval(G, BGP, O, RDFS9) = \{?w : Doom\}$
Another option is to first materialize the dataset and then evaluate the query without the need for the entailment regime during the query evaluation. First, we obtain the materialize dataset:

[5] https://www.w3.org/TR/rdf-sparql-query/
[6] https://www.w3.org/TR/sparql11-entailment/

(G_{rdfs9}={$(Doom\, type\, VideoGame)$, $(Doom\, type\, Software)$, $(Doom\, type\, CreativeWork)$}). Then we can evaluate the query without the entailment regime:
$eval(G_{rdfs9}, BGP, \emptyset, \emptyset) = \{?w : Doom\}$

4.3.3 Stream Processing

We introduce a window operator to be able to process the content in the stream, based on the definitions from CQL [2] and RSP-QL [13].

Definition 14. A window $W(S)$ is a set of data extracted from a stream S. A time-based window is defined based on two time instances o and c, respectively the opening and closing time instant, such that: $W(S) = \{d|(d, t) \in S \wedge t \in (o, c]\}$. With d all data in S at a specific time instant.

Definition 15. A time-based sliding window operator $\mathbb{W}$ is defined based on three parameters (α, β, t^0), such that α is the width of the window, β is the slide and t^0 is the time instant on which $\mathbb{W}$ starts to operate. The sliding window operator produces a sequence of time-based windows $W_1, W_2, \ldots$ such that: 1) the opening of the first window (W_1) is t^0; 2) each window has width α, i.e. window W_i is defined through (o_i, c_i) with $c_i - o_i = \alpha$; 3) the differences between the opening times of two consecutive windows is β, i.e. the difference between the opening time o_{i+1} of W_{i+1} and o_i of W_i is β.

4.3.4 Hierarchies in Stream and Event Processing Languages

Since we are exploiting the hierarchical language feature found in IFP, we introduce an example to show the hierarchical definitions in these languages. The Event Processing Language (EPL) used in Esper[7] allows to define the hierarchies of events in its language. Listing 4.1 shows an example of defining a small part of the category taxonomy from Figure 4.1 in EPL.

Listing 4.1: Hierarchical definition in EPL

```
create schema CreativeWork(id string, ts double);
create schema Software() inherits CreativeWork;
```

For other examples, we direct the reader to Eckert et al. [15].

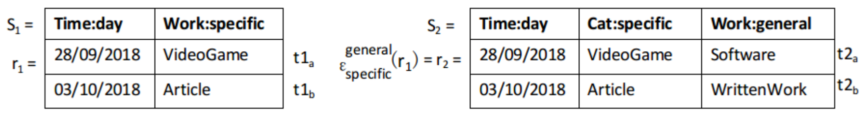
4.3.5 TRA

The Taxonomy-based Relational Algebra (TRA) [20] introduces taxonomies to relax query answering in relational databases. We introduce some of the key concepts of TRA, since they will be used later in the formalization of C-Sprite.

First, we define an h-domain and taxonomies.

Definition 16. An h-domain h is composed of:

[7] http://www.espertech.com/esper/


S_1 =
r_1 =
$t1_a$
$t1_b$
S_2 =
$\varepsilon_{specific}^{general}(r_1) = r_2 =$
$t2_a$
$t2_b$

Time:day	Work:specific
28/09/2018	VideoGame
03/10/2018	Article

Time:day	Cat:specific	Work:general
28/09/2018	VideoGame	Software
03/10/2018	Article	WrittenWork

Figure 4.2: T-relation, t-schema and upward extension TRA example.

- a finite set $L = \{l_1, ..., l_k\}$ of levels, each associated with a set of values, i.e. the members of the level and denoted by $M(l)$;
- a partial order $\leq_L$ on L having a bottom ($\perp_L$) and a top element ($\top_L$).
- a family of functions $LMAP_{l_1}^{l_2} : M(l_1) \to M(l_2)$, called level mappings.

A taxonomy is a set of h-domains.

Example 12. In our running example from Section 4.2 we can create a set of levels
$L = \{creativework, general, specific, ...\}$ with
$M(creativework) = \{CreativeWork\}$,
$M(general) = \{Software, WrittenWork, ...\}$,
$M(specific) = \{VideoGame, Article, Book, ...\}$.

Besides ordering between levels, there is also an ordering between members:

Definition 17. Let h be an h-domain and m_1 and m_2 are members of respectively l_1 and l_2. There exists an ordering on the members $m_1 \leq_M m_2$ if $l_1 \leq_L l_2$ and $LMAP_{l_1}^{l_2}(m_1) = m_2$.

We can now define a schema over taxonomies and t-relations, as the natural extension of a relation table built over taxonomy defined values:

Definition 18. Let T be a taxonomy. A t-schema (schema over taxonomies) for T, is denoted by $S = \{A_1 : l_1, ..., A_k : l_k\}$, with A_i the attribute name of the h-domain and l_i the level of some h-domain in T.

Example 13. Figure 4.2 depicts the t-schema $S_1 = \{Time : day, Cat : specific\}$.

Definition 19. A t-tuple over a t-schema $S = \{A_1 : l_1, ..., A_k : l_k\}$ for a taxonomy T is a function mapping each attribute A_i to a member of l_i. A t-relation r over S is a set of t-tuples over S.

Example 14. In Figure 4.2 we can see the t-tuples $t1_a$ & $t1_b$ in the t-relation r_1.

Last but not least, we introduce the upward extension operator that allows to take the taxonomy into account:

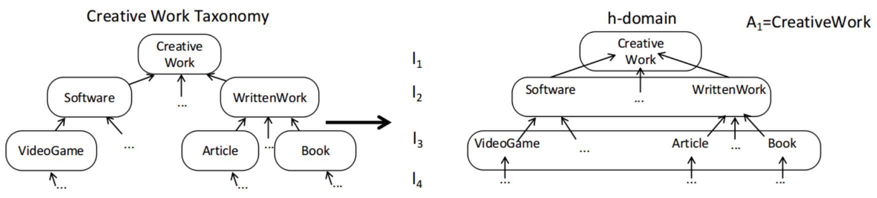


Figure 4.3: Example ontology and taxonomy alignment

Definition 20 (upward extension). Let r be a t-relation over S, A an attribute in S defined over a level l, and l' a level such that $l \leq_L l'$. The upward extension of r to l', denoted by $\varepsilon_{A:l}^{A:l'}(r)$, is the t-relation over $S \cup \{A : l'\}$ defined as:
$\varepsilon_{A:l}^{A:l'}(r) = \{t | \exists t \in r : t[S] = t', t[A : l] = LMAP_l^{l'}(t'[A : l])$

Example 15. Figure 4.2 depicts the upwards extensions $\varepsilon_{specific}^{general}(r_1)$ in r_2.

Besides the upward extension, TRA also provide downward extension, upward/downward selections, projections, unions, differences and joins, which are omitted because they are not relevant for the remainder of the paper. We note that the TRA upward extension and the rdfs9 rule are alternative formalisms to capture the same idea.

Definition 21. TRA- is the subset of TRA without the downward extension, join and difference operators.

In the remainder of the paper, we will assume the usage of TRA-.

4.4 From TRA to SPARQL under entailment

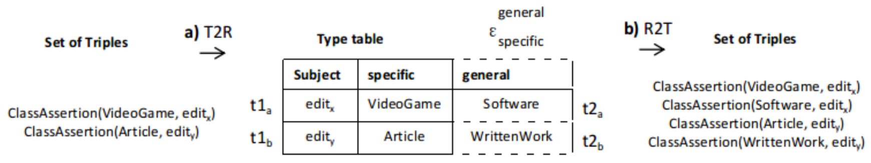
In our approach, we want to formalize the semantics of the hierarchical reasoning inside the IFP, which is built upon Relational Algebra (RA). This can be achieved by extending the RA inside the IFP to include hierarchies, which is exactly what TRA does. Therefore, in this section, we align our approach with TRA.

4.4.1 TRA for Ontologies

We first describe how we can align TRA with ontologies, we limit the ontological language to the definition of classes and properties w.r.t. RDFS rules 7 and 9, thus the hierarchical definitions of classes and properties.

4.4.1.1 Alignment of taxonomies with ontologies

Since ontologies and TRA have a different data model, we first describe how they can be aligned. TRA starts from the assumption of levels and members that is missing in ontologies. However,



ClassAssertion(VideoGame, edit$_x$)
ClassAssertion(Article, edit$_y$)

Subject	specific	general
edit$_x$	VideoGame	Software
edit$_y$	Article	WrittenWork

ClassAssertion(VideoGame, edit$_x$)
ClassAssertion(Software, edit$_x$)
ClassAssertion(Article, edit$_y$)
ClassAssertion(WrittenWork, edit$_y$)

Figure 4.4: Alignment of a) triples with relation data and b) back to triples.

we can introduce the notion of levels by visualizing the ontology classes as one or more trees and assigning all classes that have the same path length from the root to the same level in an h-domain. The members of the levels are the ontology classes themselves. The ordering between the classes is maintained by the ordering between the levels and the members. This is depicted in Figure 4.3. Note that when multiple inheritance occurs in the ontology, multiple h-domains are created. For example, RSP is a subclass of both the Semantic Web and Stream Processing. This results in two h-domains, one with RSP in a sublevel of Semantic Web, and one with RSP in the sublevel of Stream Processing. The alignment for the ontology properties is similar.

4.4.1.2 Alignment of triples with Relational Data

Since we focus on hierarchical reasoning, we limit our discussion to two types of triples, i.e. class assertions and object property assertions. We utilize the Manchester syntax[8] for this purpose: class assertions (i.e. ClassAssertion(C,s), with C an ontology class and s an individual) and object property assertions (ObjectPropertyAssertion(s,P,o), with P an object property and s and o individuals). For simplicity we focus on the class assertions, however, the definitions for object property assertions are straightforward.

Definition 22. We define the function T2R: $\{triples\} \rightarrow R$ that maps a set of triples to relational data through the use of mappings. Each class assertion triple (i.e. $ClassAssertion(C_i, x)$) has a relational presentation where the schema consists of $S = \{Subject, A_j : l_k\}$ with Subject the individual name, A_j the taxonomy attribute of C_i and l_k the level of C_i in the taxonomy. Adding a new class assertion (e.g. $ClassAssertion(C_q, x)$) results in updating the schema by adding a new column to store the additional type.

Example 16. The class assertion ($ClassAssertion(Article, edit_x)$) translates to the first row ($t1_a$) of Figure 4.4 a).

4.4.1.3 Alignment of Relational Data with triples

Definition 23. We define a function R2T: $R \rightarrow \{triples\}$ that maps relational data (obtained by $T2R$) to triples through the use of mappings. Each tuple in $S = \{Subject, A_j : l_k\}$ results in a triple ($ClassAssertion(C_i, x)$) with $t[A_j : l_k] = C_i$. When the schema

[8] https://www.w3.org/TR/owl2-manchester-syntax/

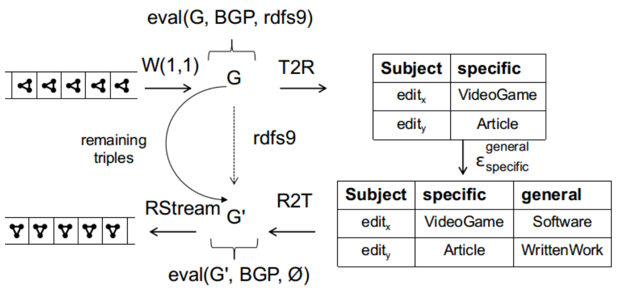


Figure 4.5: Flow of using TRA's upward extension compared to RDFS9.

contains multiple columns (e,g, $S_2 = \{Subject, A_j : l_k, A_p : l_q\}$) then multiple triples are generated.

Example 17. The first row (after extending the table through the upward extension $\varepsilon^{general}_{specific}$) ($t2_a$) of Figure 4.4 b) translates to the class assertions: $ClassAssertion(Article, edit_x), ClassAssertion(WrittenWork, edit_x)$.

The functions R2T and T2R are also known as direct mappings and its inverse application. We refer the interested reader to Sequeda et al. [25] for more a more detailed description.

4.4.2 Alignment of SPARQL under RDFS entailment with TRA

Now that we have aligned ontologies with TRA, we can further formalize our approach by aligning with SPARQL under entailment.

Theorem 1. The SPARQL evaluation of a dataset under RDFS9 entailment and a dataset under upward extension are equal.

Proof. As the evaluation of SPARQL under entailment can be implemented as first materializing the dataset and then evaluating the query, we need to prove:

$$eval(G_{rdfs9}, BGP, \varnothing, \varnothing) = eval(G_{\varepsilon}, BGP, \varnothing, \varnothing) \tag{4.1}$$

Through the materialization, we can further limit the proof to the alignment of the dataset under RDFS9 entailment and upward extension: $G_{rdfs9} = G_{\varepsilon}$.

Figure 4.5 shows how the use of TRA's upwards extension compares to RDFS9.

Assumptions:

1. We assume that the ontology O contains a hierarchy of a certain number of subclasses: $\exists C_0, .., C_{k+1} \in O : C_i \sqsubseteq C_{i+1} \wedge level(C_i) \leqslant level(C_{i+1})$ with $i <= k$ and $level(C_j)$ the mapping of each ontology class to a certain level in the h-domain.

2. There is a function $T2R$ ($R2T$) that maps triples to relation data (relation data to triples), as described in Section 4.4.1.2.

3. There is a function $rdfs9_{i..j}$ that applies the sequence of RDFS9 rules[9] $(C_i \quad subclassOf \quad C_{i+1}), .., (C_{j-1} \quad subclassOf \quad C_j)$
4. n is the depth of the hierarchy used in the reasoning.

Since RDFS9 entailment over a dataset equals the union of the entailment on each triple in the dataset [27], we can simplify the proof for a single triple.
We prove that $rdfs9(x \in C_i) = R2T(\varepsilon(T2R(x \in C_i)))$ for a certain ontology O, with $x \in C_i = ClassAssertion(C_i, x)$:

Base case (n=1): In this case we apply one hierarchical reasoning step. This means that $C_0 \sqsubseteq C_1 \wedge level(C_0) \leqslant level(C_1)$ while its known that $x \in C_0$. For readability we show the base case for $C_0 \sqsubseteq C_1$ but it holds for every i such that $C_i \sqsubseteq C_{i+1}$.
We need to prove that: $rdfs9_{0..1}(x \in C_0) = R2T(\varepsilon_{l_0}^{l_1}(T2R(x \in C_0)))$

Applying the RDF9 entailment we obtain that $x \in C_1$:

$$\frac{(x \quad type \quad C_0)(C_0 \quad subclassOf \quad C_1)}{(x \quad type \quad C_1)} rdfs9 \tag{4.2}$$

Through the upward extension we maintain a table with an additional column:

$$\varepsilon_{l_0}^{l_1}(r) = r \cap T2R((x \quad type \quad C_1)) \tag{4.3}$$

By the definition of $R2T$ and $T2R$ we can conclude that the result of (4.2) equals R2T(4.3).

Inductive hypothesis (n=k): We assume that the theorem holds for all values of n up to some $k, k \geq 0$. With k the difference in hierarchy level.
$rdfs9_{0..k}(x \in C_0) = R2T(\varepsilon_{l_0}^{l_k}(T2R(x \in C_0)))$

Inductive step (n=k+1): Lets assume that the hierarchy is of size $k + 1$.

$$\begin{aligned}
&rdfs9_{0..k+1}(x \in C_0) \\
&\quad = R2T(\varepsilon_{l_0}^{l_{k+1}}(T2R(x \in C_0))) \\
&\quad = R2T(\varepsilon_{l_k}^{l_{k+1}}(\varepsilon_{l_0}^{l_k}(T2R(x \in C_0)))) \text{ def } \varepsilon \\
&\quad = R2T(\varepsilon_{k}^{l_{k+1}}(T2R(rdfs9_{0..k}(x \in C_0))) \text{ inductive step} \\
&\quad = rdfs9_{k..k+1}(rdfs9_{0..k}(x \in C_0)) \text{ base case} \\
&\quad = rdfs9_{0..k+1}(x \in C_0) \text{ def transitivity rdfs9} \\
&Q.E.D.
\end{aligned}$$

□

The proof for rdfs7 was omitted, as it is similar to the proof for rdfs9. In Figure 4.5 we have windowed the data and used the RStream function to assign timestamps to the resulting solution mappings. The RStream function allows you to stream out the obtained answers. We refer the interested reader to Arasu et al. [2] for more information. The incorporation of the streaming operators is further detailed in Section 4.5.

[9]The semantics of the transitive property of subclassof is the same as the sequental excecution of RDFS9.

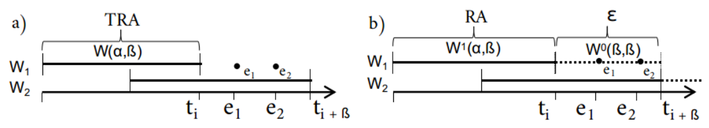


Figure 4.6: Window sequence timeline of a) TRA and b) $\varepsilon(RA)$. In a) events e_1 and e_2 need to wait for the shifting of the window (W_2) to be taken into account, while in b) this period is used to perform the upward extension.

4.5 C-TRA: Continuous TRA

Now that we have aligned our approach in a rather static context, we formalize the applicability in a streaming environment. Therefore, we extend TRA, which has been built for static environments, to Continuous TRA (C-TRA).

To extend TRA to C-TRA, we rely on the "black box" components of CQL [2] that state that instead of integrating the streaming operators in the algebra, it is possible to convert the stream to a relational form through a Stream-to-Relation (S2R) operator. The remaining operations can be performed in relational form, allowing to exploit well-understood relational semantics. CQL can be composed of a S2R operator, followed by well-known relation algebra operations and a Relation-to-Stream (R2S) operator to stream out the results:

$$\text{CQL} = S2R + RA + R2S \tag{4.4}$$

Defining a Stream-to-Stream (S2S) operator in CQL is done by combining a S2R operator with a Relation-to-Relation (R2R) operator that exploits relation algebra and an R2S operator:

$$\text{S2S}_{CQL} = S2R + R2R_{RA} + R2S \tag{4.5}$$

4.5.1 A continuous taxonomy query language

A continuous taxonomic query language (CTQL) that takes taxonomies into account can be defined as standard CQL, but operating on TRA instead of standard RA:

$$CTQL = S2R + TRA + R2S \tag{4.6}$$

The S2S operator over CTQL can then be defined as[10]:

$$\begin{aligned} S2S_{CTQL} &= S2R + R2R_{TRA} + R2S && (4.7)\\ &= \mathbb{W}(\alpha, \beta, t^0) + R2R_{TRA} + RStream && (4.8) \end{aligned}$$

We can further decompose this formula since $TRA = \varepsilon(RA)$. Figure 4.6 a) visualizes a timeline that illustrates the usage of TRA and Figure 4.6 b) its further decomposition. The figure

[10] Note that there are many options to perform the S2R operation. We opted for a sliding window and leave the further generalization for future work.

shows that the events e_1 and e_2, arriving between the shifting of the window, can be upward extended in an additional window, exploiting the wait time in between windows. The decoupling of the upwards extension form the RA allows to perform the upward extension in between window shifts while the RA is executed upon the window. We define this decoupling more formally:

$$\begin{aligned} S2S_{CTQL} &= \mathbb{W}(\alpha, \beta, t^0) + R2R_{TRA} + RStream && (4.9) \\ &= \mathbb{W}(\alpha, \beta, t^0) + R2R_{\varepsilon}(R2R_{RA}) + RStream && (4.10) \\ &= \mathbb{W}^0(\beta, \beta, t^0) + R2R_{\varepsilon} + RStream \\ &\quad + \mathbb{W}^1(\alpha, \beta, t^0) + R2R_{RA} + RStream && (4.11) \end{aligned}$$

with $\mathbb{W}_0(\beta, \beta, t^0)$ opening at the previous evaluation of $\mathbb{W}^1(\alpha, \beta, t^0)$ and closes on the next evaluation. This enforces that each window's t^0, i.e. the time instant on which each window $\mathbb{W}^i$ starts to operate, are synchronized.[11]

This shows that we can extract the extension and execute it before the rest of the processing, while we can rely on RA for the further processing steps. Furthermore, it allows to eliminate data early on, when they do not meet the hierarchical conditions.

4.5.2 C-TRA for Stream Reasoning

Since IFP operate on RA, we extended the use of RA to TRA. We have shown that TRA aligns with ontologies and that C-TRA can be used in a streaming fashion. Furthermore, the decomposition described in C-TRA allows to perform the hierarchical reasoning before the RA. This means that we can perform our triple based hierarchical reasoning inside a IFP and perform the reasoning before other operations such as joins or aggregations. This allows to eliminate triples early on based on the hierarchical requirements in the query.

4.6 Efficient Hierarchical Reasoning

This section discusses how we can efficiently store and retrieve the hierarchy for QA. We begin by describing a data structure for storing the data and the queries and then we discuss the algorithm for querying and study its complexity.

4.6.1 Data structure

A possible data structure for efficient lookup of the parent classes for a specific class in the hierarchy is by saturating the hierarchy and storing for each class a list of all the parents, as visualized in Figure 4.7 a). By storing the list of parents in a hashmap, using the class name as the key and storing the list of parents as the value, one can look up the parents for a specific class in constant time ($O(1)$).

[11] In the remainder of the paper we will focus on a window $\mathbb{W}^0(1, 1, t^0)$ without losing generality, but relaxing the need for synchronizing t^0.

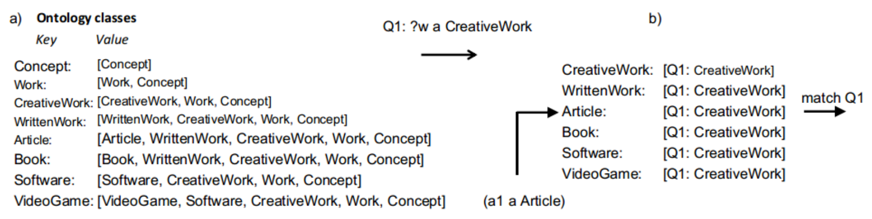

a) Ontology classes
Key
Value
Concept: [Concept]
Work: [Work, Concept]
CreativeWork: [CreativeWork, Work, Concept]
WrittenWork: [WrittenWork, CreativeWork, Work, Concept]
Article: [Article, WrittenWork, CreativeWork, Work, Concept]
Book: [Book, WrittenWork, CreativeWork, Work, Concept]
Software: [Software, CreativeWork, Work, Concept]
VideoGame: [VideoGame, Software, CreativeWork, Work, Concept]
Q1: ?w a CreativeWork
b)
CreativeWork: [Q1: CreativeWork]
WrittenWork: [Q1: CreativeWork]
Article: [Q1: CreativeWork]
Book: [Q1: CreativeWork]
Software: [Q1: CreativeWork]
VideoGame: [Q1: CreativeWork]
(a1 a Article)
match Q1

Figure 4.7: Flow of the algorithm

Since we need a way to efficiently query the data, we create a new instance of the hierarchy that only contains the concepts (keys) that have the queried type in their list of parent concepts. When the concepts have been filtered, we link each concept to the query. When multiple queries are added, each concept contains a list of queries it matches according to the hierarchy. We focus specifically on queries asking for specific type instances (queried types). In Figure 4.7, query Q1 asks for all instances of CreativeWork related categories. The concepts that are not CreativeWorks, such as Concept and Work, are dropped and a direct link is made to the query.

When new data arrives, such as an Article in Figure 4.7, a simple lookup in the hashmap allows us to detect that a match for query Q1 has been found.

4.6.2 Algorithm

A possible algorithm to query hierarchical classes can easily be defined on the data structures introduced above. Algorithm 1 shows the algorithm in pseudo-code that describes how the data structure is constructed to efficiently perform the querying. First, we convert the ontology hierarchy in a hashmap H containing for each class all its parents. Each time a query is registered, a copy of the hierarchy is pruned such that it only contains the concepts that have the queried type in their list of parents. The selected concepts are then directly linked to the queries. This allows to perform the hierarchical reasoning as a simple lookup in a hashmap.

Algorithm 2 is executed on the ingestion of a new $triple$. When a new triple is received, we execute the $CheckHierarchyMatch$ function that takes the triple and the pruned hierarchy hashmap as arguments. By looking up the asserted types of the triple in the hashmap, we can directly detect which queries the triples matches.

4.6.2.1 Complexity study:

Let's assume that the number of queried classes in all the queries is m (i.e. $m = \sum_{i=0}^{len(Q)} len(Q_i)$). Thus, the complexity of first looking up in the pruned hashmap if the triple's type matches any queries and then iterating over them is $O(1) + O(m)$). However, the number of queried classes is typically low. Furthermore, this is independent of the stream itself, i.e. for each triple in the stream, we have the same complexity. We can conclude that the complexity is linear in the number of queries, however constant in the execution over the stream.

Algorithm 1: Query registering

```
Precondition: $Q$ a collection of queries, each interested in one or more types.
 1 $H \leftarrow ConvertToHierarchy(O)$  ▷ Stores parents for each class in the Ontology $O$
 2 function PrepareHierarchy($H, Q$)
 3     $H' \leftarrow []$
 4     for $q \in Q$ do
 5         for $(concept, parents) \in H$ do
 6             if $q \in parents$ then
 7                 $H'[concept].append(q)$
 8             end if
 9         end for
10     end for
11     return $H'$
12 end function
```

Algorithm 2: Calculate the query matches on a hierarchical level

```
Precondition: $Q$ a collection of queries, each interested in one or more types.
 1 $H \leftarrow ConvertToHierarchy(O)$  ▷ Stores parents for each class in the Ontology $O$
   (preprocessing step)
 2 $H' \leftarrow PrepareHierarchy(H, Q)$  ▷ (preprocessing step)
 3 $triple \leftarrow$ ClassAssertion(type,subject)
 4 function CheckHierarchyMatch($H', triple$)
 5     $QueryMatches \leftarrow H'(types(triple))$ ▷ types extracts the type assertions of a
   triple
 6     return $QueryMatches$
 7 end function
```

4.6.3 Definition in EPL

If we want to exploit the hierarchical reasoning inside the IFP, we need to align the triples with the IFP events. One possible way to achieve this is by defining the class and property names as event definitions. For example, the class assertion triple (ClassAssertion(Article,edit)) becomes Article(edit), (with Article an event definition and edit a parameter) and similar for the property assertions. This way Article is an event definition that can exploit the hierarchy.

4.7 Related Work

In this section, we elaborate on the related approaches in the literature that are able to perform hierarchical reasoning (or more) and describe how they compare to C-Sprite. Table 4.1 summarizes the related work.

Compared to C-Sprite, current approaches suffer from the problems that accompany materialization, backward chaining or query rewriting. Either they infer too many triples, perform redundant computations or suffer from a large number of queries.

The C-SPARQL engine [6] builds on existing IFP and SPARQL engines to respectively perform the windowing of the streams and the querying of the data captured in the window. C-SPARQL supports reasoning and query through the use of Jena[12]. However, C-SPARQL is pluggable, allowing the support of other reasoners.

EP-SPARQL (JTalis) [1] is an event processing enabled SPARQL engine that builds on top of logic programming. To perform reasoning EP-SPARQL supports event-driven backward chaining by relying on Prolog.

SPARKWAVE [17] exploits the Rete algorithm [16] to materialize the RDF streams through an adaption of Rete that also incorporates the query answering. Even though it exploits Rete in a smart way, it is prone to the usual materialization problems, i.e. the inference of many unnecessary triples.

StreamQR [8] enables the execution of continuous queries under entailment by rewriting the queries in multiple parallel queries. CQELS [18] is utilized to execute the rewritten queries. It is a very promising technique, however, the query rewriting easily results in a high number of rewritten queries, drastically lowering the engine's performance.

LiteMat [9] uses an encoding scheme to encode the hierarchies to improve materialization and query rewriting in time and space complexity. The encodings allow to translate the entailment problem to a rewriting problem in terms of filtering the hierarchical entailment as numbers. C-Sprite does not encode the hierarchies in a numerical representation, but exploits the hierarchical support of the underlying IFP.

RDFox [22] is the fastest incremental reasoner currently available, utilizing an optimized version of the Delete and Rederive (DReD) algorithm [28] for efficient incremental reasoning. However, it does not provide any mechanisms to deal with high-volatile data streams.

IMARS [5, 12] keeps an incremental maintenance of the materialized knowledge that is valid within a given window of time. It adapts DReD for its applicability in SR. Even though incremental maintenance of the materialization is more efficient than rematerializing each window, its efficiency is dependent on the percentage of changes in the stream.

4.8 Evaluation

To evaluate the feasibility of C-Sprite, we compared C-Sprite's maximum throughput with other engines when increasing the window size and the size of the ontology used to perform the reasoning. Before jumping to these evaluations, we first describe the used dataset and argue the selected engines we compare against.

The evaluation itself was conducted on a 16 core Intel Xeon E5520 @ 2.27GHz CPU with 12GB of RAM running on Ubuntu 16.04 and utilizing Esper 6.1.0.

[12] https://jena.apache.org/

Table 4.1: Comparison C-Sprite and related approaches

	Reasoning	**Components**	**Problems**
C-SPARQL	Materialization	IFP + SPARQL engine	Unnecessary triples
EP-SPARQL	Backward Chaining	Prolog	Redundant computations
IMARS	Incremental Materialization	IFP + SPARQL engine	Dependent on changes
SPARKWAVE	Materialization	Rule engine	Unnecessary triples
StreamQR	Query rewriting	Rewriter + CQELS	Many Queries
LiteMat	Encodings	encoding + SPARQL engine	Vector representation
C-Sprite	Hierarchies	inside IFP	Simple entailment

Table 4.2: Dataset statistics

	Absolute Number	Relative Number
all triples	3.511.629	100%
Creative Works	56.581	1,61%
Top 5 Creative Works:		
MusicalWork	21.438	0,61%
Film	13.890	0,40%
WrittenWork	6.814	0,19%
TelevisionShow	4.579	0,13%
Software	4.493	0,13%

4.8.1 Dataset

DBpedia [4] is Wikipedia content represented as a Semantic Web. DBpedia live [21] provides the Wikipedia changes as structured data, conform to the DBpedia ontology [13]. We have used these changes to re-stream the wikipedia changes as structured data, such that we can control the stream rate in orde to evaluate the throughput of C-Sprite. The evaluation thus consists of querying all the changes to creative works that are happening to DBpedia. The RSP-QL query used within the evaluation is shown in Listing 4.2.

We loaded all the additions made between November 2013 and May 2018. As we are only interested in querying the types of the concepts, we filtered the triples in the data that did not describe a type assertion. Furthermore, when multiple types have been provided in the data, we only keep the most specific type assertions, such that the hierarchical reasoning is necessary to discover the different Creative Works in the data. The final dataset contains more than 3 million triples, of which more than 56 thousand describing Creative Works. Table 4.2 summarizes the characteristics of the used dataset.

[13] https://wiki.dbpedia.org/services-resources/ontology

4.8.2 Engines Selected as Terms of Comparison

We have selected C-SPARQL, StreamQR and SPARKWAVE as they are the most prominent RDF stream processors currently available. We also added StreamFox, i.e. a C-SPARQL approach utilizing RDFox instead of Jena. As some of these approaches can perform more advance reasoning than the hierarchical reasoning discussed in this paper, we carefully adapted their configuration such that the reasoning tasks only consists of hierarchical reasoning such that it is a fair comparison.

Esper implements a similar algorithm as the one we discussed in Section 4.6. Therefore, we build upon Esper to perform the evaluation. Note that we convert the ontology to EPL rules that can be interpreted by Esper and convert the triples in the stream to Esper events. The approach exploiting Esper's internals is further denoted as C-SpriteEsper. As Esper provides more functionality than needed to efficiently perform hierarchical reasoning over data streams and the algorithm is not as optimized as the one described in Section 4.6, i.e. it does not prune the list of parents based on the registered queries. Therefore, in this evaluation we have added our own algorithm as an upper bound. The latter is denoted as C-SpriteOpt[14] and provides the implementation of the optimized algorithm described in Section 4.6. We can thus consider C-SpriteEsper as a lower bound and C-SpriteOpt as an upper bound of the possible C-Sprite performance.

Listing 4.2: High level query in RSP-QL utilized in the evaluation

```
REGISTER QUERY <http://streamreasoning.org/csprite/s1> AS
PREFIX : <http://streamreasoning.org/csprite/>
PREFIX rdf: <http://www.w3.org/1999/02/22—rdf—syntax—ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT *
FROM NAMED WINDOW :win1 [RANGE 1s, SLIDE 1s]
ON STREAM :dbpediaChanges
WHERE {
  WINDOW ?w {
    ?change rdf:type dbpedia:CreativeWork.
  }
}
```

4.8.3 Throughput Evaluation

First, we evaluate the maximum throughput by streaming the DBpedia changes at the highest possible rate. This is done by setting up a websocket such that each engine can consume the stream at the highest possible rate. Figure 4.8 shows the throughput for each of the engines with increasing window sizes. As there are no joins, the window has no significant influence on most engines. C-SPARQL and StreamFox show limited influence as their query engine processes the whole content of the window when the window shifts. The other approaches process each triples as it is injected. We can see that both C-Sprite implementations (i.e., C-SpriteOpt and C-SpriteEsper) clearly outperform all the other approaches.

Figure 4.9 depicts the memory consumption during the same experiment. It is clear that the materialization approaches, i.e. C-SPARQL and StreamFox, have an increasing memory footprint

[14]The source code of CSprite together with the experiment data can be found on https://github.com/pbonte/C-Sprite

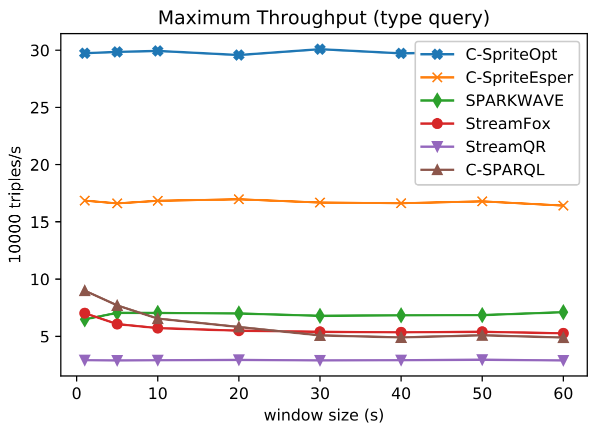


Figure 4.8: Evaluation of the maximum throughput (more is better) of C-Sprite when increasing the window size, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

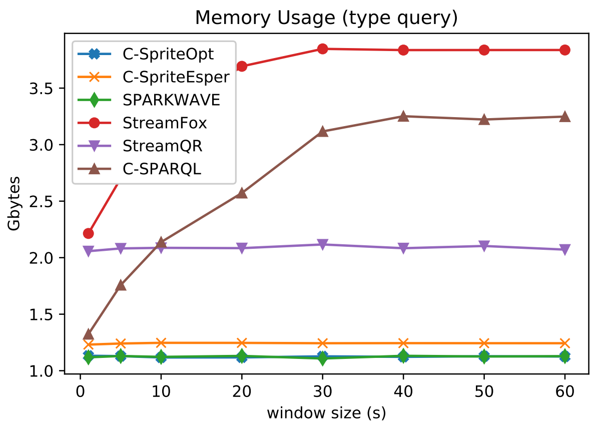


Figure 4.9: Evaluation of the memory consumption (less is better) of C-Sprite when increasing the window size, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

when the window increases. This can be expected as more and more triples need to be maintained inside the window. The other approaches are less prone to the memory increase. We see that the memory footprint of SPARKWAVE is even lower than the one of C-SpriteEsper but similar to C-SpriteOpt. This is due to encoding techniques specially incorporated in SPARKWAVE to lower the memory footprint. Said techniques have not yet been incorporated in C-Sprite. However, the most important message is that C-Sprite's memory consumption is not increasing as with the materialization approaches.

We evaluated the correctness of the various approaches, by analyzing if all the correct query matches were found. We report that each of the approaches produced correct results, even under

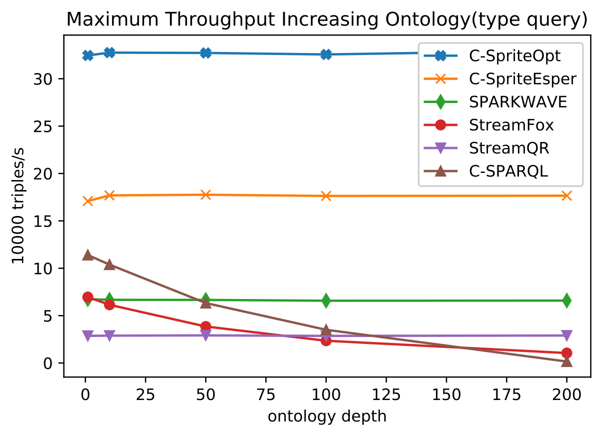


Figure 4.10: Evaluation of the maximum throughput (more is better) of C-Sprite when increasing the ontology depth, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

high load.

4.8.4 Ontology depth Evaluation

The complexity study in Section 4.6 clearly shows that the number of parents which a class has, should not influence the complexity of the approach. Therefore we made the DBpedia ontology artificially deeper. This is done by adding artificial subclasses between the CreativeWork class and the specific classes used within the stream. With the ontology depth, we mean the length of the path from the root to the classes without children if we visualize the ontology as a tree. Figure 4.10 shows the influence of increasing ontology depth on the throughput. We can clearly see that the materialization approaches become less performant when the ontology depth increases. This is due to the fact that more triples need to be inferred.

Figure 4.11 shows the consumed memory while increasing the ontology depth. Almost all the approaches are influenced by the increase in depth. C-Sprite stays rather constant as it is not materializing all the triples. StreamQR and SPARKWAVE show a small increase in memory consumption.

We also evaluated the correctness when increasing the ontology depth of the various approaches. Also in this scenario, all the engines produced the correct results.

4.8.5 Conclusion

It is clear that both C-Sprite implementations outperform the other approaches in terms of maximum throughput. Only SPARKWAVE has a smaller memory footprint than C-SpriteEsper due to its special encoding schemes but similar to it is C-SpriteOpt. C-SpriteOpt sets a realistic upper bound for the performance while C-SpriteEsper sets a possible lower bound. Even if the performance of

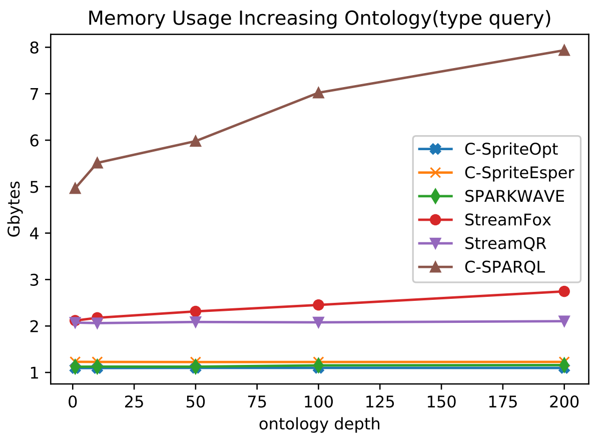


Figure 4.11: Evaluation of the memory consumption (less is better) of C-Sprite when increasing the ontology depth, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

C-SpriteOpt is too optimistic as it currently focuses only on the hierarchical reasoning functionality, the lower bound performance set by C-SpriteEsper still clearly outperforms current approaches.

4.9 Discussion

In this section, we discuss how the approach is positioned in terms of traditional reasoning techniques and we discuss if the approach is feasible for non-streaming situations. As we stated in the introduction there are three approaches to perform reasoning. However, it is not clear where our approach fits. We will now discuss each of the approaches and how they relate to C-Sprite:

- Materialization: It is clear that we are not performing materialization since we do not populate the Assertion Box (ABox) with new facts. However, we have precomputed the hierarchy and when a new triple arrives, we link in a memory efficient way its parents to the triple. In this sense, it is an efficient way of performing materialization.
- Query Rewriting: We are not performing query rewriting since we do not rewrite any queries to contain Terminological Box (TBox) information. However, we do adapt the underlying data structure to create references between the queried concepts and the hierarchies.
- Goal driven: The approach is not goal driven since we are not performing backward reasoning from the goals (the queried types) to the data. However, we are maintaining the relations between the queried types and the hierarchies in the underlying data structure.

It is clear that the C-Sprite does not fit into one specific category. We can conclude that it is a hybrid approach that allows to optimize the underlying data structures for a specific entailment that can be modeled in the form of hierarchies.

The power of the approach lies in the fact that we can check for each triple in the stream directly if it is needed for further processing. The question remains if this approach is feasible for non-streaming approaches. We argue that the approach is beneficial in situations where the data needs to be read from file, in this case we can process triple by triple and start answering the query while reading the data from file. In other cases, the approach is still feasible, but it will not result in the speed-up as in the streaming or reading from file cases. In Section 4.5, we formalized that the hierarchical reasoning can be performed before the rest of the processing, justifying that we can process triple by triple.

4.10 Conclusion

In this paper, we proposed a Stream Reasoning approach that exploits hierarchical language features from the underlying IFP. We have formalized the approach and shown in the evaluation that C-Sprite outperforms existing RSP engines for simple hierarchical reasoning tasks. We have focused on two types of triples only: class assertions and object property assertions. Furthermore, we formalized the approach for reasoning over the classes and did not take joining into account. We argue that the joins can be done at a later stage by utilizing, for example, a left-linear tree.

In future work, we wish to further formalize the approach, i.e. further generalize certain assumptions such as the sliding window of size 1 in Section 4.5. We also wish to exploit the hierarchy to enable more expressive reasoning.

Acknowledgment

This research was funded by the VLAIO Strategic Fundamental Research (SBO) DiSSeCt.

Authors' contributions

PB carried out the study, investigated the various layers, developed the platform, ran the experiments and drafted the manuscript. RT designed the query language, aided in the experimental design and proofread the paper. EDV, FDT and FO supervised the study, participated in its design and coordination and helped to draft the manuscript. All authors read and approved the final manuscript.

References

[1] Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: Ep-sparql: a unified language for event processing and stream reasoning. In: Proceedings of the 20th WWW conference. pp. 635–644. ACM (2011)

[2] Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. The VLDB Journal **15**(2), 121–142 (2006)

[3] Atzori, L., Iera, A., Morabito, G.: The Internet of Things: A survey. Computer networks **54**(15), 2787–2805 (2010)

[4] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. In: The semantic web, pp. 722–735. Springer (2007)

[5] Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: ESWC 2010, Proceedings, Part I. pp. 1–15 (2010)

[6] Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Record **39**(1), 20–26 (2010). https://doi.org/10.1145/1860702.1860705

[7] Barnaghi, P., Wang, W., Henson, C., Taylor, K.: Semantics for the internet of things: early progress and back to the future. International Journal on Semantic Web and Information Systems (IJSWIS) **8**(1), 1–21 (2012)

[8] Calbimonte, J.P., Mora, J., Corcho, O.: Query rewriting in rdf stream processing. In: International Semantic Web Conference. pp. 486–502. Springer (2016)

[9] Cure, O., Naacke, H., Randriamalala, T., Amann, B.: Litemat: a scalable, cost-efficient inference encoding scheme for large rdf graphs. In: Big Data, 2015. pp. 1823–1830. IEEE (2015)

[10] Della Valle, E., et al.: Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In: Proceedings of DEBS. pp. 394–401. ACM (2016)

[11] Della Valle, E., Ceri, S., Van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems **24**(6) (2009)

[12] Dell'Aglio, D., Della Valle, E.: Incremental reasoning on rdf streams. (2014)

[13] Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. Int. J. Semantic Web Inf. Syst. **10**(4), 17–44 (2014). https://doi.org/10.4018/ijswis.2014100102

[14] Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Science **1**(1-2), 59–83 (2017). https://doi.org/10.3233/DS-170006

[15] Eckert, M., Bry, F., Brodt, S., Poppe, O., Hausmann, S.: A cep babelfish: Languages for complex event processing and querying surveyed. In: Reasoning in Event-Based Distributed Systems, pp. 47–70. Springer (2011)

[16] Forgy, C.L.: Rete: A fast algorithm for the many object pattern match problem. In: Readings in Artificial Intelligence and Databases, pp. 547–559. Elsevier (1988)

[17] Komazec, S., Cerri, D., Fensel, D.: Sparkwave: continuous schema-enhanced pattern matching over rdf data streams. In: Proceedings of DEBS. pp. 58–68. ACM (2012)

[18] Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC. pp. 370–388. Springer (2011)

[19] Margara, A., Urbani, J., Van Harmelen, F., Bal, H.: Streaming the web: Reasoning over dynamic data. Web Semantics: Science, Services and Agents on the World Wide Web **25**, 24–44 (2014)

[20] Martinenghi, D., Torlone, R.: Taxonomy-based relaxation of query answering in relational databases. The VLDB Journal **23**(5), 747–769 (2014)

[21] Morsey, M., Lehmann, J., Auer, S., Stadler, C., Hellmann, S.: Dbpedia and the live extraction of structured data from wikipedia. Program **46**(2), 157–181 (2012)

[22] Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: Rdfox: A highly-scalable rdf store. In: ISWC. pp. 3–20. Springer (2015)

[23] Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context aware computing for the Internet of Things: A survey. IEEE Communications Surveys & Tutorials **16**(1), 414–454 (2014)

[24] Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited, (2009)

[25] Sequeda, J.F., Tirmizi, S.H., Corcho, O., Miranker, D.P.: Direct mapping sql databases to the semantic web: A survey. Univeristy of Texas, Department of Computer Sciecnces Technical Report TR-09-04 (2009)

[26] Thomas, E., Pan, J.Z., Ren, Y.: TrOWL: Tractable OWL 2 Reasoning Infrastructure. In: the Proc. of the Extended Semantic Web Conference (ESWC2010) (2010)

[27] Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: Webpie: A web-scale parallel inference engine using mapreduce. Web Semantics: Science, Services and Agents on the World Wide Web **10**, 59–75 (2012)

[28] Volz, R., Staab, S., Motik, B.: Incrementally maintaining materializations of ontologies stored in logic databases. In: Journal on Data Semantics II, pp. 1–34. Springer (2005)

5
Subset Reasoning for Event-Based Systems

Chapter 3 explained how relevant data in data streams can be selected in order to perform expressive reasoning over volatile data streams. However, this does not provide a solution to cases where there are large amounts of static background knowledge that need to be combined with event data. This chapter tackles this exact problem by approximating a relevant subset of ABox data, such that the amount of data to reason upon can be minimized. Chapter 2 and 3 described how services can subscribe to semantic IoT data. However, these services often still need to combine the subscribed event data with large amounts of background knowledge in order to make accurate decisions. Furthermore, the solution described in this chapter can be utilized in the platforms described in Chapter 2 and 3 when large amounts of static background knowledge are required to correctly handle the abstraction of the various events. This chapter investigates Research Question 4: "Can expressive reasoning over event data, that needs to be combined with large static knowledge bases, be employed in time-critical use cases?" and validates Hypothesis 4: "Using an approximation technique that extracts a subset of data to reason upon, we can speed up the expressive OWL 2 DL reasoning process at least 10 times, compared to the state of the art.".

⋆ ⋆ ⋆

P. Bonte, F. Ongenae, F. De Turck.

Submitted to IEEE Access, January 2019.

Abstract In highly dynamic domains such as the Internet of Things (IoT), Smart Industries, Smart Manufacturing or Social Media, data is being continuously generated. By combining this generated data with background knowledge and performing expressive reasoning upon this combination, meaningful decisions can be made. Furthermore, this continuously generated data typically originates from multiple heterogeneous sources. Ontologies are ideal for modeling the domain and facilitate the integration of heterogeneous produced data with background knowledge. Furthermore, expressive ontology reasoning allows to infer implicit facts and enables intelligent decision making. The data produced in these domains is often volatile. Time critical systems, such as IoT Nurse Call systems, require timely processing of the produced IoT data. However, there is still a mismatch between volatile data and expressive ontology reasoning, since the incoming data frequency is often higher than the reasoning time. For this reason, we present an approximation technique that allows to extract a subset of data to speed-up the reasoning process. We demonstrate this technique in a Nurse Call proof of concept where the locations of the nurses are tracked and the most suited nurse is selected when the patient launches a call. Furthermore, we evaluate using an extension of an existing benchmark. We managed to speed up the reasoning process up to 10 times for small datasets and up to more than 1000 times for large datasets.

5.1 Introduction

5.1.1 Problem Description

Highly dynamic domains such as the Internet of Things (IoT), Smart Industries, Smart Manufacturing, financial sector or Social Media require real-time processing of heterogeneous generated data [1]. These time critical systems need to react as quick as possible to newly generated event data. However, many of these systems need to integrate background knowledge with the event data on the fly, to enable real-time interpretation of these events and execute advanced logics to make correct decisions [2, 3]. For instance, in an IoT nurse call system, expressive reasoning is required to capture the capabilities of the nurse, the pathologies of the patients, the relation between the patients and the staff, etc [4]. To automatically determine the priority of a launched patient call, the pathology of the patient needs to be inspected. Depending on the patient's disease, the call gets a higher priority. Similar examples can be thought of in other domains such as detecting hazard situations in a smart manufacturing scenario or reacting to traffic jams in smart cities.

Semantic web technologies, such as ontologies, are the preferred model for the integration of the generated heterogeneous data with background knowledge [5, 6]. An ontology formally describes concepts, properties, and their relations, within a certain domain. By defining the relations between various concepts, a model can incorporate the knowledge about a certain domain. Through the use of a reasoner, implicit facts can be automatically inferred. For example, by modeling that a 'high priority call' is a call made by a patient that has a certain risk profile, the reasoner can automatically decide which calls should be handled with higher priority. Note that the fact that a patient has a risk profile can be inferred based on the pathology and the history of the patient. To make intelligent conclusions, the reasoner should be able to handle highly expressive definitions in the ontology. We opted for Web Ontology Language (OWL) reasoning, which uses

Description Logic (DL), as OWL is widely used and it is a web standard.

However, currently, there is still a mismatch between expressive reasoning and real-time requirements [1]. Expressive reasoning techniques such as DL reasoning can have up to NEXPTIME complexity [7], resulting in slow reasoning times with growing datasets [8, 9]. In this paper, we present a practical subset reasoning technique that combines expressive reasoning and event-based requirements by extracting and approximating a subset of data. The subset minimizes that dataset to reason upon, speeding up the reasoning process.

Many advances have been made in the Stream Reasoning domain [10–12] to combine data from multiple streams with static background knowledge. Generated event data produced by various sources can be considered as data streams. To be able to process these unbounded streams of data, stream reasoning techniques consider the data within a defined time frame, i.e., a window. Expressive reasoning platforms have mostly focused on the processing of static data [13] or slowly changing data [14]. When these systems try to process data streams, newly incoming data will pile up, eventually crashing the system, since each item needs to be processed one by one [15]. By using windows, multiple data items can be processed simultaneously. However, a problem that arises when using windows in combination with expressive reasoning, is the possible inconsistency within a window. For example, when an individual is a member of two disjunct classes due to considering the data within the window. However, the content of the window should never be inconsistent, only the most recent statement should be considered [16]. Handling these inconsistencies within a window is still an open problem [16].

5.1.2 Related Work

We now discuss the most prominent works in the literature and their drawbacks.

Traditional OWL2 DL reasoner such as HermiT [13] and Pellet [17] focus on the processing of static or very slow changing data. They provide no mechanisms for the processing of event data and are typically too slow to handle event data.

PAGOdA [18] is a hybrid approach that combines a datalog reasoner with an OWL2 reasoner. Most of the computations are executed by the fast datalog reasoner and only if necessary the OWL2 reasoner computes the missing facts. Although it is a very promising technique, in its current state PAGOdA focuses on querying and does not allow the adding and removal of facts which is necessary in a changing environment. This means that the PAGOdA reasoner would need to be restarted each time new data arrives. PAGOdA uses RDFox [14] as its datalog reasoner. RDFox is the fastest incremental OWL2 RL reasoner currently available. As we will discuss in Section 5.3, OWL2 has three profiles that minimize the expressivity to increase the efficiency of reasoning. RDFox is thus not as expressive as OWL2 DL reasoning. Furthermore, as PAGOdA and RDFox focus more on static domains, they do not provide any mechanisms to process data streams, such as windowing or update policies.

TrOWL [19] offers a subset of OWL2 DL expressiveness while maintaining tractable, by using language transformations. It supports stream reasoning by incrementally processing the addition and removal of facts. As only a subset of OWL2 DL is supported, TrOWL does not support nominals or datatype reasoning.

Many RDF Stream Processing (RSP) techniques exist [10, 11] that consider streams of RDF data within a predefined window and process only the content within the window. When new data arrives, or when time passes, the window slides over the data stream and a new portion of the data stream is processed. These techniques support the integration with background data and support various streaming operators such as aggregations. However, due to the high velocity of data these systems need to process, the reasoning capabilities are low. The most expressive RSP engine is StreamQR [20], which supports the $\mathcal{ELHIO}$ logic, which falls under the OWL2 EL fragment, one of the most expressive logics currently used for query rewriting. As OWL2 EL is another profile of OWL2 DL, it is less expressive.

Other techniques such as module extraction and ontology partitioning focus on minimizing the ontology Terminological Box (TBox). Module extraction techniques [21] allow to extract a part of the TBox to speed up the reasoning process in a specific case, e.g. to type check some specific classes. While ontology partitioning techniques split the ontology into smaller self-contained modules [22]. Both techniques focus on minimizing the TBox but provide no solution for growing Assertion Box (ABox)es. Anagnostopoulos et al. [23] highlights the importance of approximate reasoning in order to perform time-critical decision making. They utilize probabilistics to approximate certain reasoning tasks, based on the similarity with other situations, without dealing with highly expressive ontologies. This implies that the results might not always be correct. In our approximation approach, we aim for correct answers achieved by approximating a subset of data to perform the expressive reasoning upon.

5.1.3 Objectives & Solution

To allow the design of time critical systems within highly dynamic domains, we set the following objectives:

1. Heterogeneous data: Since data typically needs to be combined from various heterogeneous sources, we need to be able to integrate this heterogeneous data.

2. Event data: Data is produced continuously, therefore new data should be added to the system and old data should be updated or removed.

3. Large knowledge bases: Many domains have large knowledge bases that need to be combined with the generated data, e.g. sensors typically only describe their sensor readings and still need to be combined with the sensor itself and the location of the sensor, etc.

4. Expressive reasoning : In order to correctly interpret the domain, expressive reasoning is required to correctly analyze the domain definitions.

To tackle these challenges we propose an approximation technique that minimizes the dataset to reason upon, in order to speed up the reasoning process in an event-based environment. To solve the windowing problem, we define various update policies that describe how the most recent data in the stream should be captured. As such, instead of a window, we maintain a recent view on the stream and use this recent view to compute our subset.

We show that our subset approach scales very well, even with large amounts of data (i.e. instantiation data), making it an ideal tool for time critical systems that require expressive reasoning.

5.1.4 Paper Organization

The remainder of this paper is structured as follows: Section 5.2 introduces the nurse call use case used throughout the paper. In Section 5.3 the background to understand the remainder of the paper is explained. Section 5.4 describes the policies that allow to maintain a recent view on the data stream. Section 5.5 explains the subset approximation algorithm, while Section 5.6 describes the implementation details of the system. Section 5.7 evaluates our technique by comparing the execution time of the use case from Section 5.2 and a benchmark to existing techniques and discusses the results. We show that our technique is up to 10 times faster for really small datasets and up to more than 1000 times faster for larger datasets. Section 5.8 discusses how the results should be interpreted and Section 5.9 concludes the paper and identifies future research paths.

5.2 Use Case Description

In the remainder of the paper we focus on an IoT nurse call system to introduce and explain our approach. We note that our approach is applicable for any event-based environment requiring expressive reasoning.

The IoT plays a crucial role in providing optimal and personalized care for patients. The advances in this field allow patients to be easily monitored [24] and to localize the necessary staff members [25]. However, to achieve truly personalized care, profile, context and domain information needs to be considered. Most of this information is rather static, e.g., the patient's profile and pathology, the competences of the nurses and the floor-plan of the hospital. Discrete streams of events representing, e.g. patient' calls, person location updates, call status updates, need to be combined with the static data to derive actionable insights. In this use case, we consider a call assignment scenario, where the most suited staff member at a particular moment should be assigned to a patient call. The nurse selection procedure is made up out of a rather large decision tree consisting of 36 leaves [4]. The tree was constructed together with domain experts, i.e. nurses, doctors and patients, and a company specialized in nurse call systems (Televic Healthcare[1]). The selection procedure takes into account, amongst others, the personal relation of the staff members with the patients, the location of the staff members and their competences. We consider the following scenario within this paper, which typically occurs during the night, consisting of the following steps:

[1]http://www.televic-healthcare.com/

1. **Call Launched:** A patient launches a call and the selection procedure is started to assign the most appropriate nurse to the call. The nurse is notified of this assignment.

2. **Call Redirect:** The nurse is currently busy and indicates that the call should be redirected. The selection algorithm runs again to assign and notify another nurse.

3. **Call Temporary Accept:** The new nurse temporary accepts the call. This is a temporary accept because the call can only be completely accepted once the nurse is with the patient. This allows easy re-assignments in case of interruptions or delays.

4. **Corridor:** The nurse moves towards the room of the patient and continuous location updates are registered by the IoT system.

5. **Patient Location:** When the nurse arrives in the patient room, a new location update is sent. Some lights in the room automatically turn on at the appropriate low level, since a staff member is present in the room.

6. **Presence on:** The nurse logs into the terminal in the patient room. The call is now accepted and the correct lights, depending on the procedure, turn on. For example, for a medical procedure, the spot lights above the bed turn on, while for an assistance procedure the mood lighting is activated.

7. **Presence off:** The nurse inputs some additional administrative information about the procedure of the call on the terminal and logs out. The call is now finished and the procedure lights turn off.

8. **Corridor:** The nurse leaves the room. The location of the nurse is updated and since no staff member is in the room, all the lights turn off.

Since regulations stipulate that a nurse should be present in the room within three or five minutes (depending on the country) when a call has been made, the allowed decision time should be limited to five seconds, to allow for plenty of time for the nurse to move to the room. Therefore, the data should be processed in a timely manner to meet these timely requirements. To represent the eHealth knowledge, the ACCIO ontology[2] is used, which has been constructed in collaboration with domain experts. An elaborate description can be found in Ongenae, et al. [4].

The decision tree of the selection procedure was translated into SPARQL queries, that takes into account the background, profile and context information captured within the ACCIO ontology.

5.3 Background

This section introduces the necessary background on which the remainder of this paper is built.

[2]https://github.com/IBCNServices/Accio-Ontology/

5.3.1 Description Logics

The popularity of OWL has led to the design of OWL2, defining the foundations of OWL2 DL reasoning [26]. **Description Logics** [27] are the logical-based formalisms on which OWL2 DL has been built [28]. We introduce the syntax of a simplified DL, explaining the basic notions to understand the remainder of the paper. We refer the reader to Horrocks et. al. [29] for a more thorough description of the logic $\mathcal{SROIQ}$ (which is used within OWL2 DL) and its semantics. DL defines concepts to represent the classes of individuals and roles to represent binary relations between the individuals. Concrete roles (or data properties) are roles with datatype literals in the second argument.

DL languages contain concepts names $A_1, A_2, ...$, role names $P1, P2, ...$ and individual names $a_1, a_2, ...$ A role R is either a role name P_i, its inverse P_i^- or a complex role $R_1 \circ \cdots \circ R_n$ consisting of a chain of roles. Concepts C are constructed from: two special primitive concepts $\bot$ (bottom) and $\top$ (top) or concepts names and roles using the following grammar:

$$C ::= A_i|\top|\bot|\neg C|C_1 \sqcap C_2|C_1 \sqcup C_2|\exists R_1.C_1|\forall R_1.C_1$$

Note that the two last concepts are called, respectively existential ($\exists$) and universal ($\forall$) quantifiers. More expressive constructs such as qualified number restrictions are allowed as well:

$$C ::= \geq nR.C_1| \leq nR.C_1$$

Meaning that at least or at most a specific number n of relations R should be present. Nominal support allows to restrict to specific individuals instead of concepts:

$$C ::= \exists R.\{a\}|\exists R.\{a_1, a_2, ..a_n\}$$

Where the latter can be seen as “one-of”. Data property restrictions can restrict the values of data properties:

$$C ::= \exists R. \geq n|\exists R. \leq n$$

A TBox $\mathcal{T}$, is a finite set of concept (C) and role (R) inclusion axioms of the form

$$C_1 \sqsubseteq C_2 \text{ and } R_1 \sqsubseteq R_2$$

with C_1, C_2 concepts and R_1, R_2 roles. A concept equation ($C_1 \equiv C_2$) denotes that both C_1 and C_2 include each other:

$$C_1 \sqsubseteq C_2 \text{ and } C_2 \sqsubseteq C_1$$

An ABox $\mathcal{A}$ is a finite set of concept and role assertions of the form

$$C(a) \text{ and } R(a, b)$$

with C a concept, R a role and a and b individual names. $ind(\mathcal{A})$ denotes the set of individuals occurring in $\mathcal{A}$. A Knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ combines $\mathcal{T}$ and $\mathcal{A}$.

We can now define the semantics using an interpretation $\mathcal{I}$. $\mathcal{I}$ is a pair ($\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}$) consisting of a non-empty domain of interpretation $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$. The interpretation function assigns:

- an element $a_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to each individual name a_i,
- a subset $A_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to each concept name A_i,
- a binary relation $P_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each role name P_i.

We can now use the interpretation function to define the semantics of the above defined grammar:

$$\begin{aligned}
(P^-)^{\mathcal{I}} &= \{(v,u)|(u,v) \in P^{\mathcal{I}}\},\\
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}},\\
\bot^{\mathcal{I}} &= \emptyset,\\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}},\\
(C_1 \sqcap C_2)^{\mathcal{I}} &= (C_1)^{\mathcal{I}} \cap (C_2)^{\mathcal{I}},\\
(C_1 \sqcup C_2)^{\mathcal{I}} &= (C_1)^{\mathcal{I}} \cup (C_2)^{\mathcal{I}},\\
(\exists R_1.C_1)^{\mathcal{I}} &= \{u|\exists v \in C^{\mathcal{I}} \wedge (u,v) \in R^{\mathcal{I}}\},\\
(\forall R_1.C_1)^{\mathcal{I}} &= \{u|\forall v \in C^{\mathcal{I}} \wedge (u,v) \in R^{\mathcal{I}}\}.
\end{aligned}$$

We call $\mathcal{M} = \mathcal{K}^{\infty}$ the **materialization** of $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, i.e. all inferred axioms w.r.t. explicit individuals in $\mathcal{K}$ are computed and explicitly stored. For example, based on the knowledge defined in $\mathcal{T}$, additional axioms regarding $\mathcal{A}$ can be extracted.

OWL2 contains three profiles, each limiting the expressivity power in a different way, to ensure efficiency of reasoning:

- OWL2 RL: does not allow existential quantifiers on the right-hand side of the concept inclusion, eliminating the need to reason about individuals that are not explicitly present in the knowledge base. Furthermore, it does not allow quantified restriction, e.g. minimum, maximum or exactly a specific number of quantified roles. This profile is ideal to be executed on a rule-engine.

- OWL2 EL: mainly provides support for conjunctions and existential quantifiers. This profile is ideal for reasoning over large TBoxes that do not contain, among others, universal quantifiers, quantified restrictions or inverse object properties.

- OWL2 QL: does not allow, among others, existential quantifiers to a class expression or a data range on the left-hand side of the concept inclusion. This makes the profile ideal for query rewriting techniques.

Note that each of these profiles is a subset of OWL2 DL. Expressive logics such as OWL2 DL require special techniques to support their reasoning, such as the tableaux algorithm [30, 31] which are provided by reasoning systems [13, 17, 32, 33].

5.3.2 Reasoning Techniques

Looking at the literature, we can distinguish three categories of reasoning techniques within our domain:

- Reason on query time: These approaches perform the reasoning while executing the query. The query evaluation itself is then performed under a certain entailment, in order to incorporate the reasoning. The query evaluation can be formalized as $eval(Q, \mathcal{K}, \mathcal{E})$, with Q the query that needs to be evaluated, $\mathcal{K}$ the ontology ABox and TBox and $\mathcal{E}$ the entailment regime that defines the expressivity of the reasoning during query evaluation.
- Materialization: Materialization approaches materialize their data, such that queries can be executed without the need for reasoning during query execution. The query evaluation can be formalized as $eval(Q, \mathcal{K}^{\infty}, \emptyset)$, with $\mathcal{K}^{\infty}$ the materialization of $\mathcal{K}$. The entailment regime is $\emptyset$, denoting that there should be no reasoning while evaluating the query.
- Query Rewriting: Rewriting approaches rewrite the provided queries based on the ontology TBox, such that the query contains all the information from the ontology. The query evaluation can here be defined as $eval(Q', \mathcal{K}, \emptyset)$, with Q' the rewritten query such that the reasoning is contained within the query. The entailment regime is here also $\emptyset$, thus no reasoning should be executed while evaluating the query. We note that only a subset of OWL2 DL can be rewritten.

Figure 5.1 visualizes the flow of the different reasoning techniques. We make a distinction between the loading of the data at start-up, at the top, and the adding of event data during runtime, at the bottom. The figure shows that, once the event data has been added to the ontology at runtime, the rewriting and materialization approaches don't require reasoning at query time. However, the materialization results in computing all possible assertions in the knowledge base, some might be irrelevant for the query answering. While the query rewriting results typically in a very complex query and only a small subset of OWL2 DL can be rewritten.

5.4 Update Policies

In an event-based environment, data is continuously produced and special techniques are necessary to capture the current view on the data streams. We introduce the definition of Update Policies that allow to define how the current view should be constructed. The current view differs from the window operator as it is updated based on the previous current view, while the window is an operator that captures the stream in processable chunks. We start with the definition of a current view on a data stream:

Definition 24. Ax is a set of ABox axioms and $S = Ax_1, Ax_2, ..., Ax_n$ is a time varying sequence of axioms. $Ax_S(t)$ is a function that extracts a view at a specific time instant from the Stream S. $Ax_{current}$ is a current view on an axiom Stream S, it describes as a set of axioms how the updates in the stream should be interpreted at the current time.

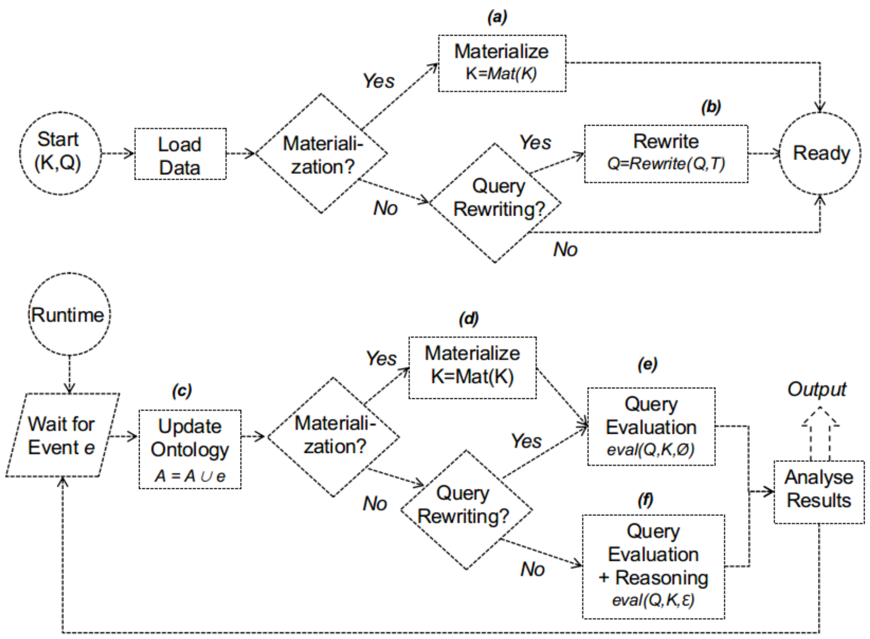


Figure 5.1: Visualization of the flow of materialization, query rewriting and reasoning on query time approaches.

We can now introduce the notion of an update policy:

Definition 25. An Update Policy U_p is a function that updates the view with the incoming axioms of the stream to a current view:
$Ax_{current}(t) = U_p(Ax_{current}(t-1), Ax_S(t))$, with t the latest time instant.

Let's say we have a stream S of axiom sets Ax_i, i.e. $S = Ax_1, Ax_2, ..., Ax_n$ with $i = 1..n$ time instances.

We define three basic update policies. Figure 5.2 visualizes the differences between these policies based on different axioms set in the data stream.

1. **Latest:** always takes the latest axiom set in the data stream, similar to the Now operator that can be defined for window functions in streams, namely:

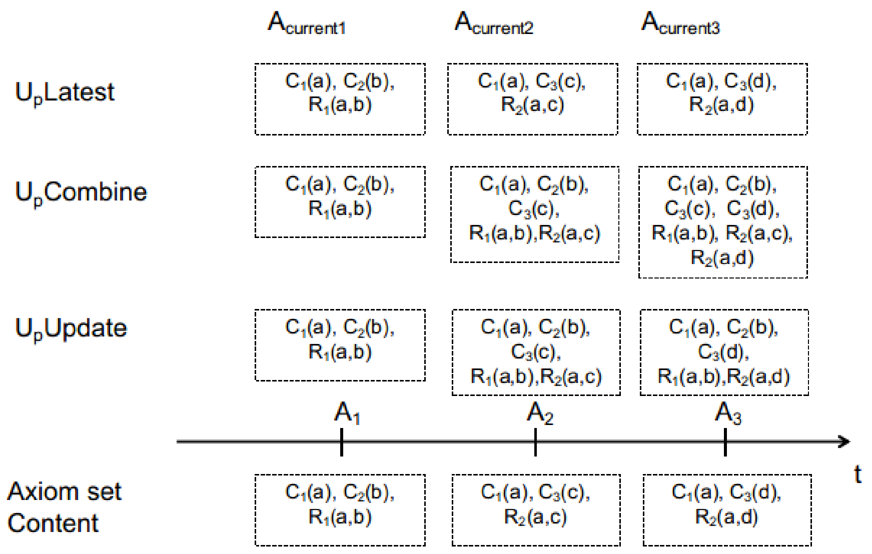
$$U_P Latest(Ax_1, Ax_2) = Ax_2$$

2. **Combine:** merges the sets of axiom's together.

$$U_P Combine(Ax_1, Ax_2) = Ax_1 \cup Ax_2$$

3. **Update:** overwrites existing relations and merges new ones.

$$U_P Update(Ax_1, Ax_2) = (Ax_1 \setminus Ax^-(Ax_1, Ax_2)) \cup Ax_2$$



$A_{current1}$
$A_{current2}$
$A_{current3}$
U_pLatest
U_pCombine
U_pUpdate
Axiom set Content
$C_1(a), C_2(b), R_1(a,b)$
$C_1(a), C_3(c), R_2(a,c)$
$C_1(a), C_3(d), R_2(a,d)$
$C_1(a), C_2(b), C_3(c), R_1(a,b), R_2(a,c)$
$C_1(a), C_2(b), C_3(c), C_3(d), R_1(a,b), R_2(a,c), R_2(a,d)$
$C_1(a), C_2(b), C_3(d), R_1(a,b), R_2(a,d)$
A_1
A_2
A_3
t

Figure 5.2: Visualization of resulting current views based on the different update policies.

with

$$Ax^{-}(Ax_1, Ax_2) = \{R(a,b) \in Ax_1 | \exists R(a,c) \in Ax_2 \wedge b \neq c\}$$

Note that more advanced policies are possible, however, out of scope for this paper.

5.5 Subset Reasoning

Now that we can maintain a view on the data stream according to the defined policy updates, we introduce the concept of subset reasoning as an approximation technique to efficiently speed up the reasoning process. The technique calculates a subset of ABox data from the knowledge base, based on the new axioms in the data stream. The extracted subset allows to efficiently calculate the materialization of the new axioms. Note that this requires the knowledge base to be in a materialized state.

5.5.1 Defining Subset Reasoning

Before we define how to extract the subset, we need to identify how big the subset should be (similar to the Modal depth [34]):

Definition 26. We define the Concept Depth as a recursive function, starting from all axioms $C_{eq} \in \mathcal{T}$ with C_{eq} a concept inclusion or concept equivalence and C_i concepts and R_i roles:

$$\begin{aligned}
depth(C_1 \sqsubseteq C_2) &= max(depth(C_1), depth(C_2)) \\
depth(C_1 \equiv C_2) &= max(depth(C_1), depth(C_2)) \\
depth(C_i) &= 0 \text{ with } C_i \text{ a concept name} \\
depth(C_1 \sqcap C_2) &= max(depth(C_1), depth(C_2)) \\
depth(C_1 \sqcup C_2) &= max(depth(C_1), depth(C_2)) \\
depth(\exists R.C_1) &= depth(R) + depth(C_1) \\
depth(\forall R.C_1) &= depth(R) + depth(C_1) \\
depth(R_1 o ... o R_n) &= depth(R_1) + ... + depth(R_n) \\
depth(R_i) &= 1
\end{aligned}$$

The Concept depth thus calculates the number of relations defined in a concept. We can now calculate the deepest concept within the TBox:

Definition 27. We define the TBox Depth as:

$$depth(\mathcal{T}) = max(\{depth(C) | C \in \mathcal{T}\}).$$

The TBox depth will define how big the subset should be.

Example 18. We calculate the TBox depth based on the following example TBox:

$$\begin{aligned}
\mathcal{T} = \{ & \\
& NormalCall \equiv Call \sqcap \exists callMadeBy. \\
& \qquad \exists hasRole.(Patient \sqcup Resident) \\
& CareCall \equiv NormalCall \sqcap \exists hasReason.CareReason \\
& PriorityCall \equiv Call \sqcap \exists callMadeBy. \\
& \qquad \exists hasProfile.RiskProfile \\
& Patient \equiv Role \sqcap \exists hasDetails. \\
& \qquad \exists isAdmittedTo.Hospital \\
& Resident \equiv Role \sqcap \exists hasDetails. \\
& \qquad \exists isAdmittedTo.ResidentCareCenter \\
\} &
\end{aligned}$$

The depths for each of these concepts is respectively:

$$\begin{aligned}depth(NormalCall) &= 2\\ depth(CareCall) &= 1\\ depth(PriorityCall) &= 2\\ depth(Patient) &= 2\\ depth(Resident) &= 2\end{aligned}$$

The TBox depth is $max(\{2, 1, 1, 2, 2\}) = 2$.

Before introducing the definition of a subset, we define the collection of all outgoing relations as the function r_{out}:

Definition 28. The outgoing relations of an individual i in a knowledge base $\mathcal{K}$ is defined as:

$$r_{out}(i, \mathcal{K}) = \{R(i, j) | R(i, j) \in \mathcal{A}\}$$

The types of the individuals that are linked through these outgoing relations are defined through the function c_{out}:

$$c_{out}(i, \mathcal{K}) = \{C(j) | R(i, j) \in r_{out}(i, \mathcal{K}) \land C(j) \in \mathcal{A}\}$$

The combination of all the outgoing relations with the types of the linked individuals is then defined as rc_{out}

$$rc_{out}(i, \mathcal{K}) = r_{out}(i, \mathcal{K}) \cup c_{out}(i, \mathcal{K})$$

We can now define how a subset for a set of ABox axioms Ax can be calculated with respect to a materialized knowledge base $\mathcal{K}^{\infty}$. We denoted $ind(Ax)$ as the collection of individuals contained in Ax and $depth_{max}$ the TBox depth.

Definition 29. A subset $Sub(Ax, \mathcal{K}^{\infty}, depth_{max}) =$
$\{R_{out}(i, \mathcal{K}^{\infty}, depth_{max}) | i \in ind(Ax)\}$ with

$$\begin{aligned}R_{out}(i, \mathcal{K}, 0) =& rc_{out}(i, \mathcal{K}),\\ R_{out}(i, \mathcal{K}, depth) =& rc_{out}(i, \mathcal{K})\\ & \cup\{R_{out}\big(i', \mathcal{K}, depth - 1\big) | i' \in ind\big(r_{out}(i, \mathcal{K})\big)\}\end{aligned}$$

So the subset method of an ABox A_x based on $\mathcal{K}^{\infty}$ extracts recursively all relations, individuals and their types linked to the individuals in Ax that are also in $\mathcal{K}^{\infty}$. The recursion is dependent on the TBox depth. Note that to make the theory work in practice two special cases need to be incorporated:

- When in the latest step of the scenario (depth=0) there are transitive relations, the recursion should continue to follow these relations (and these relations only) until no more of the transitive relations are found.

- In each step, we store the ABox relations that have been followed before and make sure we do not follow previously visited relations. This is to prevent that the algorithm gets stuck in a loop.

These special cases were not included in the formalization in order to maintain clarity.

Example 19. (cont'd) We extend the TBox from Example 18 with the following axioms:

$$\begin{aligned}\mathcal{T}' = \mathcal{T} \cup \{&BehaviourRiskProfile \sqsubseteq RiskProfile\\ &MedicalRiskProfile \sqsubseteq RiskProfile\}\end{aligned}$$

We introduce the following ABox:

$$\begin{aligned}\mathcal{A}' = \{&Person(p1), hasRole(p1, r1), Role(r1),\\ &Hospital(h1), hasProfile(p1, m1),\\ &MedicalRiskProfile(m1), Detail(d1)\\ &hasDetails(r1, d1), isAdmittedTo(d1, h1),\\ &\ldots\\ &Person(p2), Person(p3), Person(p4),\\ &hasRole(p2, r2), hasRole(pr, r3), hasrole(p4, r4),\\ &StaffMember(r2), CareRole(r3),\\ &StaffMember(r4)\\ &\}\end{aligned}$$

Note that $\mathcal{A}'$ contains more axioms (indicated by '...'), but these were omitted for conciseness reasons. When we materialize the knowledge base, we can infer (among others) that $Patient(r1)$ and $RiskProfile(m1)$. Consider now the following update in the stream:

$$Ax = \{Call(c1), callMadeBy(c1, p1)\}$$

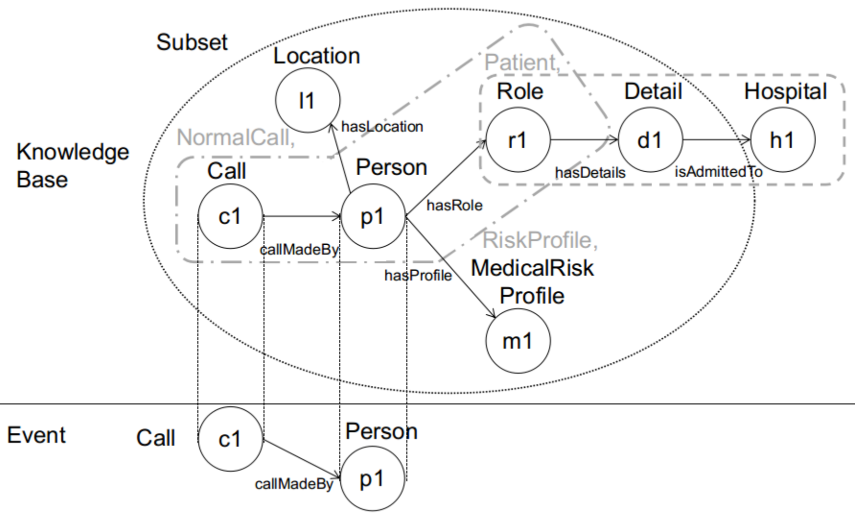


Figure 5.3: Visualization of the example illustrating that subset technique can extract a subset of data and still infer the types of the individuals in the event data due to the materialization of the knowledge base. The circle visualizes the data in the subset, the dotted rectangles depict the data necessary to infer that c1 is a NormalCall and r1 is a Patient.

If we calculate the subset for Ax in function of $\mathcal{T}'$ and $\mathcal{A}'$, we obtain the following axioms:

$$\begin{aligned}
&Sub(Ax, \mathcal{K}^\infty, 2)\\
&\quad = \{R_{out}(c1, \mathcal{K}^\infty, 2), R_{out}(p1, \mathcal{K}^\infty, 2)\}\\
&\quad = \{Call(c1), callMadeBy(c1, p1), Person(p1),\\
&\qquad hasRole(p1, r1), R_{out}(r1, \mathcal{K}^\infty, 1), R_{out}(m1, \mathcal{K}^\infty, 1)\}\\
&\quad = \{Call(c1), callMadeBy(c1, p1), Person(p1),\\
&\qquad hasRole(p1, r1), Role(r1), hasDetails(r1, d1),\\
&\qquad R_{out}(d1, \mathcal{K}^\infty, 0), hasProfile(p1, m1),\\
&\qquad MedicalProfile(m1), RiskProfile(m1)\}\\
&\quad = \{Call(c1), callMadeBy(c1, p1), Person(p1),\\
&\qquad hasRole(p1, r1), Role(r1), hasDetails(r1, d1),\\
&\qquad Detail(d1), hasProfile(p1, m1),\\
&\qquad MedicalProfile(m1), RiskProfile(m1)\}
\end{aligned}$$

Which is a subset and still allows us to calculate the fact that $c1$ is a $NormalCall$ and a $PriorityCall$.

The example is visualized in Figure 5.3 where we can see that we don't need all information in the knowledge base to infer the types of $c1$. To infer that the Call $c1$ is a NormalCall, we need

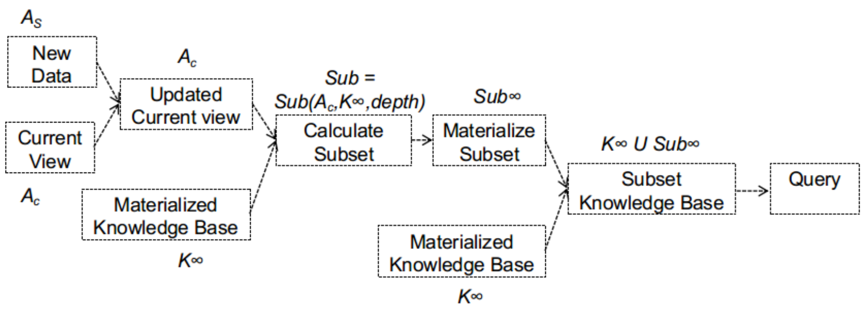


Figure 5.4: The different processing steps of the practical subset reasoning approach. With A_c the current view ($A_{current}$) on the stream A_S.

to know that $r1$ is a Patient. However, the subset does not contain all the data to infer that $r1$ is a Patient. Since we operate on a materialized knowledge base, we already inferred in a previous step that $r1$ is, in fact, a Patient. Since the inferred types are part of the subset, we can successfully infer that $c1$ is a NormalCall. Since the knowledge base is materialized, a smaller set of data can be extracted to infer the types of the data in Ax.

5.5.2 Subset Reasoning: a practical approach towards expressive Event-Based reasoning

The subsetting technique is very useful upon frequently changing data. We define an axiom streaming dataset as follows:

Definition 30. An axiom streaming dataset ADS is a set of the form $\{(A_S, U_p), ...\}$ with A_S a stream of events described as axioms and U_P its update policy, as defined in Section 5.4.

In an event-based environment such as the IoT, there are multiple streams to be considered since data from various sources needs to be combined. The calculated subsets for each of these streams should thus be combined with the knowledge base to allow querying of the integrated streams. The process of the various steps for one stream is depicted in Figure 5.4. First, we calculate the new current view through the selected policy update and based on the new view we extract a subset of data from the materialized knowledge base. The subset is then materialized and combined with the materialized knowledge base in the Subset knowledge base. The Subset knowledge base can then be queried.

Definition 31. A Subset Knowledge Base is the union of the materialized knowledge base and the materialized axioms sets from ADS: $\mathcal{K}^{\infty}_{ADS} = \bigcup A^{\infty}_S \cup \mathcal{K}^{\infty}$
with $A^{\infty}_S = \mathcal{M}(Sub(U_P(A_S current, A_S(t)), \mathcal{K}^{\infty}, depth, \mathcal{T}))$

This allows to combine the materialized views on the different streams with the static data in the knowledge base. For each A_S we use a subset of $\mathcal{K}^{\infty}$ to materialize $A_S current$. Due

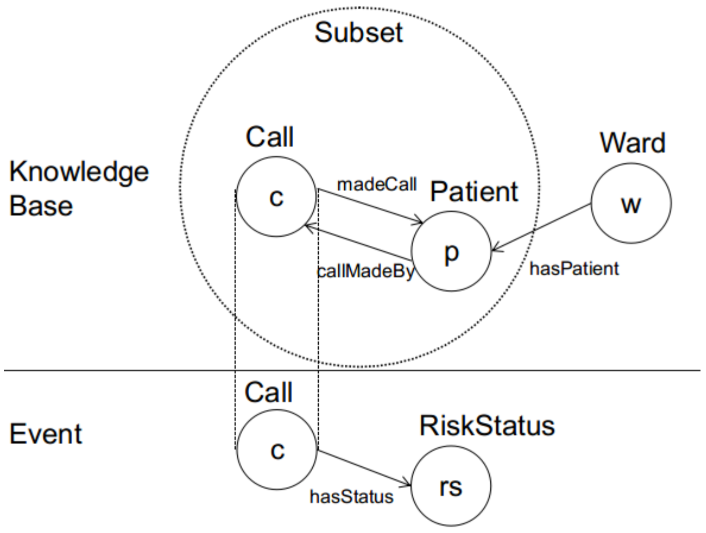


Figure 5.5: Visualization of the limitations of the subset technique when the events update parts of the knowledge base outside the subset.

to the monotonicity property, $\mathcal{K}^{\infty}$ does not need to be updated when data is removed from $A_{\mathcal{S}}current$ since the materialization of $A_{\mathcal{S}}current$ is maintained outside of $\mathcal{K}^{\infty}$. Additional information regarding $\mathcal{K}^{\infty}$ can be inferred in $A_{\mathcal{S}}current$ but it does not inflict $\mathcal{K}^{\infty}$ directly. Since the union of $\mathcal{K}^{\infty}$ and all the $A_{\mathcal{S}}current \in ADS$ are used for query answering, these results are taken into account. Note that when $A_{\mathcal{S}}current$ is updated, the previous materialization is removed and a new materialization based on the extracted subset is calculated. This methodology bypasses the difficulties attached to removals in an incremental reasoning approach [35, 36] (i.e. detecting which inferred facts should be removed when removing data).

5.5.3 Extension of Subset Reasoning

Since the subsetting technique is an approximation technique it is not always complete. It is however always sound. The technique focusses on the efficient materialization of the new events, however, updating the knowledge base according to the new events might sometimes be incomplete if these updates fall outside of the subset. We first introduce a pure fictional scenario to highlight the possible risks and afterwards we present a solution. Note that in the practical implementation of the nurse call system, we were never confronted with these limitations. Figure 5.5 visualizes these limitations. Let's consider the following TBox consisting of wards, patients and calls that might be at risk:

$$\begin{aligned} RiskWard &\equiv Ward \sqcup \exists hasPatient.RiskPatient \\ RiskPatient &\equiv Patient \sqcup \exists madeCall.RiskCall \\ RiskCall &\equiv Call \sqcup \exists hasStatus.RiskStatus \\ callMadeBy &\equiv madeCall^{-} \end{aligned}$$

Say that the knowledge base consists of the following ABox:

$$\begin{gathered} Ward(w), Patient(p), hasPatient(w,p), \\ Call(c), madeCall(p,c), callMadeBy(c,p) \end{gathered}$$

The stream that updates the call statuses A_S produces the following axioms that state that the call made by the patient has a risk status:

$$Call(c), hasStatus(c, rs), RiskStatus(rs)$$

Since the TBox depth is only one in this example, we extract the following subset:

$$\begin{aligned} &Call(c), hasStatus(c, rs), RiskStatus(rs), \\ &\qquad callMadeBy(c,p), Patient(p) \end{aligned}$$

Upon materialization of the subset based on the introduced TBox, we can infer that that call c is a RiskCall and that patient p is a RiskPatient.

$$RiskCall(c), RiskPatient(p)$$

However, since the ward w was not in the subset, we do not infer that w is a RiskWard.

$$RiskWard(w)$$

As a solution to this problem, we can gradually increase the subset in size when individuals at the edge of the subset have changed type. With the edge of the subset, we denote the individuals that are present in the subset but are not in A_S.

$$edge(A_S, \mathcal{K}^{\infty}, d) = \{c | c \in (ind(Sub(A_S, \mathcal{K}^{\infty}, d)) \setminus ind(A_S))\}$$

The set of individuals that changed type in the edge of the subset can be captured by the following function:

$$\begin{aligned} &edge_{Changes}(A_S, \mathcal{K}^{\infty}, d) = \\ &\qquad \{c | \exists c \in edge(A_S, \mathcal{K}^{\infty}, d) \wedge \exists C \in \mathcal{T} \\ &\qquad \wedge C(c) \in Sub^{\infty}(A_S, \mathcal{K}^{\infty}, d) \\ &\qquad \wedge C(c) \notin Sub(A_S, \mathcal{K}^{\infty}, d)\} \end{aligned}$$

With Sub^{∞} the materialization of the subset. In the case that the changes A_c in the edge (i.e. $A_c = edge_{Changes}(A_S, \mathcal{K}^{\infty}, depth_{max})$) is not empty, we extend the subset by adding the changes A_c to the ABox we want to calculate the subset upon:

$$Sub(A_S \cup A_c, \mathcal{K}^{\infty}, depth_{max})$$

We keep increasing the subset ABox, with the changes A_c, until there are no more individuals at the edge that have changed type, i.e. until $A_c = \emptyset$. By also including changed individuals in the edge in the calculation of the subset, individuals that should be influenced by these changes will also be correctly materialized.

Furthermore, when calculating the subset, we also take the incoming relations r_{in} and their types rc_{in} into account in this extension.

$$\begin{aligned} r_{in}(i, \mathcal{K}) =& \{R(j,i) | \exists R \in \mathcal{T} : R(j,i) \in \mathcal{A}\} \\ c_{in}(i, \mathcal{K}) =& \{C(j) | \exists R(j,i) \in r_{in}(i, \mathcal{K}) : C(j) \in \mathcal{A}\} \\ rc_{in}(i, \mathcal{K}) =& r_{in}(i, \mathcal{K}) \cup c_{in}(i, \mathcal{K}) \end{aligned}$$

When we redefine rc_{out} used in the subset extraction function, then these relations are also taken into account:

$$rc_{out}(i, \mathcal{K}) = r_{out}(i, \mathcal{K}) \cup c_{out}(i, \mathcal{K}) \cup rc_{in}(i, \mathcal{K})$$

Note that the incoming relations in the original subset approach are not necessary since there we only want to infer the types of the newly added data and its closest relations.

In the example, Patient p became a RiskPatient and p was in the edge of the subset. Therefore we add p to the arriving data and calculate the subset on their union. This results in the following subset:

$$\begin{gathered} Call(c), hasStatus(c, rs), RiskStatus(r, s), \\ callMadeBy(c, p), Patient(p), Ward(w), hasPatient(w, p) \end{gathered}$$

This subset is sufficient to infer that w is a RiskWard. We note that in most of our scenarios it is the knowledge base that influences the arriving events and its nearest relations instead of the other way around. Therefore, we propose the latter solution as an optional configuration as it needs to verify the types of the individuals at the edge and thus slightly slows down the process.

5.6 Realizing Subset Reasoning

In this section we describe how we practically implemented the subset reasoning approach[3]. Figure 5.6 visualizes the various steps.

For simplicity, we first assume that different streams of observations can be distinguished from each other. For example, in our use case, we have a location stream that transmits the

[3]The implementation itself is available on github.com/pbonte/SubsetReasoning

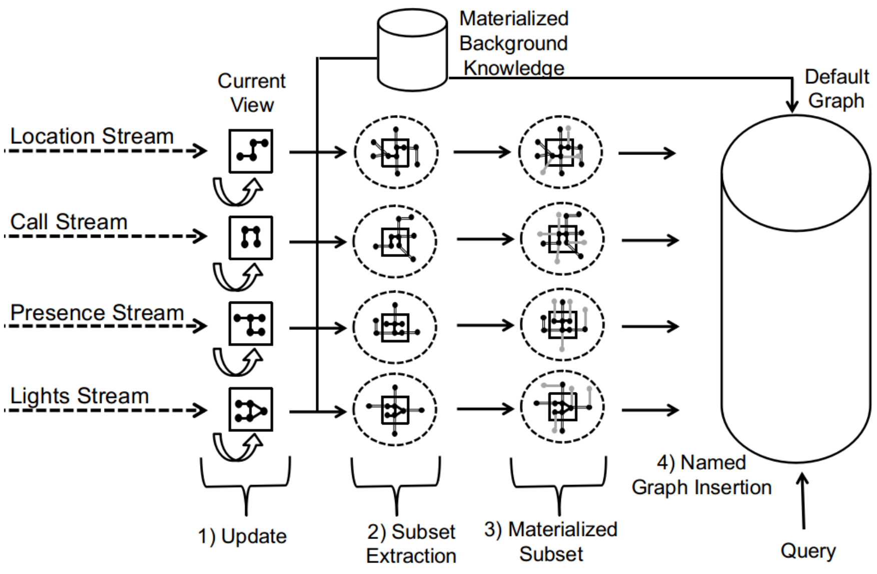


Figure 5.6: The different processing steps of the subset reasoning approach.

observations regarding the new locations of the personnel, a call stream regarding new calls and their updates, a light stream that captures the current status of lights and a presence stream that indicates whether the personnel logs in on the various devices. If these streams cannot be distinguished, additional filtering, such as defined in Section 5.6.5, can further split the streams up. The streams can be modeled as OWLAPI[4] axioms or Jena[5] statements. Furthermore, in the below explanation we assume that the background knowledge is already materialized in a preprocessing step and the queries are already defined, such that they can be continuously executed. If this is not the case, we also provide the mechanisms to materialize an ontology.

5.6.1 Updating

As each stream produces data, the stream updates are first fed through their own policy updater that calculates the current view on the stream according to the policies defined in Section 5.4. This is visualized in Figure 5.6 as step 1. The system maintains a current view for each stream and when new data arrives the current view is updated according to the policy. In Example 19 we already showed an update in the call stream.

$$\mathcal{A}_{\mathcal{S}} = \{Call(c1), callMadeBy(c1, p1)\}$$

[4]http://owlapi.sourceforge.net/
[5]https://jena.apache.org/

Since this is the beginning of a call, the current view will be empty and the whole update will be considered the new current view.

One can register a new stream by indicating the stream name and the policy update for that stream. When adding new data, the name of the stream should also be passed as an argument. This allows to extract the correct current view of the stream and update it according to the defined update policy.

5.6.2 Subset Extraction

Once the current view on the stream has been fixed, the subset can be calculated from all the individuals in the current view and the (materialized) background knowledge. This is done according to the description in Section 5.5 and visualized in Figure 5.6.2. We refer the reader to Example 19 for an example of the extracted subset based on the above defined current view.

5.6.3 Materializing Subset

Once the subset is extracted, the whole subset is materialized according to the used ontology TBox (Figure 5.6.3). An example of the materialization of the extracted subset can be found in Example 19. The reasoning is performed with the Hermit reasoner. However, other reasoners could easily be plugged in.

5.6.4 Named Graph Insertion

Each of the materialized subsets is then added as a named graph to an RDF store. The background knowledge details the default graph. Upon querying, the various graphs are merged together and the querying is done on the union of all the graphs. We utilized the jena Dataset to store the various graphs and to query them.

5.6.5 Fine Grained Filtering

As one can imagine, the performance of the system strongly depends on the size of the subset that needs to be materialized. It is possible that the arriving streams produce observations that could be further split up. For example, in our hospital setting, the location updates could come from multiple wards, while it is only necessary to combine updates from the same ward. Therefore we support finer grained filtering of the observations to split them up according to some parameter, e.g. a ward in the hospital. This is done by allowing additional SPARQL queries that select that parameter. To do this filtering, the stream observation needs to be combined with the materialized background knowledge.

Example 20. To filter each location update according to the ward it is produced in, we combine the produced data, containing the observation, with the knowledge base and execute the following query that selects the ward:

```
SELECT DISTINCT ?ward WHERE {
GRAPH :location {?s ?p ?o}
?s context:hasLocation ?loc.
?loc context:hasCentreCoordinate ?coord.
?coord context:hasZCoordinate ?ward.
}
```

The ward level is then used as an additional identifier to select the current view of the location stream for that specific ward.

5.6.6 Differences Traditional Reasoning Techniques

Looking back at the reasoning techniques introduced in Section 5.3.2, i.e. reasoning at query time, materialization and query rewriting, Subset reasoning is clearly a materialization approach. However, it differs in the way the materialization is calculated and maintained. In Figure 5.1 we have seen that data is typically just added to the ontology (step c), then the knowledge base is materialized (step d) and then the querying can be executed (step e). Our Subset reasoner takes a different approach where the updates to the ontology are stream specific and can overwrite and update data through the use of the update policies. Thus step c of Figure 5.1 aligns with Figure 5.6.1. The materialization of the runtime data is also specific for each stream and utilizes the subset extraction to minimize the data to reason upon. Thus step d of Figure 5.1 aligns with both Figure 5.5.2 and Figure 5.5.3. The querying itself requires an additional step, as first all the materializations of all the streams need to be combined with the materialized background knowledge. By extracting only a subset of data, the reasoning process can be speed up.

5.6.7 Subset Reasoning Configuration

The Subset reasoner can be configured utilizing the following parameters:

- The ontology: The ontology used for the reasoning process, this is typically the TBox.
- Static knowledge base: The static data that needs to be combined with the event data. There is an additional option to materialize this static data if this is not the case.
- Data Streams: The data streams that will produce event data.
- Update Policies: For each data stream, an update policy can be defined.
- Queries: Multiple queries can be registered, allowing to execute all the queries when new data arrives. There is also the option to execute specific queries.
- Subset size: The size of the subset can be set manually or can be automatically extracted from the registered ontology.

Table 5.1: Summary of the ACCIO ontology for different number of wards.

#Wards:	0	1	10	25	50	75	100
Axioms	2169	2942	10643	19879	41438	63399	85094
Logical Axioms	1225	1835	7893	15151	32115	49396	66467
Individuals	88	250	1894	3871	8467	13147	17771
Classes	282						
Object Properties	131						
Data Properties	37						
DL Expressivity	SHOIQ(D)						

When data arrives, it is added to the registered streams and the assigned update policy will update the current view on that stream. The subset calculation will extract the necessary data and then the event data can be materialized. Once the materializations are combined, the knowledge base can be queried.

5.7 Evaluation

This section evaluates both the performance of the proposed approach and the completeness. The evaluation was conducted on a 16 core Intel Xeon E5520 @ 2.27GHz CPU with 12GB of RAM running on Ubuntu 16.04.

5.7.1 Evaluation Set-up

To evaluate the performance and completeness of the proposed approach, we evaluate our Subset approach in a real-life use case (as described in Section 5.2) in Section 5.7.2 and through an existing benchmark in Section 5.7.3. To evaluate the performance of the system, in Section 5.7.2.1 and 5.7.3.1 we compare with reasoners with the same expressivity and for purely illustrative purposes we compare in Section 5.7.2.2 and 5.7.3.2 with less expressive reasoners, however, they are unable to infer all answers.

The reasoners with same expressivity consist of the Pellet reasoner, the Stardog triple store[6], and two versions of the Hermit reasoner, one where we always materialize the whole knowledge base with Hermit and then query it (further referred to as 'hermit') and one where the basic graph patterns are evaluated inside the query engine with OWL 2 DL reasoning [37] (further referred to as 'owlbgp').

The selected reasoners with a lesser expressivity are RDFox, TrOWL and the jena incremental rule engine inferring the fragment of the ontology that can be represented as rules. The latter was incorporated because the C-SPARQL RSP engine [10] allows the use any kind of rules within

[6]www.stardog.com

its query engine. We also incorporate the query rewriting method from StreamQR, i.e. we rewrote the queries for the $\mathcal{ELHIO}$ fragment of the ontology using the StreamQR rewritting techniques (further referred to as 'elhio'). We could not incorporate StreamQR directly, as it does not support static background data, such as the details regarding the staff members, hospital layout, etc.

We note that the results of these reasoners are not completely comparable as the expressivity of these approaches is lower than our Subset approach. These approaches could not produce all required results in the evaluated scenarios. In Section 5.7.2.3 and 5.7.3.3, we evaluate and discuss the correctness of the approaches.

5.7.2 Use-case Evaluation

We implemented the scenario from the use case introduced in Section 5.2. In each step of the scenario, an observation is sent and multiple queries are executed to determine the correct actions. To measure the scalability, we calculated how long the reasoning and querying took for each step in the scenario for a hospital ranging from 1 to 100 wards. We executed the scenario 35 times for each number of wards, dropped the first three and last 2 execution times and calculated the averages over the remaining 30 samples.

Table 5.1 shows the different amounts of axioms used for each number of wards and the complexity of the ontology. With 0 wards we denote the case where we only consider the minimal set of individuals that allow to run the scenario, namely three locations and three persons (one patient and two staff members) each with their properties. This makes it the smallest ABox the scenario can run on. Note that the TBox depth of the ontology is three.

5.7.2.1 Performance Evaluation Comparable Reasoners

In Figure 5.7, we compare the execution time for the reasoners with OWL2 DL expressivity and our Subset approach. Note that the y-axis is in logarithmic scale. Furthermore, for the other approaches we use the same update policies used in the Subset approach, but update on the whole knowledge base.

It is clear that the other reasoners with the same expressivity do not scale very well in our scenario. As the number of wards and thus the ABox data increases, all of the approaches, except the Subset approach, become unusable slow for real-time decision making.

Since it is hard to see the performance of the Subset approach, we plot each step of the scenario for each number of wards in Figure 5.8. We can clearly see that the size of the ABox has a small influence on the execution time. This is because the subset can efficiently extract the necessary data. When the subset would be affected by an increase in data that could be logically separated, e.g. the number of wards, we can use a more fine grained filtering, as described in Section 5.6.5, to separate the data in the subsets.

Note that in a preprocessing step, we need to materialize the static background knowledge for the Subset approach. This can be very time consuming for larger knowledge bases and is not taken into account in this comparison. However, since it only needs to be done once and the static knowledge typically does not change (often), it causes no real issues. In our use case, the static

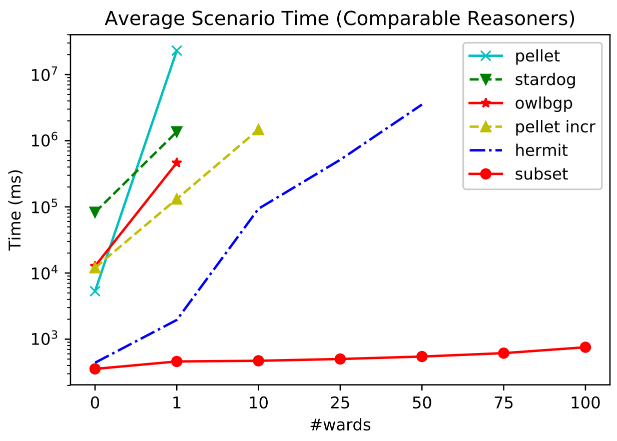


Figure 5.7: Comparison of the performance of various reasoners, with OWL2 DL expressivity, for increasing ABox data in the real-life use case.

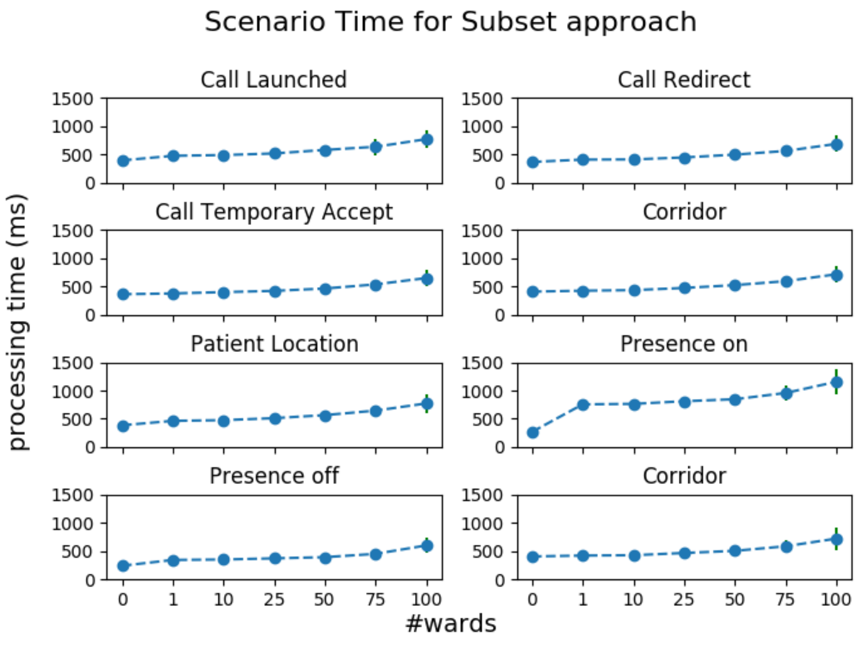


Figure 5.8: Performance of the Subset approach for increasing ABox data in the real-life use case.

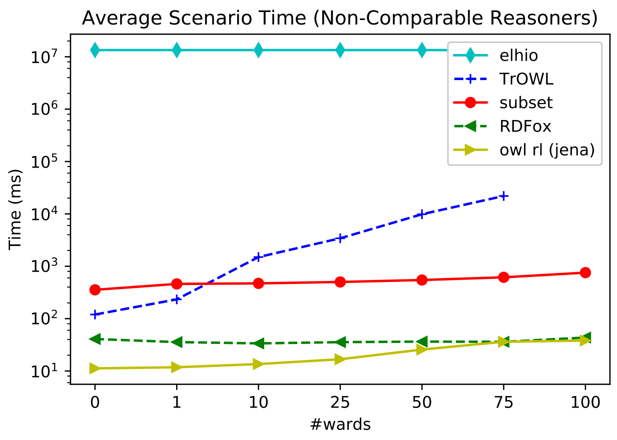


Figure 5.9: Comparison of the performance of various reasoners, with lower expressivity, for increasing ABox data in the real-life use case. Note that these lower expressive reasoner are not able execute the scenario correctly.

data contains the various locations in the hospital; the staff members and their capabilities; and the patients and their pathologies. Note that when the patients change too often, they can be modeled as a stream instead of static data.

5.7.2.2 Performance Evaluation Non-Comparable Reasoners

Figure 5.9 details the performance evaluation for the non-comparable reasoners. The less expressive reasoners (TrOWL, RDFox, owl rl (jena)) are more performant than the Subset approach. Even though TrOWL gets quickly caught up by the Subset approach when the ABox increases in size. The OWL RL approaches are faster than the Subset approach, which can be expected as they only infer a fragment of the ontology. The fragment has been selected in such a way that it can be efficiently computed. An interesting observation is that rewriting of the queries based on an expressive ontology results in queries with many unions (between 430 and 138158, with an average of 34276 unions). The execution of these queries is very expensive due to the large number of unions. Even though our Subset approach is slower, yet more expressive, we see a similar trend in terms of scalability as for the OWL2 RL reasoners.

We point out to the reader that the most important comparison is to compare the Subset approach with the two hermit approaches (hermit and owlbgp) since they use the same reasoner. We chose to use Hermit within our subset reasoner since it is the most complete. However, the technique is reasoner independent. This means that the subset technique can be ported to other reasoners as well.

Table 5.2: Correctness of the different approaches. The correctness is calculated as the number of correct class assertions. The last column details whether each engine was able to correctly run the scenario.

Engine	Correctness	Scenario Correct
Subset	100%	yes
Rdfox	98.3%	no
Jena OWL-RL	97.2%	no
TrOWL	99.6%	no
ELHIO	87.1%	no

5.7.2.3 Correctness Evaluation

Since the Subset approach (without the extension from Section 5.5.3) is an approximation technique, and we have included less expressive reasoners, we now discuss the correctness of each approach. To calculate the correctness we have run the scenario with the hermit reasoner and saved a materialized snapshot of the knowledge base in each step of the scenario. We view this as our baseline and compare for each approach the percentage of class assertions that have been correctly assigned to the individuals in the knowledge base.

Table 5.2 shows the correctness of each of the approaches. The last column indicates if the scenario was executed correctly. Our Subset approach is correct in this scenario, however none of the less expressive reasoners where able to execute the scenario correctly, even though the correctness level was already rather high. This is because the majority of concepts in the scenario do not always require expressive reasoning, however, to facility correct decision making in critical situation, e.g. correctly classify each call, expressive reasoning is necessary. Even though the frequency for the expressive reasoning might seem low, it is necessary to enable correct decision making as up to 40% of the scenario steps really depend on the results that require expressive reasoning (that cannot be inferred within the OWL2 RL fragment). The correctness results in Table 5.2 should thus be interpreted carefully as they indicate the correctness over the whole knowledge base. Even a small lack of correct results can lead to missing important events. This is clear for TrOWL, even with a correctness of 99.6% the scenario is not able to execute correctly.

In Table 5.3, we listed the OWL2 RL coverage for the IoT labeled ontologies in the Linked Open Vocabularies repository (lov.linkeddata.es)[7]. In the right column, the table shows the number of subclass definitions that each ontology contains. Note that a class equivalence can be seen as two subclass definitions. The table shows that many of these ontologies consist of definitions that are not fully covered by the OWL2 RL semantics. Even though the OWL2 RL reasoners are more performant, many existing ontologies still require higher expressive reasoning, such as OWL2 DL.

5.7.3 UOBM Evaluation

The University Ontology Benchmark (UOBM) [38] is an existing reasoning benchmark consisting of universities of various sizes containing professors, assistant professors, undergraduate students,

[7] We included all the ontologies that were accessible at the time of writing.

Table 5.3: OWL2 RL coverage of the IoT labeled ontologies in the Linked Open Vocabularies repository (lov.linkeddata.es).

Ontology Name	OWL2 RL Coverage	#subclass def
The NASA Air Traffic Management Ontology	92.7%	263
Climate and Forecast (CF) standard names parameter vocabulary	95.6%	405
Data Value Vocabulary (DaVe)	50%	6
Ontology Modeling for Intelligent Domotic Environments	63.1%	2963
FIESTA-IoT Ontology	79.6%	598
Home Weather	49.2%	313
Iot-lite ontology	100%	11
The Machine-to-Machine Measurement (M3) Lite Ontology	82.2%	544
MobiVoc: Open Mobility Vocabulary	100%	17
SAN (Semantic Actuator Network)	64.3%	42
SAREF: the Smart Appliances REFerence ontology	76.2%	248
The SEAS Device ontology	58.3%	36
The SEAS Forecasting ontology	45.4%	11
The SEAS Time Ontology.	84.6%	13
Sensor, Observation, Sample, and Actuator (SOSA) Ontology	100%	0
Semantic Sensor Network Ontology	68.8%	80
Semantic Sensor Network Ontology (old version)	80.9%	89
VoCaLS: A Vocabulary and Catalog for Linked Streams	100%	13

Table 5.4: Summary of the UOBM ontology for different number of university departments.

#Departments	**1**	**2**	**3**	**4**	**5**	**10**
Axioms	12405	26490	38849	52486	63949	133809
Logical Axioms	11943	25871	38075	51545	62870	131888
Individuals	1158	3141	4461	5937	7082	14063
Classes	113					
Object Properties	35					
Data Properties	9					
DL Expressivity	SHOIN(D)					

students, courses, publication, etc. The benchmark comes with four queries requiring expressive reasoning:

- **Q1:** Retrieve all women students. UOBM defines a woman college as: $WomanCollege \equiv School \wedge \forall hasStudent.Student \wedge \forall hasStudent.(\neg Man)$. Thus as a school where the students are not of the gender man. When a student is attending a woman college, the reasoner can infer that the student is female as Man and Women are defined as disjoint classes.

- **Q2:** Retrieve all people with many hobbies. UOBM defines people with many hobbies as: $PeopleWithManyHobbies \equiv \geq 3 \quad like.\top$. They thus should like at least three things.

- **Q3:** Retrieve all people who love sports. UOBM defines people who like sports as: $SportsLover \equiv \exists like.Sports$. This means that there should exist a sport that the person likes.

- **Q4:** Retrieve all people with at least one hobby. People with a hobby are defined in UOBM as: $PeopleWithHobby \equiv person \wedge \geq 1 \quad like.\top$. This means that the person should like at least one thing.

It is clear that the UOBM defines some complex concepts. However, compared to the ACCIO ontology used in Section 5.7.2, UOBM has a depth of one. UOBM is by default a static benchmark, so we extended the benchmark such that has a streaming characteristic. This is done by allowing students to join a college over the year. This means that we extracted the generated students from the benchmark and stream them together with the courses they take, the college they attend, their interests and friends, etc. We have evaluated the average performance of processing the students stream over a university with 1, 2, 3, 4, 5 and 10 departments. Table 5.4 describes this in detail. As each student typically changes friends, courses and interests over the course of time, we have modeled each student and its updates as a distinct stream.

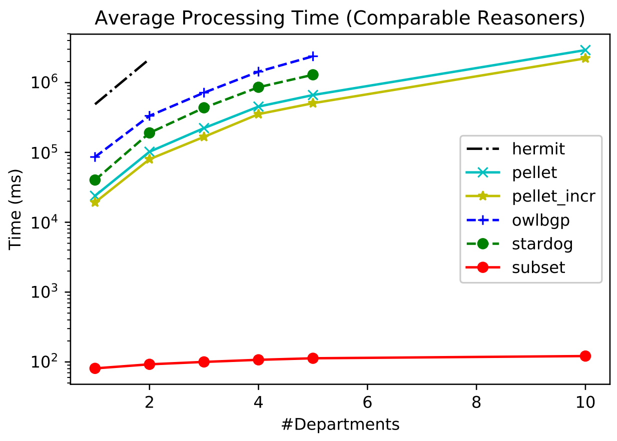


Figure 5.10: Comparison of the performance of various reasoners, with OWL2 DL expressivity, for increasing ABox data on the UOBM benchmark.

5.7.3.1 Performance Evaluation Comparable Reasoners

In Figure 5.10, we compare the average time to process a new student for the reasoners with OWL2 DL expressivity and our Subset approach. Note that the y-axis is in logarithmic scale. It is clear that the other approaches do not scale well.

5.7.3.2 Performance Evaluation non-Comparable Reasoners

In Figure 5.11 we show the comparison with reasoners with lower expressivity. It is clear that they are typically faster than our approach. However, as we will see in the completeness evaluation, in Section 5.7.3.3, these approaches are fast in this scenario but are unable to infer a complete answer set. We even see that the rewriting approach, i.e. elhio, becomes slower as well due to the many unions in the rewritten queries. Furthermore, our Subset approach is faster than TrOWL.

5.7.3.3 Completeness Evaluation

Figure 5.12 shows the correctness of the approaches in the UOBM scenario in terms of the completeness of the given query answers. For each query, we evaluate the percentage of correctly derived results. It is clear that only HermiT, Pellet, OWLBGP and our Subset approach are able to provide correct answers. TrOWL is unable to provide correct answers to the first two queries. The other reasoners fail to make most of the derivations. Even though these reasoners are fast, they are incomplete in scenarios where expressive reasoning is required.

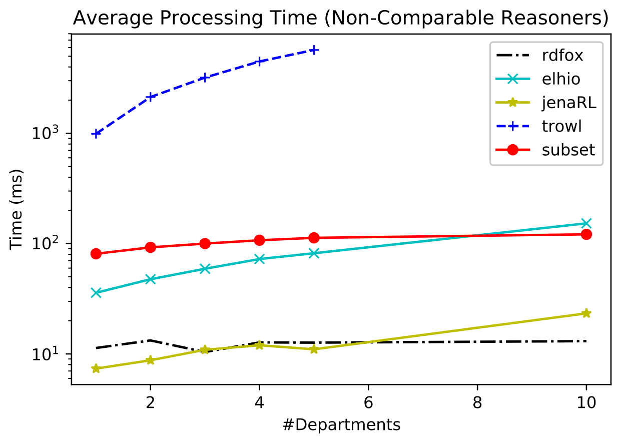


Figure 5.11: Comparison of the performance of various reasoners, with lower expressivity, for increasing ABox data on the UOBM benchmark.

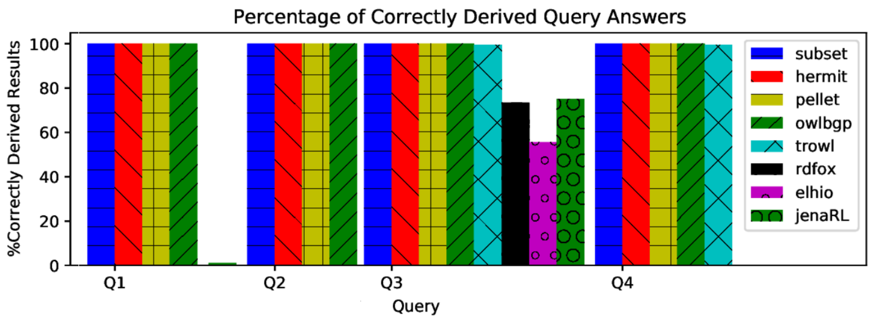


Figure 5.12: Correctness comparison UOBM evaluation for different reasoners.

5.8 Discussion

In this section, we discuss how our solutions tackles the set objectives, how Subset reasoning compares to the related work, how the evaluation results should be interpreted and the drawbacks and future work directions for Subset reasoning.

5.8.1 Objectives Discussion

Looking back at the Objectives set in Section 5.1.3, we can now discuss how our Subset reasoning approach tackles the various objectives:

1. Heterogeneous data: Our Subset reasoner can combine various heterogeneous sources by utilizing a common semantic model, i.e., an ontology.
2. Event data: The Subset approach can handle the addition and removal of event data through the use of its update policies that allows the system to have a clear view on the current context. Furthermore, the Subset approach utilizes an approximated extraction method to minimize the data to reason upon, decreasing reasoning time. Fast reasoning times are necessary in event-based system such that the system stays reactive and real-time decisions can be made.
3. Large knowledge bases: Many domains have large knowledge bases that need to be combined with the generated event data. However, reasoning over these large knowledge bases in combination with the changing event data might become slow. The Subset reasoner tackles this problem by exploiting the fact that the large knowledge bases typically consist of static data that does not change very often. Therefore, this data is materialized and due to the monotonicity property, adding data will not lead to the removal of facts in the static data. The Subset reasoner extract data from the materialized knowledge base in order to compose a minimal subset to reason upon.
4. Expressive reasoning : Expressive reasoning is necessary to interpret many complex domains. The Subset approach supports OWL2 DL reasoning. It is able to perform this highly expressive reasoning over event data by computing a subset of data to reason upon.

5.8.2 Related Work Comparison

Table 5.5 summarizes the related work and how they relate to our Subset approach. As we have seen in the evaluation in Section 5.7, traditional reasoners such as Pellet and Hermit are very expressive, however, they have problems processing event data in a timely fashion. They become very slow when the data increases and do not have any mechanisms to process event data besides adding and removing of data. The Pagoda reasoner tries to solve the performance problem by combining the Hermit reasoner with a less expressive OWL2 RL reasoner. However, Pagoda cannot be used with event data as it does not allow the addition and removal of data. RDFox is the

Table 5.5: Comparisson of the related work to the Subset approach. (With W = windowing and UP = update policies.)

	Add/ Remove	Event Data	Expressive Reasoning	Large KB	View
Traditional Reasoners	x	/	x (OWL2 DL)	/	/
Pagoda	/	/	x (OWL2 DL)	x	/
RDFox	x	/	/ (OWL2 RL)	x	/
TrOWL	x	x	x (SHIQ)	/	/
RSP	x	x	/ (RDFS)	x	x (W)
StreamQR	x	x	/ (ELHIO)	/	x (W)
SubSet	**x**	**x**	**x** (OWL2 DL)	**x**	**x** (UP)

fasted OWL2 RL reasoner currently available. It is very performant within its OWL2 RL fragment, however, as we have seen in Section 5.7.2.3 and 5.7.3.2, many use cases and ontologies require expressive OWL2 DL reasoning to correctly interpret the domain knowledge. Even though RDFox allows efficient addition and removal of data, it does not provide any mechanism such as windowing or update policies to process the event data and keep a view on the current context. TrOWL provides almost OWL2 DL expressivity, however, in Section 5.7.3.2 we saw that even some unsupported constructions lead to incomplete answers. Furthermore, in the evaluation we have seen that TrOWL becomes rather slow when the background knowledge increases. RSP engines provide all the mechanisms to handle event data, however, due to their low expressivity, they cannot interpret complex domains. Query rewriting techniques such as provided by StreamQR can only inject a small part of their expressivity in the query and tend to become slow when the static data increases. Our Subset reasoner can perform expressive reasoning over event data, in combination with large knowledge bases by approximating a subset of data to reason upon. It is the only approach that fulfills all the set requirements.

5.8.3 Evaluation Discussion

In section 5.7.2.1 we have shown that the Subset approach is a good candidate to make expressive reasoners more scalable and applicable in real-time scenarios were expressive reasoning is necessary. From section 5.7.2.2 and 5.7.3.2 it is clear that OWL2 RL reasoners, which are less expressive, are more efficient. However, as we have shown in Section 5.7.3.3, these techniques are incomplete when expressive reasoning is required. Furthermore, in section 5.7.2.3 we have shown that only a small fragment of the IoT labeled ontologies are completely covered by the OWL2 RL fragment. This implies that ontology designers either do not take efficiency into consideration when designing ontologies or ontology designers feel that the OWL2 fragments are lacking expressivity to model their domains.

We agree that some of the OWL2 DL definitions that cannot be represented in OWL2 RL, such as quantified number restrictions (e.g. there should be exactly one person present in a certain room), could (in some form) be represented as standard existential quantifiers (e.g. there should be a person present in a certain room). However, such adaptations should be conducted very carefully as the semantics of the concepts are changing.

Thus, as expressive ontologies are still being designed and it is not trivial to convert expressive ontologies to their lesser expressive variants, techniques to efficiently reason upon expressive ontologies are still needed.

5.8.4 Subset Reasoning Limitations & Future Work Directions

A first limitation of the subset approach is that the data streams are handled separately, this means that if data from stream A has influence on data from stream B, that this will not be detected. This is because each stream is handled by its own update policy. This can be bypassed by combining various streams in the same update policy, however, this will have a negative impact on the performance. A second limitation is the fact that the update policies are currently not time based. This means that one has to explicitly remove facts in order to deprecate data. Time-based windows allow facts to be removed after a certain period of time, something which is not supported yet by our update policies. A third limitation is that our approach currently extracts ABox data only and takes the complete TBox each time into account, when performing reasoning.

In future work, we wish to further extend the subset approach and provide mechanisms to detect influences between streams. This would allow to efficiently process various streams that have influences on each other. Furthermore, we wish to further extend the update policies such that facts can be automatically removed after a certain period of time. This will eliminate the need to explicitly remove facts. This would also allow us to integrate aggregation mechanisms, as supported by RSP engines. We also wish to investigate mechanisms to efficiently minimize the TBox utilized in the reasoning process to further increase the reasoning performance.

5.8.5 Problems of Using Windowing

In this Section, we describe the implications of using windowing[8], instead of the update policies, has on the scenario, as visualized in Figure 5.13. Say we have two events (event E_1 and event E_2), E_1 can be inferred through reasoning as a NormallCall, when adding the information described in E_2, that says that the call has a medical reason, to E_1, the call can now be inferred as a MedicalCall. These kinds of dependencies occur throughout the use case. Different queries are executed in a certain order on these events and when one of the queries triggers, a certain action is executed. When E_1 arrives, describing a new call has been launched by a certain patient, query Q_1 that selects staff members for new calls (status active) is executed and notifies the selected staff member. This is step 1 (Call Launched) in the scenario. In step 2 (Call Redirect) the staff member is busy and redirects the call and indicates that the call has a medical reason. This is described in E_2. Normally, query Q_2 should select a new staff member for medical calls that have been redirected.

[8] There exists various entailment regimes [1], we explain graph-level entailment as it is currently the most commonly used.

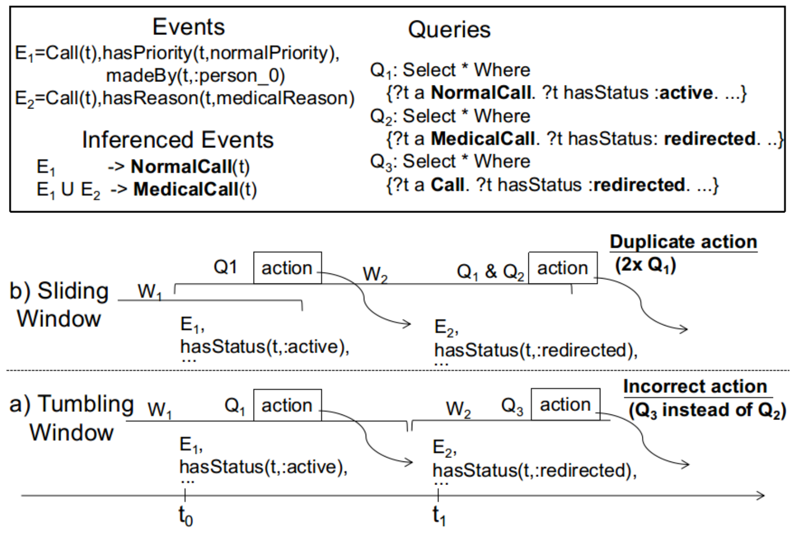

Events
E1=Call(t),hasPriority(t,normalPriority), madeBy(t,:person_0)
E2=Call(t),hasReason(t,medicalReason)
Inferenced Events
E1 -> NormalCall(t)
E1 U E2 -> MedicalCall(t)
Queries
Q1: Select * Where {?t a NormalCall. ?t hasStatus :active. ...}
Q2: Select * Where {?t a MedicalCall. ?t hasStatus: redirected. ..}
Q3: Select * Where {?t a Call. ?t hasStatus :redirected. ...}
b) Sliding Window
W1
Q1
action
W2
Q1 & Q2
action
Duplicate action (2x Q1)
E1, hasStatus(t,:active), ...
E2, hasStatus(t,:redirected), ...
a) Tumbling Window
W1
Q1
action
W2
Q3
action
Incorrect action (Q3 instead of Q2)
E1, hasStatus(t,:active), ...
E2, hasStatus(t,:redirected), ...
t0
t1

Figure 5.13: Complications of using windowing. The figure shows a typical dependency between events in the scenario. As reasoning is involved, the events need to be combined to infer the correct types that are used in the queries. However simply combining them, as in the sliding window, results in duplicate triggering queries (and thus duplicate actions). When the events are in different windows, incorrect actions are taken because the inferred information is missing.

However, when using windowing, the dependencies between the events can be lost. Figure 5.13 shows both the problems for both tumbling and sliding windows. When using a tumbling window (Figure 5.13 a), Q_1 is executed upon E_1 and as a result E_2 is sent. However, in the next window W_2, the information regarding the call described in E_1 is lost and now it is unknown that the call was a NormalCall and therefore Q_2 will not be executed. Query Q_3 that redirects any kind of call will match and an incorrect action will be taken. When using a sliding window (Figure 5.13 b), both events can be comprised in the same window, however, as the call keeps its initial status (active) query Q_1 will trigger for a second time and duplicate actions will be taken. This problem occurs due to the lack of an update policy that should overwrite the call status.

5.9 Conclusion and Future Work

In this paper, we presented a technique that allows to bridge the gap between volatile data and expressive reasoning. Our technique maintains a materialized view on the ontology ABox and uses a subset approximation to efficiently update the materialized view. To define how these updates should happen, we introduced the notion of update policies. The subsetting enables a scalable system even with highly increasing ABoxes.

In our future work, we will investigate the possibility to perform aggregations, as supported by RSP, within the update policies. Furthermore, we will investigate the integration of module extraction techniques [21] to, besides minimizing the ABox, minimize the used TBox within the reasoning process. Our technique achieves a speed-up of up 10 times for small ABoxes and more than 1000 for larger ones.

We show that the subsetting technique is a valid tool to enable expressive reasoning in time critical scenarios, allowing time-critical systems to make complex decisions based on expressive reasoning solutions.

Acknowledgment

This research was funded by the IWT O&O ORCA and Televic Healthcare.

Authors' contributions

PB carried out the study, developed the algorithm, ran the experiments and drafted the manuscript. FO and FDT supervised the study, participated in its design and coordination and helped to draft the manuscript. All authors read and approved the final manuscript.

References

[1] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein, "Stream reasoning: A survey and outlook," Data Science, no. Preprint, pp. 1–24, 2017.

[2] A. Margara, J. Urbani, F. Van Harmelen, and H. Bal, "Streaming the web: Reasoning over dynamic data," Web Semantics: Science, Services and Agents on the World Wide Web, vol. 25, pp. 24–44, 2014.

[3] M. I. Ali, N. Ono, M. Kaysar, Z. U. Shamszaman, T.-L. Pham, F. Gao, K. Griffin, and A. Mileo, "Real-time data analytics and event detection for iot-enabled communication systems," Web Semantics: Science, Services and Agents on the World Wide Web, vol. 42, pp. 19–37, 2017.

[4] F. Ongenae, P. Duysburgh, N. Sulmon, M. Verstraete, L. Bleumers, S. De Zutter, S. Verstichel, A. Ackaert, A. Jacobs, and F. De Turck, "An ontology co-design method for the co-creation of a continuous care ontology," Applied Ontology, vol. 9, no. 1, pp. 27–64, 2014.

[5] S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano, "Semantic Integration of Heterogeneous Information Sources Using a Knowledge-Based System," Data & Knowledge Engineering, vol. 36, pp. 215–249, 2001.

[6] T. Strang and C. Linnhoff-Popien, "A context modeling survey," in Workshop on advanced context modelling, reasoning and management, UbiComp, vol. 4, 2004, pp. 34–41.

[7] U. Hustadt, B. Motik, and U. Sattler, "Data complexity of reasoning in very expressive description logics," in IJCAI, vol. 5, 2005, pp. 466–471.

[8] L. Al-Jadir, C. Parent, and S. Spaccapietra, "Reasoning with large ontologies stored in relational databases: The OntoMinD approach," Data & Knowledge Engineering, vol. 69, no. 11, pp. 1158–1180, 2010.

[9] A. Hogan, A. Harth, and A. Polleres, "Saor: Authoritative reasoning for the web," The Semantic Web, pp. 76–90, 2008.

[10] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying RDF streams with C-SPARQL," SIGMOD Record, 2010.

[11] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth, A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 370–388.

[12] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "EP-SPARQL: a unified language for event processing and stream reasoning," 2011, pp. 635–644.

[13] R. Shearer, B. Motik, and I. Horrocks, "HermiT: A Highly-Efficient OWL Reasoner." in OWLED, vol. 432, 2008, p. 91.

[14] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee, “Rdfox: A highly-scalable RDF store,” in ISWC 2015 , Proceedings, Part II, 2015, pp. 3–20.

[15] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen, “Towards expressive stream reasoning,” in Semantic Challenges in Sensor Networks, 24.01. - 29.01.2010, 2010.

[16] E. Della Valle, S. Schlobach, M. Krötzsch, A. Bozzon, S. Ceri, and I. Horrocks, “Order matters! harnessing a world of orderings for reasoning over massive data,” Semantic Web, vol. 4, no. 2, pp. 219–231, 2013.

[17] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, “Pellet: A practical OWL-DL reasoner,” Web Semantics: science, services and agents on the World Wide Web, vol. 5, no. 2, pp. 51–53, 2007.

[18] Y. Zhou, B. Cuenca Grau, Y. Nenov, M. Kaminski, and I. Horrocks, “Pagoda: Pay-as-you-go ontology query answering using a datalog reasoner,” Journal of Artificial Intelligence Research, vol. 54, pp. 309–367, 2015.

[19] E. Thomas, J. Z. Pan, and Y. Ren, “TrOWL: Tractable OWL 2 reasoning infrastructure,” in Extended Semantic Web Conference. Springer, 2010, pp. 431–435.

[20] J.-P. Calbimonte, J. Mora, and O. Corcho, “Query rewriting in rdf stream processing,” in International Semantic Web Conference. Springer, 2016, pp. 486–502.

[21] A. A. Romero, M. Kaminski, B. C. Grau, and I. Horrocks, “Module extraction in expressive ontology languages via datalog reasoning,” J. Artif. Int. Res., vol. 55, no. 1, pp. 499–564, Jan. 2016.

[22] A. Schlicht and H. Stuckenschmidt, “Criteria-based partitioning of large ontologies,” in Proceedings of the 4th international conference on Knowledge Capture. ACM, 2007, pp. 171–172.

[23] C. Anagnostopoulos and S. Hadjiefthymiades, “Enhancing situation-aware systems through imprecise reasoning,” IEEE Transactions on Mobile Computing, vol. 7, no. 10, pp. 1153–1168, 2008.

[24] M. Hassanalieragh, A. Page, T. Soyata, G. Sharma, M. Aktas, G. Mateos, B. Kantarci, and S. Andreescu, “Health monitoring and management using Internet-of-Things (IoT) sensing with cloud-based processing: Opportunities and challenges,” in Services Computing (SCC), 2015 IEEE International Conference on. IEEE, 2015, pp. 285–292.

[25] D. Macagnano, G. Destino, and G. Abreu, “Indoor positioning: A key enabling technology for IoT applications,” in Internet of Things (WF-IoT), 2014 IEEE World Forum on. IEEE, 2014, pp. 117–118.

[26] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, “Owl 2 web ontology language primer,” W3C recommendation, vol. 27, no. 1, p. 123, 2009.

[27] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider, and D. Nardi, The description logic handbook: Theory, implementation and applications. Cambridge university press, 2003.

[28] R. Kontchakov and M. Zakharyaschev, "An introduction to description logics and query rewriting," in Reasoning Web International Summer School. Springer, 2014, pp. 195–244.

[29] I. Horrocks, O. Kutz, and U. Sattler, "The even more irresistible sroiq." Kr, vol. 6, pp. 57–67, 2006.

[30] I. Horrocks, U. Sattler, and S. Tobies, "Practical reasoning for expressive description logics," in International Conference on Logic for Programming Artificial Intelligence and Reasoning. Springer, 1999, pp. 161–180.

[31] "A tableau decision procedure for SHOIQ, author=Horrocks, Ian and Sattler, Ulrike, journal=Journal of automated reasoning, volume=39, number=3, pages=249–276, year=2007, publisher=Springer."

[32] D. Tsarkov and I. Horrocks, "Fact++ description logic reasoner: System description," in International Joint Conference on Automated Reasoning. Springer, 2006, pp. 292–297.

[33] V. Haarslev, K. Hidde, R. Möller, and M. Wessel, "The racerpro knowledge representation and reasoning system," Semantic Web, vol. 3, no. 3, pp. 267–277, 2012.

[34] E. A. Emerson, "Temporal and modal logic." Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), vol. 995, no. 1072, p. 5, 1990.

[35] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Incremental reasoning on streams and rich background knowledge," in Extended Semantic Web Conference. Springer, 2010, pp. 1–15.

[36] B. Motik, Y. Nenov, R. E. F. Piro, and I. Horrocks, "Incremental update of datalog materialisation: the backward/forward algorithm." in AAAI, 2015, pp. 1560–1568.

[37] I. Kollia, B. Glimm, and I. Horrocks, "Sparql query answering over owl ontologies," in Extended Semantic Web Conference. Springer, 2011, pp. 382–396.

[38] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu, "Towards a complete owl ontology benchmark," in European Semantic Web Conference. Springer, 2006, pp. 125–139.

6 Conclusion and Future Work Perspectives

"A conclusion is the place where you got tired of thinking."

–Martin H. Fischer (1879, 1962)

In IoT settings, data results from various heterogeneous sources. To enable meaningful decision making, this data needs to be combined, and background and domain knowledge should be incorporated. Ontologies serve as a common formal model, allowing the integration of various heterogeneous data sources. Furthermore, ontologies allow to describe the domain through the use of formal descriptions. Reasoning techniques can then be employed to interpret the domain and infer implicit facts regarding the data. The more detailed the domain description, the higher the expressivity the reasoner should possess. However, expressive reasoning techniques are very complex. There is still a mismatch between the rate at which data is produced in IoT settings and the throughput of existing expressive reasoning techniques.

This dissertation proposed several techniques to enable reactive question answering through expressive reasoning over IoT data streams. Firstly, a semantic publish/subscribe platform is introduced, allowing IoT services to subscribe to high-level ontological concepts. The platform employs reasoning to enable fine-grained data access. To enable expressive reasoning over volatile data, a layered cascading reasoning approach was introduced. The lowest layers employ low complexity processing techniques that allow to process large volumes of data. Going up in the layers, the amount of data decreases as each layer selects only the relevant parts and the complexity of processing increases. This allows to employ expressive and temporal reasoning techniques at the top layers. The performance of the lower layers that employ RSP solutions have been further optimized, allowing to efficiently perform hierarchical reasoning over volatile data streams. Lastly, an approximation technique is introduced, allowing to perform expressive reasoning over a large amount of static knowledge.

6.1 Review of the Research Questions

Section 1.4 proposed four research questions, their outcome can now be evaluated. To allow IoT services to consume the heterogeneous IoT data, I proposed Research Question 1:

"How can we provide IoT services fine-grained access to IoT data in a flexible manner?"

As IoT data is typically heterogeneous, a common semantics is required to enable interoperability. Ontologies serve as an ideal tool to integrate various data sources. However, data is often produced at different levels of granularity, requiring services to investigate the data sources in order to know what level of detail the produced data contains, so that they can subscribe to the correct data. The use of a reasoning-enabled publish/subscribe system allows to abstract these different levels of granularity according to the defined ontology. Compared to the state-of-the-art, our proposed solution enables expressive reasoning to perform intelligent filtering on IoT data. This allows services to subscribe to higher level concepts, without the need to worry about the lower level details at the data level. Furthermore, the platform is user-friendly, allowing services to subscribe to their data of interest in a declarative way. Compared to the state-of-the-art, our platform is fully decoupled, data-driven and the events are processed by the publish/subscribe system in a stateless fashion, allowing scalability. Furthermore, as services can filter the data very precisely, they only need to process a small subset of data, which improves performance.

This allows to validate Hypothesis 1: "Using an ontology-enabled publish/subscribe platform will allow semantic and flexible service subscription".

As there is still a mismatch between the rate data is produced in IoT settings and performance of current expressive reasoning techniques, I proposed Research Question 2:

"Can expressive reasoning be performed over highly volatile data streams?"

The introduction of a layered cascading reasoning approach allows data to be processed at different levels of update frequency and complexity of processing. At the lowest layers, data is processed at a low complexity of processing that can process large amounts of data. These lower layers can select possible relevant events from the volatile data streams, so that only a selection of data needs to be processed by the more complex layers, such as the expressive reasoning layers. Our cascading approach is the first approach that realizes the Cascading Reasoning vision, combining RSP to filter the relevant parts of the RDF stream, DL reasoning to abstract the data, and CEP to detect temporal patterns defined on these abstractions. Furthermore, we provide a unifying query language that tightly couples DL definitions and CEP patterns. The performance of the cascading approach is dependent on the selection rate, i.e. the rate of events that are selected by the lower layers. When the selection rate is high, and most events in the data streams are forwarded to the next layer, the performance of the cascading approach will be low, because the complex layers still need to process large amounts of data. However, when the selection rate is low, only a small percentage of events is selected and the performance increases significantly. We show that our approach can process data streams of up to hundreds of events per second, while performing expressive and temporal reasoning.

This allows to validate Hypothesis 2: "Using a cascading reasoning system will improve the efficiency of expressive OWL 2 DL reasoning over volatile data streams, enabling to process up to hundreds of events per second".

As the layered approach depends on the performance of the lower layers to handle large amounts of data and each layer can be more selective by employing reasoning capabilities, I proposed Research Question 3:

"Can RSP engines efficiently reason over highly volatile data streams?"

Three main approaches to perform reasoning over data streams exist, i.e. materialization, goal driven reasoning, and query rewriting. However, each of these approaches has some drawbacks. Materialization approaches infer many irrelevant facts as they infer all possible facts in the knowledge base. Goal driven approaches often redo the same work and query rewriting approaches end up in large and complex queries. Information Flow Processors often support hierarchies of events as part of their language features. They employ a hierarchical encoding of events, something that has not been exploited to perform reasoning over RDF data. To perform efficient reasoning over volatile data streams, we focus on hierarchical reasoning, i.e. subclass/subproperty reasoning. Employing hierarchical reasoning already significantly simplifies the query definition process and allows to target the data granularity problem on a hierarchical basis. We have compared our hierarchical reasoning approach with the state-of-the-art in a thorough evaluation and shown that our approach can perform hierarchical reasoning in constant time. Furthermore, our approach is at least twice as fast as the state-of-the-art and employs a minimal memory footprint. Compared to the state-of-the-art, our approach has the highest throughput of 30k triples/s compared to an average throughput of 5k triples/s when considering the various engines in the state-of-the-art. Our approach also has the lowest memory footprint, i.e. 1.2 Gbytes compared to an average of 4Gbytes considering the state-of-the-art. The latter is important when deploying the RSP engines as part of a cascading approach at the edge, i.e. close to the source and on resource constrained hardware. These findings allow to validate Hypothesis 3: " Using a hierarchical encoding of concepts will improve the throughput while performing hierarchical reasoning with at least a factor two and maintaining a minimal memory footprint, compared to the state-of-the-art".

IoT services often need to combine the selected events from the data streams with large amounts of static knowledge, however, reasoning over these large knowledge bases is typically slow, therefore I proposed Research Question 4:

“Can expressive reasoning over event data that needs to be combined with large static knowledge bases be employed in time-critical use cases?”

A cascading reasoning approach can select the events from large data streams, so that the more complex layers have to process only a selection of the data. However, when dealing with a large amount of static knowledge, special techniques need to be employed to perform expressive reasoning over event data that needs to be combined with large static knowledge bases. As incremental reasoning approaches cannot be applied to the required expressivity of reasoning, the only option is to minimize the amount of data to reason upon. This can either be the ABox or TBox. We focus on minimizing the ABox, as minimizing the TBox might still run into performance problems when the corresponding ABox keeps growing and the TBox cannot be further minimized. We propose a solution that approximates a subset of ABox data to reason upon, so that the event data can be correctly materialized. We have shown that our approach is up to 10 times faster for small datasets of up to 2,000 statements and up to more than 1,000 times faster for larger datasets up to 80,000 statements, compared to the state-of-the-art. This validates Hypothesis 4: “Using an approximation technique that extracts a subset of data to reason upon, we can speed up the expressive OWL 2 DL reasoning process at least 10 times, compared to the state-of-the-art”

6.2 Value & Impact for the IoT Domain

Looking back at the different challenges that accompany the processing of IoT data introduced in Chapter 1, we can now explain how the research in this dissertation impacts the IoT domain.

As IoT applications typically process data resulting from many different devices, with their own data format and encoding, there should be a way to abstract this heterogeneous data. Otherwise, data from different data sources cannot be integrated, eliminating interoperability. A common model is important in the IoT, as the number of involved devices is expected to become extremely high, resulting in challenges on how to represent, interconnect and search the produced IoT data [1]. We employ a common semantics through the use of an ontological model, enforcing different data sources to adhere to the same model. This means that even though two data sources have different data formats, their data can be combined and used to obtain advanced insights.

However, employing a common model might not be enough to ensure interoperability. As different data producers are involved in the IoT, it is not possible to ensure well-defined and agreed-upon content syntax [2]. Even if they employ a common model data could still be produced at different levels of granularity. For example, observations from a light sensor might be modeled as ’light sensor observations’ as well as ’observations that observe the property light’. Semantically they denote the same, however, on a data level they are not equal. The employed semantic model allows to abstract these observations to the same level by employing reasoning. We employ expressive reasoning to abstract the data even in complex domains and broaden the search space by elevating data from different sources to the same abstractions.

Many IoT domains try to aid us in our daily activities through some sort of automated decision making. In order to do this, the context and the domain should be taken into account. For example, this allows to automatically reduce the lights levels when a patient with a concussion is exposed to bright light, while this is not needed for patients who are not sensitive to light, at least during the day. We employ expressive reasoning to extract these actionable insights. This allows to define the business logic in a declarative way in the well-defined logic of the ontology model. This reduces the amount of code that needs to be written and allows to easily maintain and extend the business logic. Furthermore, reasoning techniques can explain why and how they took certain decisions [3], in comparison to black box machine learning techniques [4]. The topic of explainability will become very important, as the European Union's General Data Protection Regulation (GDPR) imposes that users have the right to an explanation regarding the automated decisions an algorithm takes [5]. It is worth noting that reasoning over well-defined logics, such as DL, ensures complete and correct answers [6]. This means that, given the available facts, the reasoner will infer correct decisions, in comparison to probabilistic methods that provide an answer with a certain probability [7].

Data streams are a common characteristic within the IoT. In order to make actionable insights on this streaming data, they should be processed as fast as possible. However, to obtain these insights, expressive reasoning capabilities are required, which by default cannot handle the update frequency of these data streams. We propose a cascading approach that allows to reach the required level of expressivity while processing data streams by filtering out parts of the streams with less complex techniques. This allows to perform expressive reasoning over volatile data streams. Furthermore, this enables extracting actionable insights in a flexible, easily adaptable and declarative manner, by defining the information need on a high-level.

Detecting temporal patterns is important in the IoT, as many domains have temporal dependencies. However, efficient techniques such as CEP fail to model the complexity of the domains employed in the IoT [8], while expressive reasoning techniques that incorporate temporal reasoning easily become undecidable [9]. We propose a solution that orthogonally combines the detection of temporal patterns with expressive reasoning. This means that temporal patterns can be defined while using high-level ontological concepts. This largely simplifies the definition of temporal patterns in complex domains. Furthermore, we provide a query language that enables the definition of temporal patterns over complex domains in streaming data. This unifying query language allows to 1) model complex domains, 2) use high-level abstraction to define temporal patterns, and 3) define how relevant parts should be selected from volatile data streams.

Edge computing allows to process the generated data as close as possible to its source. This means that the generated data could be processed as close as possible to the sensors in the IoT. However, resources are typically limited compared to a cloud computing setting. Therefore, processing techniques should be very efficient. We propose a hierarchical reasoning RSP engine that is at least twice as fast as the state-of-the-art and employs only a minimal memory footprint. This efficiency allows to extract actionable insights in a reactive fashion, close to the generated data. As data does not need to be transferred to the cloud for processing, this enables minimal response times, increased security and privacy and limited network usage.

Expressive reasoning techniques have some drawbacks when reasoning over large amounts of data, as their reasoning time can increase exponentially with the size of the data. However, in many IoT domains, large amounts of background knowledge are often necessary in order to make meaningful decisions. We propose an approximation technique that increases the reasoning scalability when reasoning upon large datasets. This allows to make actionable insights that require large amounts of contextual data in order to make the right decisions.

Specifically, we propose an IoT platform that allows services to filter the produced IoT data, hiding the heterogeneity and problems with the granularity for the service developers. This allows to obtain data produced by different sources even if they employ different content syntaxes. Since data in the IoT is often volatile, we propose solutions to handle these data streams, without losing the advantages of being able to abstract the data to a high level in order to broaden the search space. We make sure services can extract actionable insights both from volatile data streams and in scenarios where large amounts of data need to be considered in order to make correct decisions.

6.3 Open Challenges and Future Directions

We list the remaining open challenges and provide future work directions.

6.3.1 Cloud and edge deployment

We have currently focused on showing the feasibility of a cascading reasoning platform, with multiple optimizations in the various layers. However, scalability remains a challenge as we have mainly focused on single node solutions. To be deployed in a real-world scenario, the platform should be deployed in the cloud and at the edge so that the platform can automatically be scaled and duplicated. The lower layers of the cascade can easily be duplicated and deployed at the different locations in the edge, close to where the data is produced. This reduces the need to transmit large amounts of data to the back-end and improves latency. The layers that employ solutions with higher complexity of processing and thus demand more resources, can be deployed in the cloud. Intermediate modules would be necessary to combine and process the data from the edge at intermediate nodes. Additional algorithms need to be investigated to enable automatic distribution and duplication of various layers, so a more scalable system can be realized. First steps towards the investigation of how cascading reasoning can be distributed across the cloud and the edge have been made by my colleague Mathias De Brouwer [10], who will focus his PhD on this specific problem.

6.3.2 A user-friendly query language for streaming data & IoT services

To provide an easy to use platform, users should be able to define an easy to express Information Need [11]. The usability of the current platform remains a challenge, even the special designed query language for the cascading platform requires some knowledge regarding the data. Additional research is required that allows users to define their Information Need on a high-level while the platform takes care of federating and rewriting the Information Need to the lower and possibly distributed layers.

6.3.3 Privacy and security

Privacy and security are important requirements for IoT platforms, as they often handle sensitive and personal data. For example, as the goal of the IoT is to make personalized decisions, IoT applications need to collect personal data. In a security breach, attackers could extract sensitive data, such as when a user is at home or away. Therefore, authentication and access control need to be incorporated in various layers of the IoT, so that users and their data can be kept safe [12]. By distributing parts of the platform in which data is processed as close to the source as possible, a part of this problem can be eliminated, as all data does not need to be transmitted to the back-end. Policies should be designed that allow to restrict the access to certain sensitive data.

6.3.4 Integration data-driven technologies

The IoT is a dynamic environment, resulting in rapid changes. A remaining challenge is the adaptability of the platform to changing environments. For example, concept definitions might change or new concepts might occur. We have currently focused on declaratively defining our information needs. However, when the environment is rapidly changing, data-driven techniques would allow to anticipate these changes and learn how the environment is changing, so that the platform can automatically adapt. In Appendix A, a first step towards learning concept definition from a data-driven perspective is proposed.

6.3.5 Anomaly detection

A specific use case that requires data-driven techniques is the detection of anomalous behavior. The techniques described in this dissertation can declaratively define what can be considered normal/abnormal behavior. However, in changing environments, the system needs to learn how its environment is changing, such that it can distinguish normal from abnormal behavior. Further investigation is necessary on how data-driven anomaly detection techniques can be coupled with the declarative techniques defined in this dissertation. It is worth noting that existing anomaly detection techniques are typically trained for a specific environment, lacking the flexibility to be used in other contexts. Solutions that can identify the differences between the context it was previously applied in and its new context can pinpoint how the detection techniques should be updated. Thus, the combination of the data-driven anomaly detection techniques with the contextual declarative techniques discussed here can result in a best of both worlds solution.

6.3.6 Integration with big data platforms

In order to perform cascading reasoning on an extreme scale, existing big data stream processing solutions should be integrated allowing to exploit the scalability and fault-tolerance capabilities of these platforms. Many of these platforms exist, such as Apache Flink, Apache Spark or Apache Beam. Further research is required to investigate how the different layers of the cascade can scale and how they map on the semantics of these big data platforms.

6.4 Lessons Learned & Limitations

It is clear that the proposed solution in this dissertation is powerful enough to tackle the problems inherent to the IoT. However, each solution has limitations. This section describes some of the limitations of the current proposed solution.

6.4.1 Open World Assumption

We have currently focused on OWL2 DL reasoning, which uses the Open World Assumption. This is a powerful tool to deal with missing information, however, it comes with the cost that it lacks the possibility to use default negation, i.e. detect the absence of a certain event. Additional research is required to find a symbiosis between open and closed world reasoning frameworks.

6.4.2 Out-of-scope use cases

Even though the platform can deal with data streams, the solution cannot solve all use cases. Use cases requiring video stream analysis cannot be solved with the current solution. However, additional adapters that perform image processing can produce events that can be further processed with the proposed solution in order to achieve more accurate results. Use cases requiring hard real-time processing are currently also out-of-scope. These use cases would require optimized adapters that can automatically be deployed by the platform once it has determined a way to distribute the Information Need that needs to be solved.

6.4.3 Manual definitions

Declarative languages are very powerful, however, it can be time-consuming to define all required knowledge or queries. Especially the definitions of ontologies can be time-consuming as they require to model the domain knowledge. There is ongoing research that allows to extract ontology definitions form various document types in order to lower this burden.

6.5 Conclusion

This dissertation presents a cascading reasoning platform that allows to process volatile IoT data streams into actionable insights by taking into account the current context and domain knowledge. Many challenges still lie ahead, however, first steps have been taken to find valuable insights in large volumes of high-velocity data streams, regardless of their inherent heterogeneity and variety.

References

[1] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. Computer Networks, 54(15):2787–2805, oct 2010. Available from: http://linkinghub.elsevier.com/retrieve/pii/S1389128610001568, arXiv:arXiv:1011.1669v3, doi:10.1016/j.comnet.2010.05.010.

[2] N. Alhakbani, M. M. Hassan, and M. Ykhlef. An Effective Semantic Event Matching System in the Internet of Things (IoT) Environment. Sensors, 17(9):2014, 2017.

[3] D. L. McGuinness and P. P. Da Silva. Explaining answers from the semantic web: The inference web approach. Web Semantics: Science, Services and Agents on the World Wide Web, 1(4):397–413, 2004.

[4] M. T. Ribeiro, S. Singh, and C. Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pages 1135–1144. ACM, 2016.

[5] B. Goodman and S. Flaxman. European Union regulations on algorithmic decision-making and a" right to explanation". arXiv preprint arXiv:1606.08813, 2016.

[6] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider, and D. Nardi. The description logic handbook: Theory, implementation and applications. Cambridge university press, 2003.

[7] C. Robert. Machine learning, a probabilistic perspective, 2014.

[8] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. Data Science, (Preprint):1–25, 2017.

[9] S. Batsakis, E. G. Petrakis, I. Tachmazidis, and G. Antoniou. Temporal representation and reasoning in OWL 2. Semantic Web, 8(6):981–1000, 2017.

[10] M. De Brouwer, F. Ongenae, P. Bonte, and F. De Turck. Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions. Sensors, 18(10):3514, 2018.

[11] R. Tommasini, P. Bonte, E. Della Valle, F. Ongenae, and F. De Turck. A Query Model for Ontology-Based Event Processing over RDF Streams. In European Knowledge Acquisition Workshop, pages 439–453. Springer, 2018.

[12] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao. A survey on security and privacy issues in internet-of-things. IEEE Internet of Things Journal, 4(5):1250–1258, 2017.

A

Towards Optimizing Hospital Patient Transports by Automatically Identifying Interpretable Causes of Delays

The previous Chapters explained how expressive reasoning can efficiently be executed over IoT data streams. However, over the course of time, the definition of concepts might change, or new concepts might occur. This Appendix explains how new concepts can be learned. More specifically, we provide a solution to learn concept definitions that explain groups of certain events. Data-driven techniques can be incorporated in the cascading reasoning approach, allowing to constantly expand the domain with new knowledge. Starting from a data perspective, clusters of events can be semantically explained, such that these definitions can update the knowledge base and be used to detect future events in the various layers of the cascading reasoning approach. This is presented through a hospital use case, where we learn the reason why hospital transports are delayed. Historical transport data is used to learn definitions that explain why transports were late or on time. These definitions can then be used to detect, while the transports are being scheduled, which transports will be late. However, learning techniques can be employed in any domain which requires to adapt to dynamic changes. These techniques can be employed in the services that subscribed to the IoT data through the use of the platforms described in Chapter 2 and 3, to update the definitions used to reason about large knowledge bases as in Chapter 5 or to directly update the knowledge that is employed to filter the data streams in Chapter 3.

⋆ ⋆ ⋆

P. Bonte, F. Ongenae, F. De Turck.

Accepted to International Journal of Software Engineering and Knowledge Engineering, October 2018.

Abstract The continuous financial pressure on hospitals forces them the rethink various workflows. We focus on optimizing hospital transports, within the hospital, as they count up to 30% of the overall hospital cost. In this paper, we discuss a self-learning platform that learns the causes of transport delays, in order to avoid these kinds of delays in the future. We pay special attention to the explainability of the self-learning system, such that management understands the learned causes and remains in control over the automated process. This is achieved by providing the learned causes as sentences that can be understood by non-technical personnel and allowing these causes to first be supervised before the system takes them into account. Once approved, the system will calculate how much more time should be assigned to these transports in order to avoid future delays. As a result, the scheduling of patient transportation can be automatically optimized, while management remains full control of the process.

A.1 Introduction

A.1.1 Background

Due to the continuous financial pressure, hospitals struggle to balance budget while maintaining quality of care [1, 2]. Hospitals are forced to rethink and optimize various workflows in order to meet the financial constraints. In this paper, we focus on the transportation of patients within hospitals, as in-hospital transportation counts up to 30% of the total hospital cost [3]. Furthermore, not all transport tasks are performed by logistic personnel. It is estimated that nurses spent up to 10% of their time performing transportations of patients or goods [4], instead of taking care of patients. This leads to cost and care implications, but most importantly, due to the shortage of healthcare personnel, to social implications, such as stress-related diseases [4].

The increase in ICT infrastructure in hospitals can aid in the optimization of hospital's workflows, as the better use of ICT infrastructure is essential to providing better care at lower cost [5, 6]. The advent of the Internet of Things (IoT) allows the usage of non-intrusive sensors and devices to capture the environment through sensor readings [7]. These sensors and devices can be used to track the locations of various transports, to localize beds & wheelchairs, to easily notify staff members, etc.

Existing solutions have three major shortcomings. Firstly, the algorithms to schedule transports are based on fixed predefined parameters. For example, a transport from a patient room to a medical room always takes the same amount of time. They do not take the hospital's context into account. For example, the current occupancy level of the hospital is disregarded, thus regardless of the hospital's occupy, the same amount of time is scheduled to perform a transport. During visitor hours, the additional visitors might cause the transports to move more slowly. Secondly, the transport schedule is made in advance and last minute changes in the schedule are hard to achieve as the scheduler cannot be dynamically updated. Lastly, since these solutions lack the ability to model the context of what is happening inside the hospital, they are oblivious to why certain transports are late. As such, they cannot learn from past delays and keep making the same sub-optimal decisions.

The high cost and involvement of the nursing staff and the lack of automation make in-hospital patient transport an ideal candidate for optimization.

A.1.2 Related Work

Previous work has focused on learning delays in the health, financial and transport domain. Laskowski et al. [8] investigated how to reduce the patient wait time in the emergency department. The technique is specifically for the emergency department and does not provide interpretable explanations on the algorithmic suggestions.

Markovic et al. [9] proposed a system to learn the passenger train arrival delays, while Rebollo et al. [10] focused on predicting air traffic delays. Xu et al. [11] have focused on predicting traffic delays and Silva et al. [12] investigated the influence and delays on the public transports when certain stations or lines have been closed. However, these techniques predict the amount of time a certain transport will be late, but do not provide interpretable rules that can be supervised and provide an explanation of why the delays occur.

Lecue et al. [13] have shown the importance of explainability, as they allow flagged expenses to be explained to auditors in the financial sector. This allows the auditors to understand why certain expenses were flagged. Diagnosis of traffic congestion has also been investigated, allowing to explain and identify why certain roads are congested [14]. Even though these systems enable explanation, they provide the explanation in the form of rules, which still require domain experts to understand them.

Previous research has also focused on the scheduling of dynamic changing tasks in hospitals. Fiegl et al. [15] describe an online algorithm for dynamic scheduling of pick-up and delivery tasks in hospitals. Hanne et al. [16], on the other hand, have focused on transport between hospital buildings. Beaudry et al. [17] provide a solution to scheduling dynamic hospital transport requests. They take both in-house transports, i.e. transports within the same hospital building, as campus-based transports, i.e. transports between hospital buildings that need to be provided by an ambulance. Kergosien et al. [18] take into account additional constraints, such as disinfection of a vehicle or type of vehicle needed. These algorithms are able to cope with the dynamic nature of the tasks requests, however, no insights are provided in why these transports are late.

A.1.3 Objective

In this paper, we present a solution, designed in collaboration with two Flemish hospitals, that models the hospital's context by integrating various sources of information: static information regarding the hospital layout, patient & staff information and dynamic data resulting from the sensor stream, such as sensor readings that capture the location of the transports. Based on a historical view of this context, we provide a self-learning module that learns the causes of transport delays. These causes are presented as human-interpretable sentences that can be supervised by management. Upon supervision, when one or more of these causes are accepted by management, the system takes these causes into account and when transports that adhere to the selected causes are requested in the future, the system will calculate the additional time needed to enable accurate scheduling and avoid future delays. Note that highly accurate scheduling is important. When too much time is assigned and a transport is finished too early, the next transport might not be ready for transportation yet. When too little time is assigned, the next transports need to wait until the transport is ready and the whole schedule gets turned over.

A.1.4 Requirements

To provide a system that can learn the causes of delayed hospital transports, provide them in a fashion that non-technical users can understand them and allow these causes to be taken into account such that future delays can be avoided, the following requirements should be adhered:

- **Extendability**: since the ICT infrastructure in hospitals keeps evolving, it should be possible to easily incorporate new sources of information, such that this new information can also be used to learn the causes of delays. These sources can be either sensors & devices providing dynamic data or databases providing descriptions regarding the hospital's static context.
- **Human-involvement**: since it is sensitive to allow an automated system to directly adapt the hospital work processes, it should, therefore, be possible to involve human decision making.
- **Explainability/Interpretability**: to involve human decision making, it should be possible for non-technical users to interpret and understand which causes the self-learning algorithm is suggesting. It is important that users understand the decisions of an automated system [19].
- **Scalability**: to be applicable to different sizes of hospitals, the algorithms should scale adequately.
- **Usability**: non-technical users must be able to operate the system and evaluate the decisions from the self-learning module. The system should be easily accessible through a Graphical User Interface (GUI). Furthermore, the GUI should provide an easily interpretable overview of the findings of the module. Lastly, it should be intuitive to accept certain identified causes and update the system.

A.1.5 Paper organization

The remainder of the paper is organized as follows. Section A.2 details the designed architecture to optimally schedule requested hospital transports, notify the staff, capture the data to learn upon and describes how it can be deployed as a whole in a hospital. In Section A.3, we zoom in on the self-learning component and detail the devised algorithms. Section A.4 describes how we enabled explainability of the learning component, such that non-technical personnel, can understand the outcome of the learning system. The evaluation of the learning module is described in Section A.5 and the outcome is discussed in Section A.6. In Section A.7 we highlight the most important conclusions and describe opportunities for future work.

A.2 The AORTA platform

In this section, we describe how the data originating from various sources can be integrated and interpreted and detail the overall architecture of the designed Adaptive Optimization for Resource & Task Assignment in Hospitals (AORTA) platform.

A.2.1 Ontologies & Reasoning

To enable heterogeneous data integration and interpretation, an ontology [20] is composed that models the hospital's context. Ontologies are formal models that semantically describe a certain domain, in this case, the hospital domain. This description is made by modeling the different concepts within the domain and how they relate through the use of relations. Each concept, relation or individual (an instance of the former) is referenceable through a unique Uniform Resource Identifier (URI), e.g. http://aorta.intec.ugent.be/ontology/aorta.owl#PatientTransport. Ontologies are also an ideal tool for integrating IoT data [21], as it provides a uniform model for multiple heterogeneous data sources to adhere to. A part of the ontology designed to describe the hospital transports is depicted in Figure A.1[1]. As Figure A.1 describes, there are two types of transports, i.e. LogisticTransportTasks and PatientTransportTasks, that each are executed by a certain Person, that has a certain Role and each TransportTask has a specific Location as destination, etc. Since the ontology is a uniform and formal model, different data sources can map their data onto the ontology allowing to integrate data from various sources. As such we get a complete, interpreted overview of the current context and status of the hospital transports.

The ontology can define implicit relations within the data, that can be inferred through the use of a reasoner. The reasoning process is comparable to the execution of rules but in a more formal environment. For example, as depicted in Figure A.1, a Nurse is a subclass of Staff, which is a subclass of Role. The reasoner will infer that each Nurse is also a type of Role, when it is provided with an instance of the type Nurse. However, more complex constructions can be defined in the ontology. For example, we could define that a PatientTransport that has a relation hasTransportType to a Bed and a relation hasErrorCode to a PatientNotReady error code can be considered a

[1]The full ontology can be found on http://pbonte.github.io/Ontologies/aorta/aorta.owl

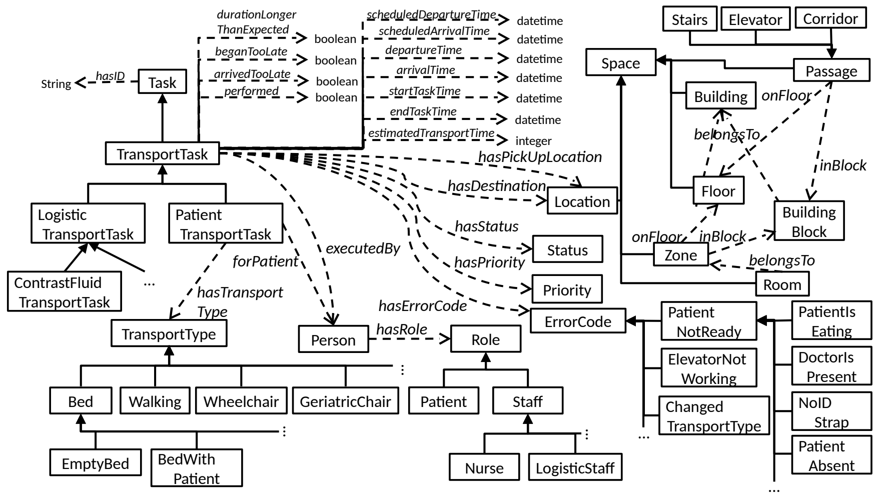


Figure A.1: Overview of the most important concepts (visualized in the rectangles) and relationships between concepts (visualized through the dotted arrows) and the hierarchical interpretation of the concepts (interpreted as subconcepts and visualized through full arrows) of the designed hospital transport ontology.

late transport. This could be defined in a formal way as[2]:

$$\begin{aligned} LateTransport \equiv & \exists hasTransportType.Bed \\ & \sqcap \exists hasErrorCode.PatientNotReady \end{aligned}$$

When a transport is requested that adheres to this definition, the reasoner will infer that the transport is a LateTransport. Say the following fragment describes a newly requested transport:

$$PatientTransportTask(p1), hasTransportType(p1, t1), BedWithPatient(t1)$$
$$hasErrorCode(p1, e1), PatientIsEating(e1)$$

The reasoner will infer that the individual $p1$ is a $LateTransport$ because it knows that BedWithPatient is a subclass of Bed and PatientIsEating is a subclass of PatientNotReady. Note that $p1, t1, e1$ are data instances (individuals) that have a certain type and relations, e.g. $p1$ has the type PatientTransportTask and has a relation hasTransportType to $t1$.

The definition of these rules allows to incorporate the logic that is specific to a certain domain. Here, we will use the reasoning capabilities to identify transports that adhere to the previously learned causes that identify that a transport might be late. Note that the system assigns more time to these late transports, such that the scheduling can be defined more accurately.

[2]The $\exists$ denotes an existential quantifier and can be interpreted as 'there exists'.

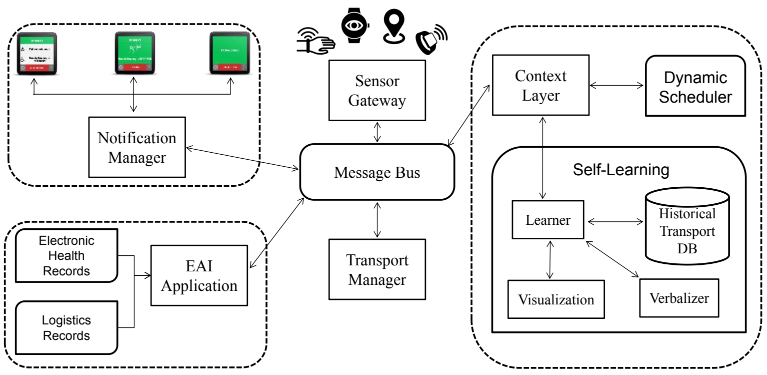


Figure A.2: Architecture of AORTA project.

A.2.2 Architecture

Now that we can model the context in our hospital and integrate the data from various data sources, we can describe the components in the AORTA architecture. Figure A.2 provides a visual overview of the different components and how they interact.

To capture the hospital's environment, **smart devices, wearables, and sensors** are introduced into the hospital's environment. These devices generate the dynamic data, e.g. data describing the location of the staff or the status of the transport. Each staff member is equipped with a smart wearable that allows to receive, accept/decline transports notifications and transmit location updates. Furthermore, the smart wearable allows to scan, through the use of NFC or QR codes, each patient that needs to be transported, eliminating mix-ups. All this information is pushed on the **Message Bus** that routes the generated data to the interested services, which can subscribe to particular data on the bus. The **Notification Manager** communicates with the wearable devices and notifies the staff members about new tasks or updates. It captures whom of the staff are available on which devices, allowing the Context Layer to target the correct staff members when dispatching tasks.

The **EAI Application** module is responsible for integrating the existing hospital data tools, such as the electronic health records of patients and information regarding the staff members and the logistics. This module is responsible for extracting from these existing tools, the information relevant for scheduling and executing the transportation tasks and providing it to the Context Layer.

The **Transport Manager** allows the different hospital departments to request transport tasks, which are forwarded to the Context Layer for optimal scheduling.

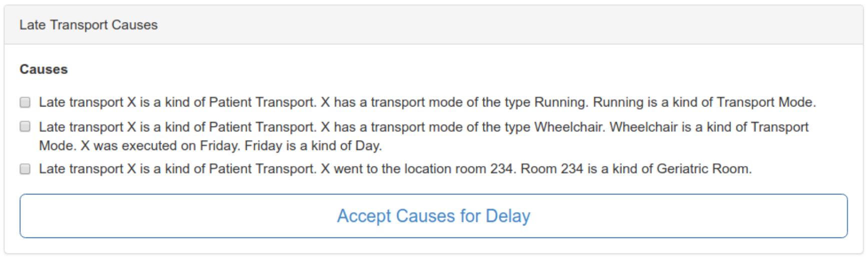
The **Context Layer** captures the current context in the hospital by combining and integrating the data resulting from the EAI Application and Transport Manager with the dynamic sensor data. The ontology described above is exploited for this purpose. This contextual information is stored in a triple store, i.e. a database for ontological data. The Context Layer provides data to the Dynamic Scheduler that needs to know which tasks to be scheduled, which personnel is available and what are their locations, the achievable walking speed considering the current commotion of the hospital, etc. The Context Layer uses reasoning to infer missing and implicit data, based on the ontological definitions. The reasoning can indicate which transports need more time to be executed, e.g. by detecting delays and interruptions.

The **Dynamic Scheduler** receives the transportation requests from the Context Layer and uses its context to construct an optimal schedule such that all the requests can be handled in a timely manner with an optimal use of resources. To achieve this optimal rostering, the scheduler will request the dynamic context information from the Context Layer, e.g., the locations, availability, competences, work load & average walking speed of the staff, busy areas and possible causes of delay. It constantly maintains an overall optimal schedule and updates this schedule as new requests and status updates of on-going transports come in. When a staff member indicates that a transport has been finished, the Context Layer will communicate this to the Dynamic Scheduler, which will then assign a new task to this staff member based on this overall optimized schedule. Note that it is the Context Layer that indicates how much more time should be assigned to the transports that are expected to be late. The scheduler will try to optimally schedule the tasks based on the provided information from the Context Layer. To be able to dynamically update its schedule when new transports are requested, the scheduler uses a dynamic pick-up and delivery model [22].

The **Self-Learning** module consists of four components, the learner, the visualizer, the verbalizer and a historical database. The later keeps a historical view of the Context Layer. The learner requests the data from the historical database, such that it can learn from the historical data why certain transports were delayed. Based on this historical context, the learner identifies why transports in the past were delayed, such that these delays can be prevented in the future. For example, the module could learn that certain transports during the visiting hour on Friday are often late and more time should be reserved for them. Once the learner has learned the causes as ontological rules, the verbalizer can transform these rules to human readable sentences such that management can access the identified causes through the visualization and argue their validity while remaining control over the automated system. Once approved, these rules are added to the Context Layer. When similar transports are scheduled, they will be identified through the use of the reasoner and the platform will calculate how much more time will need to be incorporated to schedule these kinds of transports accurately. This module is further detailed in Section A.3.

A.2.3 Implementation

We now explain some of the implementation details of the components that are not discussed in detail in the remainder of the paper.


Late Transport Causes

Causes

- ☐ Late transport X is a kind of Patient Transport. X has a transport mode of the type Running. Running is a kind of Transport Mode.
- ☐ Late transport X is a kind of Patient Transport. X has a transport mode of the type Wheelchair. Wheelchair is a kind of Transport Mode. X was executed on Friday. Friday is a kind of Day.
- ☐ Late transport X is a kind of Patient Transport. X went to the location room 234. Room 234 is a kind of Geriatric Room.

Figure A.3: Example of the verbalized rules.

The **Message Bus** is based on Mirth Connect[3], a message broker optimized for the transmission of healthcare messages.

The **Notification Manager** receives the scheduled tasks from the Message Bus, which are already targeted for a given user. The Notification Manager chooses the best way to notify the user, i.e. choose the most convenient device that the user is carrying at a given moment. Then, the Notification Manager transforms the tasks into notifications that are tailored to be presented on the selected device.

The **Context Layer** utilizes RDFox [23] to store the contextual data and perform the reasoning on it. RDFox is the fastest reasoning-enabled triplestore, i.e. a database to store ontological data, currently available. The Context Layer also needs to map the healthcare messages from the Message Bus to the ontological data, this is done through the use of RML [24], that allows to map raw data to the ontology model.

A.2.4 Workflow Self-Learning component

When a new transport is requested, it is captured by the Context Layer that models the current view of the hospital's context. After its execution, the details of the transport are communicated with the Self-Learning module, such that the module can store a history of past transports. Figure A.2 shows the components of the learning module. The new transport arrives through the Message Bus, the Learner component first captures the transport and stores it in the historical database. It also does some quick preprocessing such that a real-time overview of the transports can be shown in the Visualization.

When a more in-depth description of the causes of delays is necessary, through the visualization one can request the Learning component to start learning the delays of the transports in the selected time range, e.g. the last month. It will load the data from the historical database and start one of the learning algorithms detailed in Section A.3. The Learner passes the learned rules to the Verbalizer and sends them to the Visualization so they can be shown in the visualization dashboard. Figure A.3 shows an example of the verbalized rules in the dashboard.

When one of the verbalized causes get accepted by management, the learning module cal-

[3]https://www.nextgen.com/products-and-services/integration-engine

culates the average delay that was caused by these transports. The additional time is added as part of the rule. The Context Layer is then updated by adding the new cause as a rule that will be invoked by the reasoner when a similar transport is being scheduled. Since the extra time necessary to execute the transport within accepted time is part of the rule, the additional time is automatically added to the new transport.

Example 21. (Adding time to late transports) Say the following rule has been accepted:

$$hasTransportType.Bed \wedge hasPeriod.visitingHour \rightarrow LateTransport$$

When adding this rule to the reasoner and a transport is requested with the transport type bed and requested during visiting hours, the reasoner will know that the transport will be late. We can now calculate how much more time, on average, is necessary to provide the scheduler with as accurate data as possible, to prevent future delays. This can be done by calculating how much more time is necessary to finish this task within time. This calculated time can then be added as part of the rule. If 10 additional minutes are required, we can update the rule, to automatically add this time to the transport:

$$\begin{aligned} hasTransportType.Bed \wedge & hasPeriod.visitingHour \\ & \rightarrow LateTransport, addTime(10min) \end{aligned}$$

When the Context Layer adds this rule to the reasoner, and a new transport is requested that adheres to this rule, it will know that the transport will be late and that 10minutes additional time should be reserved to perform the transport.

A.3 Rule Learner module

The Self Learning module and more specifically the Learner, learns from historical information regarding the transports why certain transports are delayed. By indicating the causes of these delays, future delays can be prevented. We investigated and optimized two techniques to learn the transport causes: an Assocation Rule Mining (ARM) [25] and an Inductive Logic Programming (ILP) [26] technique. We detail each technique and how we combined them for the best performance.

A.3.1 Association Rule Mining

ARM was originally developed to discover hidden knowledge from transactional data, such as relational databases. A transaction is an observation of the co-occurrence of a set of items.

$I = \{i_1, i_2, \ldots, i_m\}$ is defined as a set of m items describing the different elements the database could contain and $D = \{t_1, t_2, \ldots, t_n\}$ as a database of n transactions, where each transaction in D is a subset of I and can be seen as a database entry. We name a subset of items an itemset. $supp(X)$ is the support of an itemset X, i.e. the percentage of transactions in the database D that contain X.

Table A.1: Example transactions

transactionID	Walking	WheelChair	VisitingHour	Late
t_1	1	0	1	0
t_2	0	1	0	0
t_3	0	1	1	1
t_4	1	0	0	0
t_5	0	1	1	1

An association rule r can then be defined as a rule of the form $X \Rightarrow Y$ where X and Y are non-empty subsets of I, and $X \cap Y = \emptyset$. X is called the antecedent of r and Y is the consequent of r. The support and confidence of a rule are respectively denoted as

$$supp(r) = \frac{|\{t \in D \wedge X \subseteq t\}|}{|D|} \tag{A.1}$$

and

$$conf(r) = \frac{supp(X \cup Y)}{supp(X)} \tag{A.2}$$

where the confidence describes the how confident one can be that the antecedent is related to the consequent of the rule.

Mining association rules is the process of finding all association rules with a support and confidence greater than a predefined threshold. This mining process can be divided into two phases. First, frequent itemsets of the transactions have to be computed according to the minimum support threshold. Second, rules are generated from these frequent itemsets with respect to the minimum confidence threshold.

Example 22. (Association Rule Mining) Table A.1 shows a database with four items: $I = \{Walking, WheelChair, VisitingHour, Late\}$ and five transactions. If we want to calculate if the association rule $r = \{WheelChair, VisitingHour\} => \{Late\}$ holds, we calculate the support as $supp(r) = \frac{2}{5}$, as only two transactions (t_3, t_5) consist of the itemset $\{WheelChair, VisitingHour\}$ and the database consists out of five transactions. The confidence is calculated as
$conf(r) = \frac{supp(X \cup Y)}{supp(X)} = \frac{supp(\{WheelChair,VisitingHour,Late\})}{supp(\{WheelChair,VisitingHour\})} = \frac{2/5}{2/5} = 1.$

Since ARM works specifically on items and transactions, it needs to be adapted to work with semantic data. Our previous work [27] describes in detail how the ontological data can be converted to items and transactions and how various optimizations can be executed. This conversion is necessary as the semantic data described by the ontology can be seen as a graph rather than a set of transactions. The conversion consists of the following steps:

1. We identify a concept in the ontology we want to get insights from and retrieve all individuals from that concept, e.g. all patient transports in a certain hospital.

2. We follow all the relations the selected individuals have to other individuals, store them as so-called features, and follow the relations from the new concepts until we reach a certain threshold that indicates how many concepts to follow.

3. We also store the types of each followed individual, i.e. the ontology concept they have been assigned to and look up the hierarchy of these concepts in the ontology and store the hierarchy as well.

4. The stored features can now be used as items for the transactions database.

Example 23. (Converting ontological data to transactions)
Say we have the following five (simplified) transports in our ontology:

- PatientTransport(t1), hasTransportType(t1,walking), Walking(walking), duringPeriod(t1,visitorHours),
- PatientTransport(t2), hasTransportType(t2,wheelchair), Wheelchair(wheelchair), duringPeriod(t2,morning),
- PatientTransport(t3), hasTransportType(t3,wheelchair), Wheelchair(wheelchair), duringPeriod(t3,visitorHours),
- PatientTransport(t4), hasTransportType(t4,walking), Walking(walking), duringPeriod(t4,morning),
- PatientTransport(t5), hasTransportType(t5,wheelchair), Wheelchair(wheelchair), duringPeriod(t5,visitorHours)

When we follow the above described steps, we can convert the ontological data into transactions:

1. The individuals we want to learn about are t1, t2, t3, t4, and t5, which are all PatientTransports.

2. When following their relations we obtain for each individual the feature hasTransportType and when coupling the relation to the linked individual we obtain the features: hasTransportType.walking and hasTransportType.wheelchair. Following the next relations, we obtain the features: duringPeriod,duringPeriod.visitingHours and duringPeriod.morning.

3. When taking the types into account we obtain the features: hasTransportType.Walking, hasTransportType.Wheelchair, duringPeriod.TimePeriod. Taking the hierarchy of the ontology into account we obtain the feature: hasTransportType.TransportType.

4. We can now convert the selected features to items and use each patient transport individual as a transaction. Table A.2 shows a part of these transactions. Note that some items should be filtered out as they do not provide any information gain, e.g. all transaction have the item hasTransportType.

Table A.2: Example ontology conversion to transactions

t	hasTransportType	hasTransport-Type.wheelchair	hasTransport-Type.walking	hasTransport-Type.Wheelchair	hasTransport-Type.Walking
1	1	0	1	0	1
2	1	1	0	1	0
3	1	1	0	1	0
4	1	0	1	0	1
5	1	1	0	1	0

A.3.2 Inductive Logic Programming

ILP is a machine learning technique that combines inductive machine learning and logic programming. ILP is able to learn rules as ontology concepts and fully exploits the semantics describing the data. Thus, ILP can work directly with the semantic data and generates very accurate rules, however, it is less scalable than statistical approaches such as ARM. Statistical relational learning [28] is an extension of ILP that incorporates probabilistic data and can handle observations that may be missing, partially observed, or noisy. However, since our observations are not possible missing or partially observed, we do not consider it here.

ILP starts from the idea of positive and negative examples and a background describing the domain. ILP tries to learn a hypothesis such that the positive examples follow from the hypothesis but the negative examples do not. Finding the hypothesis is, of course, the difficult part. The Class Expression Learning for Ontology Engineering (CELOE) algorithm [29] from DL-learner[4] takes a generate and test approach where it appends ontology concepts and relations to the learned hypothesis in order to achieve the highest possible accuracy. This is possible by making the hypothesis more generic or more specific. The latter is calculated on the fact that more positive examples are contained by the hypothesis compared to negative examples.

Example 24. (Inductive Logic Programming) We reuse the transports from Example 23 where both transport t3 and t5 are positive examples. The algorithm will take the following steps:

1. Create a new concept, e.g. LateTransport;
2. Add a new concept or relation to the concept, e.g. adding the relation hasTransportType. We thus generate a new concept that says that a LateTransport has a relation hasTransportType.
3. The algorithm tests the coverage of the new concept and sees that both positive and negative examples have the relation hasTransportType. The coverage is tested by adding the

[4] http://dl-learner.org/

new concept to the ontology and ask the reasoner for all the individuals that have the new concept as a type. The accuracy is calculated as the percentage of individuals that have the type LateTransport and were in fact in the positive examples.

4. The algorithm can decide to make the generated class more specific by changing any hasTransportType relation specifically for one type of transport, i.e. for wheelchairs. After specifying this relation, all the positive examples are contained, however, one negative example (transport t2) is also contained.

5. Therefore, the algorithm tries to add another relation, i.e. the duringPeriod.visitorHours relation. Now all the positive examples are contained and none of the negative examples.

Eventually the algorithm generates a new class:
$LateTransport \equiv PatientTransport \wedge \exists hasTransportType.\{wheelchair\}$
$\wedge \exists duringPeriod.\{visitorHours\}$.

In a realistic dataset, there are many reasons that might cause transport delays. This means that multiple rules need to be identified. In previous work, we coupled the ILP technique with an ontological clustering technique to split the dataset into some more manageable clusters, such that the algorithm can easier find the various delays [30].

A.3.3 Combining ARM & ILP for optimal identification of causes of delays

We have combined the two approaches such that we can benefit from the scalable statistical analysis from ARM and the correctness of ILP. Our technique is based on a statical evaluated generate and test method. Thus the statistical evaluation from ARM combined with the generate and test methodology from CELOE. ARM generates rules that are applicable to the whole dataset, however, since we are only interested in rules detailing the lateness of transports, many rules need to be filtered. Furthermore, since we have positive and negative examples, more rules need to be filtered that occur both in the positive and negative examples. Thus, there are many unnecessary computations as many rules need to be filtered to make the technique applicable. CELOE has the advantage of testing each addition it generates but requires for each addition a call to the reasoner, that does not scale very well, to compute the coverage.

In this approach, we directly compare the support, see Equation A.1, of each item in the rule in both the positive and negative examples. We take a Breadth-First Search (BFS) through the graph to compose the rules. Each edge we transverse, we test if the information is adding value to the generation of the rule. For each edge, we take four steps which are visualized in Figure A.5 that executes the algorithm on an example detailed in Figure A.4. These four steps are:

1. We add the edge (relations) to the path and use it as a feature.

2. We add the URIs of the individuals to the path and use it as a features.

3. Instead of the URI we add the type of the individual to the path and use it as a feature.

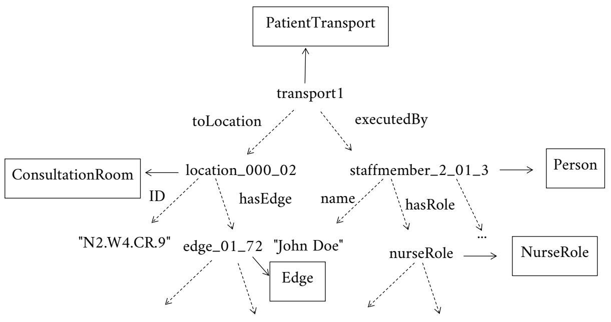


Figure A.4: Example of a fragment of a transport described by the ontology schema.

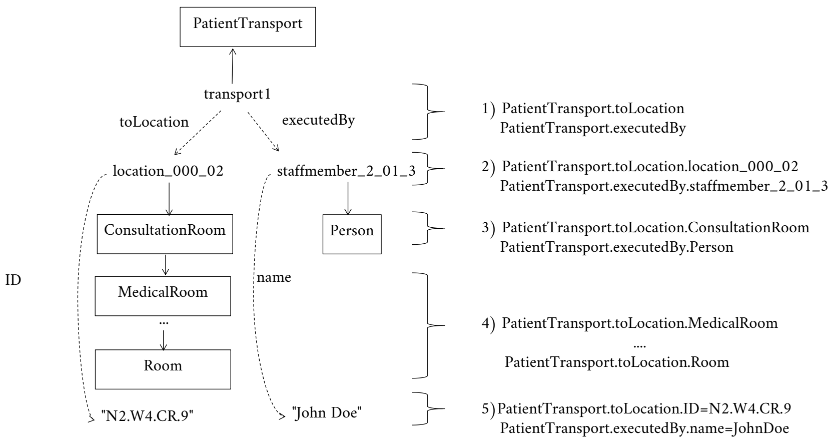

PatientTransport
transport1
toLocation
executedBy
location_000_02
staffmember_2_01_3
ConsultationRoom
Person
ID
name
MedicalRoom
Room
"N2.W4.CR.9"
"John Doe"
1) PatientTransport.toLocation
PatientTransport.executedBy
2) PatientTransport.toLocation.location_000_02
PatientTransport.executedBy.staffmember_2_01_3
3) PatientTransport.toLocation.ConsultationRoom
PatientTransport.executedBy.Person
4) PatientTransport.toLocation.MedicalRoom
PatientTransport.toLocation.Room
5) PatientTransport.toLocation.ID=N2.W4.CR.9
PatientTransport.executedBy.name=JohnDoe

Figure A.5: The algorithm traverses the graph in BFS mode and first adds the relations as items, then the individual names, followed by the types and super types and finally the data properties.

Table A.3: Example of conjunction.

t	hasTransportType .Wheelchair	hasPeriod .visitingHours	isLate
1	0	1	0
2	1	0	0
3	1	1	1
4	0	0	0
5	1	1	1

4. Instead of the type we generate for each super type (defined in the ontology hierarchy) of the type a new feature.
5. Instead of the super types, we add the data properties with their values to the path and use them as features.

We test which one of the four steps is the best feature candidate, both in terms of support and interpretability. The interpretability hierarchy is configurable, standard the most specific type has priority over the concepts in the type hierarchy, then the data properties, the relations and lastly the individual names. Note that this is only considered if multiple of these produce the same results. Once all the potential candidates have been generated we test which conjunctions enable a significantly higher drop in the support of the negative dataset, compared to the positive set.

Example 25. (Conjunction example) Let us consider the transports from Example 23 where t3 and t5 were late. We will calculate the conjunction between the transports with wheelchair transport type and the transports scheduled during visiting hours. Table A.3 shows the transactions that allow us to calculate the support of the conjunction. Table A.4 shows the support calculation for each of the positive (t3, t5) and negative (t1, t2, t4) examples. The table shows the support for the transports scheduled with the transport type wheelchair, i.e. a support of $2/5$ for the positive set and $1/5$ for the negative. The support for the transports scheduled during visiting hours are $2/5$ for the positive and $1/5$ for the negative. The table also shows that the support drops in the negative set when calculating the conjunction between the rules, while the support in the positive set remains. This means that this conjunction should be considered as a candidate result or temporarily result, e.g. when additional conjunctions are required to find significant difference in support.

A.3.4 Related approaches

Nebot et al. [31] proposed an ARM technique for ontological data. The concept and the features to learn about are defined through a SPARQL query , i.e. a query language for ontological data, and translated to transactions for the ARM algorithm. Our ARM approach builds upon their proposed technique in the sense that no explicit indication of the learning features is necessary and various optimizations are proposed to prune the learned rules.

Table A.4: Support calculation of the conjunction between transports with the transport type wheelchair and transports scheduled during visiting hours.

	Positive	Negative
support(hasTransportType.Wheelchair)	$\frac{\vert\{t3,t5\}\vert}{5} = 2/5$	$\frac{\vert\{t2\}\vert}{5} = 1/5$
support(hasPeriod.visitingHours)	$\frac{\vert\{t3,t5\}\vert}{5} = 2/5$	$\frac{\vert\{t1\}\vert}{5} = 1/5$
support(hasTransportType.Wheelchair ∧ hasPeriod.visitingHours)	$\frac{\vert\{t3,t5\}\vert}{5} = 2/5$	$\frac{\vert\{\}\vert}{5} = 0/5$

AMIE [32] and its successor AMIE+ [33] provide an algorithm for mining rules in large knowledge bases where there are no negative examples. The technique is more scalable than standard ILP techniques, however, it does not enable reasoning during the learning phase. This means that the algorithm cannot make a rule more specific or more generic to match the examples. Furthermore, it is not optimized to cope with positive and negative examples.

A.4 Interpretable Results

In Section A.3, we have shown how we can learn rules that describe why transports are late. However, as can be seen in Example 24 and 25, these rules are not very interpretable or intuitive for non-technical end-users, e.g. management of a hospital.

A.4.1 Verbalizing Rules

To make the learned rules more interpretable, we can convert the rules to human readable sentences. Since we make use of an ontological model, the model describing the rules is fixed. Therefore, we can make use of a verbalizer such as NaturalOWL [34] that can convert ontology concepts to readable sentences. By defining how the classes and properties in the ontology should be verbalized, NaturalOWL can generate fluent human-readable text. This makes it easier for the management to interpret the learned rules. In practice, to enable this, the ontology needs to be annotated and indicated which concepts should be interpreted as adjectives, nouns or verbs and how they construct readable sentences when combined. However, this typically needs to be done only once, since the ontology itself does not change (often).

Example 26. (Verbalization) The concepts in the ontology are annotated with verbalization information and various sentence plans are defined to be able to construct human-readable sentences. The class assertion definitions can be verbalized through the following sentence plan:

$[OWNER_{OWNER}][is_{verb}][a\,kind_{string}][of_{prop}][FILLER_{FILLER}]$

This means that the assertion $PatientTransport(trans_1)$ will be verbalized as "$trans_1$ is a kind of PatientTransport" . Where $OWNER$ is the individual assigned to the class, here $trans_1$ and $FILLER$ is the class itself, here the class $PatientTransport.$

The relation transportMode can be verbalized through the sentence plan:

$$[OWNER_{OWNER}][has_{verb}][a_{string}][transportmode_{noun}]$$
$$[of\ the\ type_{string}][FILLER_{FILLER}]$$

This means that the relation $hasTransportMode(trans_1, bed)$ will be verbalized as "$trans_1$ has a transport mode of the type bed". Here $OWNER$ is the individual from where the relation starts, here $trans_1$ and $FILLER$ is the individual that is linked by the relation, here the individual bed.

A.4.2 Dashboard

We are now able to learn the causes of the delays and verbalize them such that management can interpret them. However, they are still not usable as distinct tools. Therefore, we provide a visualization through a dashboard that enables insights into the transports and allows to activate the learning-verbalization-chain. Figure A.6 visualizes the dashboard. It provides some graphical analytics such as:

- An overview of the number of tasks that were on-time versus the ones that were late.
- An overview of all the transport modes and how they influence the arrival times.
- An overview of both the location the transport came from/is going to and how they relate to the arrival times.

The exact metric for what is shown in the overview can be easily configured through the use of a query. For example, in Figure A.6 two queries are defined, one for selecting the transport types and one for selecting the destination location of the transports. When a more in-depth analysis is necessary, the learning module can be activated from the dashboard to inspect the causes of transports delays over a specific time range. Figure A.3 shows an example of how the verbalized rules in the dashboard are shown. Each of the learned rules is translated into readable sentences and can, after inspection of management, be incorporated into the system to avoid future delays.

A.5 Evaluation

This section elaborates on the evaluation of the Self-Learning module and more specifically on its learning capabilities. We make a comparison between the learning capabilities of the different algorithms discussed in Section A.3.

A.5.1 Dataset

As the IoT system described in Section A.2.2 can only be deployed in a real hospital setting after thorough evaluation and proof that the system functions correctly, we do not have enough real-time data to be used in the evaluation. We note that the IoT platform has been evaluated in

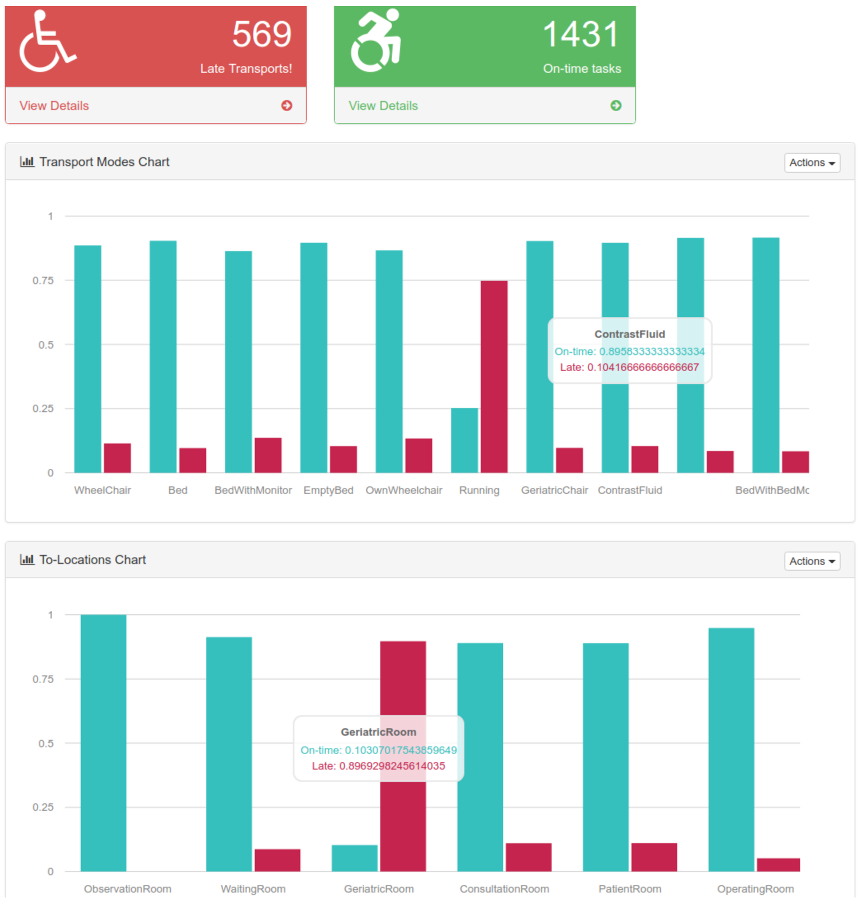


Figure A.6: Overview of the AORTA Self-Learning Dashboard.

a controlled hospital environment, to prove its feasibility. To enable the learning phase, data of many transports is necessary and since the platform could only be deployed in a smaller controlled environment, capturing enough data to enable the learning phase was not possible. However, the hospitals currently have a static scheduling system, describing the various dispatched tasks. Even though the static schedule does not contain all the context as it would in the IoT case, it is still a good starting point to show the feasibility of the learning component. As this static data is maintained in a relational database, we extract the data, map the data to the semantic model through the use of RML [24], which allows non-semantic data to be mapped to a semantic model, such that it can be used by the learning algorithms. We note that in the IoT deployment, when more data is available, more accurate rules can be learned.

We received static transport data from two Flemish hospitals describing over three months worth of patient transports details, based on 40 variables. On average, around 10000 transports are scheduled each month and about 26% of these transports are late. For the first two evaluations, i.e. Section A.5.2 and Section A.5.3, we adapted the hospital dataset so we can manipulate its distributions in order to illustrate the underlying mechanics of the learning algorithms. We selected one month worth of data and removed all transports that were late and added on time transports from the next months to obtain a total of 10000 transports. To evaluate the learning capabilities of the algorithm, we injected several causes of delays in the dataset, allowing to evaluate accurately if the algorithm is capable of detecting these causes. Among these causes are 1) transports from a patient room to a consultation room where the patient had to walk, 2) transports on Friday in the evening and 3) transports in the afternoon towards the operating room. For the last evaluation, Section A.5.4, we used the received dataset to explain why certain transports were late.

The dataset itself contains 474 unique locations, each mapped to the hospital layout and specific function of the location, 8 transport modes (e.g. bed, wheelchair, walking, etc.), the period of the day (i.e. morning, afternoon, evening), the exact time, etc. Since we started from the received dataset, the data distributions are realistic.

A.5.2 Minimum number of late transports

The cause of the late transports can only be detected if a sufficient number of these transports are contained in the positive examples. The question remains, how frequent should they occur to be detected? This is defined by the support parameter that defines the minimum frequency a candidate item should occur before it can be considered. This filters out very low occurring items and reduces the number of generated rules. Thus, the lower the support parameter the higher the chances it is detected by the algorithm. However, since lower support parameters imply more generated rules, more noise will be produced and complicates the interpretability by management. Figure A.7 shows the influence of the support parameter on the dataset, OWN indicates our combined algorithm, ARM our ARM approach and DL the CELOE approach provided by DL-learner. It is worthing noting that the CELOE does not have a support parameter and needs to be configured in the function of the number of results it may generate and the amount of time

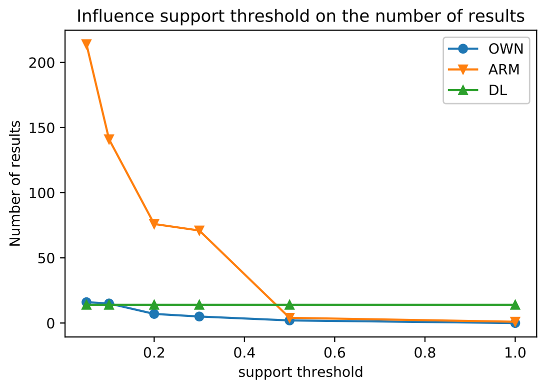


Figure A.7: Influence of the support parameter on the number of rules.

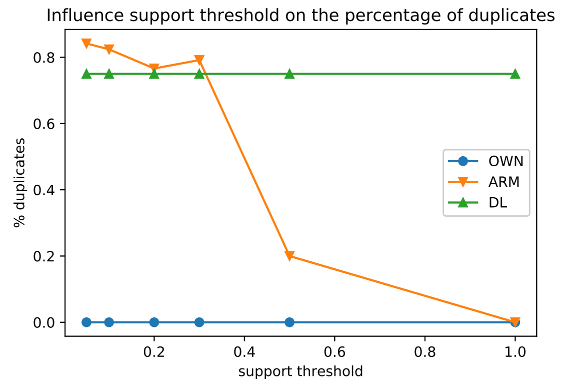


Figure A.8: Influence of the support parameter on the duplicates.

it may execute. We fixed the number of results to the same number as the expected number of results, i.e. the number of rules contained in the data and the correct execution time was obtained by iterating over various execution times until the causes were detected by the algorithm. The figure shows that for ARM and OWN, the lower the parameter, the more rules are generated. This makes sense as none of the injected rules are contained in more than 30% of the late transports. Therefore, the causes are only starting to be detected as the support threshold decreases below 0.3. It is clear that the ARM approach generates many more rules, but more rules do not necessarily mean better results.

Figure A.8 shows the percentage of duplicates contained in the results. These duplicates are unique rules that have the same meaning, e.g., each transport mode has a certain ID, resulting in two rules, one stating that the transport mode 'Running' causes transports to be late or one stating that the transport mode with id '131' causes transports to be late. These are different rules

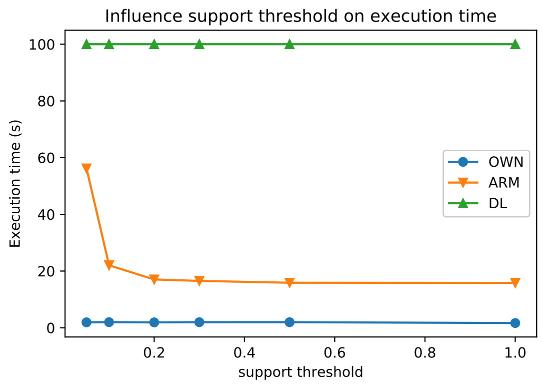


Figure A.9: Influence of the support parameter on the execution time.

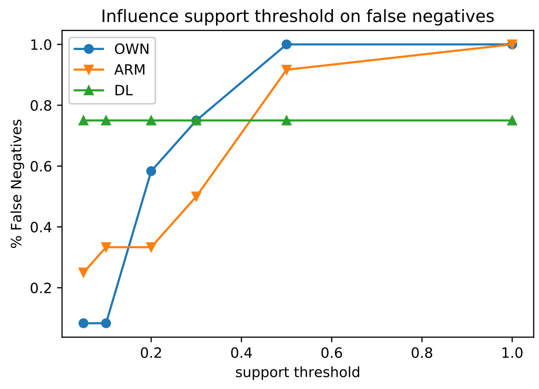


Figure A.10: Influence of the support parameter on the false negatives.

but have the same meaning. As explained in Section A.3.3 our own algorithm is tailored to only generate the most meaningful features and rules. This is reflected in the results of Figure A.8 as the percentage of duplicates is low for our algorithm.

In Figure A.9 the execution time of each algorithm is plotted. The ARM and OWN algorithms are faster than the CELOE algorithm as they take a more scalable statistical approach. For low support thresholds the execution time increases, this is because more features are selected which results in more combinations that need to be tested to detect the rules. Our own algorithm is less prone to this, as it uses a generate and test approach and only considers combinations of features if they are improving the accuracy. The ARM approach generates more combinations and is thus slower.

Figure A.10 shows the percentage of false negatives, i.e. the percentage of rules that should have been detected but were not. It is clear that as the support parameter decreases, more rules

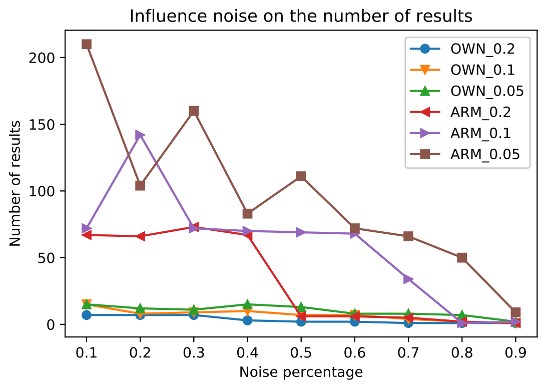


Figure A.11: Influence of the noise on the number of results.

are found for the ARM and OWN approach and the percentage of false negatives drops. As the support parameter has no influence on the DL approach, it remains constant. However, it fails to find most rules even after long execution times. The false negatives decrease faster in the ARM approach, however, only our OWN approach finds all rules. The reason for this slower decrease is because our OWN approach is very selective in which rules to generate.

A.5.3 Noise in the dataset

A second important aspect is the ability to cope with is noise. Many transports are late for no reason and are thus adding noise to the positive examples as there is not a straightforward explanation.

Figure A.11 shows the number of found rules in function of the percentage of noise in the dataset for a support threshold of 0.2, 0.1 and 0.05 for the ARM and OWN approach. We did not further include the DL approach, as it has troubles to deal with noisy data. We artificially added additional late transports to the dataset, which were selected from the set of transports that were on time and thus do not contain any real causes for their delays and can be considered as noise. The figure shows that as the noise increases, the number of found rules decreases. This is because the percentage of the late transports in the dataset that should be detected decreases and random causes increase. When further decreasing the support parameter, the rules can be detected again. However, when decreasing too much, random rules will start populating the results. Table A.5 shows this trend for a dataset containing 80% noise. The table shows for both OWN and ARM that as the support parameter decreases, more rules are generated. However, as the support parameter decrease, more random rules are considered as well. This can be expected, as with a support parameter of 0.001, a rule only needs to occur in 0.01% of the examples. For a dataset of one month, this means that only two occurrences should be present in the data. This leads to the production of many random results. The ARM approach is not able to produce rules for a support

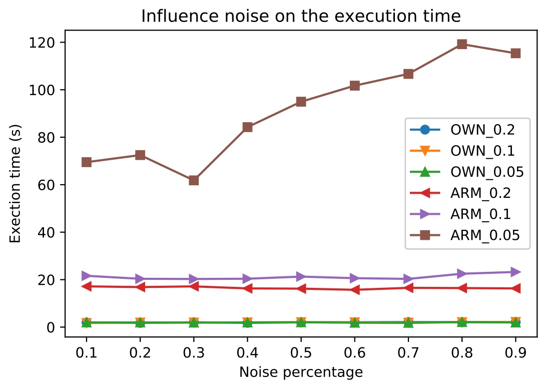


Figure A.12: Influence of the noise has the execution time.

Table A.5: The influence of the decreasing support parameters for a dataset with high number of noise (80%). Both the number of generated results (#res) are compared to the number of correct results (#correct).

Support	OWN		ARM	
	#res	#correct	#res	#correct
0.2	1	1	2	1
0.1	2	2	2	1
0.05	7	7	50	16 (56% duplicates)
0.01	16	14	-	-
0.005	35	14	-	-
0.001	198	14	-	-

parameter of 0.01 and lower. This is due to the fact that too many conjunctions need to be tested and therefore the execution time explodes.

A.5.4 Evaluating the unmodified dataset

We also executed the algorithms on the datasets received from the hospitals, i.e. the dataset described in Section A.5.1 without the artificially injected late transports. Since we did not inject the transports, there is no objective metric to evaluate the correctness of the learned rules. Therefore, we provide a discussion of our findings. Figure A.13 depicts the dashboard of the received dataset and shows some of the dataset characteristics.

While executing the learning algorithms, we found that some of the learned causes are rather trivial, such as the fact that if the task started on time or the priority of the task is low, then transports are often late. Other causes are however less trivial. In one of the hospitals, transports in the morning that need a wheelchair are often late. Transports on Friday or Saturday or to the consultation room share the same fate. However, transports that need a bed with a bed mover or

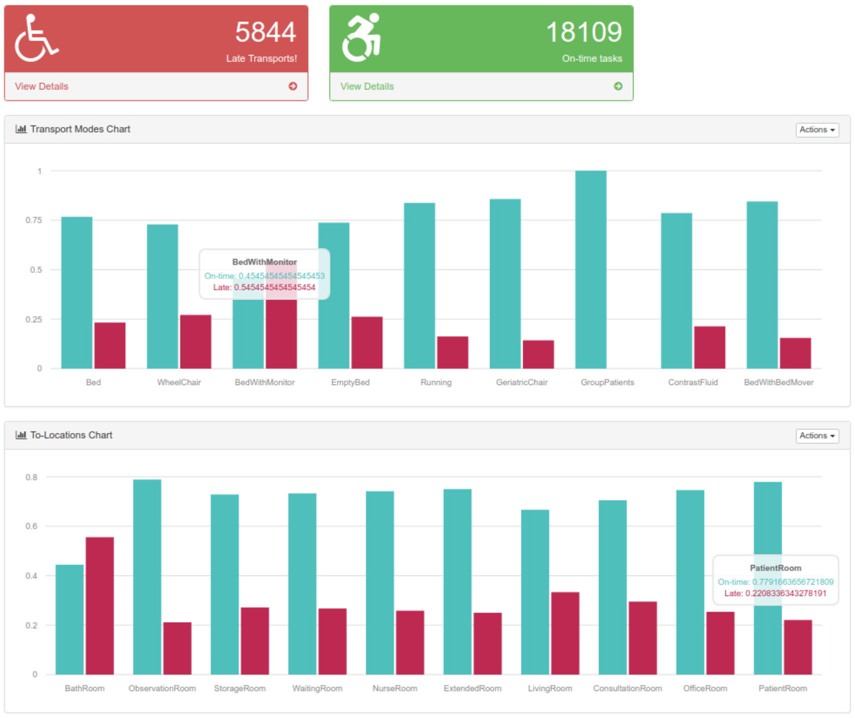
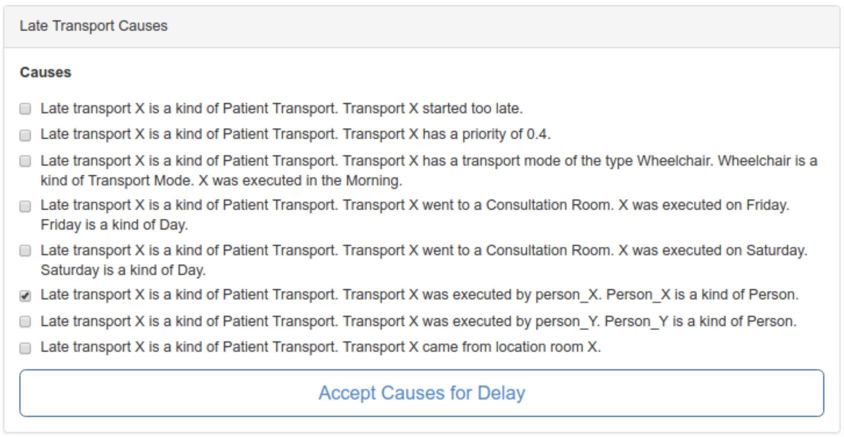


Figure A.13: The dashboard visualizing some of the characteristics of the dataset received from the hospitals.


Late Transport Causes

Causes

- [ ] Late transport X is a kind of Patient Transport. Transport X started too late.
- [ ] Late transport X is a kind of Patient Transport. Transport X has a priority of 0.4.
- [ ] Late transport X is a kind of Patient Transport. Transport X has a transport mode of the type Wheelchair. Wheelchair is a kind of Transport Mode. X was executed in the Morning.
- [ ] Late transport X is a kind of Patient Transport. Transport X went to a Consultation Room. X was executed on Friday. Friday is a kind of Day.
- [ ] Late transport X is a kind of Patient Transport. Transport X went to a Consultation Room. X was executed on Saturday. Saturday is a kind of Day.
- [✓] Late transport X is a kind of Patient Transport. Transport X was executed by person_X. Person_X is a kind of Person.
- [ ] Late transport X is a kind of Patient Transport. Transport X was executed by person_Y. Person_Y is a kind of Person.
- [ ] Late transport X is a kind of Patient Transport. Transport X came from location room X.

Figure A.14: Example of the verbalized rules for the hospital datasets.

transports planned on Wednesday/Thursday afternoon are mostly on time. However, the algorithm also reveals more sensitive data, such as certain persons or teams that cause more delays than other. Figure A.14 provides an example of the verbalization of some of the learned rules on the static dataset.

In Section A.6 we discuss how to deal with sensitive data and the advantages of having a system that provides explanations in these situations.

A.5.5 Comparison

Compared to ARM our technique scales very well as it does not need to generate rules regarding the whole dataset that later on needs to be filtered out. Furthermore, since we pick the features very carefully, the number of elements that are used to calculate the conjunctions is limited and thus faster. We also have a different algorithm for creating conjunctions of rules. ARM has multiple algorithms to achieve this but essentially checks if the support of the conjunction is above a certain threshold. We take a different approach where we only consider conjunctions that enable a greater drop in the support of the negative dataset, compared to the positive dataset. This means that the conjunction occurs more frequently in the positive dataset compared to the negative dataset and it also appears relatively more as a conjunction. Furthermore, because we filter the features early on by selecting only the most interpretable features, fewer conjunctions need to be calculated and less duplicate results are produced.

Compared to ILP (DL/CELOE) we take a statistical approach to test the coverage of the rule and not a logical one. This is possible by converting the graph to features, by the generation of the paths. This is more scalable than the ILP approach as the coverage is easier to compute. The CELOE algorithm generates possible rules from the ontology concepts and checks if it matches the dataset. We take another approach by starting from the dataset and generalize the found rules by incorporating the knowledge in the ontology. The ILP algorithm has also troubles to find rules in a noisy dataset. The ILP technique is more suited when one specific and possibly complex rule needs to be found. In our case, because multiple causes for delay exist in the dataset and noise can be present, the technique is not ideal. Furthermore, the configuration of our algorithm is easier than to configure the CELOE algorithm, as it requires to indicate the execution time.

It is clear that our approach is the fastest in execution time and also produces the most correct results. By incorporating the filtering techniques during the generation of the features, the number of duplicate rules is minimized. Also, the detection of the correct rules is higher than in the other algorithms. This allows to provide only the essential rules and give a clear overview and insights into the data.

A.6 Discussion

The proposed system is able to learn rules that identify possible causes of why transports in hospitals are late. By identifying the context in which transports are often late, we can predict which transports will be late in the future and more importantly avoid future delays. Both patients, staff members and hospital management benefit from more accurate scheduling. Currently, patients often have to wait before being picked up before or after a medical intervention, which is often uncomfortable. For staff members, it is stressful to see their tasks pile up as the assigned transports take longer than expected.

Furthermore, by explaining the cause as human-readable sentences, management gets understandable insights into their underlying hospital's mechanics. The fact that these causes can be understood by non-experts, allows management to be involved in the automated process and provides them with the final judgment.

The use of the semantic model allows to easily extend the platform and integrate additional data that could offer more accurate insights. For example, a new sensor could be added that captures the exact route a transport takes. This would allow to detect various bottlenecks in the transport routes, such as taking a specific elevator that is slow during certain times of the day (maybe visitors tend to use the elevator as well) or routes that pass a certain corridor in the hospital that is often very busy.

The learning phase can also detect sensitive data, such as certain staff members or teams that are underperforming. These are scenarios where management should open a discussion with their employees to find out the real cause of the problem. By first providing the causes to management for verification, management is provided with the opportunity to have this discussion and staff members are not rewarded for executing their tasks more slowly. In a fully automated system, the system would detect that a certain staff member takes more time and automatically assign more time to the tasks this staff member has to execute. Other insights can also provide opportunities for optimizations, e.g. transport towards certain specific locations that are always late could indicate that there is a structural problem in the department and perhaps a reorganization of the department would be beneficial.

The dashboard provides an easy access to the learning tool. By providing some graphical overviews of the transport distributions between timely and late transports, management can have a quick visual overview. By making the overviews adaptable through queries, the content can be easily adapted. However, the construction of these queries might not be trivial for non-technical persons. Therefore, we provide some basic queries and allow the option to monitor the transports that adhere to the previously selected causes of delays. This allows to quickly validate if the system is now assigning the required time to execute these transports more accurately.

A disadvantage of learning from past transports is that data about past transports need to be available. If management restructures the execution of transports, it takes time to see the influence in the learned causes. One solution to solve this is to only take the transports into account that were conducted from the time the restructuring took place. However, data about the transports is still necessary.

Besides late transports, there might also be cases where the transports are assigned too much time, i.e the transports arrive too early compared to the assigned delivery time. These transports can be identified in the same way we identified late transports. By alternating both identifying the early and late tasks, the system will converge to an optimal setting.

A.7 Conclusion & Future Work

In this paper, we propose a learning system that can indicate the causes of why certain hospital transports are late. Special precautions are taken to make sure that the learned causes can be explained to management, enabling management to remain in full control of the automated system. We have shown that our platform is capable of learning said causes and verbalize them in interpretable sentences for further inspection by the hospital management.

In future work, we wish to incorporate additional sensors to allow the detection of more accurate and complex rules. We also wish to further extend the usability of the dashboard. For example, allow management to easily construct the overview queries in a natural and interpretable manner for non-technical users.

Acknowledgment

This research was funded by the iMinds ICON AORTA project, co-funded by VLAIO, iMinds, Xperthis, Televic Healthcare, AZ Maria Middelares and Ziekenhuis Netwerk Antwerpen.

Authors' contributions

PB carried out the study, developed the algorithm, ran the experiments and drafted the manuscript. FO and FDT supervised the study, participated in its design and coordination and helped to draft the manuscript. All authors read and approved the final manuscript.

References

[1] Mladovsky P, Srivastava D, Cylus J, Karanikolos M, Evetovits T, Thomson S, et al. Health policy responses to the financial crisis in Europe edited by Philipa Mladovsky et al. 2012;.

[2] Karanikolos M, Mladovsky P, Cylus J, Thomson S, Basu S, Stuckler D, et al. Financial crisis, austerity, and health in Europe. The Lancet. 2013;381(9874):1323–1331.

[3] Hastreiter S, Buck M, Jehle F, Wrobel H. Benchmarking logistics services in German hospitals: a research status quo. In: Service Systems and Service Management (ICSSSM), 2013 10th International Conference on. IEEE; 2013. p. 803–808.

[4] Landry S, Philippe R. How logistics can service healthcare. In: Supply Chain Forum: An International Journal. vol. 5. Taylor & Francis; 2004. p. 24–30.

[5] Bates DW, Ebell M, Gotlieb E, Zapp J, Mullins H. A proposal for electronic medical records in US primary care. Journal of the American Medical Informatics Association. 2003;10(1):1–10.

[6] Marschollek M. Recent progress in sensor-enhanced health information systems – slowly but sustainably. Informatics for Health and Social Care. 2009;34(4):225–230.

[7] Atzori L, Iera A, Morabito G. The internet of things: A survey. Computer networks. 2010;54(15):2787–2805.

[8] Laskowski M, McLeod RD, Friesen MR, Podaima BW, Alfa AS. Models of emergency departments for reducing patient waiting times. PloS one. 2009;4(7):e6127.

[9] Marković N, Milinković S, Tikhonov KS, Schonfeld P. Analyzing passenger train arrival delays with support vector regression. Transportation Research Part C: Emerging Technologies. 2015;56:251–262.

[10] Rebollo JJ, Balakrishnan H. Characterization and prediction of air traffic delays. Transportation research part C: Emerging technologies. 2014;44:231–241.

[11] Xu J, Deng D, Demiryurek U, Shahabi C, Van Der Schaar M. Mining the situation: Spatiotemporal traffic prediction with big data. IEEE Journal of Selected Topics in Signal Processing. 2015;9(4):702–715.

[12] Silva R, Kang SM, Airoldi EM. Predicting traffic volumes and estimating the effects of shocks in massive transportation systems. Proceedings of the National Academy of Sciences. 2015;p. 201412908.

[13] Lecue F, Wu J. Explaining and predicting abnormal expenses at large scale using knowledge graph based reasoning. Web Semantics: Science, Services and Agents on the World Wide Web. 2017;44:89–103.

[14] Lecue F. Applying Machine Reasoning and Learning in Real World Applications. In: Reasoning Web International Summer School. Springer; 2016. p. 241–257.

[15] Fiegl C, Pontow C. Online scheduling of pick-up and delivery tasks in hospitals. Journal of Biomedical Informatics. 2009;42(4):624–632.

[16] Hanne T, Melo T, Nickel S. Bringing robustness to patient flow management through optimized patient transports in hospitals. Interfaces. 2009;39(3):241–255.

[17] Beaudry A, Laporte G, Melo T, Nickel S. Dynamic transportation of patients in hospitals. OR spectrum. 2010;32(1):77–107.

[18] Kergosien Y, Lente C, Piton D, Billaut JC. A tabu search heuristic for the dynamic transportation of patients between care units. European Journal of Operational Research. 2011;214(2):442–452.

[19] Huh J, Pollack M, Katebi H, Sakallah K, Kirsch N. Incorporating user control in automated interactive scheduling systems. In: Proceedings of the 8th ACM Conference on Designing Interactive Systems. ACM; 2010. p. 306–309.

[20] Gruber T. Toward principles for the design of ontologies used for knowledge sharing. International Journal of Human-Computer Studies. 1995;43(5-6):907–928.

[21] STRANG T. A context modeling survey. In: 1st International Workshop on Advanced Context Modelling, Reasoning and Management, 2004; 2004. p. 34–41.

[22] Vancroonenburg W, Esprit E, Smet P, Vanden Berghe G. Optimizing internal logistic flows in hospitals by dynamic pick-up and delivery models. In: Proceedings of the 11th international conference on the practice and theory of automated timetabling; 2016. p. 371–383.

[23] Nenov Y, Piro R, Motik B, Horrocks I, Wu Z, Banerjee J. RDFox: A highly-scalable RDF store. In: International Semantic Web Conference. Springer; 2015. p. 3–20.

[24] Dimou A, Vander Sande M, Colpaert P, Verborgh R, Mannens E, Van de Walle R. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: LDOW; 2014. .

[25] Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. In: Acm sigmod record. vol. 22. ACM; 1993. p. 207–216.

[26] Lavrac N, Dzeroski S. Inductive Logic Programming. In: International Conference on Inductive Logic Programming. Springer; 1994. p. 146–160.

[27] Bonte P, Ongenae F, Hoogstoel E, De Turck F. Mining semantic rules for optimizing transport assignments in hospitals. In: ISWC2016, the 15th International Semantic Web Conference; 2016. p. 1–6.

[28] De Raedt L, Kersting K. Statistical Relational Learning. Encyclopedia of Machine Learning. 2011;p. 916–924.

[29] Lehmann J, Auer S, Bühmann L, Tramp S. Class expression learning for ontology engineering. Web Semantics: Science, Services and Agents on the World Wide Web. 2011;9(1):71–81.

[30] Bonte P, et al. Learning Semantic Rules for Intelligent Transport Scheduling in Hospitals. In: European Semantic Web Conference (ESWC2016); 2016. .

[31] Nebot V, Berlanga R. Finding association rules in semantic web data. Knowledge-Based Systems. 2012;25(1):51–62.

[32] Galárraga LA, Teflioudi C, Hose K, Suchanek F. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In: Proceedings of the 22nd international conference on World Wide Web. ACM; 2013. p. 413–422.

[33] Galárraga L, Teflioudi C, Hose K, Suchanek FM. Fast rule mining in ontological knowledge bases with AMIE + +. The VLDB Journal. 2015;24(6):707–730.

[34] Androutsopoulos I, Lampouras G, Galanis D. Generating natural language descriptions from OWL ontologies: the NaturalOWL system. Journal of Artificial Intelligence Research. 2013;48:671–715.